University of Turin
School of Management and Economics
Master's Degree in
Quantitative Finance and Insurance

Optimizing the behavior of trading agents using genetic algorithms in a stock exchange simulation framework with real data.

Main supervisor:
Prof. Pietro Terna

Second supervisor:
Prof. Sergio Margarita

Candidate:
Gerson Massobrio

Academic Year 2012/2013

Contents

1 NetLogo and BehaviorSearch **7**
1.1 Genetic Algorithms 7
1.1.1 Introduction 7
1.1.2 Background 8
1.1.3 Structure 11
1.1.4 Limitations 16
1.2 BehaviorSearch 18
1.2.1 Overview 18
1.2.2 What is BehaviorSearch 18
1.2.3 How it works 19
1.3 BehaviorSearch Tutorial 20
1.3.1 Installation and Structure 20
1.3.2 The BehaviorSearch experiment editor 22
1.3.3 Run BehavorSearch 37
1.3.4 Examples 39
1.3.5 Comparison between search algorithms 54
1.3.6 Technical achievements 82

2 Stock exchange simulation and search of the optimal agents behavior **84**
2.1 User manual 84
2.1.1 g1 _ CDA _ basic _ model 84
2.1.2 Level Price Real Data Agents 86
2.1.3 Trend Agents 89
2.1.4 Volume Agents 91
2.1.5 Stop Loss Agents 92
2.1.6 Covered Agents 98
2.1.7 Bollinger Bands Agents 106
2.2 Basic Framework 108
2.3 Market structures 115

2.3.1 Market structures: basic framework plus one trading agents breed . . . 115
2.3.2 Market structures: basic framework plus more than one trading agents breed . . . 125
2.3.3 Agents effect on the market . . . 129
2.4 Simulations . . . 143
2.4.1 Comparisons between artificial and real market : AV.nlogo . 145
2.4.2 Comparisons between artificial and real market : AT.nlogo . 145
2.4.3 Comparisons between artificial and real market : AB.nlogo . 145
2.4.4 Comparisons between artificial and real market : ASL.nlogo 149
2.4.5 Comparisons between artificial and real market : AC.nlogo . 149
2.4.6 Comparisons between artificial and real market : ABT.nlogo 152
2.4.7 Comparisons between artificial and real market : A-C-SL.nlogo 152
2.4.8 Comparisons between artificial and real market : A-B-C-T-SL-V.nlogo . . . 155
2.5 Optimize Agents Behavior . . . 157
2.5.1 Preliminary BehaviorSearch test: the price structure . . . 158
2.5.2 BehaviorSearch test number 1: VolumeAgents . . . 160
2.5.3 BehaviorSearch test number 2: trendAgents . . . 162
2.5.4 BehaviorSearch test number 3: BBAgents . . . 164
2.5.5 BehaviorSearch test number 4: SLAgents . . . 167
2.5.6 BehaviorSearch test number 5: CoveredAgents . . . 170
2.5.7 BehaviorSearch test number 6: trendAgents + BBAgents . . 172
2.5.8 BehaviorSearch test number 7: SLAgents + CoveredAgents . 176
2.5.9 BehaviorSearch test number 8: all trading agents breeds . . 179
2.6 Statistical tests with R upon results . . . 186
2.6.1 ANOVA test on the fitness results of agents breeds . . . 188
2.6.2 Linear regressions between unconstrained fitness of breeds and constrained fitness of breeds . . . 191
2.7 Technical achievements . . . 206

3 Conclusions **210**

Aknowledgements

I would like to express my deep gratitude to Professor Pietro Terna, my main supervisor, for his patient guidance, enthusiastic encouragement and useful critiques of this experimental work.

Finally, I wish to thank my parents, my girlfriend, and my classmates, for their support and encouragement throughout my study.

Introduction

Uncertainty in economics is an unknown prospect of gain or loss, whether quantifiable as risk or not. Without it, household behavior would be unaffected by uncertain employment and income prospects, financial and capital markets would reduce to exchange of a single instrument in each market period, and there would be no communications industry.

Economic theories are frequently tested empirically, largely through the use of econometrics using economic data. The controlled experiments common to the physical sciences are difficult and uncommon in economics, and instead broad data is observationally studied; this type of testing is typically regarded as less rigorous than controlled experimentation, and the conclusions typically more tentative. However, the field of experimental economics is growing, and increasing use is being made of natural experiments.

Statistical methods such as regression analysis are common. Practitioners use such methods to estimate the size, economic significance, and statistical significance ('signal strength') of the hypothesized relation(s) and to adjust for noise from other variables. By such means, a hypothesis may gain acceptance, although in a probabilistic, rather than certain, sense. Acceptance is dependent upon the falsifiable hypothesis surviving tests. Use of commonly accepted methods need not produce a final conclusion or even a consensus on a particular question, given different tests, data sets, and prior beliefs.

Criticism based on professional standards and non-replicability of results serve as further checks against bias, errors, and over-generalization, although much economic research has been accused of being non-replicable, and prestigious journals have been accused of not facilitating replication through the provision of the code and data.

Prior to, and in the wake of the financial crisis, interest has grown in ABMs as possible tools for economic analysis. ABMs do not assume the economy can achieve equilibrium and "representative agents" are replaced by agents with diverse, dynamic, and interdependent behavior including herding. ABMs take a "bottom-up" approach and can generate extremely complex and volatile simulated economies. ABMs can represent unstable systems with crashes and booms that develop out

of non-linear (disproportionate) responses to proportionally small changes.

However, the idea of agent-based modeling was developed as a relatively simple concept in the late 1940s. Since it requires computation-intensive procedures, it did not become widespread until the 1990s.

In fact ABMs are typically implemented as computer simulations,either as custom software, or via ABM toolkits, and this software can be then used to test how changes in individual behaviors will affect the system's emerging overall behavior.

ABM

An agent-based model (ABM) is a class of computational models for simulating the actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) with a view to assessing their effects on the system as a whole. It combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming.

With the appearance of *StarLogo* in 1990, Swarm and *NetLogo* in the mid-1990s and *RePast* and *AnyLogic* in 2000, or *GAMA* in 2007 as well as some custom-designed code, modelling software became widely available and the range of domains that ABM was applied to, grew.

Citing Axtell (2006): 'Compactly, in agent based computational models, a population of data structures representing individual agents is instantiated and permitted to interact. One then looks for systematic regularities, often at the macro level, to emerge from the local interactions of the agents. The shorthand for this is that macroscopic regularities 'grow' from the bottom up. No equations governing the overall social structure are stipulated in multi-agent computational models, thus avoiding any aggregation or misspecification bias. Typically, the only equations present are those used by individual agents for decision-making. Different agents may have different decision rules and different information; usually, no agents have global information, and the behavioral rules involve bounded computational capacities, the agents are 'simple'. This relatively new methodology facilitates the modeling of agent heterogeneity, boundedly rational behavior, nonequilibrium dynamics, and spatial processes. A particularly natural way to implement agent-based models is through ' object-oriented' programming. '

NetLogo

NetLogo is an agent-based programming language and integrated modeling environment.

NetLogo was designed, in the spirit of the Logo programming language, to be "low threshold and no ceiling". It teaches programming concepts using agents in the form of turtles, patches, "links" and the observer. *NetLogo* was designed for multiple audiences in mind, in particular: teaching children in the education community, and for domain experts without a programming background to model related phenomena.

The *NetLogo* environment enables exploration of emergent phenomena. It comes with an extensive models library including models in a variety of domains, such as economics, biology, physics, chemistry, psychology, system dynamics. *NetLogo* allows exploration by modifying switches, sliders, choosers, inputs, and other interface elements. Beyond exploration, *NetLogo* allows authoring of new models and modification of existing models. *NetLogo* is freely available from the *NetLogo* website. It is in use in a wide variety of educational contexts from elementary school to graduate school. Many teachers make use of *NetLogo* in their curricula. *NetLogo* was designed and authored in 1999 by *Uri Wilensky*, director of Northwestern University's Center for Connected Learning and Computer-Based Modeling.

It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of agents all operating independently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from their interaction.

My thesis is mainly organized in two parts:

1. the fist one (Chapter 1) can be defined as a 'methodological' part. It summarizes the characteristics of the *genetic algorithm*, and describes the properties of *BehaviorSearch* software tool, showing some simple examples of how it interfaces with *NetLogo*.

2. The second one (Chapter 2), is a more 'empirical' part. It develops the interaction between *NetLogo* and *BehaviorSearch*, considering more complex problems. It treats the analysis with *BehaviorSearch* of the behavior and the effect of some categories of trading agents, inserted in a stock exchange simulation with real data, created with *NetLogo*.

   The results coming from the interaction of the two programs are finally examined using two statistical tools: ANOVA and linear regression.

Chapter 3 summarizes the results of the two parts, concluding my work.

Chapter 1

NetLogo and BehaviorSearch

1.1 Genetic Algorithms

1.1.1 Introduction

In the computer science field of artificial intelligence, a *genetic algorithm* (GA) is a search heuristic that mimics the process of natural selection. This heuristic (also sometimes called a metaheuristic) is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

Those alghorithms are generally based on: Holland (1992) two primary natural processes; natural selection and sexual reproduction. The first determines which members of a population survive to reproduce, and the second ensures mixing and recombination among the genes of their offspring. When sperm and ova fuse, matching chromosomes line up with one another and then cross over partway along their length, thus swapping genetic material. This mixing allows creatures to evolve much more rapidly than they would if each offspring simply contained a copy of the genes of a single parent, modified occasionally by mutation. Selection is simple: if an organism fails some test of fitness, such as recognizing a predator and fleeing, it dies.

Genetic algorithms find application in bioinformatics, phylogenetics, computational science, engineering, economics, chemistry, manufacturing, mathematics, physics, pharmacometrics and other fields.

In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are repre-

sented in binary as strings of 0s and 1s, but other encodings are also possible. Initially many individual solutions are (usually) randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, allowing the entire range of possible solutions (the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

Genetic algorithms are simple to implement, but their behavior is difficult to understand;citing Holland (1992) «*by harnessing the mechanisms of evolution, researchers may be able to 'breed' programs that solve problems even when no person can fully understand their structure*».

So in general it is difficult to understand why these algorithms frequently succeed at generating solutions of high fitness when applied to practical problems. *The building block hypothesis* (BBH) consists of:

1. A description of a heuristic that performs adaptation by identifying and recombining "building blocks", i.e. low order, low defining-length schemata with above average fitness.

2. A hypothesis that a genetic algorithm performs adaptation by implicitly and efficiently implementing this heuristic.

Goldberg describes the heuristic as follows:

> short, low order, and highly fit schemata are sampled, recombined (crossed over), and resampled to form strings of potentially higher fitness. In a way, by working with these particular schemata (the building blocks), we have reduced the complexity of our problem; instead of building high-performance strings by trying every conceivable combination, we construct better and better strings from the best partial solutions of past samplings.

Because highly fit schemata of low defining length and low order play such an important role in the action of genetic algorithms, we have already given them a special name: building blocks. Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks.

1.1.2 Background

To understand how the genetic algorithm properly works, I have to give some definition, about the concepts previously mentioned.

- **Search algorithms**: In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure.

  Algorithms for searching virtual spaces are used in constraint satisfaction problem, where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical equations and inequations. They are also used when the goal is to find a variable assignment that will maximize or minimize a certain function of those variables. Algorithms for these problems include the basic brute-force search (also called "uninformed" or "random" search), and a variety of heuristics that try to exploit partial knowledge about structure of the space, such as linear relaxation, constraint generation, and constraint propagation.

  A heuristic function, or simply a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.

  An important subclass are the local search methods, that view the elements of the search space as the vertices of a graph, with edges defined by a set of heuristics applicable to the case; and scan the space by moving from item to item along the edges, for example according to the steepest descent or best-first criterion, or in a stochastic search. This category includes a great variety of general metaheuristic methods, such as simulated annealing, tabu search, A-teams, and genetic programming, that combine arbitrary heuristics in specific ways.

- **Schemata**: A schema is a template in computer science used in the field of genetic algorithms that identifies a subset of strings with similarities at certain string positions. Schemata are a special case of cylinder sets; and so form a topological space. For example, consider binary strings of length 6. The schema 1**0*1 describes the set of all words of length 6 with 1's at the first and sixth positions a 0 at the fourth position. The * is a wildcard symbol, which means that positions 2, 3 and 5 can have a value of either 1 or 0. The order of a schema is defined as the number of fixed positions in the template, while the defining length $\delta(H)$ is the distance between the first and last specific positions. The order of 1**0*1 is 3 and its defining length is 5. The fitness of a schema is the average fitness of all strings matching the schema. The fitness of a string is a measure of the value of the encoded problem solution, as computed by a problem-specific evaluation function.

- **Cylinder sets** In mathematics, a cylinder set is the natural open set of a product topology. Cylinder sets are particularly useful in providing the base

of the natural topology of the product of a countable number of copies of a set. If V is a finite set, then each element of V can be represented by a letter, and the countable product can be represented by the collection of strings of letters. In general Consider the cartesian product $X = \prod_\alpha X_\alpha$, of topological spaces X_α, indexed by some index α. The canonical projection is the function $p_\alpha : X \to X_\alpha$ that maps every element of the product to its α component. Then, given any open set $U \subset X_\alpha$, the preimage $p_\alpha^{-1}(U)$ is called an open cylinder. The intersection of a finite number of open cylinders is a cylinder set. The collection of open cylinders form a subbase of the product topology on X; the collection of all cylinder sets thus form a basis.

- **Topological spaces**: In topology and related branches of mathematics, a topological space is a set of points, along with a set of neighbourhoods for each point, that satisfy a set of axioms relating points and neighbourhoods. The definition of a topological space relies only upon set theory and is the most general notion of a mathematical "space" that allows for the definition of concepts such as continuity, connectedness, and convergence. Other spaces, such as manifolds and metric spaces, are specializations of topological spaces with extra structures or constraints. Being so general, topological spaces are a central unifying notion and appear in virtually every branch of modern mathematics. The branch of mathematics that studies topological spaces in their own right is called point-set topology or general topology. The utility of the notion of a topology is shown by the fact that there are several equivalent definitions of this structure. The most commonly used is that in terms of open sets.

Open sets definition: Given such a structure, we can define a subset U of X to be open if U is a neighbourhood of all points in U.Let define N to be a neighbourhood of x if N contains an open set U such that x $\in$ U . A topological space is then a set X together with a collection of subsets of X, called open sets and satisfying the following axioms:
(1) The empty set and X itself are open.
(2) Any union of open sets is open.
(3) The intersection of any finite number of open sets is open.
The collection τ of open sets is then also called a topology on X, or, if more precision is needed, an open set topology. The sets in τ are called the open sets, and their complements in X are called closed sets. A subset of X may be neither closed nor open, either closed or open, or both. A set that is both closed and open is called a clopen set.

1.1.3 Structure

In Holland (1992) the GA structure is introduced by giving, first of all, the definition of *classifier system*; it consists in a set of rules, each of which performs particular actions every time its conditions are satisfied by some piece of information. From a general point of view, any program that can be written in a standard programming language such as *Python* can be rewritten as a *classifier system*.

Ferraris and Lamieri (2004) describe a non learning *classifier system* through four principal components:

- List of classifiers (population of classifiers).
- List of messages that plays the role of a 'message board' for communications and short term memory.
- Input interface (detector) that represents the environment state.
- Output interface (effector) that ensures interaction with the environment or its change.

At any time the classifier list can contain zero or more classifier. Each classifier consists of a string of fixed length and binary alphabet. A classifier list consists of a set of classifiers looking as follows:

$$condition1, condition2, ..., conditionN : action$$

When the condition part of the classifier matches the input message, activation of the classifier occurs, i.e. the classifier puts one or more messages on the message list. The output interface is a device or sub-program that receives action messages and on their basis performs manipulations with the environment.

Classifier system is being optimized by using learning rule called 'bucket brigade' and evolutionary algorithms (genetic algorithms). During learning process rules priorities (strengths) are changed. In case of success current and previous activated rules are encouraged. Evolutionary methods are used for new rules searching.

In general, to evolve classifier rules that solve a particular problem, one simply starts with a population of random strings of l's and 0's and rates each string according to the quality of its result. Depending on the problem, the measure of fitness could be business profitability, game payoff, error rate or any number of other criteria. High-quality strings mate; low-quality ones perish. As generations pass, strings associated with improved solutions will predominate.

The conditions and actions are represented by strings of bits corresponding to the presence or absence of specific characteristics in the rules' input and output.

For what concerns *Genetic Algorithms*, the chief problem is the construction of a "genetic code" that can represent the structure of different programs, just as DNA represents the structure of a person or a mouse. The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A typical genetic algorithm requires:

- a genetic representation of the solution domain;

- a fitness function to evaluate the solution domain.

A standard representation of each candidate solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The

main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations.

During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as the former process may be very time-consuming.

The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. For instance, in the knapsack problem one wants to maximize the total value of objects that can be put in a knapsack of some fixed capacity. A representation of a solution might be an array of bits, where each bit represents a different object, and the value of the bit (0 or 1) represents whether or not the object is in the knapsack. Not every such representation is valid, as the size of objects may exceed the capacity of the knapsack. The fitness of the solution is the sum of values of all objects in the knapsack if the representation is valid, or 0 otherwise.

To be more precise I give some notations,to define a genetic algorithm:

Let Ω be the space of length l binary strings, and let $n = 2l$. For u, v $\in \Omega$, let u $\oplus$ v denote the bitwise-and of u and v, and let u $\oplus$ v denote the bitwise-xor of u and v. Let $= u$ denote the ones-complement of u, and $\sharp u$ denote the number of ones in the binary representation of u.

Integers in the interval $[0, n) = [0, 2l)$ are identified with the elements of Ω through their binary representation. This correspondence allows Ω to be regarded as the product group

$$\Omega = Z_2 \times ... \times Z_2$$

where the group operation is $\oplus$. The elements of Ω corresponding to the integers $2^i, i = 0, ..., l-1$ form a natural basis for Ω.

The column vectors of length l form the elements of Ω. Let 1 denote the vector of ones (or the integer $2l1$). Thus, $u^T v = \sharp(u \oplus v)'$ `and` $= u = 1 \oplus u$.

For any u $\in \Omega$, let Ω_u denote the subgroup of Ω generated by $(2^i : u \oplus 2^i = 2^i)$. In other words, v $\in \Omega_u$ if and only if v $\oplus u = v$. For example, if $l = 6$, then $\Omega_9 = \{0, 1, 8, 9\} = \{000000, 000001, 001000, 001001\}$.

A schema is a subset of Ω where some string positions are determined (fixed) and some are not determined (variable). Schemata are traditionally denoted by

pattern strings, where a special symbol is used to denote a not determined bit. Using the symbol $*$ for this purpose: the schema denoted by the pattern string $10*01*$ is the set of strings {100010, 100011, 101010, 101011, }.

Alternatively a schema can also be defined as: the set $\Omega_u \oplus$ v, where u, v $\in \Omega$, and where $u \oplus v = 0$. In this notation, u is a mask for the variable positions, and v determines the fixed positions. For example, the schema $\Omega_{0,0,1,0,0,1} \oplus 100010$ would be the schema $10*01*$ described above.

This definition makes it clear that a schema $\Omega_u \oplus$ v with v $= 0$ is a subgroup of Ω, and a schema $\Omega_u \oplus$ v is a coset of this subgroup.

Following standard practice, we will define the order of a schema as the number of fixed positions. In other words, the order of the schema $\Omega_= u$ is $\sharp u$ (since u is a mask for the fixed positions).

A population for a genetic algorithm over length *l* binary strings is usually interpreted as a multiset (set with repetitions) of elements of Ω. A population can also be interpreted as a 2^l dimensional incidence vector over the index set Ω. if X is a population vector, then X_i is the number of occurrences of $i \in \Omega$ in the population. A population vector can be normalized by dividing by the population size. For a normalized population vector x, $\sum_i x_i = 1$. Let

$$\Delta = \{x \in R^n : \sum_i x_i = 1, x_i \geq 0 \texttt{ for any } i \in \Omega\}$$

Thus a normalized population vector is an element of Δ. Geometrically, Δ is the $n-1$ dimensional unit simplex in R^n . Note that elements of Δ can be interpreted as probability distributions over Ω.

If *expr* is a Boolean expression, then

$$[expr] = \{1 \texttt{ if expr is true}; 0 \texttt{ if expr is false}$$

The simple genetic algorithm can be described through a heuristic function G : $\Delta \to \Delta$. G contains all of the details of selection, crossover, and mutation. The simple genetic algorithm is given by:

1 Choose a random population of size r from Ω.

2 Express the population as an incidence vector X indexed over Ω.

3 Let y = G(X/r). (Note that X/r and y are probability distributions over Ω.)

4 for k from 1 to r do

5 Select individual i $\in \Omega$ according to the probability distribution y.

6 Add i to the next generation population Z.

7 endfor

8 Let X = Z.

9 Go to step 3.

If X is a population, then y = G(X/r) is the expected population after one generation of the simple genetic algorithm. In this framework we can relate to the *schema theorem*, that is a statement about the schema averages of the population y.

The heuristic function G can be written as the composition of three heuristic functions F, C, and U which describe selection, crossover, and mutation respectively. In other words,

$$G(x) = U(C(F(x))) = U \circ C \circ F(x).$$

Holland's Schema Theorem: it is widely taken to be the foundation for explanations of the power of genetic algorithms. It says that short, low-order schemata with above-average fitness increase exponentially in successive generations. The theorem was proposed by *John Holland* in the 1970s.

A general equation is:

$$E(m(H,t+1)) \geq \frac{m(H,t)f(H)}{a_t}[1-p].$$

Here $m(H,t)$ is the number of strings belonging to schema H at generation $t, f(H)$ is the observed fitness of schema H and a_t is the observed average fitness at generation t. The probability of disruption p is the probability that crossover or mutation will destroy the schema H. It can be expressed as:

$$p = \frac{\delta(H)}{l-1}p_c + o(H)p_m$$

where o(H) is the order of the schema, l is the length of the code, p_m is the probability of mutation and p_c is the probability of crossover. So a schema with a shorter defining length $\delta(H)$ is less likely to be disrupted.

An often misunderstood point is why the Schema Theorem is an inequality rather than an equality. The answer is in fact simple: the Theorem neglects the small, yet non-zero, probability that a string belonging to the schema H will be created "from scratch" by mutation of a single string (or recombination of two strings) that did not belong to H in the previous generation.

1.1.4 Limitations

- Repeated fitness function evaluation for complex problems is often the most prohibitive and limiting segment of artificial evolutionary algorithms. Finding the optimal solution to complex high dimensional, multimodal problems often requires very expensive fitness function evaluations. In real world problems such as structural optimization problems, one single function evaluation may require several hours to several days of complete simulation. Typical optimization methods can not deal with such types of problem. In this case, it may be necessary to forgo an exact evaluation and use an approximated fitness that is computationally efficient. It is apparent that amalgamation of approximate models may be one of the most promising approaches to convincingly use GA to solve complex real life problems.

- Genetic algorithms do not scale well with complexity. That is, where the number of elements which are exposed to mutation is large there is often an exponential increase in search space size. This makes it extremely difficult to use the technique on problems such as designing an engine, a house or plane. In order to make such problems tractable to evolutionary search, they must be broken down into the simplest representation possible. Hence we typically see evolutionary algorithms encoding designs for fan blades instead of engines, building shapes instead of detailed construction plans, airfoils instead of whole aircraft designs. The second problem of complexity is the issue of how to protect parts that have evolved to represent good solutions from further destructive mutation, particularly when their fitness assessment requires them to combine well with other parts.

- The "better" solution is only in comparison to other solutions. As a result, the stop criterion is not clear in every problem.

- In many problems, GAs may have a tendency to converge towards local optima or even arbitrary points rather than the global optimum of the problem. This means that it does not "know how" to sacrifice short-term fitness to gain longer-term fitness. The likelihood of this occurring depends on the shape of the fitness landscape: certain problems may provide an easy ascent towards a global optimum, others may make it easier for the function to find the local optima. This problem may be alleviated by using a different fitness function, increasing the rate of mutation, or by using selection techniques that maintain a diverse population of solutions, although the Wright (2011) No Free Lunch theorem proves that there is no general solution to this problem. A common technique to maintain diversity is to impose a "niche penalty", wherein, any group of individuals of sufficient similarity (niche

radius) have a penalty added, which will reduce the representation of that group in subsequent generations, permitting other (less similar) individuals to be maintained in the population. This trick, however, may not be effective, depending on the landscape of the problem. Another possible technique would be to simply replace part of the population with randomly generated individuals, when most of the population is too similar to each other. Diversity is important in genetic algorithms (and genetic programming) because crossing over a homogeneous population does not yield new solutions. In evolution strategies and evolutionary programming, diversity is not essential because of a greater reliance on mutation.

- Operating on dynamic data sets is difficult, as genomes begin to converge early on towards solutions which may no longer be valid for later data. Several methods have been proposed to remedy this by increasing genetic diversity somehow and preventing early convergence, either by increasing the probability of mutation when the solution quality drops (called triggered hypermutation), or by occasionally introducing entirely new, randomly generated elements into the gene pool (called random immigrants). Again, evolution strategies and evolutionary programming can be implemented with a so-called "comma strategy" in which parents are not maintained and new parents are selected only from offspring. This can be more effective on dynamic problems.

- GAs cannot effectively solve problems in which the only fitness measure is a single right/wrong measure (like decision problems), as there is no way to converge on the solution (no hill to climb). In these cases, a random search may find a solution as quickly as a GA. However, if the situation allows the success/failure trial to be repeated giving (possibly) different results, then the ratio of successes to failures provides a suitable fitness measure.

- For specific optimization problems and problem instances, other optimization algorithms may find better solutions than genetic algorithms (given the same amount of computation time). Alternative and complementary algorithms include evolution strategies, evolutionary programming, simulated annealing, Gaussian adaptation, hill climbing, and swarm intelligence (e.g.: ant colony optimization, particle swarm optimization) and methods based on integer linear programming. The question of which, if any, problems are suited to genetic algorithms (in the sense that such algorithms are better than others) is open and controversial.

1.2 BehaviorSearch

1.2.1 Overview

The *BehaviorSearch* software was initially developed as part of *Forrest Stonedahl*'s doctoral thesis research, with adviser *Uri Wilensky* at the *Center for Connected Learning and Computer-Based Modeling* at Northwestern University. It is an open source project.

The practice of designing and building new tools is crucial to computer science; compilers are an example of a software tool that fundamentally changed the landscape of computer science. However, many other tools have had substantial impact on the discipline, and society at large. One example of the success of tool building is *NetLogo* platform that *BehaviorSearch* interfaces with.

In this chapter, we will discuss the characteristics of *BehaviorSearch* , which is an open-source cross-platform tool that offers several search algorithms and search-space representations/encodings, and can be used to explore the parameter space of any *Agent Based Model* (ABM) written in the *NetLogo* language.

If you want to show the world a model that displays elephant-trunk-wiggling behavior, *BehaviorSearch* can help you find parameter settings that will do that. Does the discovery of such parameters mean you have developed a good model? Not necessarily. It only means that the behavior you sought exists somewhere in the parameter space.

BehaviorSearch aims to facilitate model analysis by making search and optimization techniques accessible to all modelers.

1.2.2 What is BehaviorSearch

BehaviorSearch is a software tool to help with automating the exploration of agent-based models (ABMs), by using genetic algorithms and other heuristic techniques to search the parameter-space.

BehaviorSearch interfaces with the popular *NetLogo* ABM development platform, to provide a low-threshold way to search for combinations of model parameter settings that will result in a specified target behavior.

Model exploration works through four steps:

1. Design a quantitative measure for the behavior you're interested in.
2. Choose parameters to vary and what ranges are allowed.
3. Choose a search algorithm and run it.
4. Examine the results (what parameters most affect this behavior?)

1.2.3 How it works

According to Stonedahl and Adviser-Wilensky (2011), *BehaviorSearch* general features are:

1. **Parameter-type flexibility.** *BehaviorSearch* is capable of searching a combination of numerical (discrete/continuous), boolean, and categorical parameters. This is an important feature, since ABM parameters often take various forms, and are not constrained to always be of uniform type.

2. **Search method variety.** *BehaviorSearch* offers several different search algorithms and search space representations that users can employ. It has been designed as a general tool for applying any type of metaheuristic search algorithm to explore ABM parameter spaces. At present, *BehaviorSearch* supports the following search algorithms: random search, stochastic hill climbing, simulated annealing, and two variants of the genetic algorithm (generational GA and steady-state GA). This flexibility is important since different approaches can be more or less effective for exploring different models.

3. **Best-checking.** *BehaviorSearch* provides built-in support of best-checking, to prevent users of the software from being misled by high fitness values resulting from ABM stochasticity (and so that users can easily detect if the search algorithm is being misled).

4. **Multi-resolution data output**. *BehaviorSearch* can collect and store data at various levels of detail: recording eachmodel run performed, each fitness evaluation, each time a new 'best' is found, as well as the final best parameter settings at the end of each search. While novices can effectively use *BehaviorSearch* by simply looking at the final best parameters found, more advanced users can dig deeper into the search process and the results and parameters examined along the way.

5. **Parametric derivatives.** Built-in support for approximating derivatives of a behavioural objective function with respect to a specified parameter. This is useful for detecting phase transitions and critical points in the parameter space.

6. **Parallelization and multi-threading support.** *BehaviorSearch* was designed from the ground up with multi-threaded support for parallel searching, offering improved performance for multi-processor / multi-core computers. As the number of cores in desktop computers proliferates, harnessing this parallelism becomes a crucial performance issue.

7. **Extensibility**. *BehaviorSearch* was developed using an extensible object-oriented framework, allowing new search algorithms and search space representations to be easily added.

1.3 BehaviorSearch Tutorial

BehaviorSearch has been released under an open-source license.

The *BehaviorSearch* tools is also supported by the accompanying project website, located at `www.behaviorsearch.org/`. This website provides additional resources, such as a summary of features, information about new releases, links to relevant papers, and a contact form for user feedback. This site also links to a Google Code open source project website, with an issue/bug tracker, and access to the source code.

To deeper explain the characteristics of *BehaviorSearch* mentioned in the previous section, I will describe with some examples how *BehaviorSearch* works on *NetLogo* platform.

1.3.1 Installation and Structure

In an effort to make the software easy to use, the first step is making it easy to install. Since *BehaviorSearch* requires *BehaviorSearch* to perform model runs, it needs to reside in a sub-folder of the NetLogo installation folder. This can be a challenge, particularly on variants of the Windows operating system, where users may have difficulty in

finding the *NetLogo* installation folder, and may not have write-access privileges to modify its contents. As a result, there is a graphical executable installer for Windows to simplify this installation process. (Installation on Mac/Linux computers is also reasonably straightforward, and generally just requires dropping a folder into the *NetLogo* application directory).

BehaviorSearch has a modular architecture, that permits changes to one part of the code without affecting others.

At a finer level of detail, the *BehaviorSearch* codebase (written in Java) is divided into eight packages for organizational purposes:

- *bsearch.algorithms* - contains all of the search algorithms.
- *bsearch.app* - contains the main code driving the *BehaviorSearch* application.
- *bsearch.evaluation* - contains code for handling

  fitness evaluation and

  fitness caching.

- *bsearch.nlogolink* - handles all of the communication with the *NetLogo* platform.

- *bsearch.representations* - contains all of the search space representations.

- *bsearch.space* - contains a representation of the parameter space.

- *bsearch.test* - a package that contains unit testing.

- *bsearch.util* - a package containing miscellaneous utility functions.

A schematic representation of the *BehaviorSearch* structure can be seen in Figure 1.1:

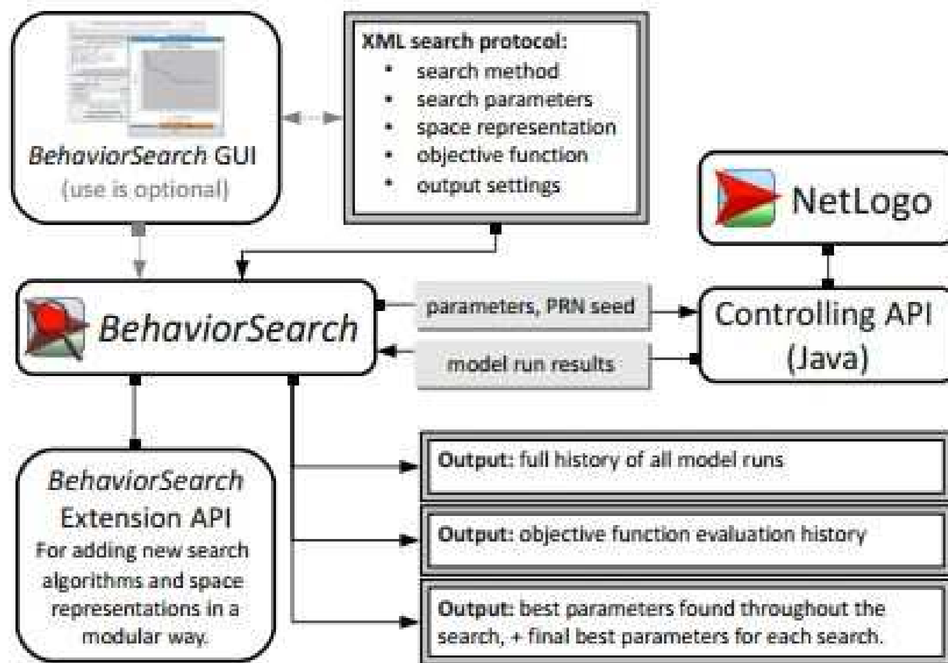

Figure 1.1: Design schematic of the *BehaviorSearch* architecture.

As shown in figure 1.1, *BehaviorSearch* contains an extensions API. This API provides a clean interface for extending its capabilities via new search algorithms and search space representations, which will help support both continued research and any special needs of end users of the tool.

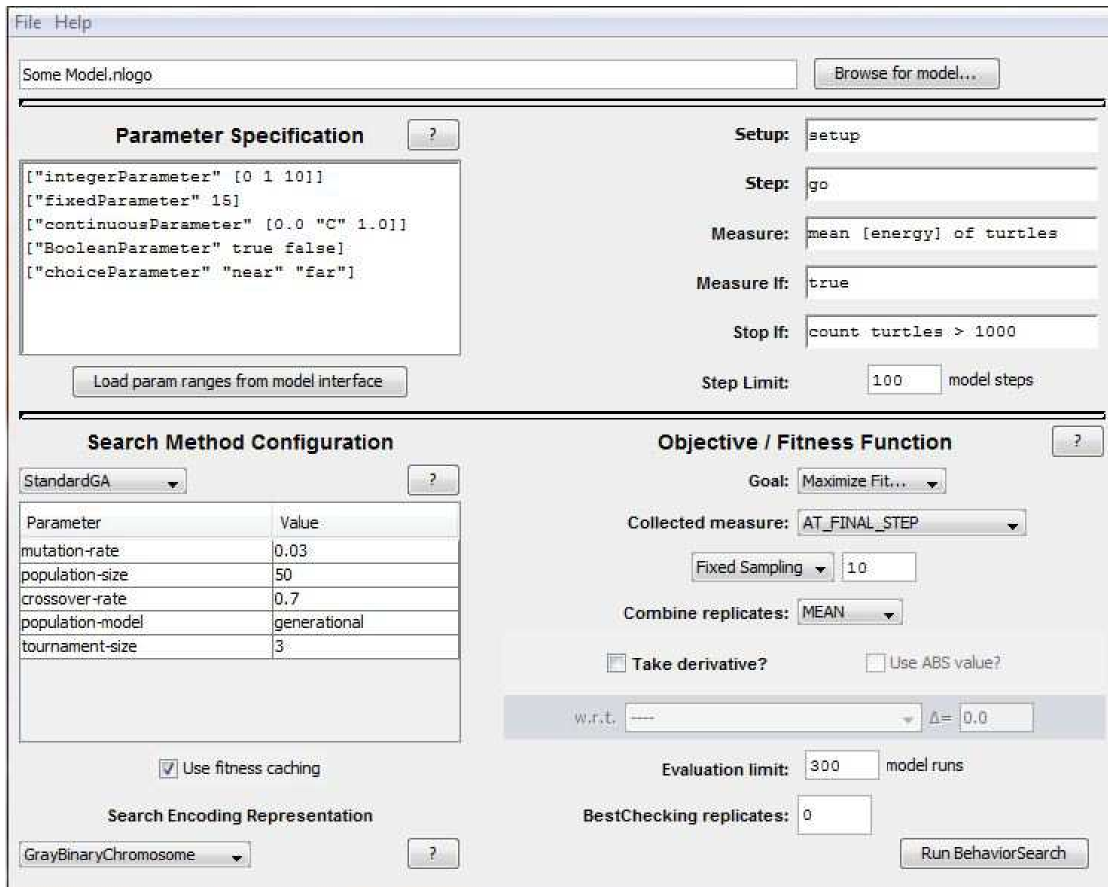

Figure 1.2: *BehaviorSearch Experiment Editor*

When you first open *BehaviorSearch*, the window that appears is the *BehaviorSearch Experiment Editor*. *BehaviorSearch* is centered around the paradigm of an experiment (or search protocol), which contains all of the information necessary to specify how the search should be performed on a model.

The *BehaviorSearch GUI* helps you create, open, modify, and save these experiments (stored as files with the ".bsearch" extension). The *BehaviorSearch* engine is separated from the GUI layer, and does not depend on it.

The *BehaviorSearch Experiment Editor* is shown in Figure 1.2:

1.3.2 The BehaviorSearch experiment editor

1. **First step: load a NetLogo program**. In order to start the *BehaviorSearch* analysis on a *NetLogo* program, first of all we have to load such

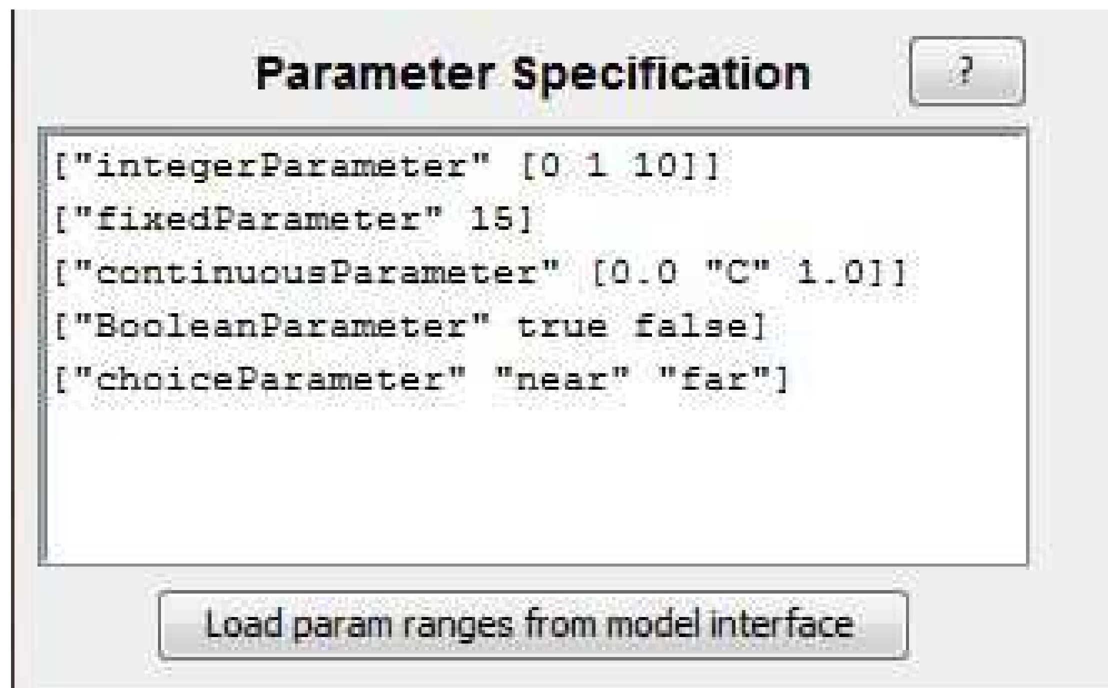

Figure 1.3: *Parameter specification*

a program clicking the '*Browse for model*' button at the top of the *BehaviorSearch Experiment Editor*

Then we can save our analyses as .bsearch files and load them clicking 'file' and then 'open'; this procedure works only if the .nlogo file is saved in the same folder of the .bsearch file.

The 'help' botton contains a direct link to the *BehaviorSearch* tutorial on `www.behaviorsearch.org`.

2. **Parameter Specification**.The next step is to specify settings, or ranges of settings, for each of the model's parameters. The easiest way to get started is to click the 'Load param ranges from model interface' button, which will automatically extract the parameters/ranges that are included in our model's interface tab (i.e. SLIDERS, CHOOSERS, and SWITCHES of our *NetLogo* program).

   The syntax in this case is the following:

- `['parameter-name'[parameter-range]]`. For example in figure 1.2 we have the slider with names `'integerparameter'`: for each kind of variable like this, the parameter-range is specified as

  `[starting-point increment ending-point]`;

  for example in the case of `'integerparameter'` the vision parameter ranges from 0 up to 10, by increments of 1 (Those kind of parameter are allowed to variate during the *BehaviorSearch* analysis).
- `['parameter-name' parameter-value]`. In figure 1.3 we have the slider named `'fixedParameter'` with this kind of syntax: the value of the parameter is fixed and does not ranges during the *BehaviorSearch* analysis; for example in the case of `'fixedParameter'` on figure 1.3, it is keep fixed at fifteen.

  In *BehaviorSearch* it is also possible to specify a continuous range for a parameter, by using "C" for the increment; in this case the syntax is: `['parameter-name'[sart 'C' stop]]`. An example can be shown in figure 1.3; the variable `continuousParameter` ranges from 0.0 to 1.0 with continuous values, specified with the command `'C'`.
- `['parameter-name' true false]` for Boolean parameters. An example is in figure 1.3: the variable `BooleanParameter` can be `true` or `false`.
- `['parameter-name' 'choice1' 'choice2'... ]` for discrete-choice parameters. Those kind of parameters are a generalization of the Boolean parameters (we can have multiple situations or conditions that can be satisfied).An example of this kind of parameter is given in figure 1.2, through the 'choices' `'near'` and `'far'` of the variable `choiceParameter`.

The parameters (that are allowed to variate, such as `'integerParameter'`) loaded for the *BehaviorSearch* analysis determine the size of the search space; this space can be seen as a multidimensional space in which each dimension is represented by a variable parameter (for example if we have only two variable parameters the search space can be represented graphically through x and y cordinates).

To determine the size of the search space we have just to : (1) calculate the total range of each parameter, it is a number; (2) multiply those numbers.

For example

BehaviorSearch is a useful tool when you have a parameter space that's too large too enumerate, and you're willing to use heuristic search methods to try to find parameters that yield behavior that you're interested in.

Setup: setup

Step: go

Measure: mean [energy] of turtles

Measure If: true

Stop If: count turtles > 1000

Step Limit: 100 model steps

Figure 1.4: *Measure specification*

3. **Specifying a measure** After the parameter specification, (including their range), the next step is to define which kind of variable we want to measure, as a function of such parameters; those information can be given in the six boxes on the upper-left side of the *BehaviorSearch Experiment Editor*. The cells for *Specifying a measure* are shown with more detail in Figure 1.4.

Looking at figure 1.4, starting from the top, we have:

- **Setup**: it refers to the name, in our *NetLogo* program, of the setup command (often it is a button called 'setup'). This command is the one that creates the framework in which the *NetLogo* program is going to be implemented/started.
- **Step**: it identifies the *NetLogo* command or commands, that let the program run (in general in *NetLogo*, it is called 'go' and it is a procedure that follows the 'setup' one). If this procedure contains the command `tick`, one step corresponds to one tick.

- **Measure**: it is the variable that we want to analyse with *BehaviorSearch*;it is a *NetLogo* expression, which somehow quantifies the behavior that we are interested in searching for.The measure can consist of any numeric *NetLogo* expression, what is important is that the measure is correlated with the behavior we would like to elicit from the model.
- **Measure If**: it is a (not compulsory) condition on the previous measure; only if that condition is satisfied, the *BehaviorSearch* analysis can be implemented (for example it can refer to : another *NetLogo* variable included in the 'step'; a true/false condition; a condition controlling on which steps the measure takes place, i.e. a condition on `ticks`).
- **Stop If**: This command works in the same way of 'Measure if', with the difference that it defines a stop condition for the model (it is optional).
- **Step Limit**: On *NetLogo*, we usually identify procedures with buttons in the interface; clicking a button, means calling all *NetLogo* commands included in such button once. But if we want to let the button work continuously, we can click on it with the right mouse button, then chose `Edit`, and then click on the option `forever`.

  Instead in *BehaviorSearch* we must specify the number of clicks on the 'go' button (we have not the option 'forever'), writing down the number of steps of the *NetLogo* program, that *BehaviorSearch* will consider, in its analysis.

4. **Search Method Configuration**. This part is not linked to the *NetLogo* program; it includes a set of *BehaviorSearch* options for what concern the algorithm that will be implemented for the search. Each algorithm owns a different number of parameters that can be modified(when you select one kind of algorithm, each window is full filled with the custom values for those parameters(figure 1.5)).

   The structure of the search method configuration can be shown in the graph of Figure 1.5.

With reference to figure 1.5, starting from the left side of the graph on the top, we have a button that permits to chose four kind of search algorithms for the *BehaviorSearch* analysis:

StandardGA . (custom): it is the genetic algorithm; its characteristics have been discussed in section 1; its convergence to an optimal solution has been guaranteed in by the *Holland schema theorem* (section 1.3).

The algorithm depends on five input parameters.

Search Method Configuration

StandardGA ▾ ?

Parameter	Value
mutation-rate	0.03
population-size	50
crossover-rate	0.7
population-model	generational
tournament-size	3

☑ Use fitness caching

Search Encoding Representation

GrayBinaryChromosome ▾ ?

Figure 1.5: *Search Method Configuration*

- `mutation-rate`: mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set too high, the search will turn into a primitive random search.
- `population-size`: the population size depends on the nature of the problem; traditionally, the initial population is generated randomly, allowing the entire range of possible solutions (the search space). During each successive generation, a proportion of the existing population is selected to breed a new generation. In our case the population-size is the number of individuals allowed in each generation. The value of population size must be an integer included in the interval [1, 1000].
- `crossover-rate`: In genetic algorithms, crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next.The word crossover is often intended as a process of taking more than one parent solutions and producing a child solution from them. There are different methods for selection of the chromosomes, in our case the crossover-rate is the probability of using two parents when creating a child (otherwise the child is created asexually).
- `population-model`: 'generational', 'steady-state-replace-random', or 'steady-state-replace-worst':
  'generational' means the whole population is replaced at once;
  'steady-state' means that only one single individual is replaced by reproduction each iteration. The individual being replaced may be randomly-chosen, or the current worst.
- `tournament-size`: During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as the former process may be very time-consuming.

Tournament selection involves running several 'tournaments' among a few individuals chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. Selection pressure is easily adjusted by changing the tournament size. If the tournament size is larger, weak individuals have a smaller chance to be selected (Usually 2 or 3 is a good value).

MutationHillClimber :In computer science, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution.

This algorithm is also defined in Holland (1992) as: «*One conventional technique for exploring such a landscape is hill climbing: start at some random point, and if a slight modification improves the quality of your solution, continue in that direction; otherwise, go in the opposite direction. Complex problems, however, make landscapes with many high points. As the number of dimensions of the problem space increases, the countryside may contain tunnels, bridges and even more convoluted topological features. Finding the right hill or even determining which way is up becomes increasingly difficult.* »

It depends on the following parameters:

- `mutation-rate`: controls the probability of mutation; it works in the same way of the mutation-rate of genetic algorithms.
- `restart-after-stall-count`: if the hill climber makes some number (restart-after-stall-count) of unsuccessful attempts to move to a random neighbor, it assumes it is trapped at a local optimum in the space, so it restarts by jumping to a new random location anywhere in the search space.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighbouring configuration) but it is not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space).

SimulatedAnnealing : is a generic probabilistic metaheuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more efficient than exhaustive enumeration, provided that the goal is merely to find an

acceptably good solution in a fixed amount of time, rather than the best possible solution.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects, both are attributes of the material that depend on its thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy. While the same amount of cooling brings the same amount of decrease in temperature it will bring a bigger or smaller decrease in the thermodynamic free energy depending on the rate that it occurs, with a slower rate producing a bigger decrease.

This search algorithm is similar to a hill climbing approach, except that a downhill (inferior) move may also occur, but only with a certain probability based on the temperature of the system, which decreases over time.

At each step, the simulated annealing heuristic considers some neighbouring state *s'* of the current state *s*, and probabilistically decides between moving the system to state s' or staying in state s. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

From a mathematical point of view, the probability of making the transition from the current state *s* to a candidate new state *s'* is specified by an acceptance probability function $P(e, e', T)$ that depends on the energies $e = E(e)$ and $e' = E(e')$ of the two states, and on a global time-varying parameter T called the temperature. States with a smaller energy are better than those with a greater energy. The probability function P must be positive even when *e'* is greater than *e*. This feature prevents the method from becoming stuck at a local minimum that is worse than the global one.

In general, in order to apply the simulated annealing to a specific problem, one must specify the following parameters: the *state space*, the *energy (goal) function E()*, the *candidate generator procedure neighbour*, the *acceptance probability function P()*, the *annealing schedule temperature* and *initial temperature*. These choices can have a significant impact on the method's effectiveness. Unfortunately, there are no choices of these parameters that will be good for all problems, and there is no general way to find the best choices for a given problem.

In *BehaviorSearch*, simulated annealing depends on four parameters:

- `mutation-rate`: it affects how much mutation occurs when choosing a candidate location for moving.
- `restart-after-stall-count`: if it doesn't manage to move to a new location after X attempts, reset the temperature, jump to a random location in the search space and try again. This parameter represents the number X of attempts.
- `initial-temperature`: the system's initial 'temperature' (a reasonable choice would be the average expected difference in the fitness function's value for two random points in the search space).
- `temperature-change-factor`: the system's current 'temperature' is multiplied by this factor (which needs to be less than 1) after each move. (Using this exponential temperature decay means that temperature will approach 0 over time. Unfortunately, the optimal rate for the temperature to decrease varies between problems).It defines the *annealing schedule temperature*.

The other parameters, i.e. the *candidate generator procedure neighbour*, the *acceptance probability function P()*, are fixed, and cannot be modified in *BehaviorSearch* analysis.

RandomSearch : it is a family of numerical optimization methods that do not require the gradient of the problem to be optimized and Random Search can hence be used on functions that are not continuous or differentiable. Such optimization methods are also known as direct-search, derivative-free, or black-box methods.

Let $f : \Re^n \rightarrow \Re$ be the fitness or cost function which must be minimized. Let $x \in \Re^n$ designate a position or candidate solution in the search-space. The basic Random Search algorithm can then be described as:

(1).Initialize x with a random position in the search-space.

(2).Repeat the following operation until a termination criterion is met:

- Sample a new position y from the hypersphere of a given radius surrounding the current position x;
- If (f(y) < f(x)) then move to the new position by setting x = y.

(3). Now x holds the best-found position.

Because of its characteristics and its degree of randomness, the random search does not have any input parameter on *BehaviorSearch Experiment Editor*.

Use Fitness Caching: this controls whether the search algorithm memorizes the result of the objective (fitness) function every time it gets evaluated, so

that it doesn't have to recompute it if the search returns to those exact same parameter settings again.

Since running ABM simulations can be time-consuming (especially when dealing with large agent populations for many ticks), turning on "fitness caching" can potentially be a considerable time-saver. However, because ABMs are usually stochastic, each time a point in the space is re-evaluated, the search process would get a new independent estimation of the value at that location.

The last button on the bottom of figure 2.5, contains four search space encodings.

- *StandardBinaryChromosome*: in this encoding, every parameter is converted into a string of binary digits, and these sequences are concatenated together into one large bit array. Mutation and crossover then occur on a per-bit basis.

- *GreyBinaryCromosome*: due to the *Hamming distance* properties of Gray codes, they are sometimes used in genetic algorithms. They are very useful in this field, since mutations in the code allow for mostly incremental changes, but occasionally a single bit-change can cause a big leap and lead to new properties.The Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. In another way, it measures the minimum number of substitutions required to change one string into the other, or the minimum number of errors that could have transformed one string into the other.

  GreyBinaryChromosome is similar to StandardBinaryChromosome, except that numeric values are encoded to binary strings using a Gray code, instead of the standard "high order" bit ordering. Gray codes have generally been found to give better performance for search representations, since numeric values that are close together are more likely to be fewer mutations away from each other.

  It is a binary numeral system where two successive values differ in only one bit. The reflected binary code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

- *MixedTypeChromosome*: this encoding most closely matches the way that one commonly thinks of the ABM parameters. Each parameter is stored separately with its own data type (discrete numeric, continuous

numeric, categorical, boolean, etc). Mutation applies to each parameter separately (e.g. continuous parameters use Gaussian mutation, boolean parameters get flipped).

- *RealHypercubeChromosome*: this encoding exists mainly to facilitate the (future) use of algorithms that assume a continuous numeric space ,such as Particle Swarm Optimization. particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. PSO optimizes a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its local best known position but, is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

  In the RealHypercubeCromosome encoding, every parameter (numeric or not) is represented by a "real-valued" continuous variable; this representation allows them to be applied even when some of the model parameters are not numeric.

5. **Objective / Fitness Function**. The *fitness function* defines how the *Measure*,i.e. the variable that we want to analyse (explained in point 3), must converge, in order to meet our work objectives. The convergence is related to an optimization problem that we want to implement on the search space.

   We know how to collect the data(measure specification), but now we need to turn it into an objective function ("fitness function"). This procedure can be done completing the table in Figure 1.6.

- `Goal`: here you specify your objective, i.e maximize or minimize the fitness function. This fitness function refers to the variable `Measure` related to Figure 1.4.
- `Collected measure`: during one model run, we may have collected the measure multiple times; you can condense all those values in the following ways(the time interval considered for the different measures across steps, i.e. mean, median, minimum, maximum, variance, sum, is referred to the command `Step Limit` ; the starting time for the mean calculation can be

Figure 1.6: *Objective / Fitness Function*

expressed in `Measure if`, specifying the number of ticks; both commands are referred to Figure 1.4):

- AT_FINAL_STEP: it reports the last measure calculated as final result of the analysis; it is useful if you are only interested in the last measure that was recorded.
- MEAN_ACROSS_STEPS: it reports the mean of the multiple measures implemented;
- MEDIAN_ACROSS_STEPS: it reports the median across steps;
- MIN_ACROSS_STEPS: it reports the minimum value measured across steps ;
- MAX_ACROSS_STEPS: it reports the maximum value measured across steps;
- VARIANCE_ACROSS_STEPS: it reports the variance of the values obtained across steps;
- SUM_ACROSS_STEPS: it reports the sum of all values calculated across steps.

The calculation of different kind of measures across steps refers to those values, that are obtained during the *BehaviorSearch* analysis. How start this analysis, is explained in the next section.

- `Fixed sampling`: how many times should the model be run? Running the model once may not give representative results, so you may want to perform multiple replicate runs (with different initial random seeds), and collect behavioral measures from each of them.

  Increasing this value will obviously rise the lasting of the *BehaviorSearch* analysis.

- `Combine replicates`: it refers to the `Fixed sampling` number. If you are doing multiple replicate runs of the model, you have to combine those results, in order to get a single number for the chosen objective function. This single number should incorporate the most useful information given by the multiple replicates. It can be calculated as:

  - MEAN: it is just the simple average of the replicates; it is, in general a good estimator for a realisation of the sample.
  - MEDIAN: may be a better choice if your measure occasionally yields extremal (too high or too low) outlier values, which you'd like to ignore.

- MIN/MAX: that is the opposite of median; you are interested in parameters that cause extreme behavior; it may mean the lowest(MIN) or the greatest(MAX) value obtained in the replicates.
- VARIANCE: In probability theory and statistics, variance measures how far a set of numbers is spread out. A small variance indicates that the data points tend to be very close to the mean and hence to each other, while a high variance indicates that the data points are very spread out from the mean and from each other. Its usefulness depends on the object of our analysis: choices may be useful for finding parameters for which there is volatility in whether the model exhibits a behavior or not. Such volatility might indicate a phase transition between two regimes of model behavior.
- STDEV: in statistics and probability theory, the standard deviation shows how much variation or dispersion from the average exists. A low standard deviation indicates that the data points tend to be very close to the mean; a high standard deviation indicates that the data points are spread out over a large range of values. It is the square root of the variance;it works in the same way of variance, except that the fitness function values will be in the same units of the original parameter, which may be preferable for human interpretation.

• `Take derivative?`: sometimes you would like to find a point in the parameter space where the change in your behavioral measure is maximized (or minimized) with respect to a small change in some parameter. Such places may indicate a phase transition, critical point, or leverage point. The Take derivative? option allows you to maximize/minimize the approximate derivative of your fitness function with respect to a specified parameter(`w.r.t.` that stays for 'with respect to') and a specified delta ($\Delta =$, i.e. the change amount).

  If you choose the special value '@MUTATE@' then finds a neighbouring point in the search space using mutation from the parameter settings being evaluated, with the mutation rate specified by delta.

  This operation can be considered as a partial derivative of a function of several variables; it is the derivative with respect to one of those variables, with the others held constant.

  The partial derivative with respect to x_1 of a function in $\Re^n$,i.e. $f(x_1, x_2, ...x_n)$, can be expressed as: $\frac{\delta f}{\delta x_1}$. In our case the $\Delta =$ is the δx_1, where x_1 is the parameter specified in the cell `w.r.t.`; the value that we want to calculate , corresponds to the variation of our fitness function, i.e. δf.

- `Use ABS value?`: if it is checked, then the reported difference is always positive.

- `Evaluation limit`: it corresponds to the number of model runs; after this number the *BehaviorSearch* analysis stops.

- `BestChecking replicates`: the number of additional replicate model runs that should be performed to get an unbiased estimate of the true objective function value, each time the search algorithm finds a new set of parameters that it thinks is "better" than any previous set.

  The motivation for this is that ABMs are usually stochastic, and when sampling a measure a small/finite number of times (such as 5, in our example here), there is likely to still be some "noise" in the objective function. Thus a search algorithm may appear to be making progress, finding better and better parameter settings, when in fact the better results are due to random noise. Using BestChecking replicates can help you identify when this is the case.

  Also, since new "bests" are found relatively infrequently, you can usually afford to specify a higher number of BestChecking replicates than you can for normal sampling, yielding more statistically significant reading of the objective function as the best parameters that the search found.

  BestChecking replicates are not counted against the total "model run" limit for the search. These replicates are extrinsic to the search process, but are included in the output results to evaluate the search performance, and verify the objective function values that are obtained.

1.3.3 Run BehavorSearch

After the completion of the *BehaviorSearch Experiment Editor*, to start the analysis you have to click on the button `Run BehaviorSearch` (shown in figure 1.6 down in the lower right corner of the window); the table that will appear to you is called *Choose experiment running option*, and can be seen in Figure 1.7.

Looking at figure 1.7, we have to complete the following five cells.

- `Output file system`: it should be written here the address where to save the output data from the search. More in deep, a number of files will be created, each starting with this same file name 'stem'.

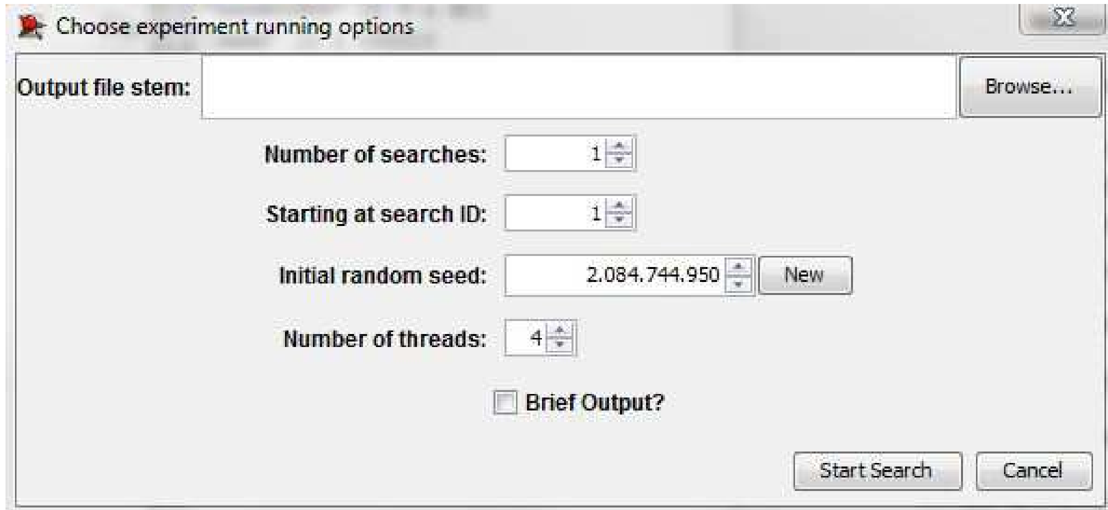

Figure 1.7: *Choose experiment running option*

- `Number of searches`: it represents the number of times the search is repeated. A single search may not find the best parameter; additional searches improve confidence (but it will require additional time to be implemented).

- `Starting at search ID`: it is a number that identifies the results of the current search(it works as an ID); this number is saved in the `Output file system`.

- `Initial random seed`: a random seed (or just seed) is a number (or vector) used to initialize a pseudo-random number generator. Starting the search with the same random seed,keeping all the parameters in the *BehavorSearch Experiment Editor* unchanged, will always lead to the same results; this is useful to repeat exately the same searches, or changing the algorithm of the search, to compare the effectiveness of the different searching algorithms (explained in section 1.3.2 point 4) for a given fitness function.

- `Number of threads`: this number indicates the number of processors/cores used for the analysis; for a multi-core/multi processor computer the running time for the search should be lower. The number of threads does not affect the results obtained, only the time spent.

`Brief Output?`: *BehaviorSearch*'s default behavior is to create a variety of output data files (discussed below), some of which can be quite large (containing the results of all model runs and all objective function evaluations). With this option suppresses the creation of the two largest output files.

1.3.4 Examples

Although *BehaviorSearch* is though as a tool to evolve models, in an ABM framework, it can be also used as a combinatorial search tool.

In fact, ABMs are typically implemented as computer simulations, either as custom software, or via ABM toolkits, and this software can be then used to test how changes in individual behaviors will affect the system's emerging overall behavior.

However, to let the reader understand deeply how *BehaviorSearch* works, I will proceed by analysing three very simple optimisation problems: I will show you, with some basic examples, how some *NetLogo* programs can be analysed by *BehaviorSearch*.

All these programs create a framework of simple optimization problems in two discrete variables. It means that I have the range of x, the range of y, and the function f(x,y); the objective of the search is to find the optimal point of the function f(x,y).

It means that the whole space can be represented by three dimensions, i.e. x, y, f(x,y).

An optimization problem with discrete variables is known as a *combinatorial optimization problem*. In a combinatorial optimization problem, we are looking for an object such as an integer, permutation or graph from a finite (or possibly countable infinite) set (to convert the problem into an optimization problem with continuous variables (in $\Re^2$) it should be sufficient to change the variable step using 'C'; it is explained in section 1.3.2 point 2).

The name combinatorial search is generally used for algorithms that look for a specific sub-structure of a given discrete structure, such as a graph, a string, a finite group, and so on. The term combinatorial optimization is typically used when the goal is to find a sub-structure with a maximum (or minimum) value of some parameter. (Since the sub-structure is usually represented in the computer by a set of integer variables with constraints, these problems can be viewed as special cases of constraint satisfaction or discrete optimization; but they are usually formulated and solved in a more abstract setting where the internal representation is not explicitly mentioned.)

The three basic examples, that I will show you in the next section, are named *localH*, *localH2*, *localH3*.

Note that all the references to loading or search times required for next *BehaviorSearch* analyses (of Section 1.3) , are related to a computer with the following characteristics:

- *Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz 2.50 GHz*

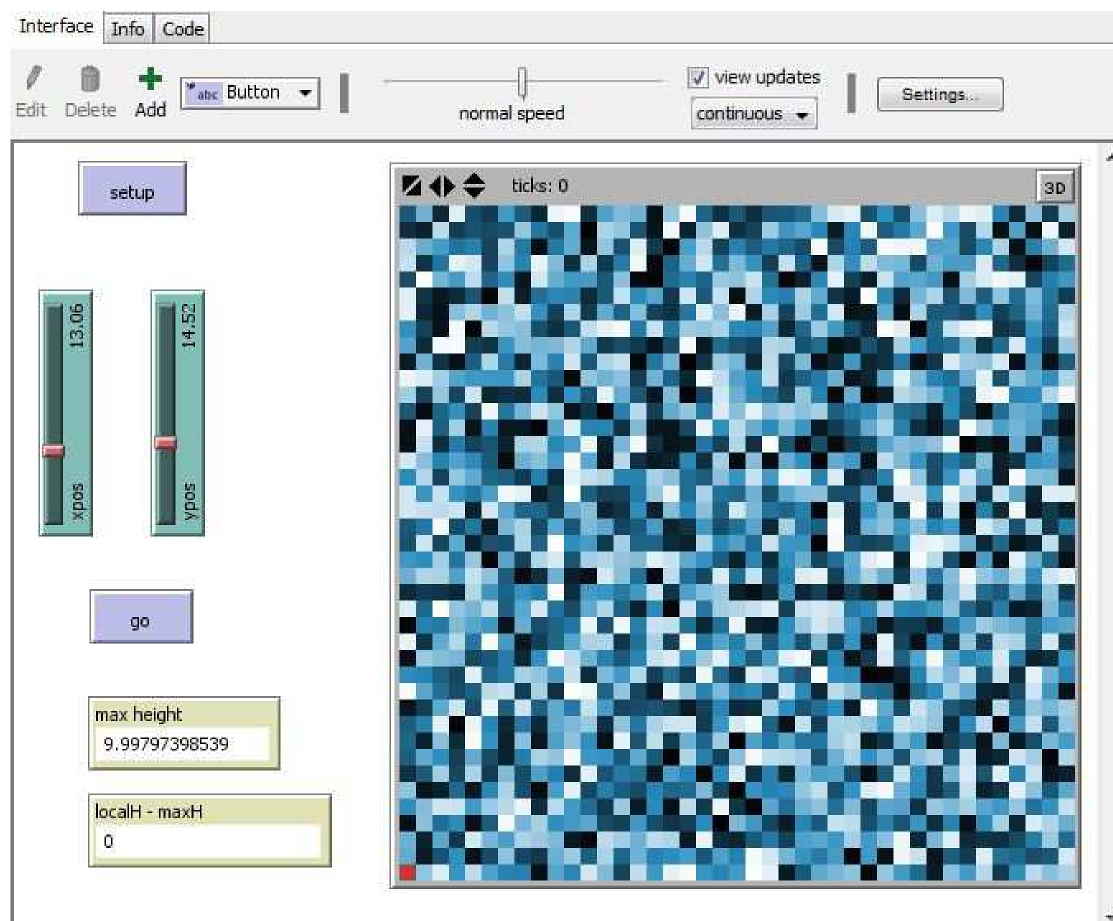

Figure 1.8: *localH.nlogo* interface

- *RAM 6,00 GB*

Case 1

localH.nlogo : the range of x and y coordinates is given through the two sliders `xpos` (position of x), and `ypos` (position of y). The third dimension z (f(x,y)) is built by the *NetLogo* program, by reading its values from the `base.txt` file. The *NetLogo* program looks as follows:

```
globals [maxH localDiff]

patches-own [height]
```

```
to setup
   __clear-all-and-reset-ticks
   file-open "base.txt"
   while [not file-at-end?]
     [
        ask patch file-read file-read
                      [set height file-read+
                        set pcolor 90 + height]
           ]
   file-close
   crt 1 [set size 1 set color red set shape "circle"]
   set maxH 0
   ask patches
     [if height >= maxH
         [set maxH height]
     ]
end
to go
    ask turtle 0
      [set xcor xpos
       set ycor ypos
       set localDiff height - maxH]
end
```

Most important parts of code:

- The function f(x,y) is created with the values red from the file, through the cycle:

```
while [not file-at-end?]
        [
         ask patch file-read file-read [set height file-read
```

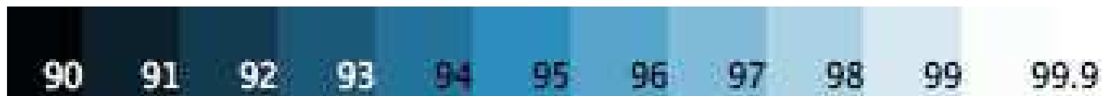

Figure 1.9: *NetLogo* table of colors: blue

```
                                    set pcolor 90 + height]
      ]
```

In *NetLogo*, you can define colors with numbers. Moreover the program associates different highs, to different colors. The benchmark used in our case is showed in Figure 1.9.

- The global variable `maxH` represents the maximum value of f(x,y) calculated with the commands:

```
ask patches
  [if height >= maxH
      [set maxH height]
  ]
```

- It is created one agent with the command

```
crt 1 [set size 1 set color red set shape "circle"].
```

With the `go` button, the program moves this agent in the *NetLogo* world-space, according to the coordinates specified by the two sliders `xpos` and `ypos`. The related commands are:

```
ask turtle 0
  [set xcor xpos
   set ycor ypos
```

- The variable `localDiff` reports the difference between the maximum value of f(x,y) and the current position of the agent. (when you click the button `setup` the agent position is set equal to the origin (0,0). The related command is:

```
set localDiff height - maxH]
```

The numbers contained in the file `base.txt` are created through a simple *Python* code, saved in the *base.py* file.

base.py

```
import random
f=open("base.txt","w")
        for i in range(41):
            for j in range(41):
               print >>f, i, j, random.random()*10
f.close()
```

In the code above, `f` is the variable that contains the command that opens the file in writing mode (`"w"`); the two `for` cycles are implemented to build the whole space, that is obtained by creating $41 \times 41 = 1681$ random numbers with the command `random.random()*10` (the command `random.random()` returns the next random floating point number in the range [0.0 , 1.0)).

localH.bsearch : In our case *BehaviorSearch* analysis is performed in order to find the maximum value of f(x,y) previously defined. Since we know in advance the value of the maximum value of the function from the file `base.txt` (it is also evaluated in the *NetLogo* program, and showed in the interface through a monitor), I will use as 'fitness function' the variable `localDiff`.

Indeed the *BehaviorSearch Experiment Editor* completed, is showed in Figure 1.10.

In this analysis the major part of the commands given to the *BehaviorSearch Experiment Editor* are default commands that are loaded when you click `Browse for model...` for the first time; the only exception is represented by the *Parameter specification* (remember to always press the button

`load param ranges from model inteface` to chose the correct variables for the analysis) and *Specifying a measure.*

In this example the search space is composed by the two variables `xpos` and `ypos` and its size is $4100 \times 4100 = 16810000$.

The `Step limit` is set equal to one, because the optimization problem lasts only one *NetLogo* tick (it is not a problem that needs to run continuously to get the solution).

In practice in our case with *StandardGA*, *BehaviorSearch* will create an initial population of random positions in the space, where each position corresponds to a particular scenario obtained with one press of the button `go`.

Then the population is evolved 5000 times (i.e. the `Evaluation limit`) through the genetic algorithm.

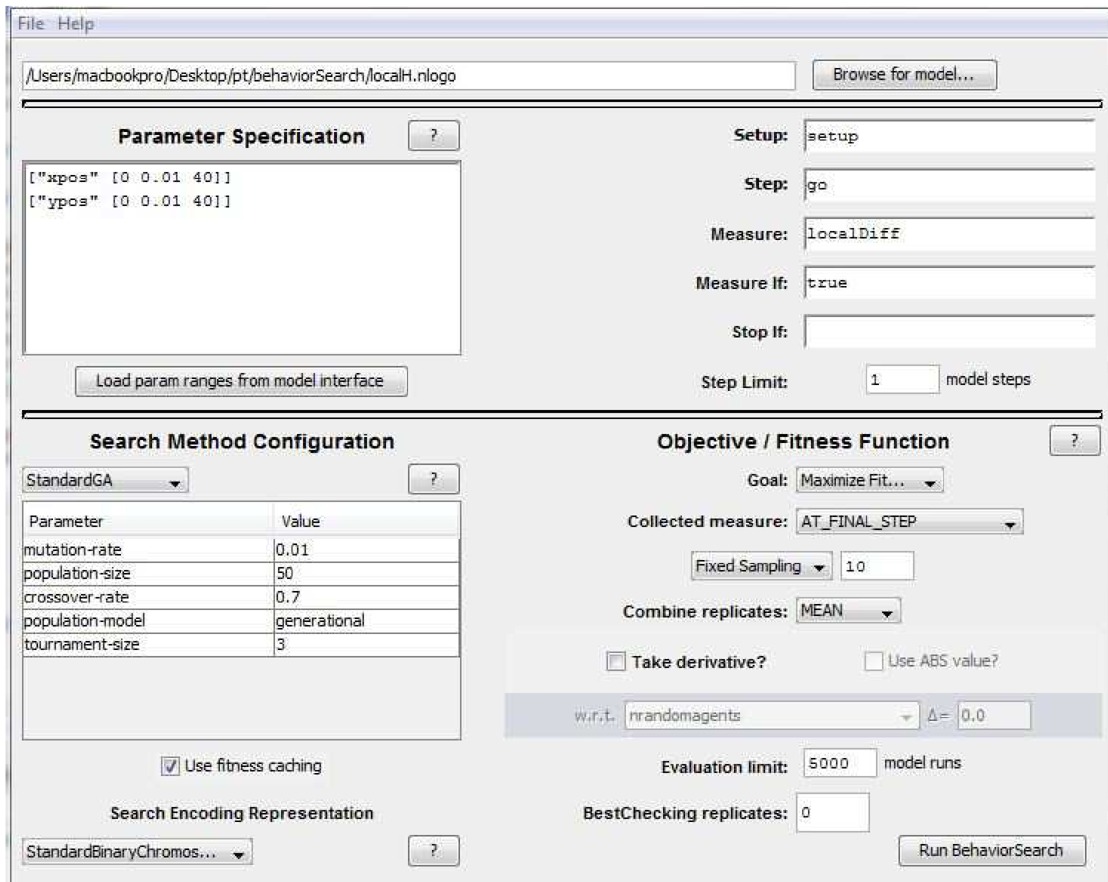

Figure 1.10: *localH.bsearch*

After the completion of the table

Choose experiment running option, the true search can finally start.

Some results can be seen in Figures 1.11, 1.12; here you can see that the results change depending on the `seed` chosen (keeping all other commands fixed).

The main analysis can be seen in the **Search Progress** table:

here it is possible to view the evolution of the search through a graph where the x axis represents the number of model runs, and the y axis is the Fitness function.

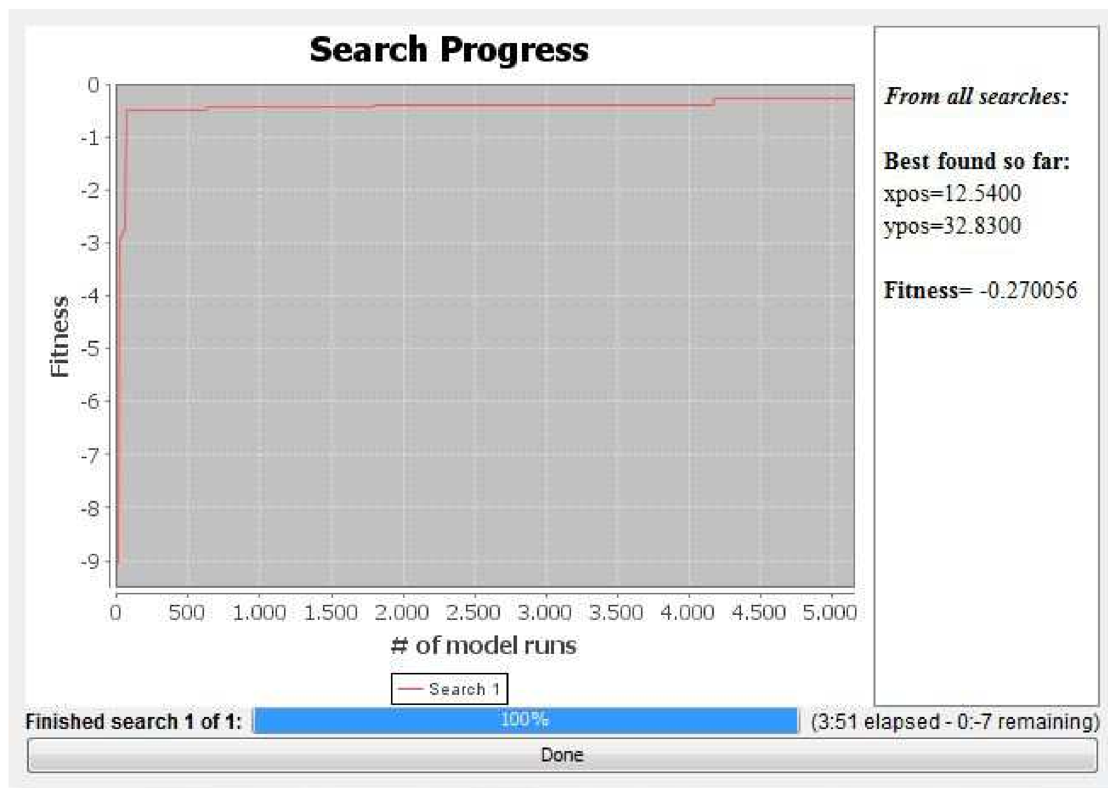

Figure 1.11: Result with initial random seed -448.067.020

Figure 1.12: Result with initial random seed 693.127.994

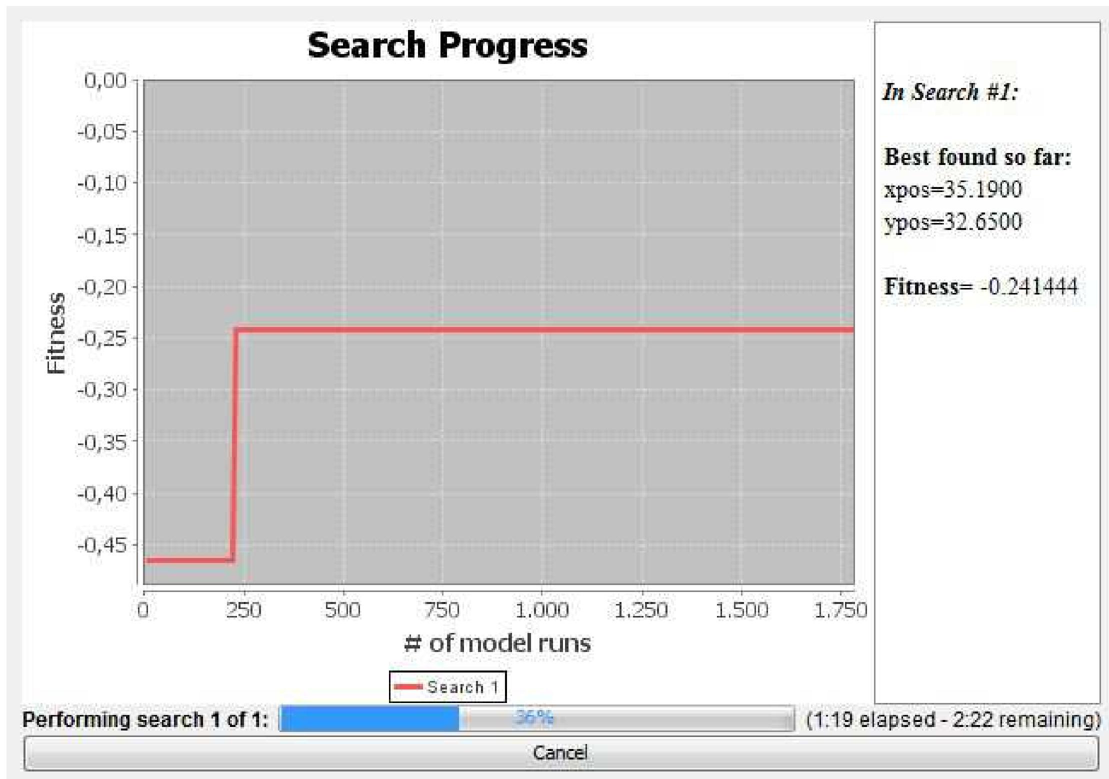

Figure 1.13: Partial result

In the rectangle on the right side of the Search Progress table, we have the values of the parameters loaded in the `Parameter Specification` table (section 1.3.2 point 2), corresponding to the Fitness value calculated so far.

Moreover consider the *BehaviorSearch* results ,with reference to Figure 1.13. We can see:

- **Performing search 1 of 1**(2 of 2): it is the percentage of completion of the search (showed through the blue bar). It can be stopped at any time by clicking the `Cancel` button.

  Note that once stopped, the search cannot be restarted.

Now consider the Figure 1.14; here we have:

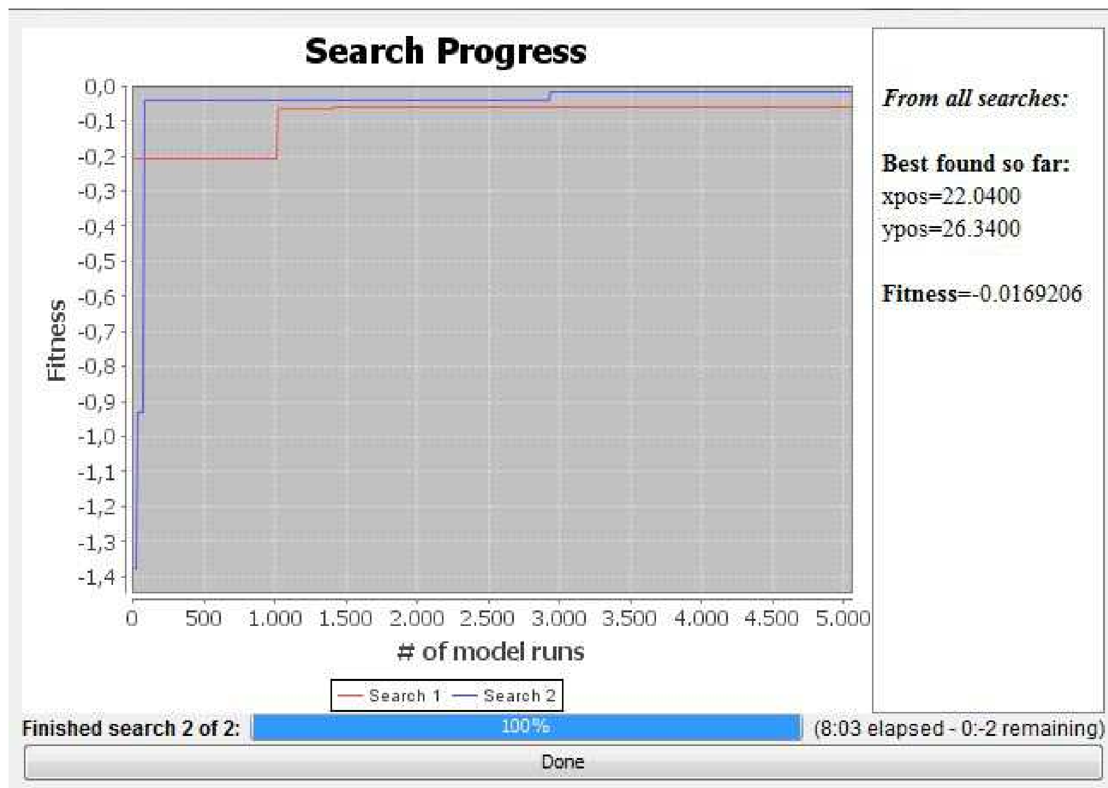

Figure 1.14: Result with *Number of Searches* = 2

- `Finished search 2 of 2`: the number `2 of 2` refers to the command given in the `Choose experiment: running options:` showed in Figure 1.7.

  Improving the number of searches will lead to better results, since in this case the different searches are not completely independent: each analysis takes into account the result obtained in the previous one.

  In the example of Figure 1.14 the two searches are showed in the `Search Progress` table (blue and red line).

Consider that the value of `max heigh` is 9.999; we expect a *Fitness function* as close as possible to zero. Performing a lot of searches with the *StandardGA*, I can conclude that the value of the *Fitness function* lays on average in an interval of [-0.3 -0.02].

Those results seem to confirm the convergence of the *GA*.

Case 2

localH2.nlogo :this program is very similar to the previous one, *localH.nlogo*; the only thing that changes is the formation of of the variable `heigh`: it is no more generated through a file, but through the following code.

```
ask patches
   [set height pxcor * 0.12499 +  pycor * 0.12499}
    set pcolor height + 90]
```

(`pxcor` and `pycor` are 'patch x coordinate', 'patch y coordinates', as in the previous case *localH.nlogo*).

With this structure we can manually calculate the variable `max heigh`, that is in fact equal to

$$'max'pxcor \times 0.12499 +' max'pycor \times 0.12499 = 9.999$$

(with 'max' pxcor = 40 = 'max' ycor)

Moreover since the value of `heigh` is conditioned by the patch coordinates, it increases smoothly as the coordinates increase, generating the graph showed in Figure 1.15.

We can say that the function that generates the space of the program *localH2* is monotone, although we are considering discrete variables.

(more in deep if we want to obtain graphs as in Figures 1.8 , 1.15, with the origin centred in the down-left corner, we must click with the right mouse

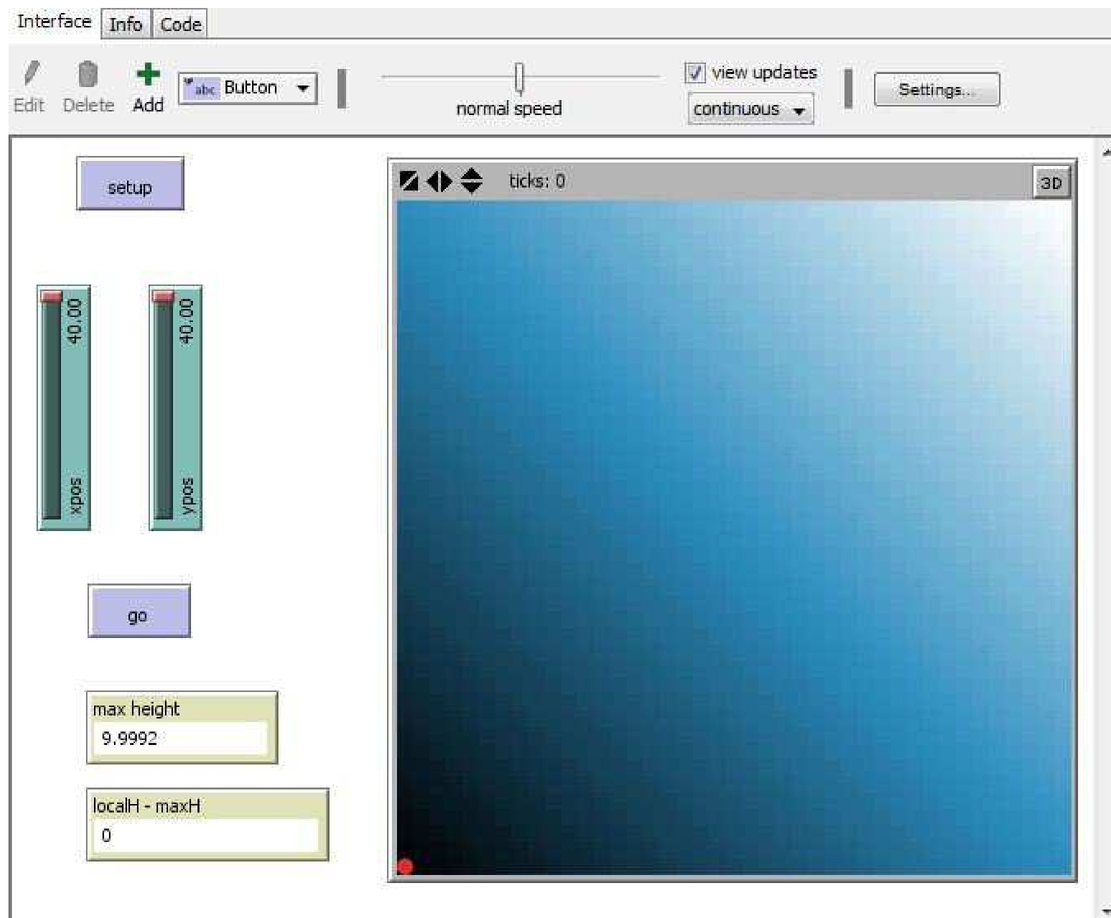

Figure 1.15: *localH2.nlogo* interface

button on the *NetLogo* monitor (world) choosing `edit` and then modify the command `Location of origin`, choosing the option `corner`).

localH2.bsearch : as in *localH.bsearch* we are looking for a value of the variable `localDiff` as close as possible to zero.

The *BehaviorSearch Experiment Editor* is equal to the one showed in Figure 1.10.

An example of *BehaviorSearch* result, for `localH2`, can be seen in Figure 1.16.

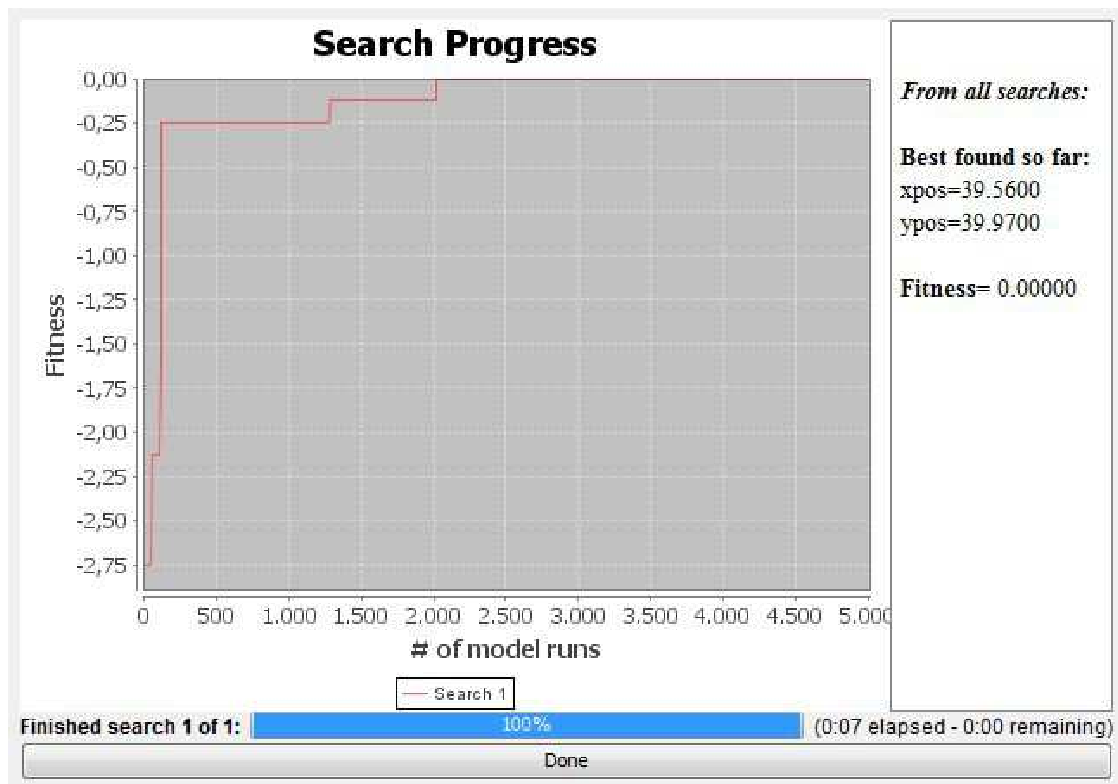

Figure 1.16: *localH2.bsearch* result

We can note that the *StandardGA* works significantly better and faster than in the previous case with *localH*, although the search space has the same size. In this case the genetic algorithm often leads to a fitness equal to zero (as in Figure 1.16).

This because the conformation of the search space(distribution of the different heigh in these examples) strongly influences the results of the search, making the different searching algorithms (section 1.3.2, point 4) more or less effective.

Case 3

localH3.nlogo : this program works as the previous *localH.nlogo* and *localH2.nlogo*; as before, the only difference is in the construction of the third dimension (heigh), that is formed with the following code:

```
ask patches
   [set height pxcor * 0.1 + pycor * 0.1 +
    sin (pxcor * 30) + sin (pycor * 30)
    if height < 0 [set height 0]
    set pcolor height + 90]
```

In this case we cannot calculate the maximum value of the variable `height` as in *localH2* . In fact if we consider the max `pxcor` and the max `pycor` we get:

$heigh = 40 \times 0.1 + 40 \times 0.1 + sin(1200) \times 2 = 9.73$

that is less than the max heigh. The effect of this code on the patch color, in the *NetLogo* interface, can be seen in Figure 1.17; here we can also see that max height is equal to 9.8.

For *localH3*, the function that characterizes the search space is oscillating; but it is also smoothed (since we are in discrete time it does not make sense to define the function 'monotone in intervals' because we are considering discrete variables).

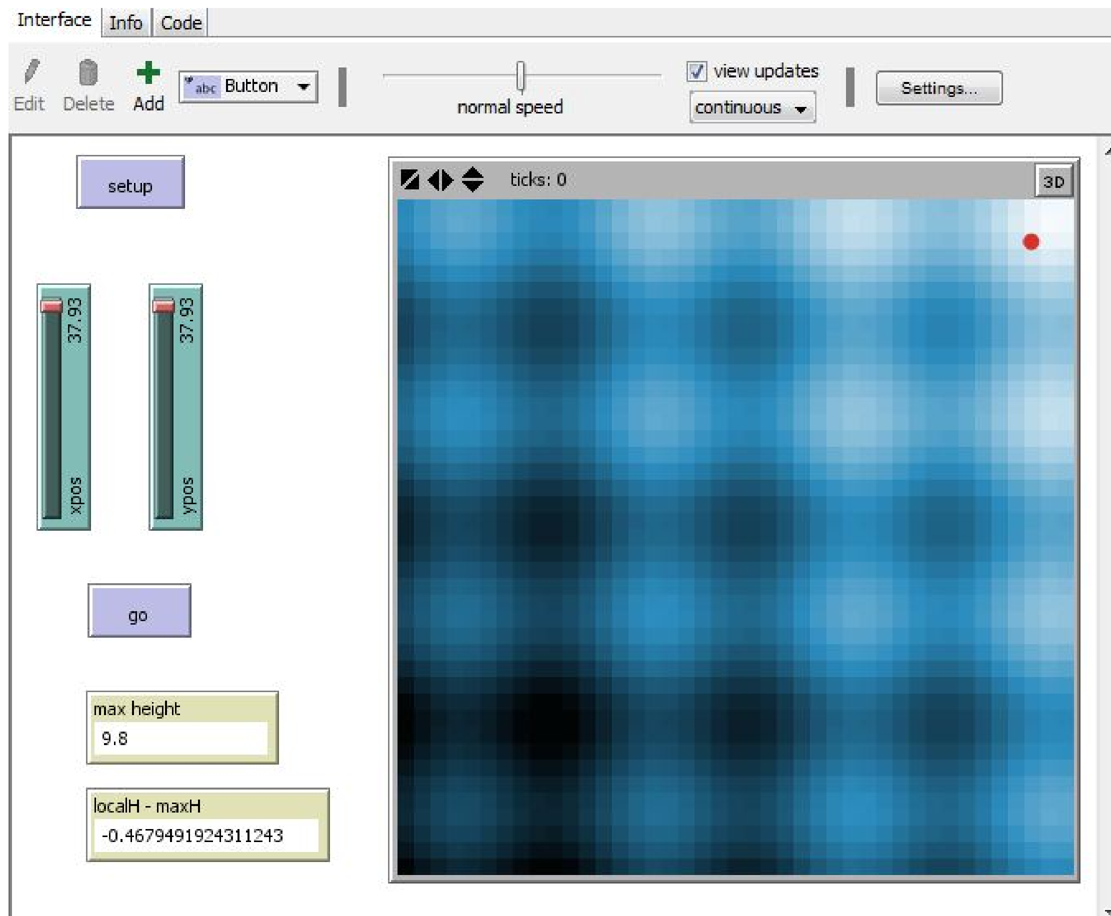

Figure 1.17: *localH3.nlogo* interface

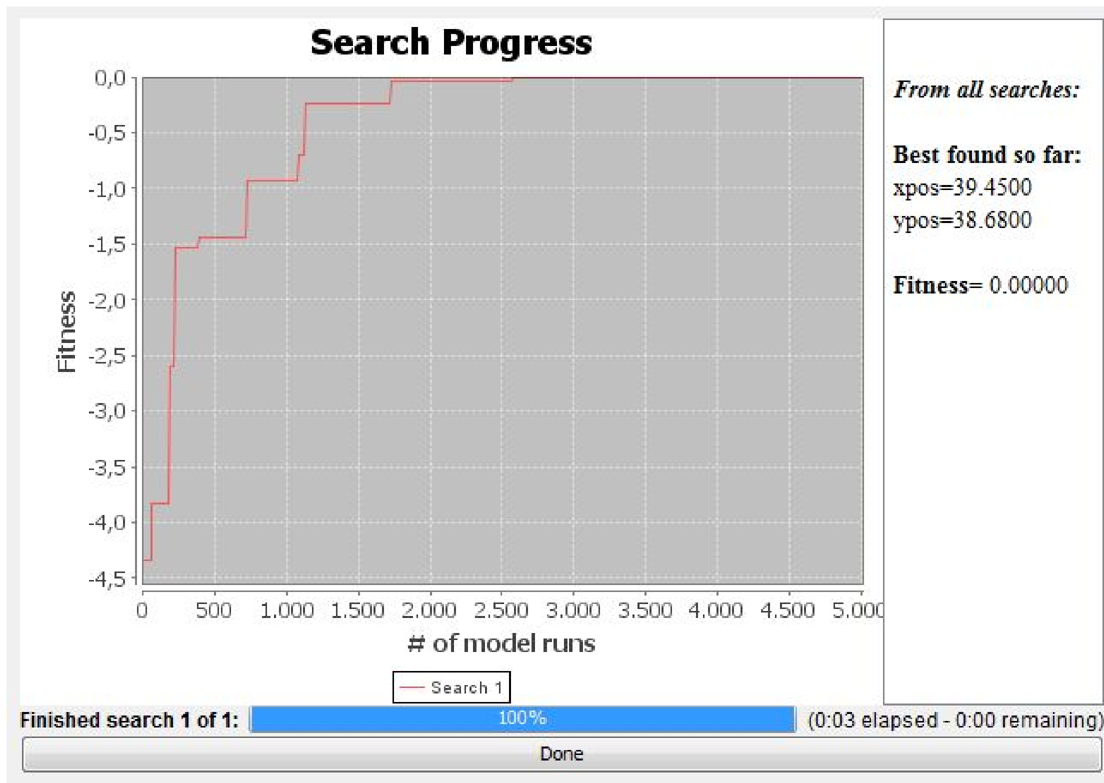

Figure 1.18: *localH3.bsearch* result

localH3.bsearch : its *BehaviorSearch Experiment Editor* looks as the one in Figure 1.10.

Performing a lot of analyses with *BehaviorSearch* we can note that the *StandardGA* works better than in *localH.bsearch* and it is faster (as happened with *localH2.bsearch*).

An example of *BehaviorSearch* result can be seen in Figure 1.18: in this graph we can also note the value of `pxcor` and `pycor` corresponding to the max height (Fitness = 0), i.e 39.45 and 38.68.

1.3.5 Comparison between search algorithms

In order to explain how *BehaviorSearch* analysis is influenced by the search space chosen, I will show some comparisons of results with the objective of establish which algorithm perform the best, and under which conditions.

In this section I will develop the following points:

- Analysis of how each search algorithm, (i.e. *StandardGA, RandomSearch, MutationHillClimbing, SimulatedAnnealing*), work on the three *NetLogo* programs, previously described (i.e. *localH, localH2, localH3*).
- Modification of the programs, *localH, localH2* increasing their search space; analysis of such new programs called *localH1.1, localH2.2*, in terms of the four search algorithms.
- *BehaviorSearch* analysis of a more complex *NetLogo* program, *Multivariate-LocalH*.

(All the analyses and results showed in this section are based on one-hundred trials, i.e considering one-hundred different seeds for each case).

Comparison between search algorithms: *localH, localH2, localH3*

(remember that `step limit=1`; `Evaluation Limit=5000`)

StandardGA : custom parameters:

`mutation-rate` = 0.01

`population-size` = 50

`crossover-rate` = 0.7

`population-model` = *generational*

`tournament-size` = 3

- *localH*: with custom parameters (Search Method Configuration explained in section 1.3.2 point 4) the *BehaviorSearch* analysis lasts 3-4 minutes; in general the best `Fitness` is reached within 1000 steps, then until step 5000 it remains substantially unchanged (or it gets little changes); the average fitness function lies in the interval [-0.3 -0.02], but it is never zero; when the starting point for the search (randomly chosen by *BehaviorSearch*) has a fitness around -9 or -7, the analysis rarely reaches a final-step fitness bigger than -0.2.

  If we change the parameters `mutation-rate`, `crossover-rate`, `tournament-size`, the analysis does not change significantly, leading in some case to better results, and in other cases to worse results (different cases are characterized by a different seed).

  A parameter that quite strongly influences the analysis is `population-size` (see Section 1.3.2 point 4): increasing this value, often leads to better

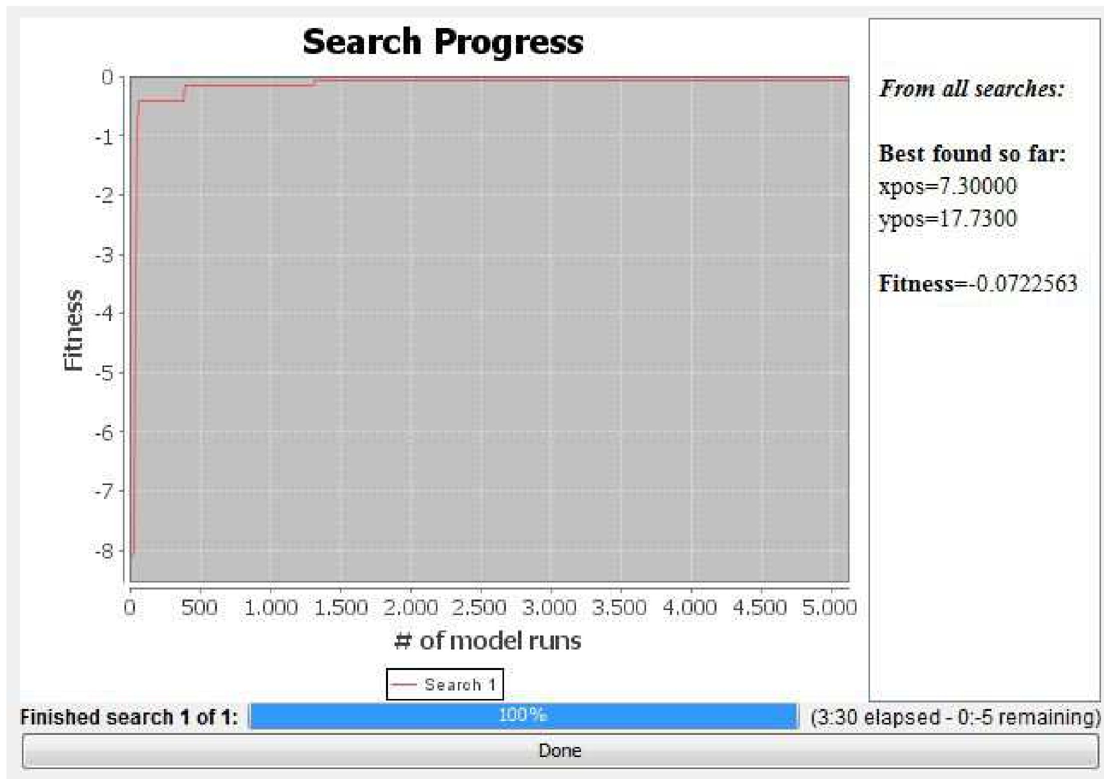

Figure 1.19: *localH.bsearch*: custom parameter of *standardGA* (population size 50), and random seed = -1.911.335.195

results, but it also increases the time due for the *BehaviorSearch* analysis (population-size must be an integer, and it lays between 1 and 1000).

For example if we set `population-size` = 200 (instead of 50), the fitness can now reach the value zero, and it is always less than 0.1.

The different effect of `population-size` on the same search attempt (seed = -1.911.335.195) can be seen in Figures 1.19, 1.20: in those graph we can note that with `population-size`= 50 the Fitness is -0.0722563; while with `population-size`= 200 the Fitness is 0.0000000.

Although the difference between the two Fitness mentioned above seems not to be relevant, in fact those values are related to very distant points in the three-dimensional search space considered:

`xpos`=7.30000 `ypos`=17.73000 for Figure 1.19;

`xpos`=37.3900 `ypos`=32.3700 for Figure 1.20.

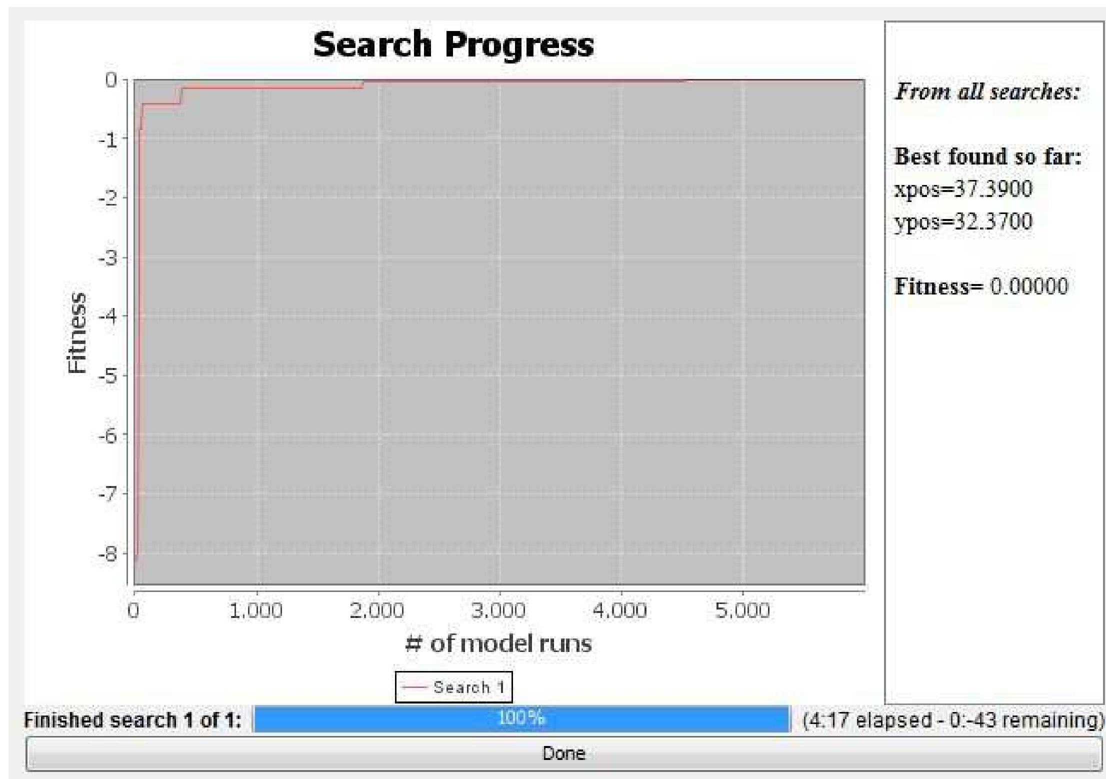

Figure 1.20: *localH.bsearch*: custom parameter of *standardGA* except population size=200; random seed = -1.911.335.195

Another parameter of the *StandardGA* that substantially changes the results of the *BehaviorSearch* analysis, is the population-model parameter (Section 1.3.2 point 4).

All examples and Figures mentioned until now, are referred to the custom population-model, i.e. *generational*.

Other models can be selected by changing the related cell; those are steady-state-replace-worst, and steady-state-replace-random.

The steady-state-replace-worst model always leads to better results providing a Fitness closer to zero; this model combined with a quite high value of the parameter population-size provides the best Fitness for *localH*, considering all the search algorithms of *BehaviorSearch*.

After the implementation of one hundred searches (with population-size=200), the average Fitness is about -0.013, and the analysis reached a mean of three zero Fitness out of ten attempts.

On average, also the steady-state-replace-random model works better than the generational one; but this model, sometimes shows a quite low fitness (around -0.2), due to the possibility, during the implementation of the *Genetic algorithm*, that one good solution is replaced. This effect is evident because the search space is quite small, and a wrong replacement can quite strongly influence the final Fitness value.

- *localH2*: for this optimization program the situation is different: the *BehaviorSearch* analysis is now faster than in *localH* case, and it lasts only 2-3 seconds.

  With custom *StandardGA* parameters: the major part of the attempts reach a zero Fitness value, while on average one search out of ten gets a Fitness value around -0.2.

  The effect of changing the population-size parameter is now less relevant: if we want to obtain a better Fitness by increasing the population-size we have also to increase the Evaluation Limit value (note that increasing these two parameters, the time due for the analysis strongly increases: for example setting population-size=1000 and Evaluation Limit=50000, we always get a zero Fitness, but the search lasts 15-16 seconds).

  Also the effect of choosing a different population-model is less remarkable: the frequency of Fitnesses different from zero is a little bit higher when using steady-state-replace-random model(on average

two-three out of ten trials); while there are no significant differences (in results) when using the other two models.

Remember that *localH2* forms a monotone three-dimensional surface as shown in Figure 1.15; in that case the variable `heigh` gradually changes its value: it means that similar values of final Fitness correspond to quite close points in the space.

The other parameters that characterize the *StandardGA* react as in *localH*.

- *localH3*: for this program, the *BehaviorSearch* results are very similar to the previous case, i.e. *localH2*:

  the analysis with custom *StandardGA* parameters requires only few seconds (3-4); the results show an often equal to zero Fitness (only one case over ten attempts gets a final Fitness within the interval [-0.25 , -0.12]).

  The effect of `population-size` is influenced by the `Evaluation Limit`(as happened with *localH2*): a small value of the first compared to the second, ensures the evolution of all members forming the initial population, for several steps; if the algorithm is repeated for several steps, the probability of convergence to the search objective increases (note that for initial population I refer to the part of the total population that is randomly chosen to evolve: in fact in *BehaviorSearch* it is the `population-size`).

  However starting from a big initial population provides a greater level of heterogeneity, that is fundamental for the well functioning of a *Genetic Algorithm*.

  It means that, keeping fixed the `Evaluation Limit`, extreme positions of `population-size` do not work well:

  for example if we set `population-size`=10 (with `Evaluation Limit`=5000), we have a small fraction 10/5000, and so a good convergence probability; but we have a small initial population (10 realisations) that does not enhance heterogeneity.

  The opposite situation happens if we set an initial population of 1000 (maximum possible value for `population-size` parameter); we now have the maximum value heterogeneity (note that I can set

  `population-size`=1000 because I have a *search space* of $400 \times 400 = 160000$ possible solution, and $1000 < 160000$);

  but the fraction 1000/5000 is quite small, and this value ensures few evolution steps from one generation to the other, and so a lower convergence probability.

As in *localH2* the effect of little changes in the other parameters characterizing the *StandardGA* is negligible.

RandomSearch : (no imput parameters)

- *localH*: with `Evaluation Limit`= 5000 model runs, the *BehaviorSearch* analysis lasts 3-4 minutes.

  The Fitness is always included in the interval [-0.04 , 0], and on average reaches the value zero three times out of ten attempts.

  The results obtained in terms of Fitness, are very similar to those showed using *StandardGA* with `population-size`= 200 and `population-model`= *steady-state-replace-worst*.

- *localH2*: the *BehaviorSearch* analysis lasts only two seconds; the algorithm performs badly compared to the *StandardGA*: the Fitness rarely gets a zero value,(on average one time out of fifteen attempts) and it is generally included in the interval [-0.6 , -0.12].

- *localH3*: the search produces Fitness result with an high variance (with respect to the other cases considered); the solutions lie in the interval [-0.7 , 0] , and on average they reach a zero Fitness one time out of three attempts.

  The time spent for the analysis is about four seconds.

MutationHillClimber : (custom parameters:
`mutation rate= 0.005`
`restart-after-stall-count=0`)

- *localH*: with custom parameters, the search is less efficient than with *RandomSearch* algorithm; this is probaly due to the random structure of the search space. The analysis lasts 3-4 minutes as in previous cases; the Fitness lies in the interval [-0.15 , 0], but rarely reaches the value zero (on average one time out of ten attempts).

  If we increase the `mutation-rate` the algorithm approaches the behavior of the *RandomSearch*, leading to slightly better results.

  The same effect can be obtained if we increase the value of `restart-after`

-stall-count; in fact this parameter strongly influence the degree of randomness of the algorithm.
(Note that the value of these two parameter influences the search result in a non proportional way: I have been empirically chosen as benchmark a framework with mutation-rate= 0.2 and restart-after-stall-count = 3; if we increase these values, increasing the degree of randomness of the algorithm, this not always leads to better results;).

Some examples of Fitness given by the *MutationHillClimber* algorithm, varying its parameters, can be seen in Figures 1.21, 1.22, 1.23 (they are characterized by the same random seed).

Sometimes a combination of mutation-rate and restart-after-stall-count can provide an even better Fitness, as shown in Figure 1.24 , but there is not a general rule; however implementing several *BehaviorSearch* trials (analysing several seeds) can be obtained an empirical rule, that properly works only on the program that we are testing.

- *localH2*: the characteristics of Hill Climber algorithm fit particularly well on search space considered: in fact it is always possible to find a greater heigh implementing slight modifications (once the right direction has been chosen, the convergence is ensured).
  More in detail, the analysis lasts three seconds, and gets always a zero Fitness, except in sporadic cases (one case out of twenty attempts, on average), in which Fitness = -0.2 .
  As explained in *localH*, the effect of increasing the two parameters characterizing the *MutationHillClimber*, rises the degree of randomness of the algorithm.
  Moreover, since the *RandomSearch* algorithm works particularly bad on *localH2*, the effect produced by mutation-rate and restart-after-stall-count works in the opposite way as that explained for *localH*: for higher values of those parameters we get a lower Fitness (close to the one computed in the *RandomSearch* case of *localH2*).

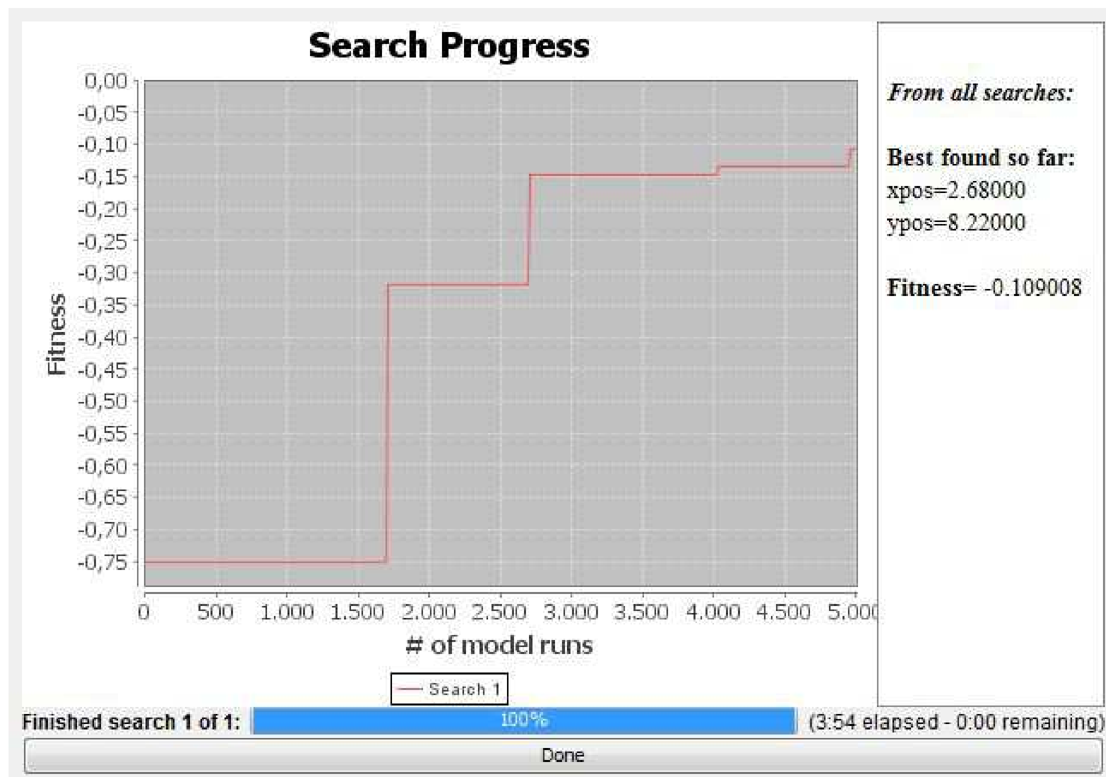

Figure 1.21: *MutationHillClimber* for*localH*: result with custom parameters and seed= 1.135.198.364

.

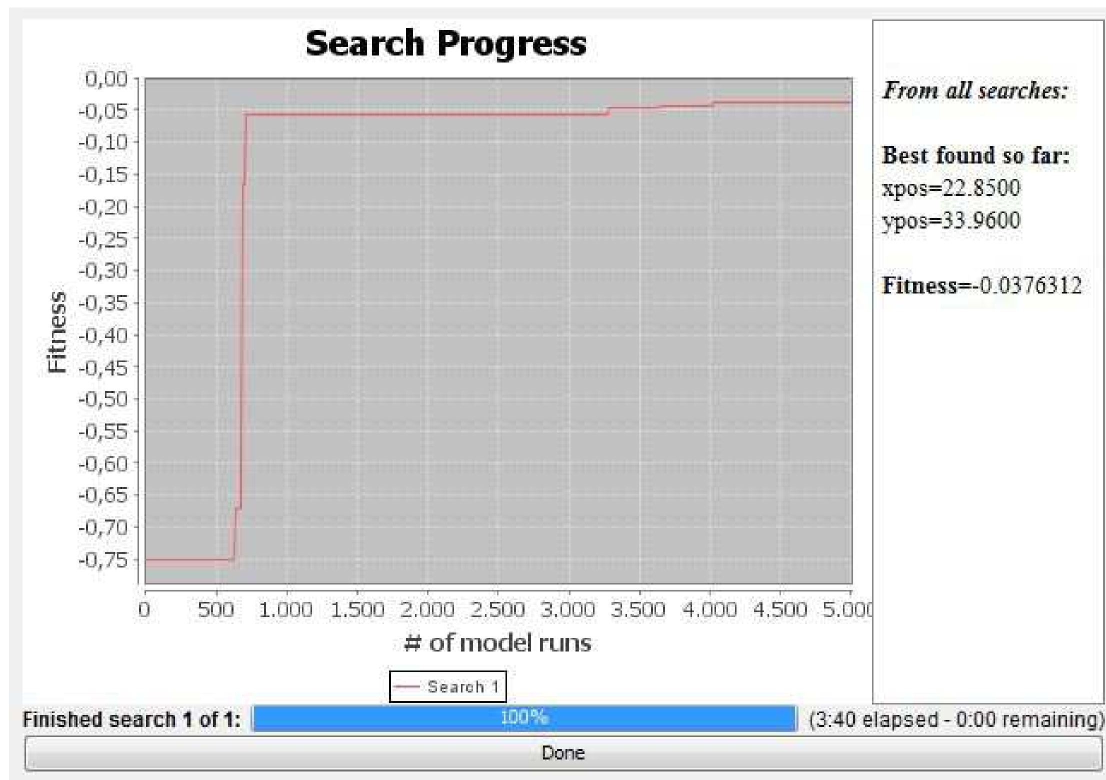

Figure 1.22: *MutationHillClimber* for*localH*: result with custom parameters except mutation-rate= 0.2 and seed= 1.135.198.364
.

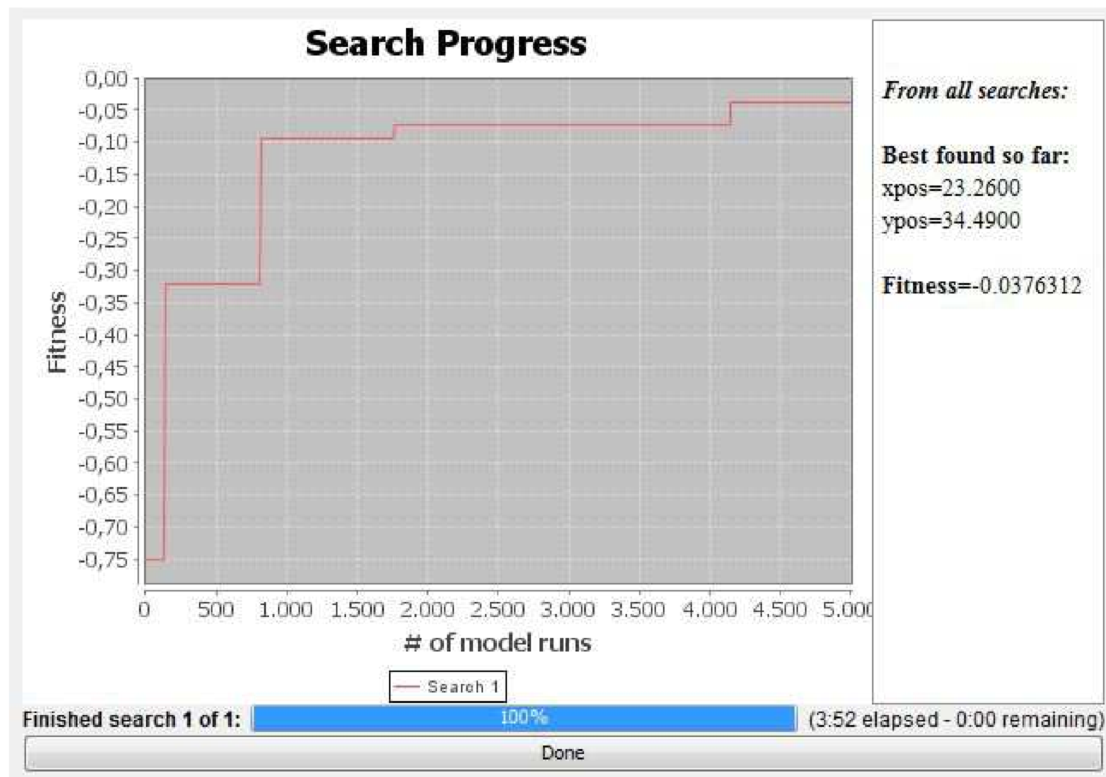

Figure 1.23: *MutationHillClimber* for*localH*: result with custom parameters except restart-after-stall-count= 3 and seed= 1.135.198.364

.

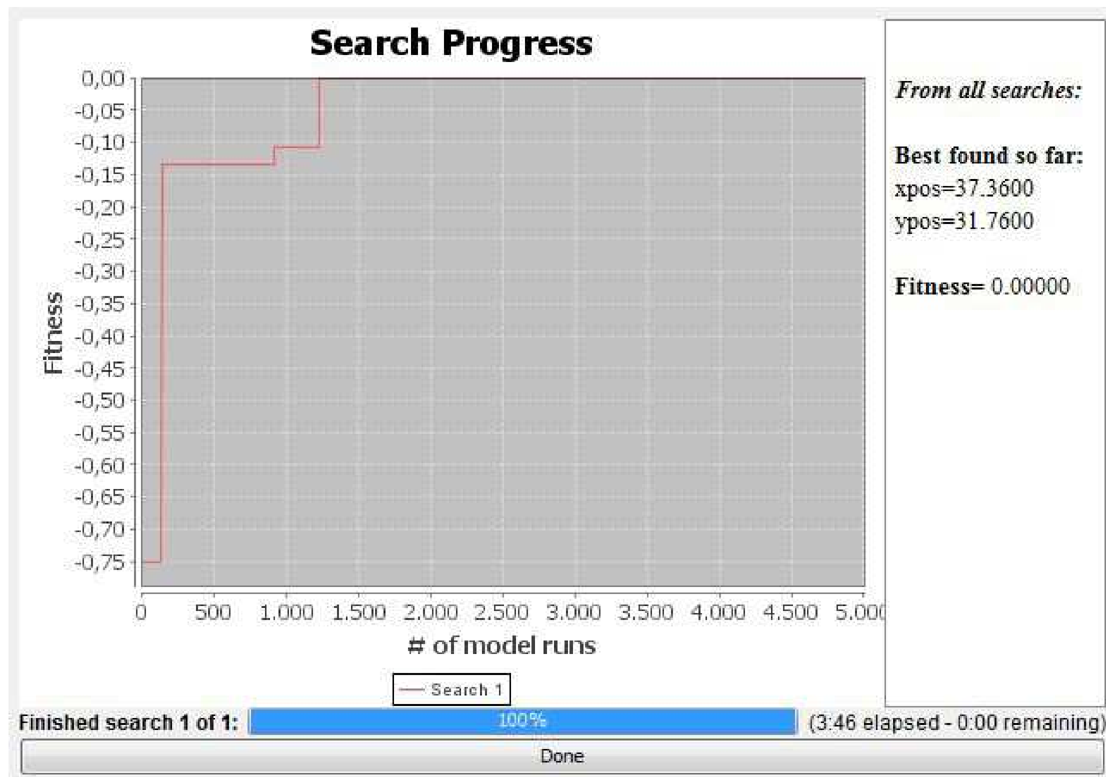

Figure 1.24: *MutationHillClimber* for*localH*: result with mutation-rate= 0.2 , restart-after-stall-count= 3 and seed= 1.135.198.364

.

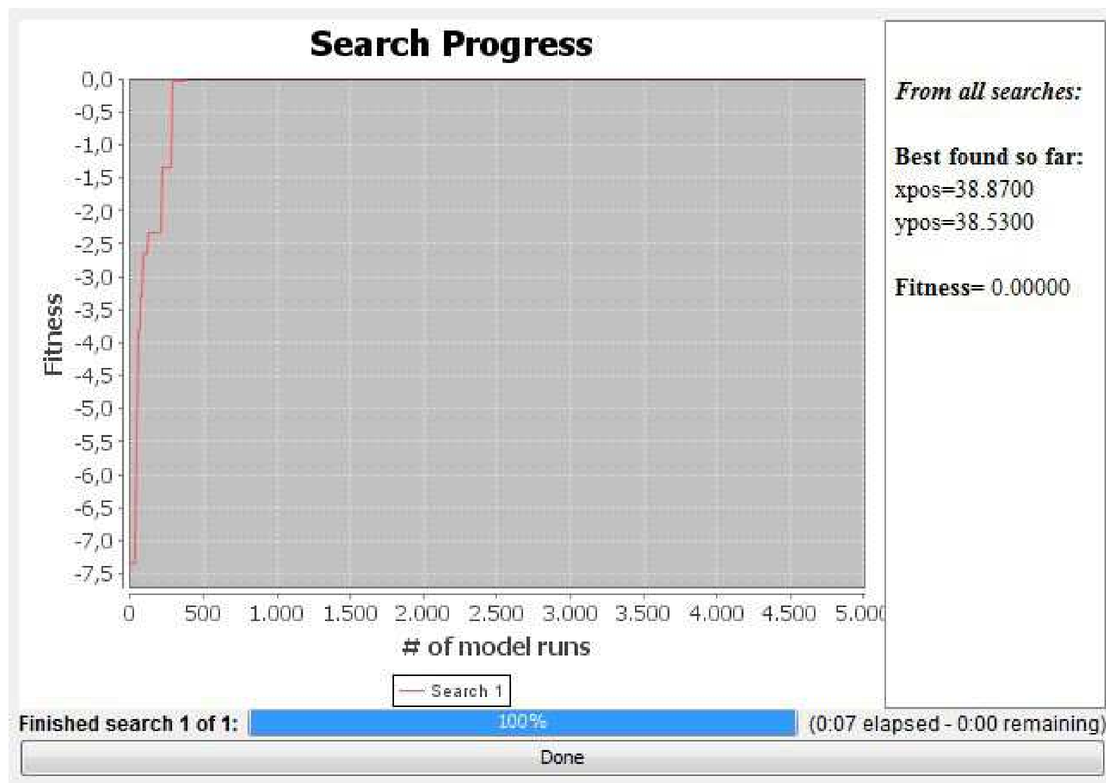

Figure 1.25: *MutationHillClimber* for*localH3*: very fast convergence to zero Fitness value

.

- *localH3*: the *BehaviorSearch* analysis lasts five-six seconds; with custom parameters the Fitness value is always zero, except in sporadic trials (on average one out of fifty attempts, with Fitness equal to about -0.2); moreover the convergence to zero is very fast, and it is generally obtained within 2000 steps(model runs): an example of this can be seen in Figure 1.25.

  For what concerns the other parameters characterizing the algorithm, they behaves as for *localH2*.

SimulatedAnnealing (custom parameters:

`mutation-rate`= 0.05

`temperature-change-factor`= 0.99

`initial-temperature`= 1.0

`restart-after-stall-count`= 0)

- *localH*: with custom parameters the algorithm gives solutions included in the interval [-0.1 -0.01], and the Fitness never gets the value zero. In this case the search analysis is less effective than the one implemented with the *RandomSearch*, and results have a smaller variance than in the case with the *MutationHillClimber* algorithm.

  If we increase the parameters `mutation-rate` and

  `restart-after-stall-count` the search becomes more random as happened with *MutationHillClimber* algorithm, leading on average to slightly better results.

  The other parameters, i.e. `temperature-change-factor` and

  `initial-temperature` are linked to the search space defined, and there is not a general rule for their effectiveness. I have found that for this specific problem, `temperature-change-factor` leads to opposite results, so I cannot define a range of possible values; for what concerns `initial-temperature`, I have found that a reasonable value for *localH* is around 8.0: in this framework, keeping all other parameters with custom settings, the algorithm can reach a Fitness equal to zero.

  As happened with the *MutationHillClimber* algorithm, a combination of different parameters value, can provide a faster convergence to zero, but there is not a general rule.

- *localH2*: the algorithm, with custom parameters, is very efficient for this program, providing the same results of the *MutationHillClimber* case; the Fitness is often zero, while on average only one trial out of twenty gives a convergence to -0.15, or -0.25 .

  The effect of changing parameters provides worse results if we increase the value of `mutation-rate` and `restart-after-stall-count`, since they rise the level of randomness of the program, and as explained before, the *RandomSearch* algorithm performs poorly on *localH2*.

  The missing parameters, i.e. `temperature-change-factor` and

  `initial-temperature` do not change remarkably the search; in some cases, a higher or a lower value of these parameters can lead only to a faster convergence to a zero Fitness.

- *localH3*: to implement the 5000 model runs, *BehaviorSearch* requires eight seconds.

If we consider the model with custom parameters, it gives results very similar to those obtained applying *MutationHillClimber* algorithm to *localH3*.

While if we change the parameters characterizing the algorithm, we sort the same effects as in *localH2* applying *SimulatedAnnealing*; in fact the two programs *localH2* and *localH3* have in common the fact that they create a smoothed three-dimensional surface, that is not completely random as in *localH* case.

Comparison between search algorithms: *localH1.1, localH2.2*

Until now we discussed about the effectiveness of the different *BehaviorSearch* algorithms, only as a function of their input parameters (`Search Method Configuration` table); but we haven't yet analysed the effect on *BehaviorSerch* of increasing the search space.

In fact all the programs *localH* *localH2*, *localH3* own a search space equal to range of `x` times range of `y` , i.e.

$$\frac{41}{0.01} \times \frac{41}{0.01} = 4100^2 = 16810000$$

But in practice, considering how the programs are built through *NetLogo*, we have only

$$41 \times 41 = 1681$$

changes of the variable `heigh`.

The objective of the analyses of this section, is to study the response of *BehaviorSearch* with respect to changes in the magnitude of the search space: for this purpose, I have modified the programs *localH* and *localH2*; the new programs updated are called *localH1.1* and *localH2.2*.

I have not built a new version of *localH3*, because it gives results very similar to *localH2*; my impression is that the *BehaviorSearch* analysis performs likewise for monotonic functions, or at least monotonic at large increments.

For the *BehaviorSearch* analyses of *localH1.1* and *localH2.2*, I will consider the same conditions previously set in the preceding analyses: in terms of the parameters characterising each search algorithm, and in terms of *BehaviorSearch Experiment Editor* (remember `Step Limit` = 1 , `Evaluation Limit` = 5000 model runs).

localH1.1 : the *NetLogo* program maintains the same structure of its predecessor (see section 1.3.4 case 1); the only part that changes is the formation of the three dimensional space, through the Python program.

As in *localH*,each value of the variable `heigh`, that characterise the function f(x,y) is saved into a text file; for *localH1.1* it is called `base_mod.txt`.

The Python program, called `base_mod.py`, creates that file in the following way:

```
import random
f=open("base_mod.txt","w")
for i in range(81):
    for j in range(81):
        print >>f, i, j, random.random()*10
f.close()
```

With respect to its predecessor, i.e. `base.py` (explained in section 3.4.1), this one builds $81 \times 81 = 6561$ points for the variable `heigh`, instead of $41 \times 41 = 1681$.

More in deep, the only difference is in each `for` cycle, that ranges from 1 to 81, doubling the range of the `x` and `y` coordinates.

Instead, the dimension of the search space is:

$$\frac{81}{0.01} \times \frac{411}{0.01} = 8100^2 = 65610000 \tag{1.1}$$

since the step for the variables `xpos` and `ypos` is 0.01 .

The surface obtained with *localH1.1.nlogo*,(after have pressed the `setup` button) is shown in Figure 1.26.

localH1.1.bsearch:

- *StandardGA*: *BehaviorSearch* takes slightly more than one minute (about 70 seconds on average) to start the search, i.e. to run the analysis (loading time); then the proper analysis lasts about 11.30 - 13.30 minutes.

  We can note that comparing *localH1.1* with *localH*, we have a search space of 65610000 instead of 16810000: the first is about 3.9 times greater than the second;

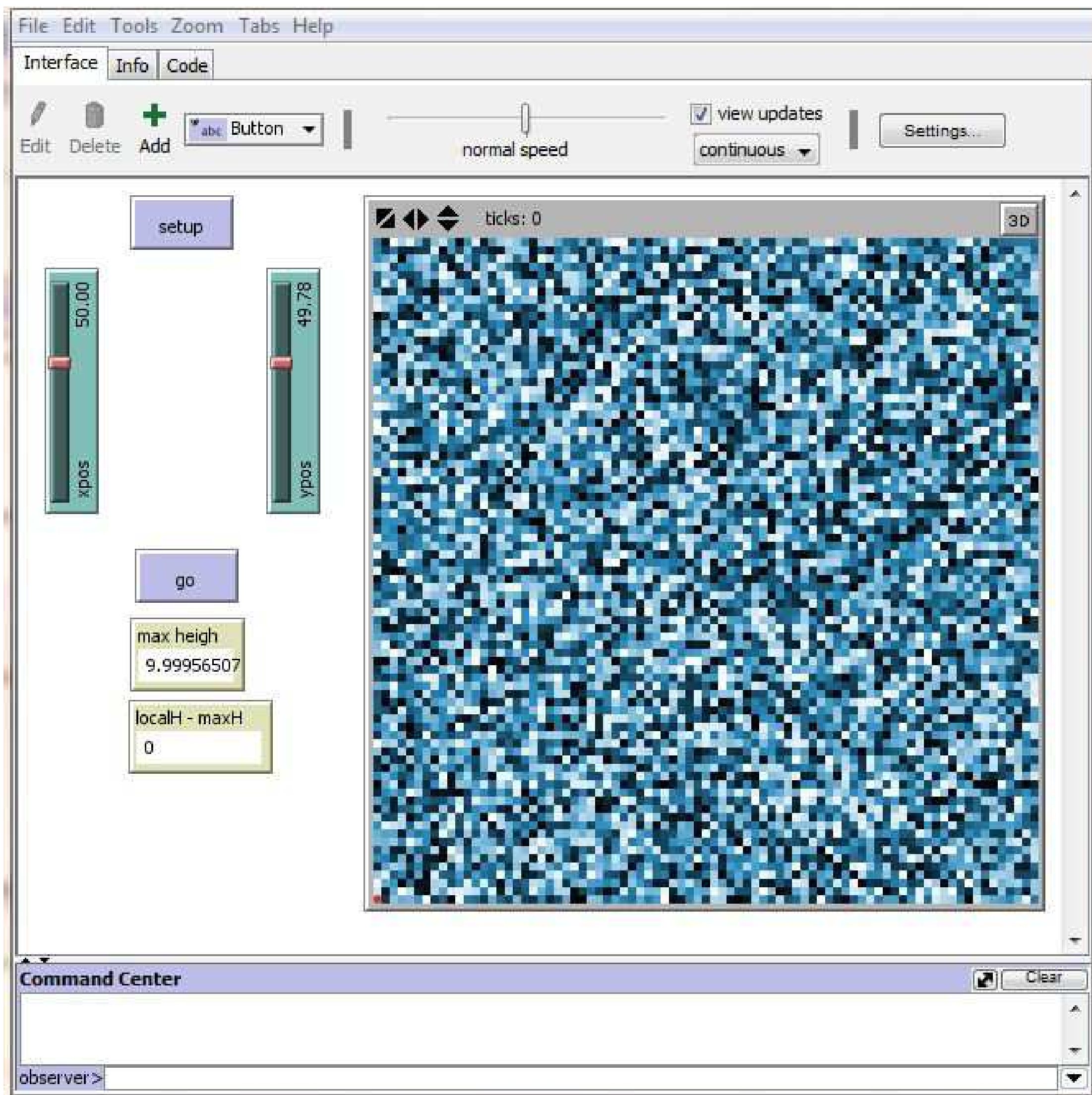

Figure 1.26: *localH1.1* interface

.

$$\frac{65610000}{16810000} = 3.903033908$$

(note that a loading time necessary to start the analysis, greater than few seconds, is present only in the *StandardGA* case with high values of `population-size`).

Also the time due to complete the search and get the one-hundred percent of completion, is about four times greater than in *localH* case (see section 1.3.5 case 1).

The genetic algorithm, with custom parameters provides a `Fitness` on average slightly more than - 0.05 (- 0.048 on 100 trials) ; moreover it is never zero, and rarely lower than than -0.1 .

The fitness obtained is greater than the one observed for *localH*: it seems that with the *StandardGA*, a larger search space size leads to better results in terms of convergence to the optimal point.

If now we set `population-size` = 200, the *BehaviorSearch* loading time, to start the analysis lasts more than four minutes; my impression is that, this waiting time is proportional to the population size considered, since that value influences the definition of the portion of initial population, randomly selected to evolve by the algorithm: a greater `population-size` requires more computations.

However,although this difference for the starting time, the duration of the proper analysis does not change significantly (the *BehaviorSearch* time spent for the analysis is proportional to the `Evaluation Limit`, and depends on the structure of the search space).

As for *localH*, the `Fitness` is closer to zero, and can reach in some trials exactly the value zero (on average it is equal to about - 0.02).

The effect of changing the `population-model`, are the opposite with respect to those observe so far (Section 1.3.5 case 1): if we set the `steady-state-replace-worst` framework, it gives always worse results in terms of Fitness (there are no strong differences with the `generational` case, but on average each trial differ of about -0.005). This effect is probably due to the elimination of the worst realisation during the *BehaviorSearch* analysis, causing a reduction in the degree of heterogeneity of the search algorithm.

Moreover we can note that the *StandardGA* is more sensitive to heterogeneity when we are considering a greater search space size.

The `steady-state-replace-random` model does not provide significant changes with respect to the custom case, reaching on average the same Fitness.

The other parameters characterising the *StandardGA*, i.e. `mutation-rate`, `crossover-rate` and `tournament-size`, are not relevant in the the search analysis, since a change of those variables produces sometimes slightly better or slightly worse results, without a identifiable pattern (as happened for *localH* in Section 1.3.5 case 1).

If we want to obtain the best results in terms of Fitness, getting always zero, we must set: `population-size` = 1000, (*Search Method Configuration*) to enhance heterogeneity , and `Evaluation Limit` equal to at least 20000 (*Objective/Fitness Function*), to ensure enough evolution steps. This structure is quite burdensome, with respect to the custom framework: the initial loading time is about twenty times greater, while the time for the search analysis is about $\frac{20000}{5000} = 4$ times greater.

- *RandomSearch*: (no input parameter in the *Search Method Configuration*) after 5000 model runs it attains an average fitness of about -0.02 (the fitness is always greater than -0.1, but it is rarely 0, one trial out of thirty attempts).

  Surprisingly the *RandomSearch* suits well for this program: increasing the search space, the random space generated by *localH* is more efficiently analysed by the algorithm.

  This can be explained as a consequence of the fact that the space points randomly generated through the *Python* program `base.py` are not enough differentiated: it means that the command `random.random()*10` does not ensures the formation of a maximum point significantly higher than the other space points; in facts there are a lot of points with `heigh` close to 9.9 .

  For that reason the Function characterising the *localH1.1* space, turn out to be more similar to the one characterising *localH2*, and so more smoothed.

  The time necessary to complete the analysis is 13 - 14 minutes.

- *MutationHillClimber*: the effect of changing the parameters of the algorithm is the same explained in section 1.3.5 case 1 .

  With custom parameters the Fitness lies in the interval [-0.2 , -0.004] with mean of about -0.06; it never reached the value zero.

As for *localH*, the algorithm works worse than the *RandomSearch*, although the surface considered is more smoothed that the one of its predecessor .

– *SimulatedAnnealing*: this algorithm gets the same results of the *MutationHillClimber* one; the average fitness is about -0.06, and the analysis lasts 13 - 14 minutes.

  The effect of changing the parameters of the algorithm, for *localH1.1* is equivalent to that observed in section 1.3.5 .

localH2.2 : in this case, the changes with respect to *localH2* (see section 1.3.4 case 2) comes from a different core structure of the *NetLogo* program; the related part of code is:

```
ask patches     [set height pxcor * 0.02 + pycor * 0.02
     set pcolor height + 80
     ]
```

These updates are implemented to obtain the graph in Figure 1.27.

We can note that now the variables `xpos` and `ypos` range from 0 to 250; it means that the *NetLogo* program *localH2.2.nlogo* creates $251 \times 251 = 63001$ points for the third dimension `heigh`, while the size of the search space is:

$$\frac{251}{0.01} \times \frac{251}{0.01} = 25100^2 = 630010000 \tag{1.2}$$

since the step is 0.01 .

LocalH2.2.bsearch The dimension of the search space is big with respect to the range of the variable `heigh` (the range is represented by the interval [0 10]): this fact together with the structure of the search space ensures a very smoothed surface: as we have seen in previous cases, this characteristic should favour *StandardGA*, *MutationHillClimber*, *SimulatedAnnealing* algorithms; while it should work badly with *RandomSearch* algorithm.

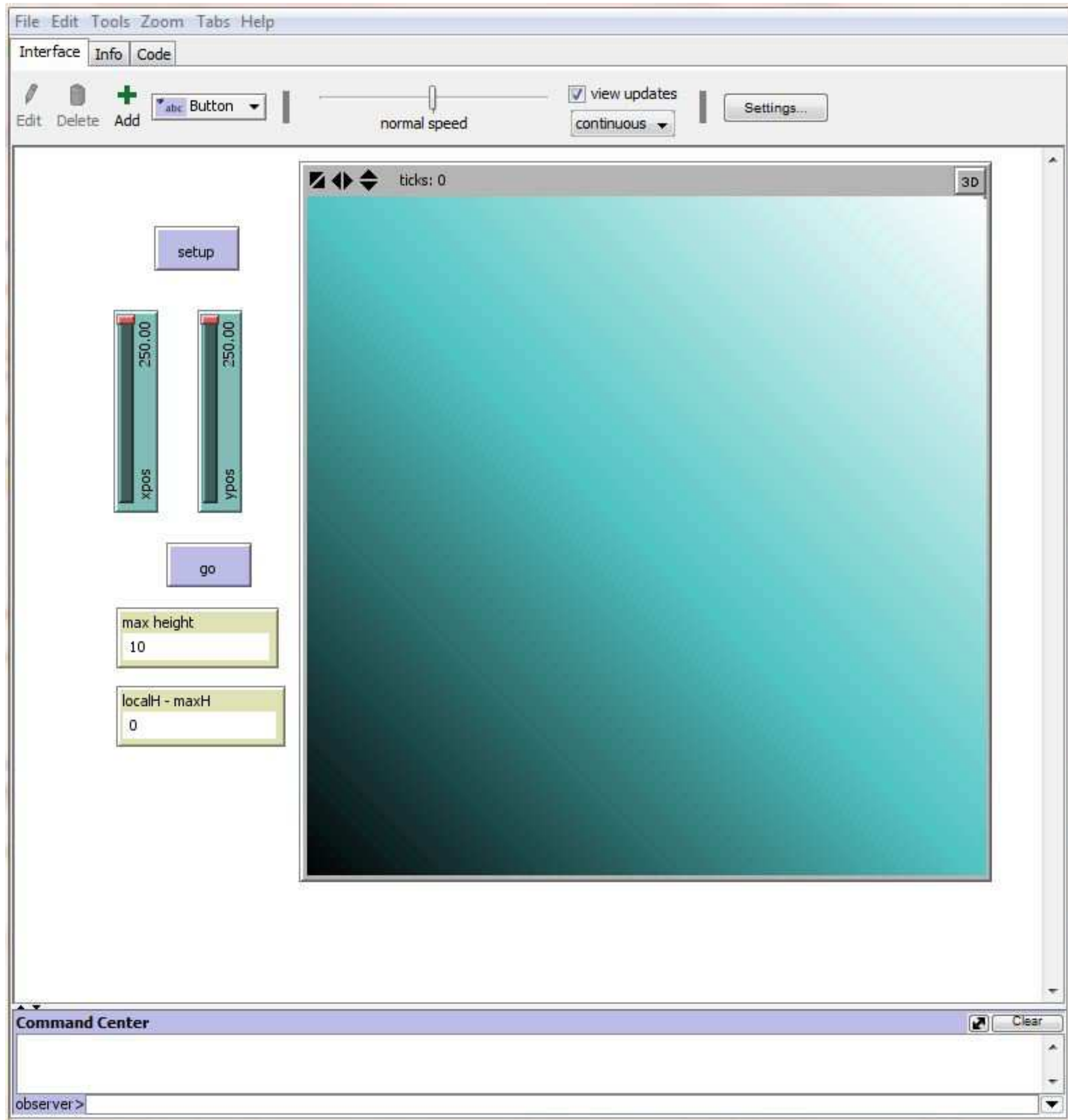

Figure 1.27: *localH2.2* interface

.

- *StandardGA*: the *BehaviorSearch* analysis lasts about two-three minutes.

  With custom parameters the fitness gets always the value zero within the step 500 (out of 5000 model runs);

  the *StandardGA* works well on *localH2.2* with any value of `population-size` . This parameter affects only the time due to get a zero fitness: a smaller value ensures a faster convergence, and vice-versa.

  This effect is probably due to the fact that if the surface considered is very smoothed, the *genetic algorithm* requires less heterogeneity to converge, and the number of evolutionary steps becomes more relevant than the magnitude of `population-size` (given by the ratio

  `population-size` / `Evaluation Limit`).

  Empirically the best population size lies between 15 and 20 individuals : the zero fitness value is reached, on average within the step 300.

  The model `steady-state-replace-worst` produces on average, a slightly faster convergence , with respect to `generational` and `steady-state-replace-random` models; but this difference seems not significant.

  Changing the parameters `crossover-rate` and `tournament-size` does not give rise to evident changes in the search, while increasing the value of `mutation-rate` slows the convergence to zero of the algorithm; this happens because a greater value of `mutation-rate` alters the *genetic algorithm* in a random search.

- *RandomSearch* (no input parameters): the analysis lasts about two minutes.

  The algorithm rarely reaches a zero fitness value (one case out of fifty trials): after 5000 model runs the fitness fills always in the interval [-0.2 0].

  It is the worst search algorithm for *localH2.2*.

- *MutationHillClimber*: *BehaviorSearch* lasts about four minutes to implement 5000 model runs with this algorithm.

  The fitness gets always the value zero within the first 300 steps.

  The algorithm, with custom input parameters, performs well on *localH2.2*, as happened with the *StandardGA*; the only difference consists in a little longer time needed to complete the search.

  As explained in previous section, the input parameters characterising the algorithm influence the randomness degree of the search: if we increase the value of `mutation-rate` and `restart-after-`

`stall-count`, the algorithm behavior approaches the random search, leading to worse results.

However the effect of the first is stronger than the effect of the latter (for example, if we set `mutation-rate` = 0.8 , the fitness lies in the interval [-0.5 , -0.05]; while if we set `restart-after-stall-count` = 5, the fitness limits are [-0.08 , -0.02]).

- *SimulatedAnnealing*: the search analysis lasts about three-four minutes to complete the 5000 model runs.

  The algorithm, with custom input parameters, gets always a zero fitness within the step 1000: for *localH2.2* it works as the *MutationHillClimber* algorithm , with the only difference that it has a slower convergence.

  The parameters `mutation-rate` and `restart-after-stall-count`, sort the same results explained for the *MutationHillClimber* algorithm: they increase the degree of randomness of the search, providing a worse average fitness.

  The effect of changing the parameter `initial-temperature` seems not too significant, although empirically with higher values of the parameter, the convergence is a little slower.

  For the parameter `temperature-change-factor` the convergence seems to be faster for values included in the interval [0.3 , 0.7].

Comparison between search algorithms: MultivariateLocalH

In order to continue the study on the effectiveness of *BehaviorSearch* algorithms, obtained by enlarging the search space, I have built a new program, i.e. *MultivariateLocalH.*

- *MultivariateLocalH.nlogo*: this program is focused on the creation of a six-dimensional space, with variables `x1, x2, x3, x4, x5, x6`.

  Obviously such kind of search space cannot be represented graphically; for that reason my idea is to create a *n-sphere*, with n = 6, because it is quite simple to understand and to find critical points.

  In mathematics, an *n-sphere* is a generalization of the surface of an ordinary sphere to a n-dimensional space. For any natural number n, an n-sphere of radius r is defined as the set of points in (n + 1)-dimensional Euclidean space which are at distance r from a central point, where the radius r may be

any positive real number. Spheres of dimension n > 2 are sometimes called hyperspheres.

The set of points in (n + 1)-space: (x1,x2,...,xn+1) that define an n-sphere, (Sn) is represented by the equation:

$$r^2 = \sum_{i=1}^{n+1} (x_i - c_i)^2$$

where `c` is a center point, and `r` is the radius.

The *NetLogo* code that generates our 6-sphere is the following:

```
 globals [maxH localDiff Y]
 turtles-own [point]
to setup
   __clear-all-and-reset-ticks
 crt 1 [set shape "person"]
end
to go
  ask turtles[
       set Y (x1 ^ 2 + x2 ^ 2 + x3 ^ 2 + x4 ^ 2
               + x5 ^ 2 + x6 ^ 2 + 10 )
       set point Y
       show point
  ]
  set localDiff Y - 60010
end
```

The range of the six variables mentioned is: [0 , 100] with increment 0.01 for `x1, x2, x3, x4, x5` ; [-100 , 100] with increment 0.01 for `x6` .

The range of each variable is set up with the respective slider, as shown in Figure 1.28 .

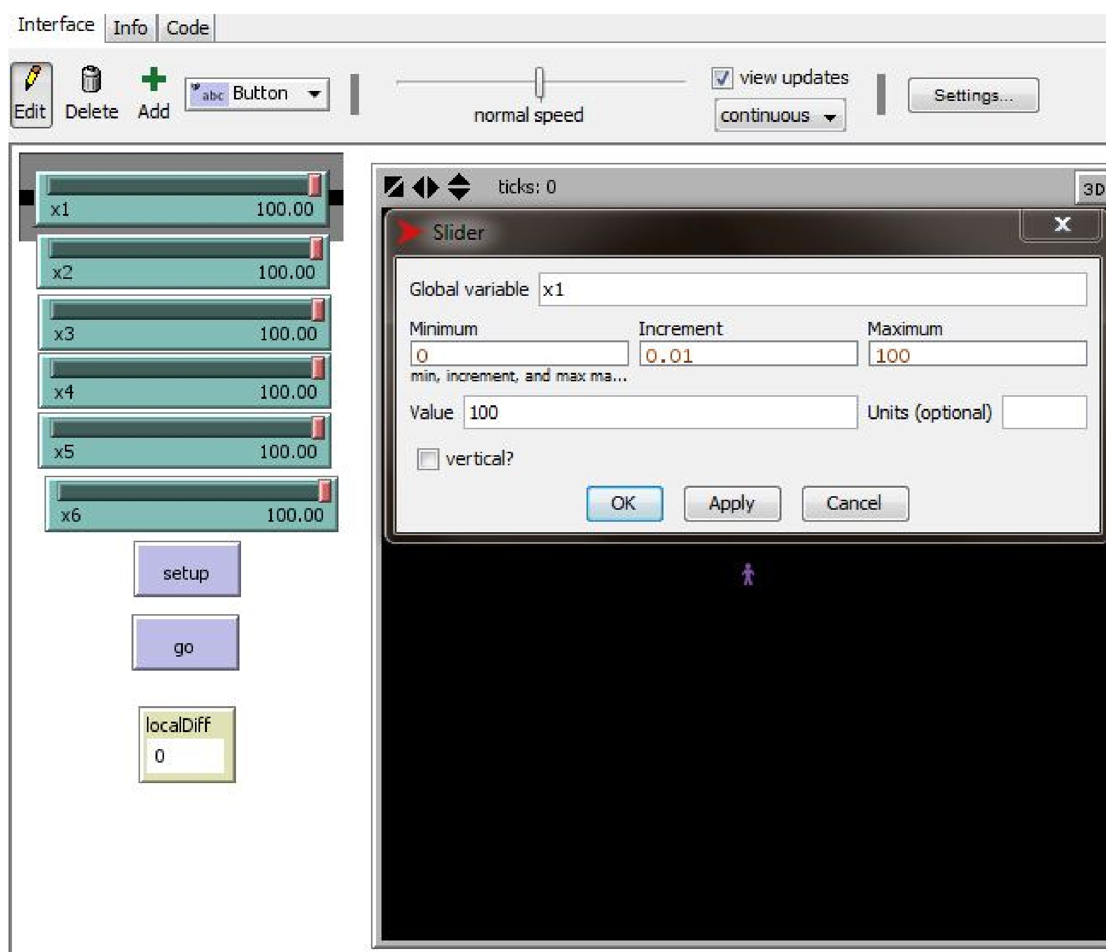

Figure 1.28: *MultivariateLocalH* interface after the implementation of the setup procedure.

The construction of the hypersphere is obtained through the equation

```
set Y (x1 ^ 2 + x2 ^ 2 + x3 ^ 2 + x4 ^ 2 + x5 ^ 2 + x6 ^ 2 + 10 )
```

that puts inside the variable `Y` every possible space point (note that all calculations are computed through the `ask` command, that requires the creation of at least one agent; this is done in the `setup` procedure with the command `crt 1 [set shape "person"]`).

Although we are considering a multidimensional space, it is easy to calculate its critical points; in facts the maximum can be reached in two possible points in the search space :

(`x1 = x2 = x3 = x4 = x5 = x6 = 100`) and

(`x1 = x2 = x3 = x4 = x5 = 100 , x6 = -100`)

since a *hypersphere* maintains the same characteristics of a sphere.

As in the previous programs analysed, (*localH*, etc...) the variable `localDiff` measures the distance between the current point, obtained with the related value of the sliders, and the maximum value of the function.

We can see this fact from the command:

```
set localDiff Y - 60010
```

where 60010 can be calculated substituting the optimal points into the *hypersphere* equation:

$$100^2 + 100^2 + 100^2 + 100^2 + 100^2 + (\pm 100)^2 + 10 = 60010$$

We can see the value of `localDiff` in the homonym monitor in the *NetLogo* interface, by pressing the `go` button (see Figure 1.28).

The dimension of the search space is the biggest analysed so far: it is equal to:

$$\frac{100}{0.01} + \frac{100}{0.01} + \frac{100}{0.01} + \frac{100}{0.01} + \frac{100}{0.01} + \frac{200}{0.01} = 2 \times 10^{24}$$

The multidimensional space so created is smoothed and monotone.

- *MultivariateLocalH.bsearch* : as happened with previous programs, the *BehaviorSearch* analyses are implemented using the variable `localDiff` as fitness: with this framework the best obtainable fitness is equal to zero.

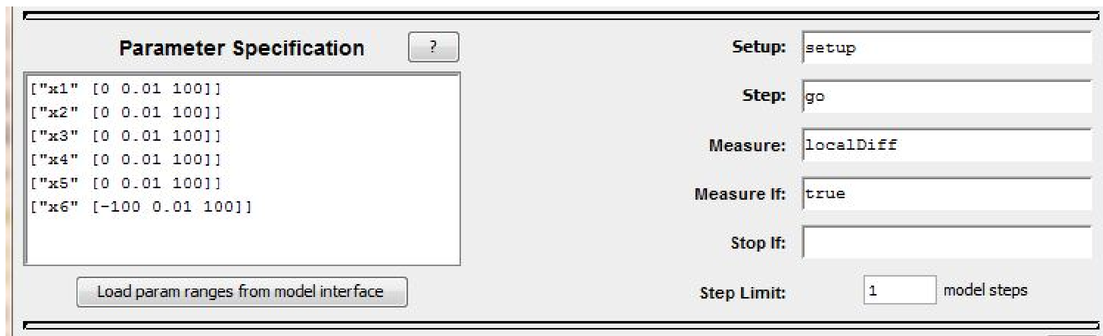

Figure 1.29: *MultivariateLocalH.bsearch*: parameter specification and identification of the fitness function.

The number of total model runs, i.e. the `Evaluation Limit`, is set equal to 5000, as in previous analyses.

- *StandardGA*: the time due to complete the *BehaviorSearch* analysis is only 3-4 seconds.

  With custom parameters the average fitness is about -5000 (it never reaches the value zero): although the search is fast, it seems that 5000 model runs are not enough to obtain a fitness close to zero.

  The parameter `mutation-rate` works in the same way for all algorithms of *BehaviorSearch*; in this case, if we increase the degree of randomness of the algorithm, we get worse results in terms of fitness.

  As in previous examples, the effect of the parameters `crossover-rate` and `tournament-size` on the search are not well defined; a change in those values can lead to opposite results depending on the trial considered (seed).

  For what concern the `population-model` :

  the `steady-state-replace-worst` model is more efficient than the `generational` one, providing an average fitness of about -3000;

  the `steady-state-replace-random` model is less efficient than the `generational` one, getting a final average fitness of about -6000.

  The input parameter that the most influences the search is `population-size` : if we increase this value, we get a lower final fitness and vice-versa; this happens because we are considering a smoothed multidimensional space, and it is more preferable to increase the number of evolutionary steps, instead of the degree of heterogeneity of the population selected to evolve (keeping fixed the evaluation limit).

For example if we set the *Search Method Configuration* with custom parameters, except `population-size`= 10, we obtain an average fitness of about -2000 (however note that the degree of heterogeneity cannot be too small: for population sizes smaller than 10, the results become worse).

The best fitness for *MultivariateLocalH* can be reached by setting:

`population-size`= 10

`population-model`= `steady-state-replace-worst`

In this framework the average fitness is about -1000 (it lies in the interval[-5000 , -200]).

- *RandomSearch*: with this algorithm the analysis lasts 3 seconds.

  The search is very inefficient with this search algorithm : the average fitness is about -16000 (it is the worst fitness, compared to the results of any other *BehaviorSearch* algorithm).

  These are not surprising results: the reliability of a random algorithm is inversely proportional to the search space magnitude.

- *MutationHillClimber*: the analysis requires 3 seconds.

  The average fitness, using custom input parameters, is about -1500, with a small variance: in fact the range of fitness values is included in the interval [-2000 , -500].

  As we have seen for *localH2*, *localH2.2* and *localH3*, the *MutationHillClimber* algorithms works well on smoothed (and monotone) functions.

  In general the efficiency of the algorithm, in terms of fitness is inversely proportional to the magnitude of `mutation-rate`, and

  `restart-after-stall-count` : as observed for smoothed functions (`localH2`, `localH2.2`) the if we increase these parameters, *MutationHillClimber* becomes similar to the `RandomSearch`, working badly on smoothed functions.

- *SimulatedAnnealing*: the analysis can be completed in 3-4 seconds.

  The fitness function `localDiff` reaches values in the interval [-4000 -200], if we consider the custom parameters of the algorithm.

  While if we change the values of these input parameters:

  we get the same effect explained for *MutationHillClimber*, if we modify `mutation-rate` and `restart-after-stall-count` ;

the parameters `initial-temperature` and `temperature-change-factor` do not modify the average fitness.

1.3.6 Technical achievements

What mainly emerge from my analyses is :

1. if we rise the value of the input parameters `mutation-rate`,

   `restart-after-stall-count`, (for any algorithm that owns these parameters) we increase the degree of randomness of the search (the search algorithm considered becomes similar to a *RandomSearch*);

2. *StandardGA*: it is the more flexible algorithm of *BehaviorSearch*. With the right combination of input parameters it provides a faster convergence to the optimal solution; however it can be difficult to find such combination of input parameters, since it depends on the conformation of the search space considered, that is not always note.

   The two parameters that affect the most, the effectiveness of the *StandardGA* are:

   (a) `population-size` : if we increase this value, we get a higher level of heterogeneity of the sample that is randomly chosen to evolve; this should ensure a greater probability of convergence to the optimal fitness. But the value of `population-size` should be considered together with the value of `Evaluation Limit`, i.e. the number of model runs: in fact a small value of the first with respect to the last, enlarges the number of evolutionary steps, providing a faster convergence.

   If we are analysing a search space that is smoothed, it is better to set a low value of `population-size`, since the degree of heterogeneity becomes not too relevant, while the number of evolutionary steps should be high; and vice-versa.

   (b) `population-model` : the effectiveness of this parameter is connected mainly to the magnitude of the search space; if we consider a quite small search space (as in *localH*, *localH2*, *localH3*) the model

   `steady-state-replace-worst` seems to work better than the others; but if we enlarge the search space the model `generational` becomes the most effective, while the model `steady-state-replace-worst` results the least effective.

One possible explanation of this effect can be given by considering the fact that if we replace one solution, we can decrease the degree of heterogeneity of the population.

The distinction between population models is less significant if we consider smoothed search spaces.

3. The `RandomSearch` algorithm is efficient only for small search spaces (as happened for *localH*, *localH2*, *localH3* cases).

4. The algorithms *MutationHillClimber* and *SimulatedAnnealing* provide a fast convergence only if the search space is smoothed.

Further considerations can be done by analysing in deep the `Search Encoding Representation`.

Chapter 2

Stock exchange simulation and search of the optimal agents behavior

2.1 User manual

This section contains the description of the behaviour of the agents interested in the simulation, through the explanation of the most interesting parts of the code.

2.1.1 g1 _ CDA _ basic _ model

My work in *NetLogo* starts from the program *g1 _ CDA _ basic _ model*.

This program creates a variable number of agents (according to the slider nRandomAgents) and displays them orderly.

Each agent has an equal probabilty of being a buyer or a seller, moreover, there is a probability to pass: it is chosen using a slider called `pass-level`.

The *NetLogo* code related to the definition of the trading decisions is:

```
ifelse out-of-market [set color white]
      [ifelse random-float 1 < passLevel [set pass True][set pass False]
       ifelse not pass
         [ifelse random-float 1 < 0.5
                    [set buy  True set sell False]
                    [set sell True set buy  False] ]
         [set buy False set sell False]
if pass          [set color gray]
if buy           [set color red]
if sell          [set color green]
```

```
;set price 501 + random 999
;set price random-normal 1000 100
set price exePrice + (random-normal 0 100)
```

Next, the agent is given a different random price (specifically, it is composed by a fixed part plus a random part). The series of different prices is ordered in a vector. From that vector are created two different vectors with agents prices, `logS` for sellers and `logB` for buyers. `logS` is sorted (set in increasing order for sellers) while `logB` is reverse sorted (set in decreasing order for buyers). Then the first elements of the two vectors are compared: if the buying price is greater than the selling price, two market prices are created: the ask price is the one held by the agent buyer (`agB`), while the bid price is the one held by the agent seller (`agS`).

That procedure is repeated continuously, eliminating the first element of both vectors of prices (`logB` and `logS`) once they generated a bid and an ask price; in this way it is always taken a different price as market price, resulting in the formation of bid and ask prices through an auction mechanism.

The most interesting part of the *NetLogo* program, for the price formation is the following:

```
ask randomAgents
  [if not pass and not out-of-market
     [
       let tmp[]
       set tmp lput price tmp
       set tmp lput who tmp
       if buy [set logB lput tmp logB]
       set logB reverse sort-by [item 0 ?1 < item 0 ?2] logB
       ;show logB
       if sell [set logS lput tmp logS]
              set logS sort-by [item 0 ?1 < item 0 ?2] logS
              ;show logS
```

Where `tmp` is a temporary vector that contains all agents prices; each price is inserted in the vector through the *NetLogo* command `lput`, that stays for 'last put'(the command puts the price inside the vector as last element, increasing the vector length by one).

Moreover the temporary vector is built to contain also the agent number for the specific considered price (through the command `who`), in order to do not loose the identity of the agent that participates to the auction mechanism (in fact without this command, the identity of each agent can be loosen because of the `ask` command, that shuffles continuously the order of the agents).

Then the content of `tmp` is released into the two vectors `logB`, if the agent is a buyer, and `logS`, if the agent is seller.

Finally the price is generated through the code:

```
if (not empty? logB and not empty? logS) and
          item 0 (item 0 logB) >= item 0 (item 0 logS)
            [set exePrice item 0 (item 0 logS)
             let agB item 1 (item 0 logB)
             let agS item 1 (item 0 logS)
```

for selling price (bid);

```
if (not empty? logB and not empty? logS) and
          item 0 (item 0 logB) >= item 0 (item 0 logS)
            [set exePrice item 0 (item 0 logB)
             let agB item 1 (item 0 logB)
             let agS item 1 (item 0 logS)
```

for buying price (ask).

The specification `item 0 (item 0 logB)` and `item 0 (item 0 logS)`, that locates the first element of the vector, have that form because `logB`, `logS` are matrices: the number of rows corresponds to the number of buying and selling prices; the number of columns is two: the first column contains the prices, the second column contains the number that identifies the corresponding agent.

2.1.2 Level Price Real Data Agents

Starting from the work of *Terna*, *g1_CDA_basic_model*, that is a clear example of how a class of non intelligent (simple) agents acts in an organized framework, I put the basis of my work, defining a new class of agents: the *Level Price Real Data Agents* (`arbitrageur`).

Those agents are 'intelligent' because they do not adopt a market strategy based on randomness, but they act as arbitrageurs.

They are inserted in the *g1_CDA_basic_model* framework: their participation to the market depends always on an external real market price(in our case *Ftse All Share* stock price) ;

they can decide

- to propose a bid price if the corresponding real price is lower than the lowest price of the sellers vector `logS`, otherwise they will propose an ask price;
- to propose an ask price if the contemporaneous real price is higher than the highest price of the buyers vector `logB`, otherwise they will propose a bid price.

For this reason this category of agents works as it knows, some second in advance the level that the market price will reach, basing their evaluations on a real index.

In practice, the main part for the creation of that class, consist in:

1. creating a vector with the real prices, that can be red at any moment;

2. the creation of an efficient counter for the number of the artificial market prices,that are formed continuously.

This because once formed, an artificial market price should be synchronized with each real market price, in order to compare them efficiently.

The creation of the real price vector is obtained in the `Setup` procedure, through the following code:

```
to read-file
  file-open "Ftse_All_Share_nice_format.txt"
  while [not file-at-end?][
    ;read one line
    let in1 file-read
    set realPvec1 lput in1 realPvec1
    set realPvec2 lput in1 realPvec2
  ]
  file-close
end
```

As usually happens the file `Ftse_All_Share_nice_format.txt` must be in the same folder of the *NetLogo* program, to be opened.

The counter `j` is putted at beginning of the artificial price formation part of code:

```
ask turtles
  [
    set j j + 1
```

The proper trading strategy of *arbitrageurs* is implemented through the code:

```
       if (not empty? logB and not empty? logS) and
          item 0 (item 0 logB) >= item 0 (item 0 logS)
            [
              if j <= 10095[
                ask arbitrageurs[
                  ifelse (not empty? logB and not empty? logS)
                   and item 0 (item 0 logS) > item j realPvec1
                   [let tmp2 []
```

```
                  set tmp2 lput price tmp2
                  set tmp2 lput who tmp2
                  set logS fput tmp2 logS
                  set AE AE + 1
                  set realPvec2 but-first realPvec2]
                 [let tmp2 []
                  set tmp2 lput price tmp2
                  set tmp2 lput who tmp2
                  set logB fput tmp2 logB
                  set realPvec2 but-first realPvec2
                       ]]]
          set exePrice item 0 (item 0 logS)
          let agB item 1 (item 0 logB)
          let agS item 1 (item 0 logS)

 if (not empty? logB and not empty? logS) and
           item 0 (item 0 logB) >= item 0 (item 0 logS)
             [
               if j <= 10095[
                  ask arbitrageurs[
                   ifelse (not empty? logB and not empty? logS) and
                    item 0 (item 0 logB) < item j realPvec1
                    [let tmp2 []
                      set tmp2 lput price tmp2
                      set tmp2 lput who tmp2
                      set logB fput tmp2 logB
                      set AE AE + 1
                      set realPvec2 but-first realPvec2]
                    [let tmp2 []
                      set tmp2 lput price tmp2
                      set tmp2 lput who tmp2
                      set logS fput tmp2 logS
                      set realPvec2 but-first realPvec2
                           ]]]
               set exePrice item 0 (item 0 logB)
              let agB item 1 (item 0 logB)
              let agS item 1 (item 0 logS)
```

The control `if j < 10095` has been used to continue the artificial price formation, even after the last element of the real price vector has been considered, i.e. when *arbitrageurs* stop their strategy.

The **Ftse All Share** stock price is an index of the *Telematic Stock Market*, governed by the *Italian Stock Exchange* (that replaces *Mibtel* from 2009).

It is composed by the aggregation of all the elements of *FTSE MIB, FTSE Italia Mid Cap e FTSE Italia Small Cap* financial indexes, excluding *FTSE Italia Micro Cap.*

In my thesis the `arbitrageur` agent will consider the time series generated by 10095 realisations of the real index, starting from the 13.12.2010.

2.1.3 Trend Agents

Another class of agents, that reflects the real behaviour of lots of investors, that trust in *technical analysis*, is the class of*Trend Agents.* The word 'trend' in finance is often associated to the moving average of a given index or price: the easier moving average to implement is the 'simple moving average'; it is a mean always calculated on the past k realizations of the price.

The Trend Agents are able to calculate a simple moving average on a given sample; they base their trading strategy on the value of this trend indicator, comparing it with the current market price (produced through the auction mechanism).

In my *NetLogo* program the sample number (k) is different for each *Trend Agent*, and it is given through the variable `sample`.

That variable is defined in the `Setup` procedure, in the following way:

```
set sample 50 - random 40.
```

The moving average is often used in *technical analysis* as a tool to predict market prices movements: The bigger is the sample considered the more consistent is the estimator, but at the same time the smaller is the sample the more sensitive to current price variation the moving average is.

For what concern the strategy adopted in my program: a *Trend Agent* will propose a bid price if the value of the sample mean is bigger than the current market price, while he will propose an ask price if the value of the sample mean is smaller than the current market price.

The parts of the code that characterize the behaviour of these agents are divided into:

1. sample mean calculation for each agent according to the variable `sample`.

```
ask trendAgents
         [
          let p 0
          let n length SboxP
```

```
if n  > (sample + k) and sample > 0
     [
      let i 0 + k
      while [i < sample + k - 1]
               [ set sm sm + item i SboxP
                 set i i + 1 ]
      set sm sm / (sample )
      set mv1 lput sm mv1
      set mv2 lput sm mv2
      set k k + 1
]
```

2. bid price formation condition:

```
ask trendAgents
           [
            if length days > sample and not empty? mv2 and
               item 0 (item 0 logS) > item 0 mv2
               [set sell false set buy true set pass false
                ;set mv2 but-first mv2
                 ]
            if length days > sample and not empty? mv2 and
               item 0 (item 0 logS) < item 0 mv2
               [set buy false set sell true set pass false
                ;set mv2 but-first mv2
                ]
            if length mv2 = 2 [set mv2 but-first mv2]
           ]
```

3. ask price formation condition:

```
ask trendAgents
           [
```

```
        if length days > sample and not empty? mv2 and
           item 0 (item 0 logB) > item 0 mv2
           [set sell false set buy true set pass false
           ; set mv2 but-first mv2
             ]
        if length days > sample and not empty? mv2 and
           item 0 (item 0 logB) < item 0 mv2
           [set buy false set sell true set pass false
            ;set mv2 but-first mv2
            ]
        if length mv2 = 2 [set mv2 but-first mv2]
       ]
```

2.1.4 Volume Agents

This particular category of agents acts in market depending on *trading volumes*.

In capital markets, volume, or trading volume, is the number of shares or contracts traded in a security or in an entire market during a given period of time. In the context of stock trading on a stock exchange, the volume is commonly reported as the number of shares that changed hands during the day.

In our case the market is made of only one stock, and the prices generated in one transaction are prices per minute. To be consistent with reality, those agents should evaluate the difference in market volumes , about at least every seven hundred prices, i.e. one trading day (for consistency with respect to the common definition of volumes).

In my program this kind of step seems too big as minimum, since the time series of real prices consists in only 10095 elements and the effect of `VolumeAgents` could be too small.

For that reason I have considered as step between successive trades, the value of a slider (`VolumeAgentStep`) multiplied by ten: the range of the slider is [1 , 100] with step 1.

In this way we get a minimum step of ten realisations of market prices, and a maximum of 1000 realisations of market prices .

The main part of the program that characterize the *Volume Agents* behavior is the following:

```
ask volumeAgents[ifelse jv >= VolumeAgentStep * 10[
```

```
if diff > 0
  [set buy true set sell false set pass false set jv 0]
if diff < 0
 [set sell true set sell false set pass false set jv 0]
if diff = 0
 [set pass true set buy false set sell false set jv 0]]
[set pass true set buy false set sell false]
                      ]
```

This part of code is located inside the auction price mechanism, and it is repeated twice, i.e. for ask and for bid prices.

The variable `jv` counts the number of prices per minute.

The variable `diff` takes into account the difference between the purchase volumes and the sale volumes. The related code that defines the variable `diff` is:

```
set diff (length logB - length logS)
```

Where `logB` is the vector of buyers, and `logS` is the vector of sellers (as explained in Section 1.1).

2.1.5 Stop Loss Agents

The Stop-Loss strategy is a strategy involving the shortage of a call and the trading of a stock.

While the former is an operation made once, the latter could require more trades depending on the path of the underlying.

The investor can take two possible positions:

Naked : the investor is short a call. It produces a profit equal to the call price in $t = 0$ if the call is OTM (out-of-money) at the expiry date, but leads to significant losses when the call expires ITM (in-the-money).

Covered : the investor is short a call and long the underlying (assumed to be bought at K, the strike price). It produces a profit equal to the call price in $t = 0$ if the call expires ITM, but leads to significant losses when it expires OTM.

The strategy kernel lies in the intersection between the two position.

It is a simple hedging strategy, and its main objective is to ensure that at time T, the bank owns the stock if the option closes in the money and does not own it if the option closes out of money.

The basic assumption for the functioning of the strategy are:

- Lognormality of asset values (underlying).
- Call priced using the Black-Scholes formula (BS).

In t = 0 the investor sells a call and has an inflow equal to the call price. Since in t = 0 the spot price is lower than the strike price the stop-loss investor is only short a call. As soon as the stock price touches the strike price barrier from the down the investor goes long on the underlying, whose value is K.

If the stock price hits again the barrier (this time from the up) the trader sells the asset, remaining naked on the call. This procedure is repeated every time the stock price equals K.

Therefore, the strategy is a combination of naked and covered positions that exploits only the advantages of the two. In fact, when the option is ITM the loss is avoided because the seller of the call holds the stock in the portfolio and the call's payoff(from the seller point of view) becomes `Max (K - K, 0)`. On the other hand, when the option is OTM is not exercised, hence the payoff is 0. In any case, the stop loss investor obtains a payoff equal to the call price received in t = 0.

To give a consistent explanation of the strategy, I will explain the Black-Scholes model.

Black-Scholes model

The Black-Scholes model for calculating the premium of an option was introduced in 1973 in a paper entitled, "The Pricing of Options and Corporate Liabilities" published in the Journal of Political Economy. The formula, developed by three economists, Fischer Black, Myron Scholes and Robert Merton , is perhaps the world's most well-known options pricing model. Black passed away two years before Scholes and Merton were awarded the 1997 Nobel Prize in Economics for their work in finding a new method to determine the value of derivatives (the Nobel Prize is not given posthumously; however, the Nobel committee acknowledged Black's role in the Black-Scholes model).

The Black-Scholes model is used to calculate the theoretical price of European put and call options, ignoring any dividends paid during the option's lifetime. While the original Black-Scholes model did not take into consideration the effects of dividends paid during the life of the option, the model can be adapted to account for dividends by determining the ex -dividend date value of the underlying stock.

The model makes certain assumptions, including:

- The options are European and can only be exercised at expiration.
- No dividends are paid out during the life of the option.
- Efficient markets (i.e., market movements cannot be predicted).
- No commissions.

- The risk-free rate and volatility of the underlying are known. and constant.
- Follows a lognormal distribution; that is, returns on the underlying are normally distributed.

The formula takes the following variables into consideration:

1. Current underlying price.
2. Options strike price.
3. Time until expiration, expressed as a percent of a year.
4. Implied volatility.
5. Risk-free interest rates.

The Black-Scholes pricing formula for call options is:

$$C(0) = S \cdot N(d_1) - K \cdot \exp(-r \cdot T) \cdot N(d_2) \tag{2.1}$$

where

$$d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2}) \cdot t}{\sigma \cdot \sqrt{t}} \qquad d_2 = d_1 - \sigma \cdot \sqrt{t} \tag{2.2}$$

C(0) = Call premium at time zero
S = Current stock price or underlying
K = Option strike price
r = Risk-free interest rate
N = Cumulative standard normal distribution
σ = St.deviation

The model is essentially divided into two parts: the first part, SN(d1), multiplies the price by the change in the call premium in relation to a change in the underlying price. This part of the formula shows the expected benefit of purchasing the underlying outright. The second part, `N(d2)Kexp(-rt)`, provides the current value of paying the exercise price upon expiration (remember, the Black-Scholes model applies to European options that are exercisable only on expiration day). The value of the option is calculated by taking the difference between the two parts.

To calculate the B-S formula with *NetLogo* we must consider that S, the stock price, is basically generated by the `randomAgents`, through the code explained in section 1.1 (*g1_cda_basic_model*) ; the calculations of input parameters `K` and `r`, are exogenous, and so their value is represented by:

- the value of the `r` is inserted into a slider, i.e. `risk-free`;
- `K` is the strike, it represents a bet for the investor, so its value depends on the circumstances; in the code the parameter is obtained through the command:

  ```
  set strike exePrice + (random-float 2 - 1) * 200
  ```

The last parameter necessary to perform the B-S formula is `N`: unfortunately there is not a *NetLogo* command to calculate the cumulative Standard Normal distribution;

To solve the problem can be used the error function.

In mathematics, the error function (also called the Gauss error function) is a special function (non-elementary) of sigmoid shape which occurs in probability, statistics and partial differential equations. It is defined as

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}$$

The complementary error function, denoted *erfc*, is defined as:

$$erfc(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} = e^{-x^2} erfcx(x)$$

which also defines *erfcx*, the scaled complementary error function.

An approximation (with a polynomial of degree one) of *erfcx*(x) can be given through the following *NetLogo* code.

```
to-report erfcc [x]
        let z abs x
        let q 1.0 / (1.0 + 0.5 * z)
        let r q *  exp ( - z * z - 1.26551223
        + q * (1.00002368 + q * (0.37409196 +
            q * (0.09678418 + q * ( - 0.18628806 + q *
(0.27886807 + q * ( - 1.13520398 + q * (1.48851587
 + q * ( - 0.82215223 + q * 0.17087277 )))))))))
        ifelse (x >= 0) [ report r ] [report 2.0 - r]
end
```

Stop-Loss strategy on NetLogo

In order to properly set up the strategy we need to use all the parameters involved in the computation of the call price, according to the Black-Scholes formula. A crucial role is played by strike which creates a partition of the space changing the

composition of the portfolios of Stop-Loss investors and the magnitude of the cash flows as soon as exeprice hits its barrier.

Another important parameter is T, since for the strategy to work properly we need to calibrate the correct timing. In fact, when the call expires the cash flow occurring can either be 0 or K.

In my program the value of `T` is given by a slider; it represents the step between progressive implementation of the strategy: `T` = 1 means 700 price realisations, i.e. about one trading day, since each market price is assumed to have one minute frequency.

In order to generate the volatility of the stock we have to consider the log-returns of the underlying, generated by:

```
set priceVector fput exePrice priceVector
if length priceVector > 1
[let tmp2 ln ( item 0 priceVector / item 1 priceVector )
set logVector fput tmp2 logVector]
```

Each price generated is put in the pricevector in first position, that collects prices generated in different periods thanks to interaction among agents. Starting from this vector we will define the volatility of the log-returns, defined as the natural logarithm of the last and second to last price.

When the vector gets too big, we eliminate the non relevant components by:

```
if length logVector > 700
[
set logVector butlast logVector
set sigma standard-deviation (logVector) * sqrt (36400)
]
```

The strategy duration is 700*T, where T can be arbitrarily chosen. The call price is computed, according to the Black-Scholes formula, as a function of sigma, T, r, S and K.

The strike price is formed on the basis of the last exeprice and a random float which takes values between -1 and 1 , multiplied by 200. The strike price is only computed at the beginning of the strategy and does not change until the call reaches its expiry.

The part of code related to the check of the strategy (includes checking the `naked` and `covered` conditions), that occurs at the end of each trading day, is the following:

```
if jsl = 700 [ ask SLAgents[
      if sigma > 0 [
```

```
if naked and exeprice > strike
 [
 set covered true set naked false
 set portf portf - exePrice
 set price strike set buy true set sell false
 ]
if covered and exeprice < strike
 [
 set naked true set covered false
 set portf portf + exeprice
 set price strike
 set sell true set buy false
 ]
 ]
                ]
 set jsl 0
 ]
```

Where the index `jsl` memorizes the number of price realisation until 700, then it restarts from zero.

The implementation of the strategy occurs at two different times: `T-1` and `T` ; the *NetLogo* code is,

Note that agents affect the price formation mechanism when the spot price hits the barrier of the strike price. The final step of the procedure occurs in T -1, when the strategy terminates and the call expires. If the stop loss investor has a naked position, then there is not any cash flow since the option is not exercised. If the stop loss investor has a covered position, he benefits of a positive cash flow of K, that is the amount paid by the buyer of the call for getting the underlying at a price lower than the spot price observable in the market at that moment.

```
if k = 700 * (T - 1) and p > 0[
                       ask SLAgents[
                                    if exePrice > strike
                                      [
                                       set portf portf + strike
                                       ]
                                    ]
                              ]
```

Where `k` works as `jsl`, but it is restarted once reached the time `T` ; the variable `portf` takes into account the value of the `SLAgents` portfolio.

For time `T` :

```
if k = 700 * T [
          ask SLAgents[
             set strike exePrice + (random-float 2 - 1) * 200
             if sigma > 0 [
 set d1 (ln (exeprice / strike)
 + (risk-free + ((sigma ^ 2) / 2)) * T) / ( sigma * sqrt ( T))
 set d2 d1 - sigma * sqrt (T)
            let a d1 / (2 ^ 0.5)
            let b d2 / (2 ^ 0.5)
            set Nd1 1 - 0.5 * erfcc a
            set Nd2 1 - 0.5 * erfcc b
 set BScall exePrice * Nd1 -
 strike * exp( - risk-free * T ) * Nd2
            set BSvec fput BScall BSvec
            ifelse exePrice < strike
                [set portf portf + BScall
                 set naked true
                 set covered false]
                [set portf portf + BScall - exePrice
                 set covered true
                 set naked false
                 set price strike
                 set buy true
                 set sell false]
                 ]
                    ]
                  set k 0
                  set p p + 1
                      ]
```

In the code above the portfolio of `SLAgents` is modified according to the `naked` or `covered` situations; the parameters for the calculation of the option price are reset and the naked and covered position are restated: this happens to let the strategy restart (remember that these codes are repeated twice, in order to work in both, buying market price and selling market price formation, generated through the commands of the program `g1_cda_basic_model` explained in section 2.1.1).

2.1.6 Covered Agents

The Covered Agents breed, that I describe in this paragraph, represents a particular class of trading agents that focuses its market strategy on both *technical*

analysis and options.

In fact, first of all `CoveredAgents` are able to evaluate the market price trend using a moving average, whose sample is a random number, that is different for any `CoveredAgent` (exactly as happened with `trendAgents`).

In this way those agents can identify two different market situations:

1. increasing market, if the current artificial market price (`exeprice`) is bigger than the last value of the moving average;

2. decreasing market, if the current artificial market price is smaller than the last value of the moving average.

The last step of the strategy requires the calculation of the option price using *Black and Scholes*, as happened with `SLAgents` (for the explanation of the B-S formula, and its implementation on *NetLogo* , see Section 2.1.5 point 1).

Then each `CoveredAgent` will cover the two market situations described above, in the following way:

- if there is an increasing market (case 1), the agent will sell a CALL option OTM (out-of-the-money);

- if there is a decreasing market (case 2), the agent will sell a PUT option OTM.

In finance, a *call option*, often simply labeled 'call', is a financial contract between two parties, the buyer and the seller of this type of option. The buyer of the call option has the right, but not the obligation to buy an agreed quantity of a particular commodity or financial instrument (the underlying) from the seller of the option at a certain time (the expiration date) for a certain price (the strike price). The seller (or 'writer') is obligated to sell the commodity or financial instrument to the buyer if the buyer so decides. The buyer pays a fee (called a premium) for this right.

Instead a *put* or *put option* is a stock market device which gives the owner the right, but not the obligation, to sell an asset (the underlying), at a specified price (the strike), by a predetermined date (the expiry or maturity) to a given party (the seller of the put). If the price of the stock declines below the specified price of the put option, the owner of the put has the right, but not the obligation, to sell the asset at the specified price, while the seller of the put, has the obligation to purchase the asset at the strike price if the buyer uses the right to do so (the buyer is said to exercise the put or put option). In this way the owner of the put will receive at least the strike price specified even if the asset is worth less.

In my *NetLogo* program, the expire date of the option is a random number, different for any agent; the trading strategy starts after the calculation of the first moving average value, then it is always re-implemented after the expire date; the strike changes depending on the agent considered (since its value contains a random number), and on the current marked situation: to generate an OTM CALL the strike should be greater than the current market price; while to generate an OTM PUT, the strike should be smaller than the current market price (in this case for current market price I mean the price used in the calculation of the B-S formula, i.e. the one compared to the moving average at the beginning of the trading strategy). For the reasons described above, each `CoveredAgent` owns a different B-S price.

Moreover the sale of the option is traded OTM, to earn a profit from both: initial inflow equal to the option price, spread between buying and selling price at maturity.

All steps of the strategy in terms of cash flows, can be synthesized as follows (to briefly analyse such steps I assume a generic starting and ending times for the strategy, t = 0 and t = T respectively, for the generic `CoveredAgent` i).

CASE 1 (a) t = 0: if the current market price is bigger than the current value of the moving average :

- BUY the underlying ($-S_0$)
- SELL an OTM CALL ($+c_0$)

(b) t = T:

- if the current value of the underlying is bigger than the strike: SELL the underlying at the strike price (+ K)
- if the current value of the underlying is smaller than the strike: SELL the underlying ($+S_T$)

CASE 2 (a) t = 0: if the current value of the underlying is smaller than the current value of the moving average:

- SELL an OTM PUT ($+p_0$)

(b) t = T:

- if the current value of the underlying is bigger than the strike: BUY the current market price and then sell it immediately (- $S_T + S_T$)

- if the current value of the underlying is smaller than the strike: BUY the underlying at the strike price (- K)

(Where S is the market price or underlying; K is the strike; c and p are respectively the call and the put prices).

The *NetLogo* codes that characterizes the `CoveredAgents` behavior are divided in the following parts.

- Calculation of the moving average, according to the variable `sample` (has happened with `trendAgents`) :

```
ask coveredAgents
         [
          let pc 0
          let nc length SboxPc
          if nc  > (samplec + kc) and samplec > 0
               [
                let i 0 + kc
                while [i < samplec + kc - 1]
                         [ set smc smc + item i SboxPc
                           set i i + 1 ]
                set smc smc / (samplec )
                set mvc1 lput smc mvc1
                set mvc2 lput smc mvc2
                set kc kc + 1
 ]
 ]
```

- Definition of the two possible position, that can be assumed at the beginning of the strategy, comprehending the calculations of both call and put prices with B-S formula (the implementation of the B-S formula on *NetLogo* is described in detail in Section 2.1.5 point 1).

```
ask coveredAgents
```

```
[
   if length vecC > samplec and not empty? mvc2
   and item 0 (item 0 logS) < item 0 mvc2 and not
   finish and not stopc
    [ set strikec exePrice -
      (random-float 2 * expiredate)
       set pass true
      set color green set buy2 true set buy1 false
      if sigmac > 0 [
           set d1c (ln (exeprice / strikec) +
           (risk-free + ((sigmac ^ 2) / 2)) *
           ((expiredate * Tc) / 61200)) / ( sigmac *
            sqrt ( ((expiredate * Tc) / 61200)))
            set d2c d1c - sigmac * sqrt
           (((expiredate * Tc) / 61200))
            ;show d1
            ;show d2
          let acc d1c / (2 ^ 0.5)
          let bcc d2c / (2 ^ 0.5)
          let cc -d1c / (2 ^ 0.5)
          let dc -d2c / (2 ^ 0.5)
          set Nd1c 1 - 0.5 * erfcc acc
          set Nd2c 1 - 0.5 * erfcc bcc
          set N-d1c 1 - Nd1c
          set N-d2c 1 - Nd2c
          set BSputc (strikec * exp( - risk-free *
          ((expiredate * Tc) / 61200) )
           * N-d2c - exePrice * N-d1c)
           ;show BSputc
           show "Put"
           set covPort covPort + BSputc
```

```
]
      set stopc true
     ]
 if length vecC > samplec and not empty?
 mvc2 and item 0 (item 0 logS) > item 0 mvc2
 and not finish and not stopc
    [ set strikec exePrice +
     (random-float 2 * expiredate)
     set pass true
     let tmc []
     set price exeprice
     set tmc lput price tmc
     set tmc lput who tmc
     set logB lput tmc logB
     set color blue set buy1 true set buy2 false
      if sigmac > 0 [
                 set d1c (ln (exeprice / strikec) +
                 (risk-free + ((sigmac ^ 2) / 2)) *
         ((expiredate * Tc) / 61200)) / ( sigmac *
         sqrt ( ((expiredate * Tc) / 61200)))
         set d2c d1c - sigmac *
         sqrt (((expiredate * Tc) / 61200))
            ;show d1
            ;show d2
           let acc d1c / (2 ^ 0.5)
            let bcc d2c / (2 ^ 0.5)
            let cc -d1c / (2 ^ 0.5)
            let dc -d2c / (2 ^ 0.5)
           set Nd1c 1 - 0.5 * erfcc acc
            set Nd2c 1 - 0.5 * erfcc bcc
            set N-d1c 1 - Nd1c
```

```
                      set N-d2c 1 - Nd2c
                      set BScallc (exePrice * Nd1c - strikec *
                      exp( - risk-free *
                      ((expiredate * Tc) / 61200) ) * Nd2c)
                      ;show BScallc
                      show "Call"
                     set covPort covPort + BScallc - exeprice
                ]
               ;set mv2 but-first mv2
                set stopc true
                ]
           if length mvc2 = 2 [set mvc2 but-first mvc2]
          ]
```

Where the variables `BSCallc` and `BSputc` contains the current call and put prices respectively; the variable `covPort` memorizes the cash flows obtained through the implementation of the strategy.

Remember that the strike to have an OTM situation should be: greater than the underlying for the call, and smaller than the underlying for the put; this fact can be seen in the code by the commands:

- `set strikec exePrice + (random-float 2 * expiredate)` for the OTM call;
- `set strikec exePrice - (random-float 2 * expiredate)` for the OTM put.

- The end of strategy depends on the variable `expireDate`, initialised in the `setup` procedure through the command

  `set expireDate  (random 50 * Tc + 10 )`

  where the range of `Tc` is defined by a slider.

  The *NetLogo* code that defines the behavior of `CoveredAgents` at the `expireDate`, is the following.

```
ask coveredAgents[
           if jcc >= expiredate * Tc
```

```
[
 if buy1[
  ifelse exeprice > strikec
   [set buy false set  sell true
    set pass false set price strikec
    set stopc false set jcc 0
    set covPort covPort + strikec]
   [set pass false set buy false
    set sell true set price exeprice
    set stopc false set jcc 0
    set covPort covPort + exeprice]]
 if buy2[
  ifelse strikec > exeprice
   [set buy true set sell false
    set pass false set price strikec
    set stopc false set jcc 0
    set covPort covPort - strikec]
   [set pass false set buy true
    set price exeprice set sell false
    set stopc false
    let tmc []
    set tmc lput exeprice tmc
    set tmc lput who tmc
    show "buy sell"
    set logS lput tmc logS
    set covPort covPort - exeprice + exeprice
]]]
```

The maturity of the options is set up at `expireDate * Tc`. The variable `jcc` works as counter for the steps of the strategy, and it is specific of the `CoveredAgents` breed.

2.1.7 Bollinger Bands Agents

This kind of agents can be considered as an implementation of the *Trend Agents* category.

Bollinger Bands strategy

Their strategy is based on the concepts of 'support' and 'resistance' levels; These values are said to be price levels above which it is difficult for stock prices to rise, or below which it is unlikely for them to fall, and they are believed to be levels determined by market psychology.

John A. Bollinger (born 1950) is an American author, financial analist, contributor to the field of technical analysis and the developer of Bollinger Bands. His model was developed in the 80's and it consider:

1. Simple moving average, it is a mean of fixed amount of data (often twenty days), generally it use the closing prices of the market. The term 'moving' is referred to the fact that are consider the last willing prices. Indeed moving average are smoothed lines that can show more easily price trends.

2. An upper band and a lower band which are calculated through k times (generally k equal to two) the volatility (statistically is the standard deviation).

So the upper band is obtained by adding to the moving average k times the standard deviation, while the lower band is calculated subtracting to the moving average k times the standard deviation. If stock prices follow a normal distribution then Bollinger bands with k=2 will capture around 95 percent of price movements (level of confidence). The region above the upper band will be considered as overbought; the region under the lower band will be considered as oversold.

So one possible strategy to buy or sell can be:

- Buying strategy:

  $P_N(t-1) < BB_N^{LOW}(t-1)$ and $P_N(t) > BB_N^{LOW}(t)$

- Selling strategy:

  $P_N(t-1) > BB_N^{UP}(t-1)$ and $P_N(t) < BB_N^{UP}(t)$

Where BB_N^{LOW} and BB_N^{UP} and represent the lower and the upper band respectively.

The computational aspect, as for the other categories of agents is to modify the program g1_basic_model in *NetLogo*, which generates a random market composed by investors who sell or buy randomly.

The elaborate cares to put in the program, the active strategy, that is, enter a code that calculates the moving average and Bollinger Bands, whereby then the investor will decide whether to sell or buy.

Bollinger Bands on NetLogo

The first step is the creation of Bollinger Bands agents (`BBAgents`) according to the slider `nBBAgents`. The crucial point of the Bollinger Bands strategy (`BBStrat`) is the capability of `BBAgents` to compare each price realization with the value of the bands at the same time. For this purpose it is necessary to memorize every market price and every bands value in different vectors (`SboxP` for prices, `SboxLB` for lower band, `SboxUB` for upper band). Then in each tick (if the length of those vectors is greater than two) the first two elements of the vectors are compared to identify the overbought and oversold conditions.

```
to BBStrat
  ask BBAgents[
   if length SBoxLB >= 2 and length SBoxP >= 2 and
      item 0 SBoxP < item 0 SBoxLB and
      item 1 SBoxP > item 1 SBoxLB and
      sold >= 1 [set BBpocket BBpocket - item 1 (SBoxP)
                                  set overbought True
                                  set oversold False
                                  set ovB ovB - 1
                                  set AR lput BBpocket AR
                                  set purchase purchase + 1
 set sold sold - 1]
   if length SBoxUB >= 2 and length SBoxP >= 2 and
      item 0 SBoxP > item 0 SBoxUB and
      item 1 SBoxP < item 1 SBoxUB and
      purchase >= 1 [set BBpocket BBpocket + item 1 SBoxP
                                  set oversold True
                                  set overbought False
                                  set ovS ovS + 1
                                  set AR lput BBpocket AR
                                  set purchase purchase - 1
 set sold sold + 1]]
```

To perform the strategy in continuous time, as imposed for the moving average and the standard deviation, we must eliminate the first item of each vector, after the command to BBStrat.

```
if length SBoxP >= 2 and length SBoxLB >= 2 and length SBoxUB >= 2
```

```
[set SBoxP but-first SBoxP
 set SBoxLB but-first SBoxLB
 set SBoxUB but-first SboxUB]
```

In our program BBAgents memorize every step of the strategy in the global variable BBpocket : for what concerns the cash flow statement the buy position (overbought) has negative sign while the sell position (oversold) has positive sign. BBAgents starts the strategy with a long position, and next they sell or buy only if they respectively have bought or sold in previous periods.

Moreover BBAgents will enter in the price formation, that is the auction mechanism, only if there are the Bollinger signals of overbought or oversold.

In the program this is implemented through:

```
ask randomAgents
    [if not pass and not out-of-market
          [
             let tmp[]
             set tmp lput price tmp
             ask BBAgents[if overbought [set tmp lput price tmp]]
             ask BBAgents[if oversold [set tmp lput price tmp]]
             set tmp lput who tmp
```

where first all random prices of randomAgents are putted in the vector tmp ; then if the Bollinger strategy suggest to buy (overbought) or to sell (oversold), also all random prices of BBAgents are putted in the vector tmp. Remember that this vector is the main vehicle for the auction mechanism, as explained in Section 1.1.

2.2 Basic Framework

The basic idea that I want to develop in my thesis, refers to the creation of a basic market structure, built with both RandomAgents and

Level Price Real Data Agents classes.

These two categories of trading agents are grouped in a *NetLogo* program, called *Artificial_ VS_ real_ mkt*. Its interface is shown in Figures 2.1, 2.2.

Figure 2.1 shows how appears the real price trend (of *Ftse all share* stock), that is created in the setup procedure (explained in Section 2.1.2).

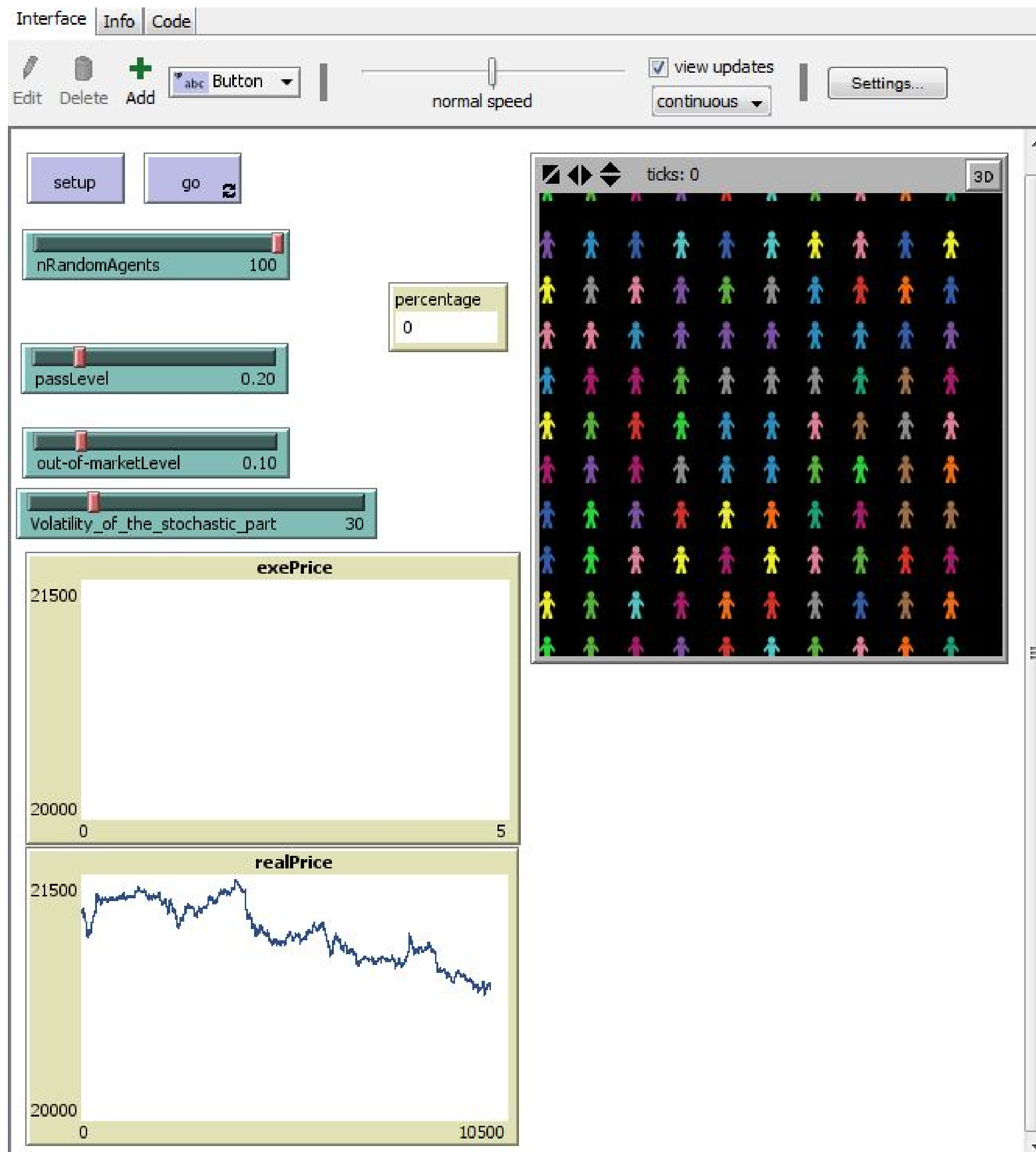

Figure 2.1: *Artificial_ VS_ real_ mkt* interface after have pressed the setup button.

That trend is also an useful indicator, in order to compare the market price generated step by step with the real one;

At the moment the program created the two categories of agents (one-hundred `RandomAgents` according to the slider `nRandomAgents`, and one arbitrageur), and their variables have been initialised; but they have not started the creation of the artificial market yet.

The next step is to let the auction mechanism begin (explained in Section 2.1.1), and this is implemented by pressing the `go` button.

Figure 2.2 shows the appearance of the *NetLogo* interface when the market is generated, i.e. pressing the *go* button.

Remember that(explained in Section 1.1):

- the slider `nRandomAgents` represents the number of `RandomAgents` that are partecipating to the auction price formation. It is kept always fixed at one-hundred;

- the slider `passLevel` represents the probability that a `RandomAgent` will not enter in the auction price formation; It is allowed to variate in an interval with extremes 0.00 and 0.80, with step 0.01 (the possible framework with pass probability equal to one seems too strong, meaning that the possibility that the market does not exist is taken into account);

- the slider `Volatility_of_the_stochastic_part` represents the variance of the stochastic part of the price: the price that each `RandomAgent` will propose in the auction mechanism is in facts obtained as the sum of a fixed part (21300) and a stochastic part: it is a number coming from a Standard Normal distribution with mean zero and variance equal to the value of the slider.

- the monitor `percentage` shows the ratio between the number of interventions of the agent `arbitrageur` in the market and the total number of real market prices considered (10095).

Note that the values of the parameters set up in the sliders are chosen based on a rule of thumb (except for `nRandomAgents` that is kept fixed); in facts there is not a specific combination of such values (for example the deterministic part and stochastic part of the price), that ensures the smallest distance between the artificial and the real market.

With *BehaviorSearch* we can state (see Section 5.1) that in the basic framework, the difference in absolute value, between the two markets considered is at least fifty

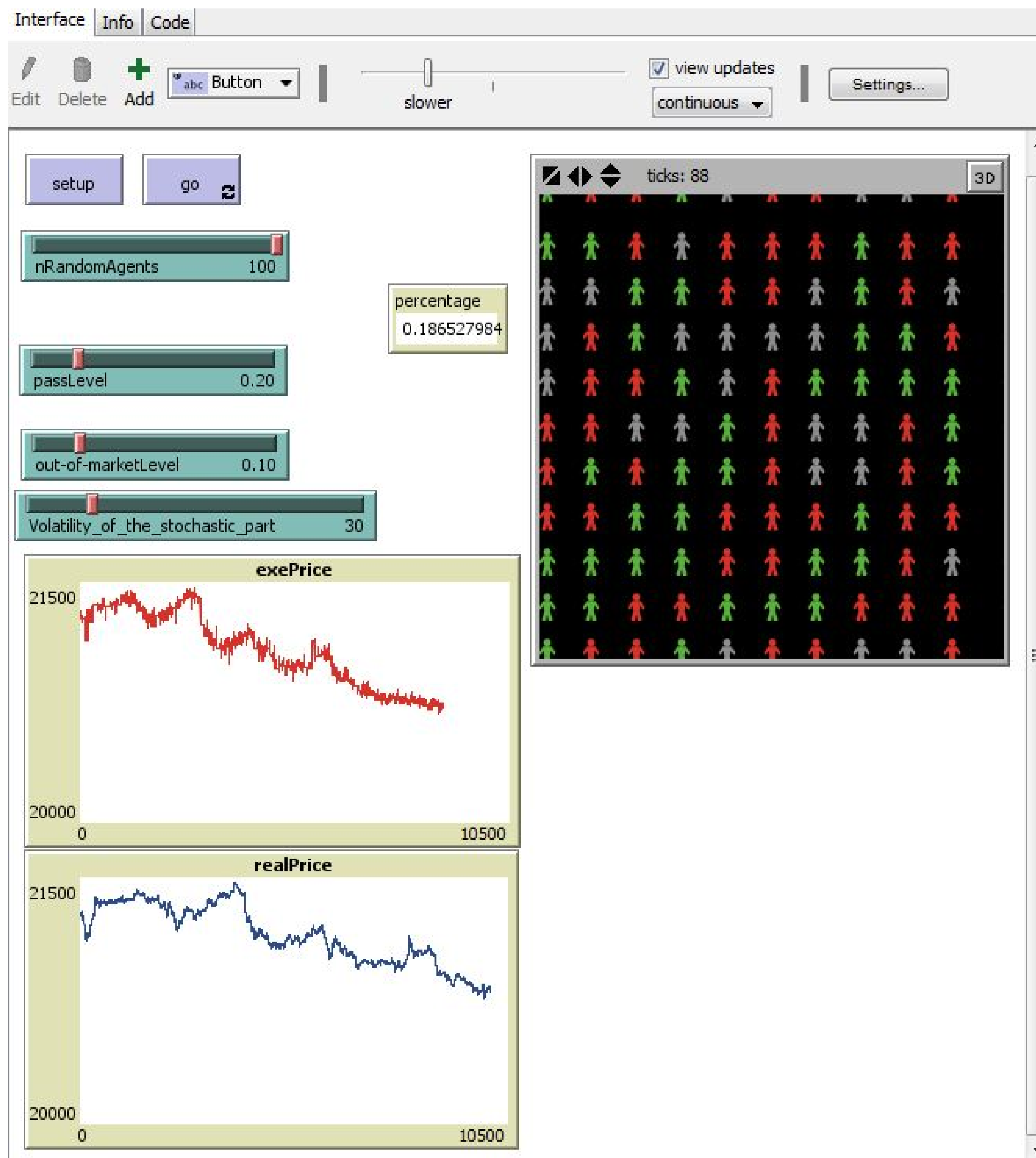

Figure 2.2: *Artificial_ VS_ real_ mkt* interface: press the go button, wait until tick 88, then stop the artificial market.

units, while at max it is about sixty units per-price; it means a difference of about 0.0025 percent in the best situation, and a difference of about 0.0030 percent in the worst situation. So in this framework it seems not too relevant the value assumed for the sliders.

Following a rule of reason, I have chosen as custom parameters:

- Deterministic part of the price (variable `price`) equal to 21300, that is close to the mean of the 10095 real prices considered, related to the *Ftse All Share* stock prices with frequency one minute.
- The stochastic part of the price, governed by the slider

  `volatility_of_the_stochastic_part`, is set equal to 30. (changes in this value modify the appearance of the artificial market,however the distance with respect to the real market remains negligible).
- `passLevel` is set equal to 0.20.
- I am not interested in the possibility that the `RandomAgents` have to be out of market; indeed I will not consider the slider `out_of_marketLevel` in the analyses.

We can see that the artificial price trend (the red one), generated by the combination of both the agents categories `RandomAgent` and `arbitrageur`, is very similar to the real one (the blue one); while the percentage of direct arbitrageur's intervention in the market is quite small, since it is about 19 percent (note that the variable `percentage` is proportional to the number of agents involved in the market, and to the variability of each RandomAgents' price, that is governed by the value of the slider `Volatility_of_the_stochastic_part`) .

What happens to the artificial market when I press the `go` button? In practice the `arbitrageur` agent forces the artificial market to replicate the real one, in a more or less consistent way (depending most of all on randomness, and in part on the value of the sliders).

A deeper comparison of the two typologies of markets ca be seen in Figure 2.3.

In this framework, my work is focused on the study of how the inclusion of other categories of trading agents can influence the artificial market trend so generated; these categories are the ones described in Section 1.

The objective of my thesis is to identify the optimal parameters (if any), characterising each typology of agents (the sliders), that minimize the distance between the artificial and the real market.

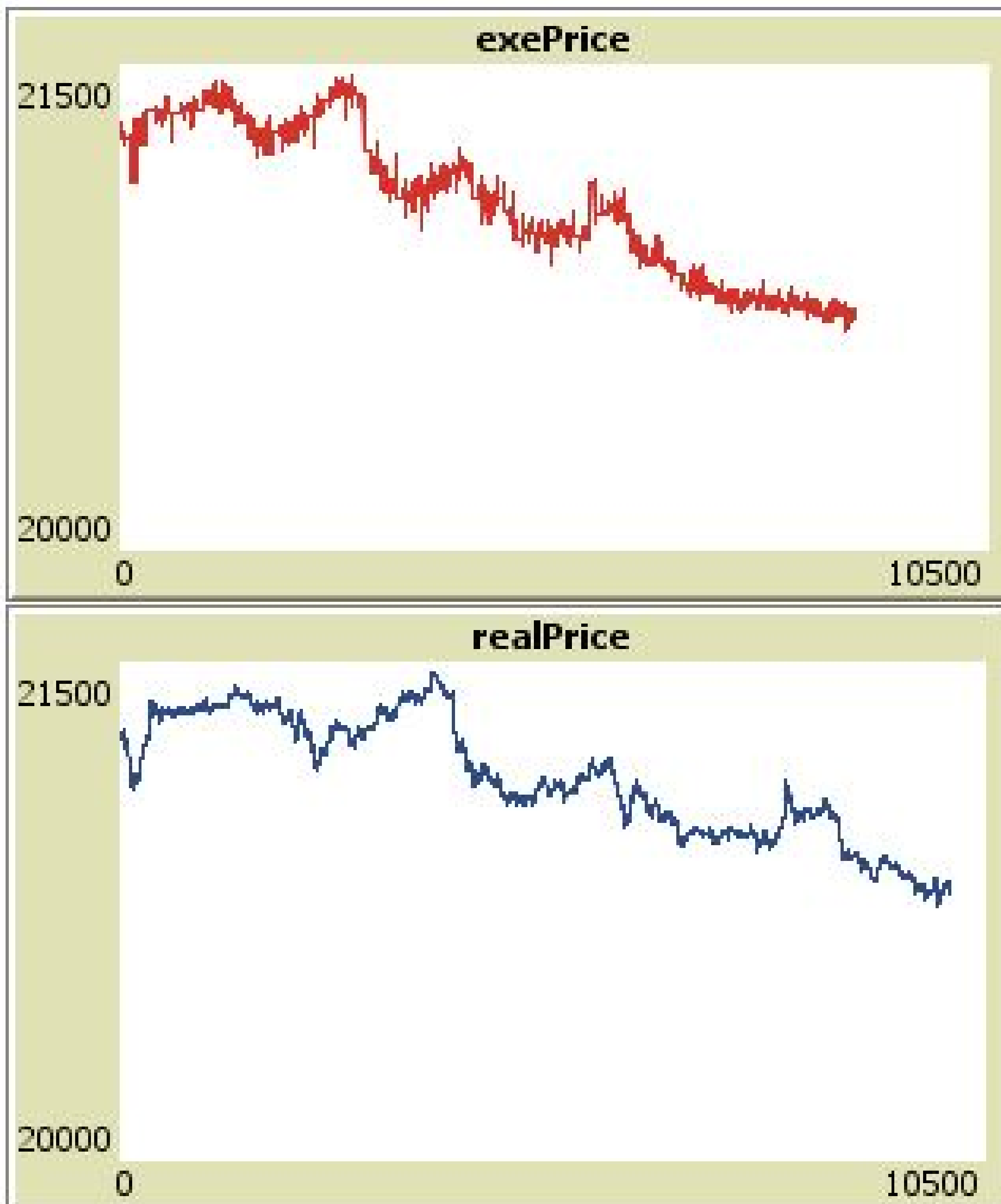

Figure 2.3: Detail of artificial (red line) and real (blue line) market, in the *NetLogo* interface.

It means optimize the agents behavior: in particular I will look for different optimal agents parameters, related to various frameworks, characterised by some market structures.

Such kind of analysis (Section 5) can be performed using *BehaviorSearch* software tool (deeply explained in Chapter) that exploits *genetic algorithm* to find the optimal solution.

Then the last step that will conclude my thesis will contain the implementation of some statistical tests, in order to compare the optimal values of the trading strategies obtained with *BehaviorSearch*.

Briefly next sections contain:

1. definition of the basic market structures: starting from the framework with one-hundred `RandomAgents` and one `arbitrageur`, the effect of inserting each trading agents category is analysed individually;

2. definition of more complex market structures: in order to study the aggregate effect on the artificial market of multiple categories of agents;

3. I will report the most significant simulations, (obtained pressing the `go` button) to see the appearance of the different market structures (*NetLogo* plots);

4. I will show the *BehaviorSearch* analyses and the most interesting results, related to the market structures defined: the fitness is the distance between artificial and real markets, and it must be minimized;

5. conclusions on the analyses implemented: comments will refer to the statistical tests of ANOVA, and to some regressions on the data collected with *BehaviorSearch*.

Note that the steps above are focused on the study of the effects of the individual breed in the market; it is simpler to investigate this behavior case by case, instead of considering all agents parameters as variables in each *BehaviorSearch* trial.

So the approach of my thesis starts from the particular to approach the much complex step by step.

The evaluation of the difference between the artificial and the real markets is obtained by inserting the following *NetLogo* command, just after the definition of the variable `exeprice` in the `go` procedure:

```
if j < 10095 [ set artificialVSreal artificialVSreal
        + (abs (exeprice - item 0 realPvec1)) / 10095]
```

The variable `artificialVSreal` memorizes the average distance between the two markets, in absolute value.

2.3 Market structures

After the definition of the basic framework, the next step is to develop different market structures in order to analyse separately the influence that each category of agent have on the artificial market showed in the previous section.

For this purpose I start with the simpler market structures, composed by the basic framework plus one category of intelligent trading agent, such that it can be easy to analyse the behavior of each breed one at a time.

Moreover, Section 2.3.2 contains two more complex market structures, obtained by adding to the basic framework more than one category of trading agents: in this way should be possible to study the aggregate effect on the artificial market of such breeds.

2.3.1 Market structures: basic framework plus one trading agents breed

First of all I define five new market structures, obtained by adding each category of trading agent explained in Sections 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7 to the basic framework.

To be consistent with reality, for each market structure, the number of trading agents inserted in the basic framework (it is always represented by a slider) is included in the interval [0 , 5]; it means that the percentage of trading agents with respect to `RandomAgents` cannot be more than five percent.

The new markets so created, are implemented through the following programs:

1. *AV.nlogo*: it is created by inserting the volume agent category (Section 2.1.4) in the basic framework; it means that we are considering three typologies of trading agents, i.e. `RandomAgents`, `arbitrageur`, `volumeAgents`. However in the proper analysis I will study only the variables affecting the behavior of `volumeAgents`, in order to find the optimal values of their sliders, with *BehaviorSearch*.

   Remember that `volumeAgents` participate to the auction mechanism generated by the `RandomAgents`, depending on buying and selling volumes. It means that when they establish to enter in the market, they decide to become buyers if there are more buyer agents than seller agents; while they decide to become sellers, if there are more seller agents than buyer agents; as

explained in Section 2.1.1, `RandomAgents` become buyers or sellers depending on randomness, in fact they participate to the market with probability 0.8, then they have probability 0.5 to either buy or sell.

Figure 2.4 shows the *NetLogo* interface of *AV.nlogo* program, after the implementation of the *setup* procedure.

The slider `VolumeAgentStep` indicates the magnitude of the step between successive interventions in the market auction mechanism, produced by `Volumeagents` ; if the value of the slider is equal to one, the trading agents considered will participate to the market at each new price formation.

2. *AT.nlogo*: it is created by inserting the trend agent category (Section 2.1.3) in the basic framework; so this market structure is composed by the following agents: one-hundred `RandomAgents`, one `arbitrageur`, a certain number of `trendAgents` (according to the slider `nTrendAgents`). As explained in Section 2.1.3, `trendAgents` decide whether to buy or sell if the current market price (`exeprice`) is respectively under or above the value of a moving average; the moving average considered in each evaluation is different for any `trendAgent`.

   Figure 2.5 shows the *NetLogo* interface of the *AT.nlogo* program, after the implementation of the `setup` procedure.

   The slider `StartingMA` defines the fixed number that is fundamental in the creation of a different moving average for each trading agent considered: in the `setup` procedure any `trendAgents` is initialised with a different moving average, calculated as the sum of the number selected for the slider `StartingMA`, and a stochastic number that is an integer in the interval [0 , 40] .

   I have added this slider, because this can lead to interesting results in term of difference between artificial and real markets, as a function of the width of the sample used for the moving average considered (I have chosen as custom value for this slider the number fifty as shown in Figure 2.5).

3. *AB.nlogo*: the current program can be obtained by adding the Bollinger Bands agents category (`BBAgents`) to the basic framework discussed in Section 2.2 . This typology of agents, explained in Section 2.1.7, will decide to

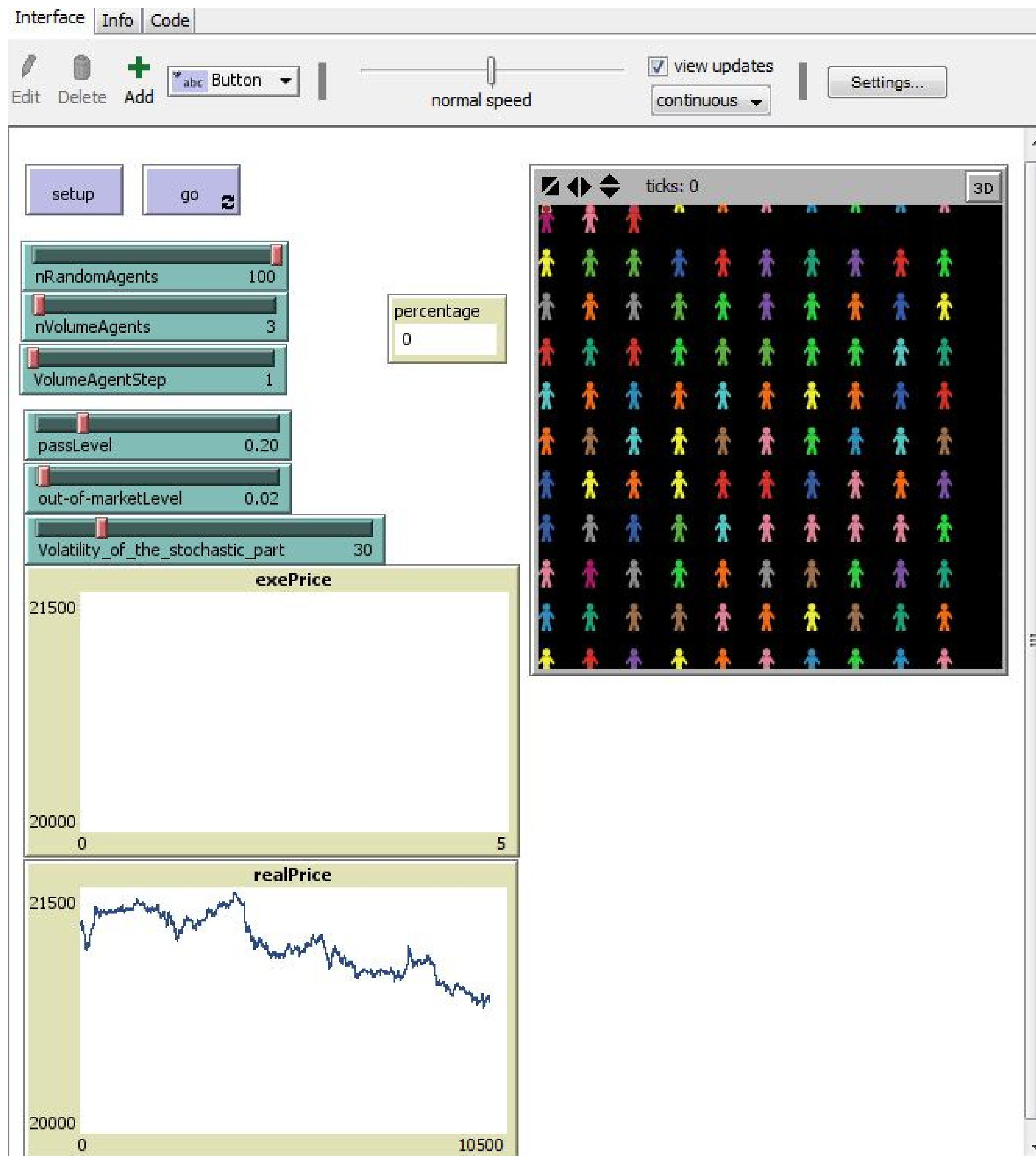

Figure 2.4: *NetLogo* interface of the program *AV.nlogo*.

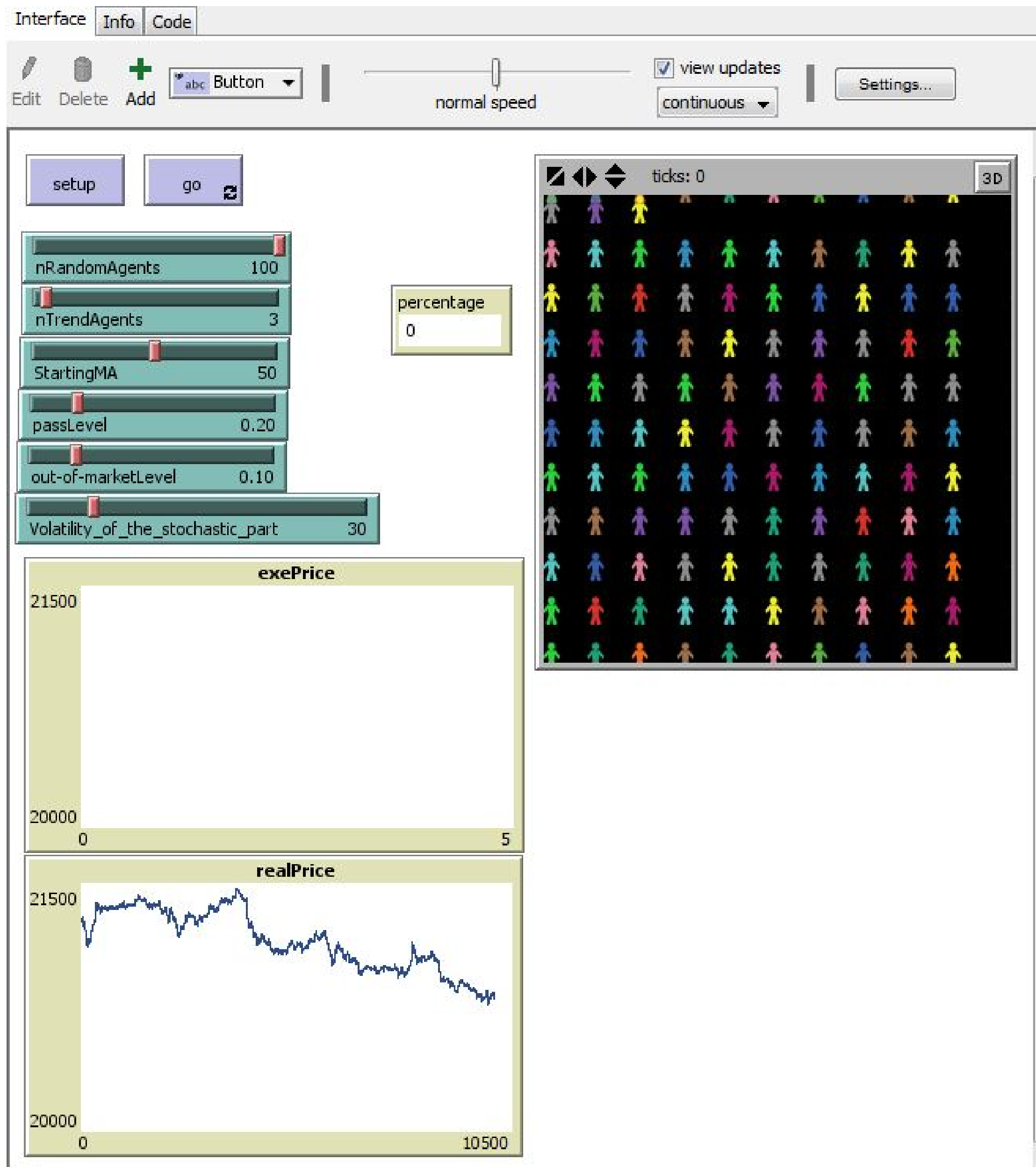

Figure 2.5: *NetLogo* interface of the program *AT.nlogo*.

buy or sell in the market, depending on support and resistance levels defined in the *John Bollinger*'s method.

The method requires the calculation of two statistical measures: a moving average (that in this case is equal for all `BBAgents`, while it was different for any `trendAgent`) and a standard deviation (calculated on the market prices and on the moving average). The Bollinger Bands are built by summing and subtracting to the moving average, the standard deviation multiplied by a constant (upper and lower band).

Then suppose that time `t` corresponds to the present: if the market price at time `t - 1` is less than the corresponding value of the lower band at the same time (under the lower band), and at time `t` it is greater than the corresponding value of the lower band (above the lower band), `BBAgents` will become buyers; vice-versa if the market price at time `t - 1` is above the upper band, and at time `t` it is under the upper band.

Figure 2.6 shows the *NetLogo* interface of the *AB.nlogo* program.

The sliders that directly influence the *Bollinger Bands* trading strategy are:

- `nMovingAverage` : represents the magnitude of the sample considered for the moving average calculation (it is equal for all `BBAgents`)
- `Bandwidth` : it is the constant, that multiplied by the standard deviation, is equal to one half of the total width of the bands: in fact by summing this value to the moving average we get the value of the upper band, while by subtracting it to the moving average we get the value of the lower band.

The plot named `BollingerBands` shows four variables together: the market price(`exeprice`), the lower (`LB`) and the upper band (`UB`), the moving average (`MA`)

4. *ASL.nlogo*: this program, (as the previous ones) consists of three categories of af agents: `RandomAgents` , `arbitrageur` , and `SLAgents` (explained in Section 2.1.5).

   The behavior of `SLAgents` in the artificial market built with the basic framework conditions, follow a *Stop-Loss strategy*: it requires the calculation of the option price, obtained through the *Black and Scholes* formula;

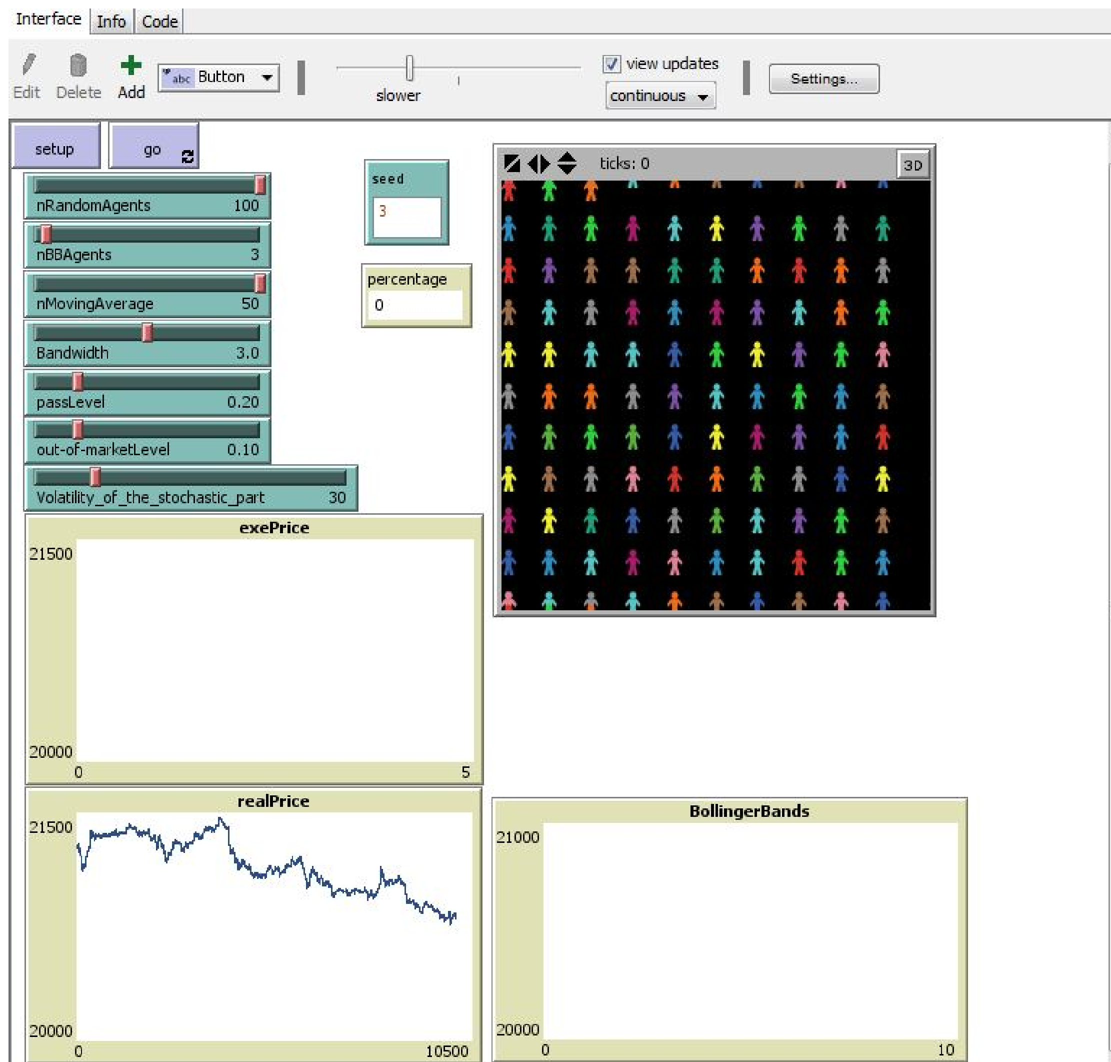

Figure 2.6: *NetLogo* interface of the program *AB.nlogo*.

Those agents can hold two possible positions depending on the market price trend: *naked* and *covered*.

In the *naked* position the investor is short a call. It produces a profit equal to the call price in t = 0 if the call is OTM (out-of-money) at the expiry date, but leads to significant losses when the call expires ITM (in-the-money)

In the *covered* position the investor is short a call and long the underlying (assumed to be bought at K, the strike price). It produces a profit equal to the call price in t = 0 if the call expires ITM, but leads to significant losses when it expires OTM.

The *NetLogo* interface of the program *ASL.nlogo*, after the implementation of the `setup` procedure, can be seen in Figure .

The slides that control the behavior of `SLAgents` are two.

- `risk-free` : it represents the risk-less rate of return in the market; its value is exogenous with respect to the artificial market created through the auction mechanism. However it is necessary to calculate the value of the option with the *Black and Scoles* formula.
- `T` : it is the step between successive checks of the strategy; it is measured in trading days, in which one trading day is assumed to generate about 700 market prices (frequency one minute).

The plot named `StopLossPortfolio` memorizes the cash flows owned by `SLAgents`.

5. *AC.nlogo*: the program combines the basic framework with the `CoveredAgents` breed, explained in Section 2.1.6 ; indeed also in the market structure so created, we have three categories of agents in total.

   Covered agents invest in the market through a strategy that both uses options and trusts in *technical analysis.*

   First of all this breed is programmed to identify two different circumstances: an increasing market if the current artificial price is greater than a simple moving average (that is an internal variable specific of the breed), a decreasing market in the opposite situation; those circumstances are covered by selling an out-of-money call in the first case, and selling an out-of-the-money put in the second case.

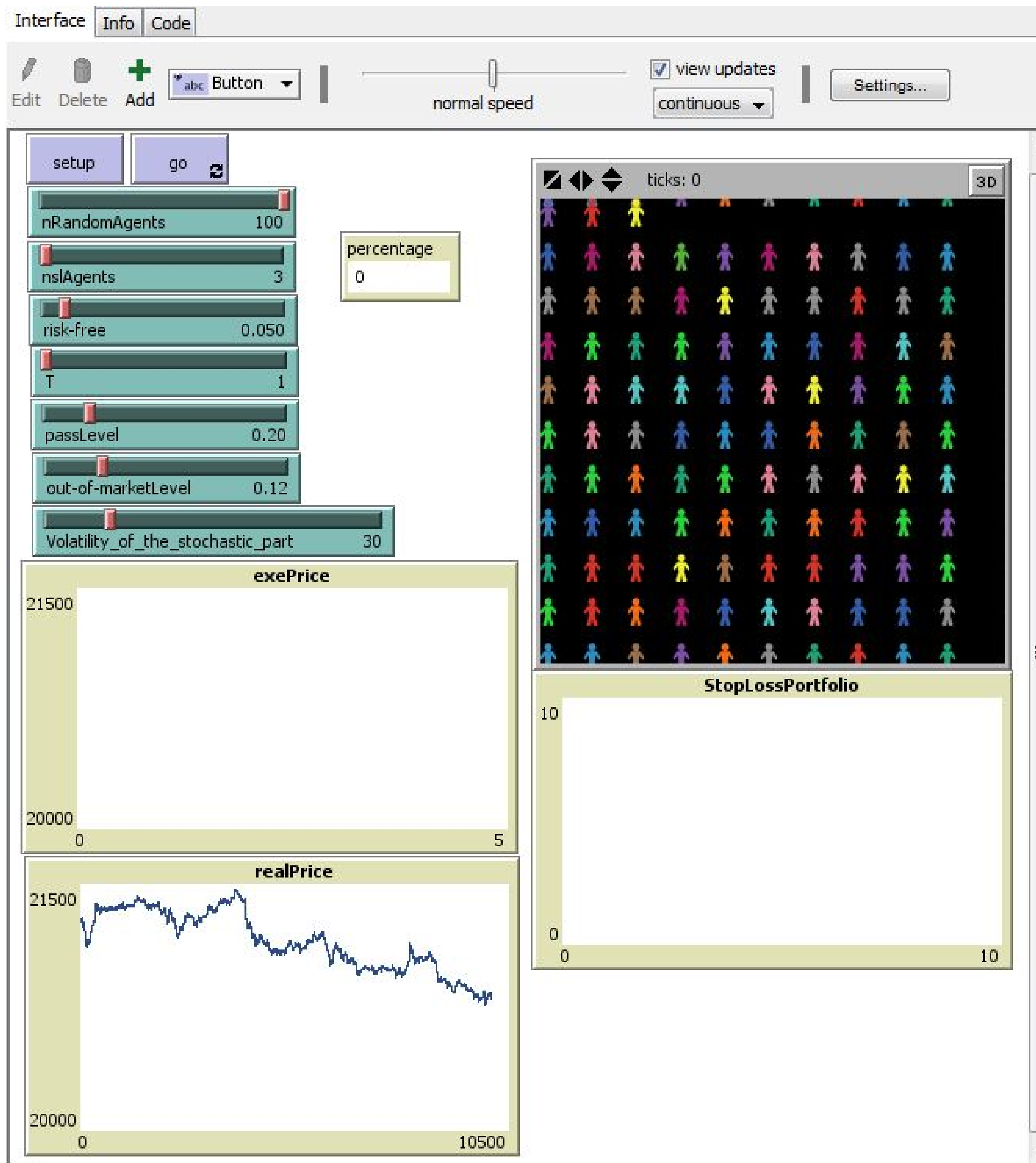

Figure 2.7: *NetLogo* interface of the program *ASL.nlogo*.

The strategy ends at the maturity date of the options, producing cash flows and market investments, due to the rights of such options (see section 1.6).

The *NetLogo* interface of *AC.nlogo*, after the initialisation of all agents variables (pressing the `setup` button), is shown in Figure 2.8.

The behavior of `CoveredAgents` is affected by the following sliders, that are breed specific.

- `nCoveredAgents` defines the number of covered agents that are created in the market through the `setup` procedure (there is a slider like this, for any category of trading agents).
- `risk-free` : it works as for `SLAgents` (see point 4 of this Section) ; it defines the risk-less rate, that is one of the input required for the calculation of the B-S formula.
- `Tc` : this slider affects the expire date of each option that will be trade in the market, during the implementation of the `go` procedure. In fact, this maturity date is driven by both `Tc` and the variable `expireDate`, through the formula

  maturity = `expireDate` * `Tc` ;

  while `Tc` ranges from one to fifty with step one, `expireDate` is a random number, that is different for any `CoveredAgent` (it is an integer random number, included in the interval [11 , 60]).
- `sampleStoch` influences the magnitude of the sample (variable `samplec`) considered for the calculation of the simple moving average (necessary to start the `CoveredAgents` strategy, i.e. to identify an increasing or decreasing market).

  This sample is computed for each covered agent in the `setup` procedure; its formula is :

  sample = 100 - random number in the interval [1 , `sampleStoch`]

  So if I set a bigger `sampleStoch`, the probability of getting a lower sample for the moving average increases, and vice-versa.

  Moreover, this slider affects the starting time of the trading strategy, since each `CoveredAgent` will calculate for the first time the B-S price , when the first value of the moving average is available.

The monitor `sigma` indicates the value of the standard deviation of logarithmic returns, calculated on the past `sample` realisations.

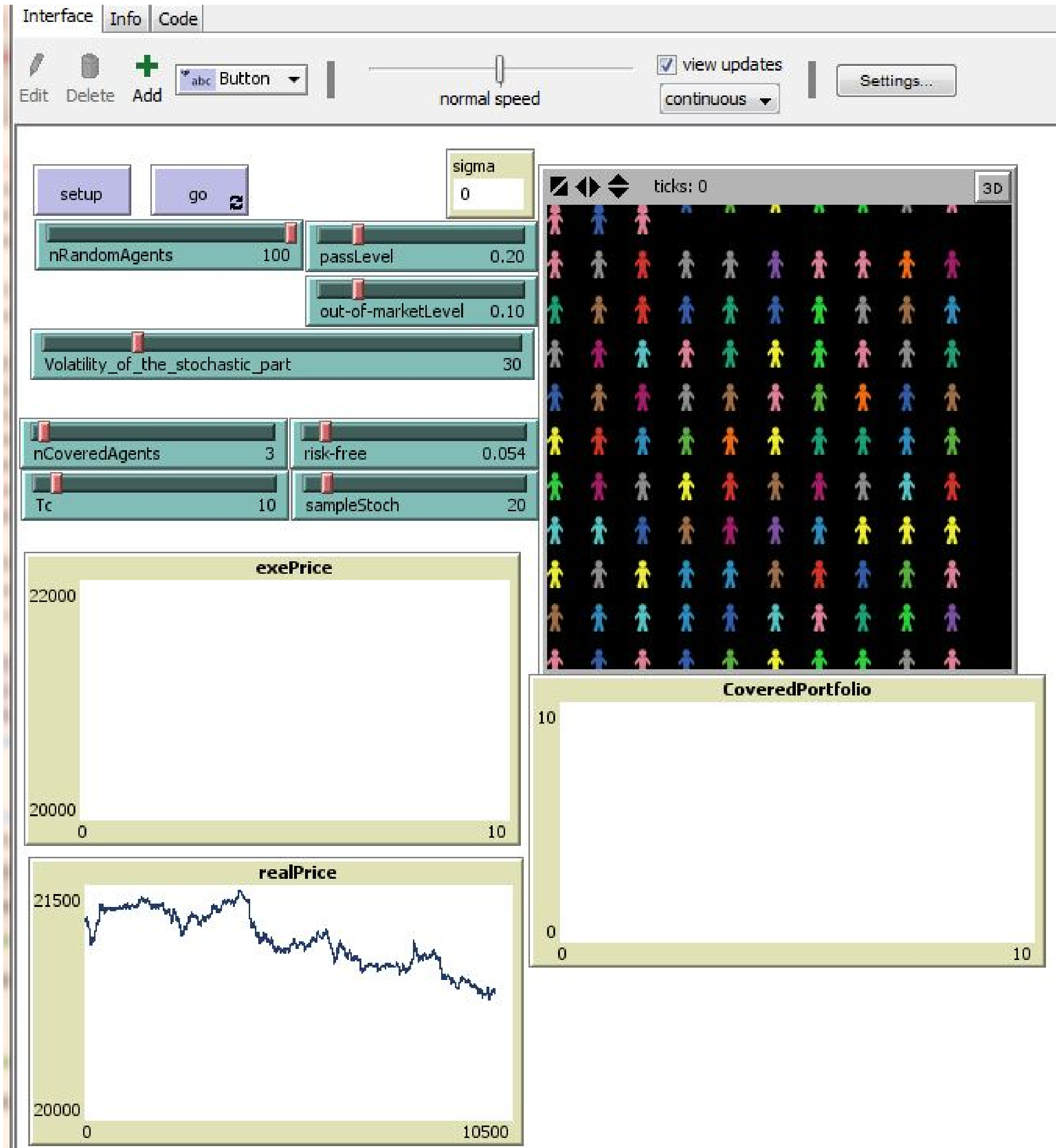

Figure 2.8: *NetLogo* interface of the program *AC.nlogo*.

Instead, the plot `CoveredPortfolio` memorizes the cash flows of all `CoveredAgents` ; in this way it is possible to check the profitability of their strategy.

2.3.2 Market structures: basic framework plus more than one trading agents breed

Up until now, in this section we spoke about market structures with only one class of trading agents operating in the basic framework defined in section 2.1 .

It can be interesting to study the aggregate effect of different categories of agents in the same (basic) framework; for that reason in this section I introduce three different market structures, that are more complex with respect to those explained in Section 2.3 .

The first market structure so defined, comprehends the agents that base their behavior on technical analysis: `trendAgents`, and `BBAgents`.

The second market structure adds to the basic framework, the two categories of agents that trade options, to cover their investments in the underlying, i.e. the artificial market price.
These breeds are `SLAgents` and`CoveredAgents`.

The third market structure, comprehends all trading agents defined so far: `volumeAgents`, `trendAgents`, `BBAgents`, `SLAgents`.

The *NetLogo* programs related to these three market structures are:

1. *ABT.nlogo*: this program is characterised by the following breeds;
   (a) `RandomAgents` : they are fundamental for the creation of the market price. their behavior is explained in Section 2.1.1 .
   (b) `arbitrageur` : it is necessary to drive the artificial market price near a real market price (*Ftse All Share* stock). This breed together with `RandomAgents` forms the basic framework.
   (c) `trendAgents` : they base their trading strategy on the value of a simple moving average, that is different for any `trendAgent`.
   (d) `BBAgents` : this class decide to trade depending on the value of the current market price with respect to the *Bollinger Bands* : these bands require the calculation of a moving average, that is fixed for any `BBAgent`, and the standard deviation calculated on the moving average.

   The *NetLogo* interface showed in Figure 2.8 , contains all the sliders that characterise the breeds mentioned.

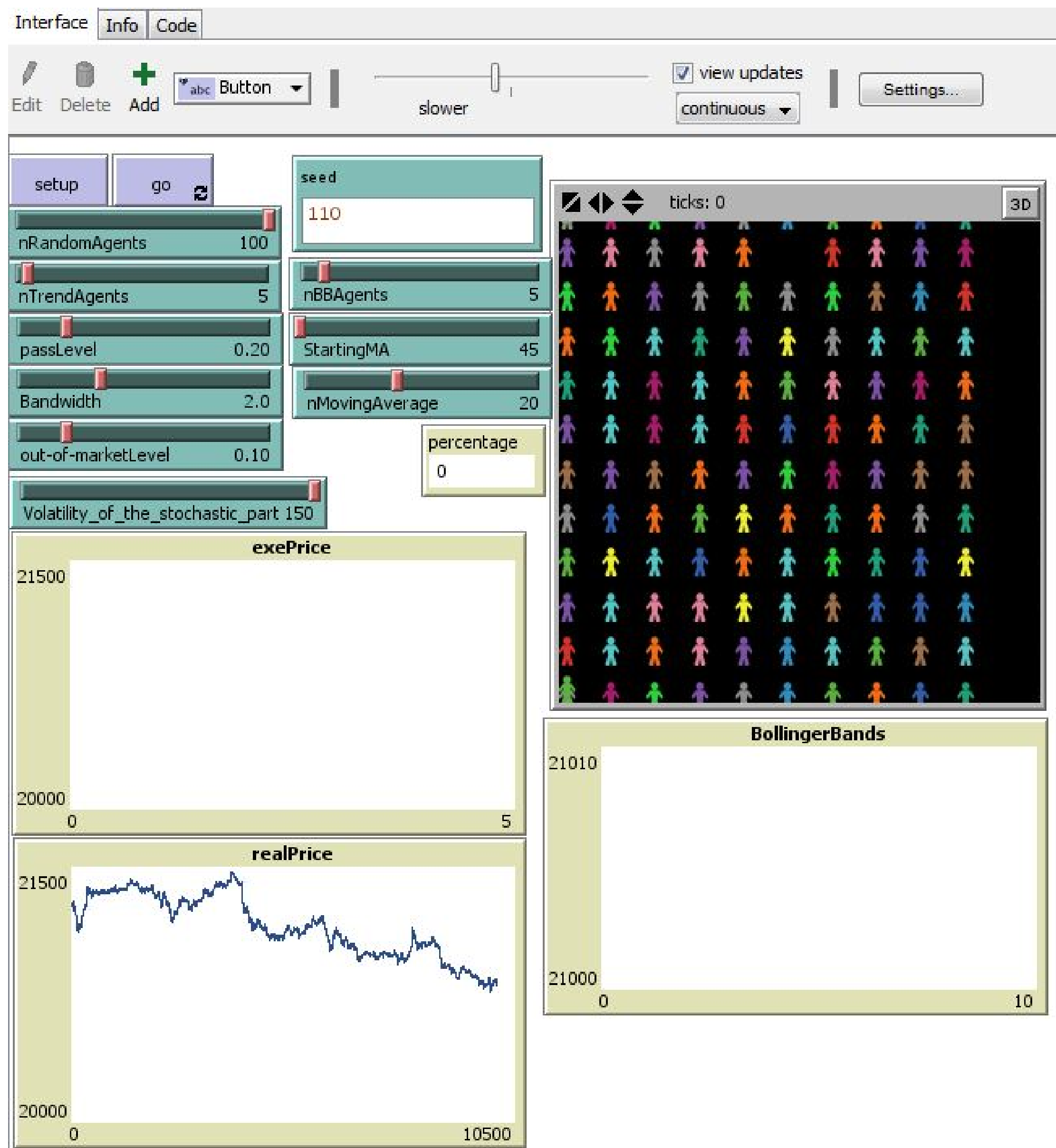

Figure 2.9: *NetLogo* interface of the program *ABT.nlogo*.

2. *A-C-SL.nlogo*: overall the program consist of the following agents breeds.

   (a) `RandomAgents` : they are the main vehicle of the auction mechanism; their investments are driven by randomness.

   Considered alone in the market, this breed is able to generate a realistic price trend. For this reason their number should be much larger than that of other agents.

   (b) `arbitrageur` : it drives the artificial market price, near the values of a real price (*Ftse All Share* stock), whose values are red from a file, and are characterized by a high frequency (one minute).

   (c) `SLAgents` : they invest in the artificial market, through the identification of a *naked*, and a *covered* positions, both characterized by a call option trade.

   For that reason, this breed requires the calculation of the call option price using the *Black and Scholes* formula.

   (d) `CoveredAgents` : they have a mixture of the capabilities of `trendAgents` and `SLAgents`. They can evaluate the price level with respect to a moving average, identifying an increasing or decreasing market situation; according to these two circumstances they can trade an OTM call, or an OTM put respectively.

   Then, also `CoveredAgents` are able to compute the option price using the *Black and Scholes* formula.

   The *NetLogo* interface of the program *A-C-SL*, containing all the sliders characterising the four breeds discussed above, can be seen in Figure 2.10 .

   Note that the risk-less rate , should be unique in the market; for this reason the corresponding slider, `risk-free`, is used by both `SLAgents` and `CoveredAgents`.

3. *A-B-C-T-SL-V.nlogo*: this program contains all breeds defined so far (described in Sections 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7).

   Briefly they are:

   (a) `RandomAgents` : they decide to buy or sell depending on randomness.

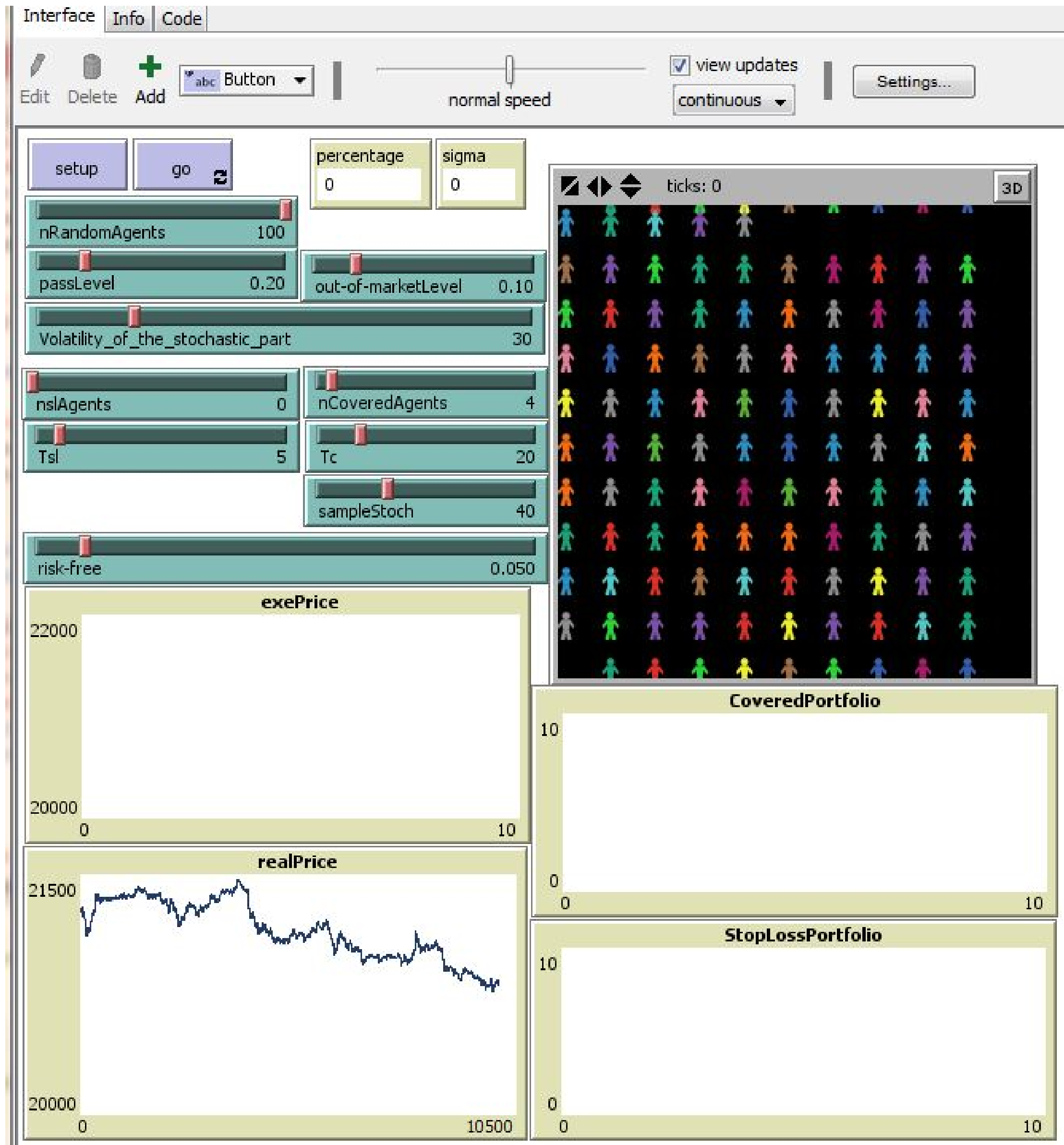

Figure 2.10: *NetLogo* interface of the program *A-C-SL.nlogo*.

(b) `arbitrageur` : he participates to the auction mechanism for the formation of the market price, depending on the values of *Ftse All Share* stock.

(c) `trendAgents` : they base their trading strategy on the value of a moving average, that is different for any `trendAgent`.

(d) `BBAgents` : they identify support and resistance levels depending on the value of the market price with respect to *Bollinger Bands* (explained in Section 2.1.7).

(e) `VolumeAgents` : they participate to the market depending on trading volumes.

(f) `SLAgents` : their investments in the market are driven by a *Stop Loss* strategy (explained in Section 2.1.5), that requires the calculation of the value of a call option, obtained through the *Black and Scholes* formula.

(g) `CoveredAgents` : they base their trading strategy on both : the value of a moving average, different for any `CoveredAgents`, and the trade of call and put options, calculated through the *Black and Scholes* formula.

The *NetLogo* interface is showed in Figure 2.9 .

2.3.3 Agents effect on the market

The core structure of the *NetLogo* program that defines the auction mechanism (of *g1_CDA_basic_model* explained in Section 2.1.1), ensures a realistic price trend that comprehends bull an bear situations.

In general the terms bull market and bear market describe upward and downward market trends, respectively, and can be used to describe either the market as a whole or specific sectors and securities.

In particular

- A bull market is associated with increasing investor confidence, and increased investing in anticipation of future price increases : in our framework it happens if there are more buyers than sellers.

- A bear market is a general decline in the stock market over a period of time. It is a transition from high investor optimism to widespread investor fear and pessimism : in our framework it happens if there are more sellers than buyers.

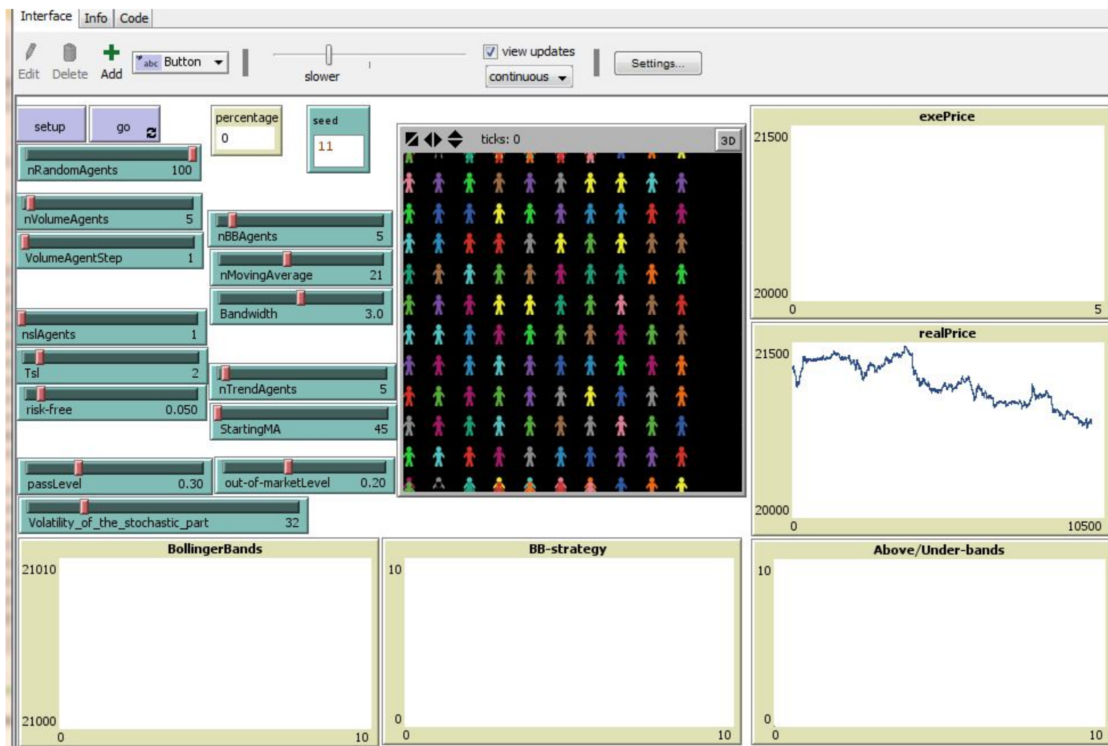

Figure 2.11: *NetLogo* interface of the program *A-B-SL-T-V.nlogo*.

The framework created in *g1_CDA_basic_model* allows also the verification of *economic bubbles*.

An economic bubble (in our case price bubble) is defined as a 'trade in high volumes at prices that are considerably at variance with intrinsic values '.

Bubbles are often conclusively identified only in retrospect, when a sudden drop in prices appears. Such a drop is known as a crash or a bubble burst.

Both the boom and the burst phases of the bubble are examples of a positive feedback mechanism, in contrast to the negative feedback mechanism that determines the equilibrium price under normal market circumstances. Prices in an economic bubble can fluctuate erratically, and become impossible to predict from supply and demand alone.

In our case bubbles can happen during extreme bull (bear) situations, that produces a big increase in the variance of buying (selling) prices (ordered in `logB` and `logS` vectors); moreover they are impossible to predict since they are linked to randomness.

Now we know that considering *g1_CDA_basic_model*, the situations described above are completely linked to the random behavior of `RandomAgents`.

But it can be different if we consider other breeds of intelligent agents, together with `RandomAgents`.

1. The clearest effect on the market is the one of `VolumeAgents` : they trade only when there is a difference between buying and selling volumes, enforcing the bigger one.

   If the value of the slider `nVolumeAgents` is big with respect to `nRandomAgents` (at least twenty percent), and the frequency of their trading decision is sufficiently high (every ten market prices realisations), this breed can often enforce bull and bear markets, easily producing bubbles.

   However in general, if the number of `VolumeAgents` is quite low with respect to the one of `RandomAgents`, their effect on the price trend is difficult to see, and the variable `diff` (that measures the difference between buying volumes and selling volumes) oscillates more frequently between positive and negative values over time (Figure 2.10).

   Figure 2.12 shows the behavior of the variable `diff` defined as

   `length logB - length logS`, in a market populated by 100 `RandomAgents` and 10 `VolumeAgents` ; moreover the trading frequency of `VolumeAgents` is one out of ten price realisations.

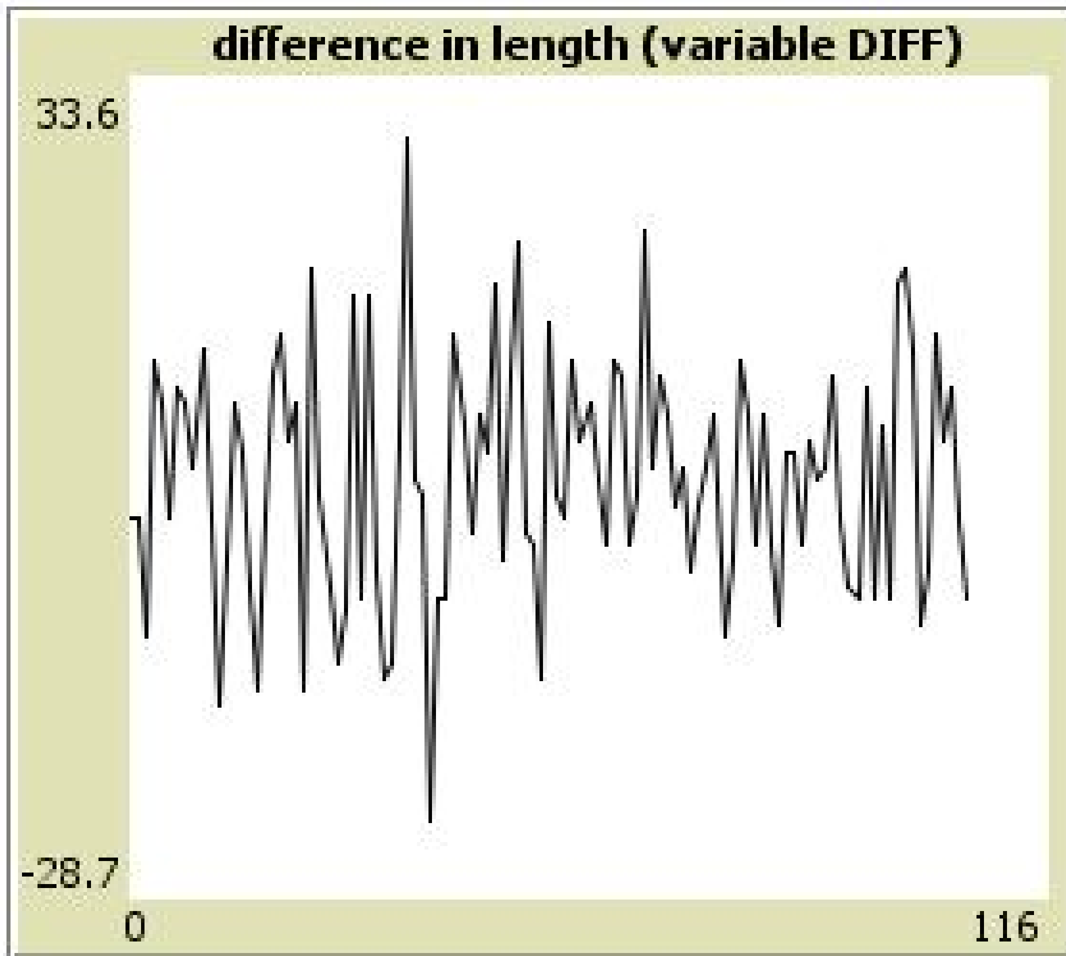

Figure 2.12: plot of the variable *diff* after one-hundred ticks.

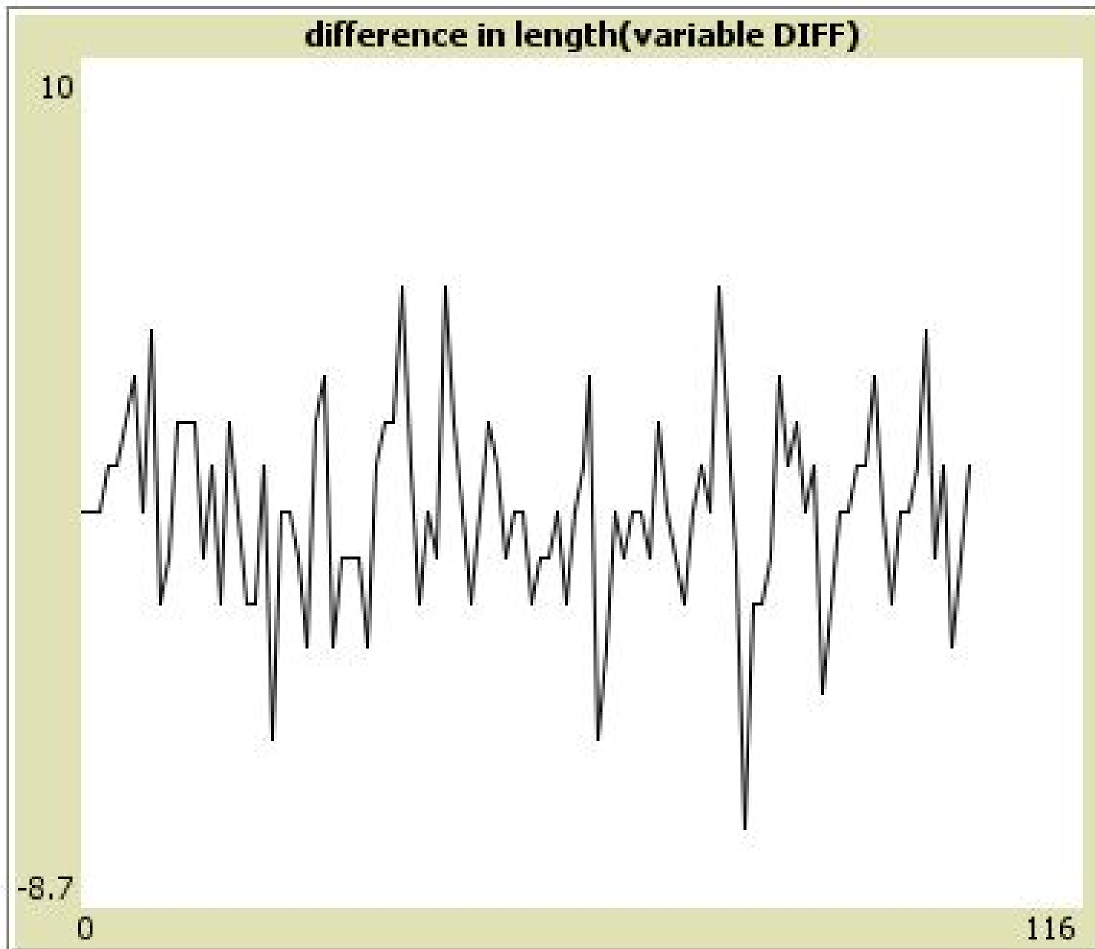

Figure 2.13: plot of the variable *diff* after one-hundred ticks.

Conversely, if we consider an extreme situation with only ten `RandomAgents` and fifty `VolumeAgents`, the trend of the variable `diff` results more smoothed, (and with smaller variance, since it is more difficult to have a market price reversal once the variable `diff` becomes positive or negative), as shown in Figure 2.11.

2. Since all `BBAgents` invest following the same moving average, the effect on the market of applying their Bollinger strategy can produce results very similar to those expected for `VolumeAgents` : they should drive the market according

to their overbought and oversold conditions pushing to bull markets in the first case and to bear markets in the second. However as happened for `VolumeAgents`, the magnitude of the slider `nBBAgents` strongly affects this phenomenon.

The difference with respect to `VolumeAgents` should be the fact that reversals in the market price are no more driven by volumes, but they are influenced by the position of the market price with respect to the bands (defined in Section 2.1.7).

However, the effect of `BBAgents` on the artificial price, is difficult to see in reasonable conditions, (i.e. magnitude of `nBBAgents` smaller or equal than five percent of the slider `nRandomAgents`) : for this purpose I will show two opposite situations.

(a) Market populated by one-hundred `RandomAgents` and one `BBAgent` : strong oscillatory trend, overbought and oversold situations cannot be seen (Figure 2.12).

(b) Market populated by only ten `RandomAgents` and fifty `BBAgents` : quite smoothed trend, with overbought (oversold) conditions corresponding to strong bull (bear) markets Figure 2.15.

.

Both market frameworks are evaluated considering: fixed part of market price equal to 21000, variable part of market price distributed as a Normal random variable with mean zero and variance thirty; `passLevel` equal to 0.2 ; seed equal to 2.

In my study on the behavior of trading agents I have opted for a moving average, that is the same for all `BBAgents`, in order to precisely analyse its value with *BehaviorSearch* (moreover in my analysis framework, I will consider only five `BBAgent` at most, and so their effect should not be too visible, without leading to extreme price variations).

3. For `SLAgents` the situation is different: their investment strategy depends mainly on a bet, performed through the identification of a strike, whose calculation is exogenous with respect to the variables that define the market.

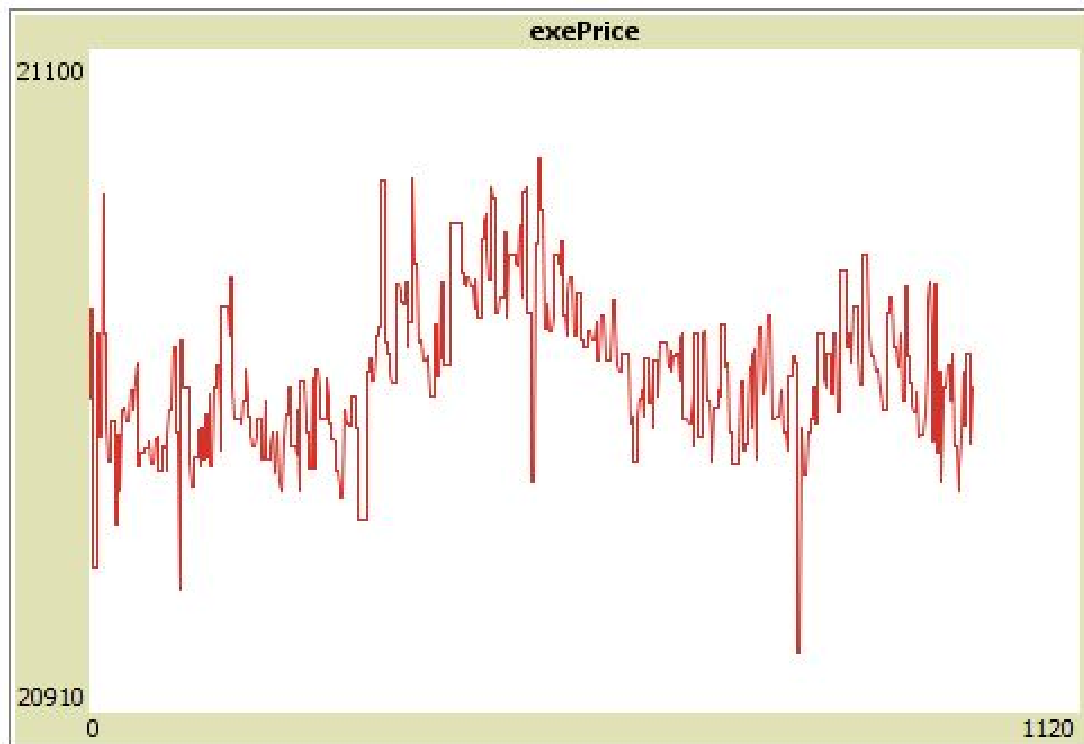

Figure 2.14: plot of the variable *exeprice*: market with 100 RandomAgents and one BBAgent.

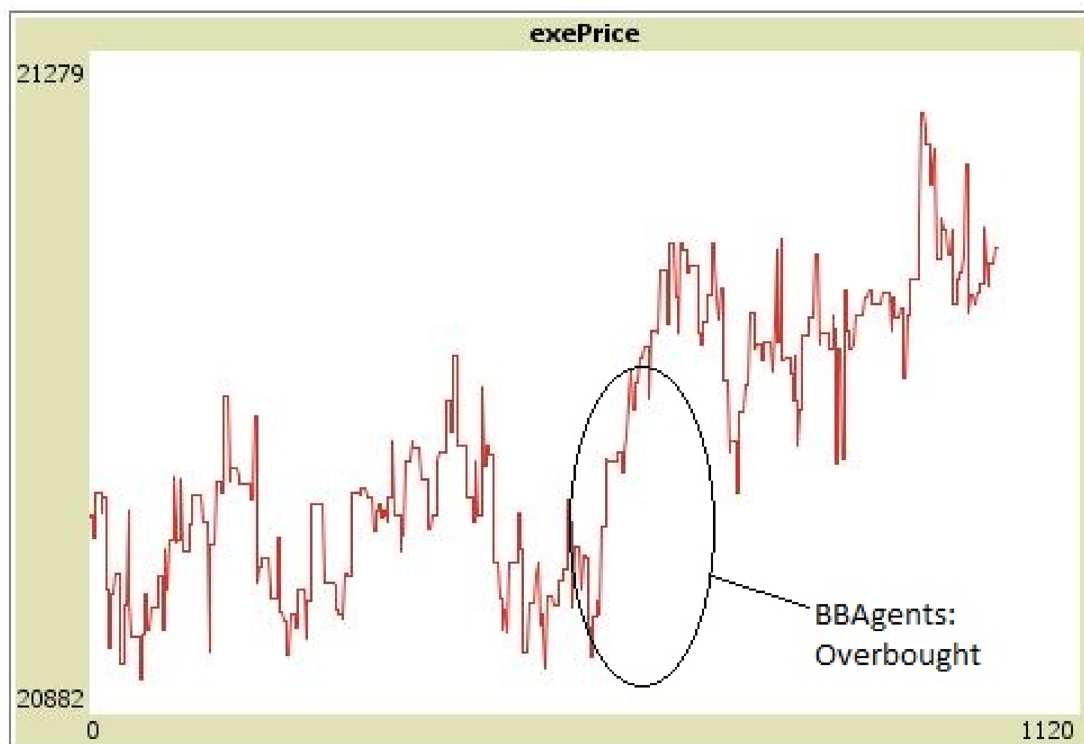

Figure 2.15: plot of the variable *exeprice*: market with 10 RandomAgents and 50 BBAgent.

Each `SLAgent` compares periodically the value of such strike, with the current market price; then at the expire date(after a certain step, whose possible set of values is defined in Section 2.1.5 point 2) according to `naked` and `covered` situations (described in section 2.1.5) he decides whether to be a buyer or a seller: in both cases he puts in the auction mechanism the strike, as possible ask or bid price (note that, contrary to overbought and oversold conditions for `BBAgents`, `naked` and `covered` positions may be different for any `SLAgent` at a given time, since such agents own different strikes).

For that reason the effect on the artificial market of such breed depends on the value of the strike, and it is not visible even in extreme situations, if the range of possible strike values is close to that of the market price.

However these considerations are made considering a market populated only by `RandomAgents` and `SLAgents`.

4. `trendAgents` invest according to the market price level, with respect to a simple moving average, that is calculated on a different sample for each `trenAgent`.

   If the market price is smaller than the value of the moving average the agent becomes a buyer and vice-versa.

   In a framework consisting of only the two breeds, `RandomAgents` and `trendAgents`, an excessive number of the last with respect to that of the first, leads to extreme situations very similar to those observed for `VolumeAgents` and `BBAgents`, but even more marked.

   If we build a market with ten `Randomagents` and fifty `trendAgents` we always obtain one of the following market trends:

   (a) Perpetual bull market: there are no artificial price reversals; all `trendAgents` become buyers (Figure 2.16). It is normal the fact that in a market composed by only sixty agents, if there are always (at any time) at least fifty buyers, the market price will continue to grow perpetually.

   (b) Perpetual bear market: there are no artificial price reversals; all `trendAgents` become sellers (Figure 2.17).Unsurprisingly, in a market composed by only sixty agents, if there are always at least fifty sellers, the market price will continue to decrease perpetually.

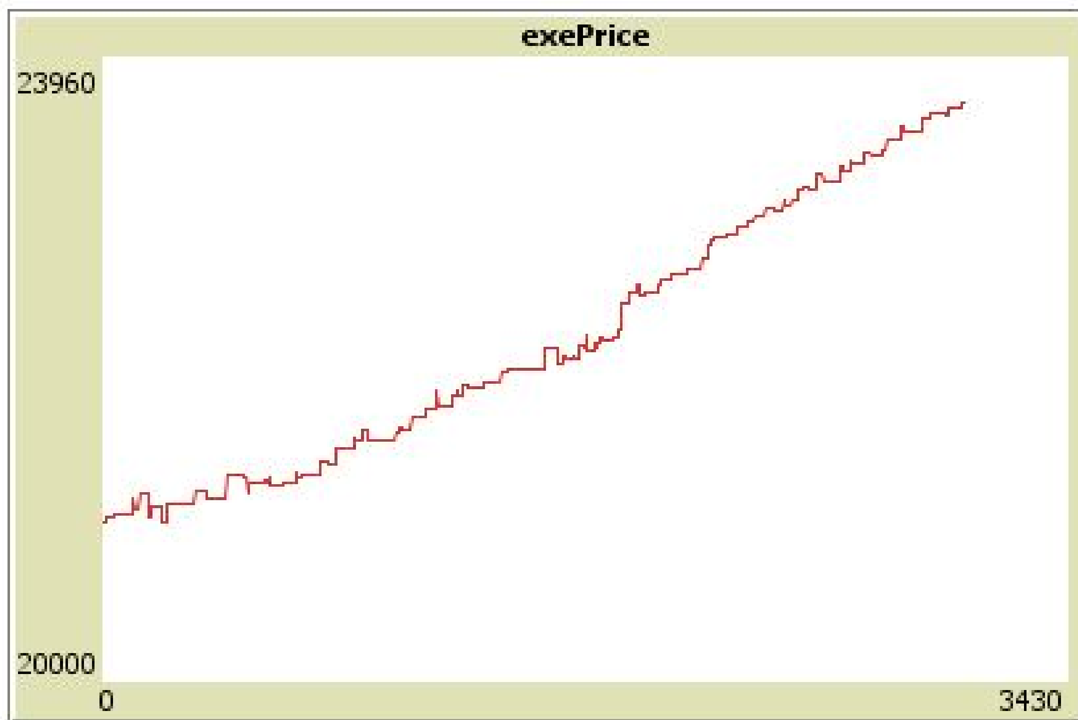

Figure 2.16: *trendAgents*, case(a) :plot of the variable *exeprice* after fifty ticks, seed = 3.

Figure 2.17: *trendAgents*, case(b) :plot of the variable *exeprice* after fifty ticks, seed = 7.

Figures 2.14, 2.15, are constructed using a price (owned by agents) with fixed part equal to 21000, changeable part distributed as a Normal random variable with mean zero and variance thirty; `passLevel` equal to 0.2 .

As happened for previous breeds, also for `trenAgents`, their effect on the market is not visible for reasonable values of the slider `nTrendAgents` with respect to the slider `nRandomAgents` (the ratio should be at least 1 / 20).

5. `CoveredAgents` effect on the market is very difficult to predict, since their strategy is rather complex. This breed starts its investment approach, by evaluating the market price trend, exactly as happen with `trendAgents` ; but then `CoveredAgents` do not become only buyers or sellers depending on the trend conditions checked (Section 2.1.6) : their investments are also driven by the sale of either an OTM call or an OTM put options.

   If we consider a market populated by only `RandomAgents` and `CoveredAgents`, it seems that this last category of agents leads to an increasing price trend, when it chooses to sell the call option, while it leads to a decreasing price trend when market conditions suggest to sell the put option (for a more detailed description of the behavior of these agents see Section 2.1.6).

   In this case, the number of 'intelligent' trading agents seems not as relevant as for previous breeds: even in small number (3-4), `CoveredAgents` can easily produce slightly bull or bear markets, that lasts for many ticks.

   Some examples of the effect on the artificial market, produced by those agents, can be seen in Figures 2.18, 2.19.

Figures 2.18, 2.19 represents the plot of the variable `exeprice` : the values of the sliders characterising `RandomAgents` and `CoveredAgents` are:

- `nRandomAgents` = 100
- `nCoveredAgents` = 4
- `pass-level` = 0.2
- `risk-free` = 0.05
- `Tc` = 20
- `sampleStoch` = 30

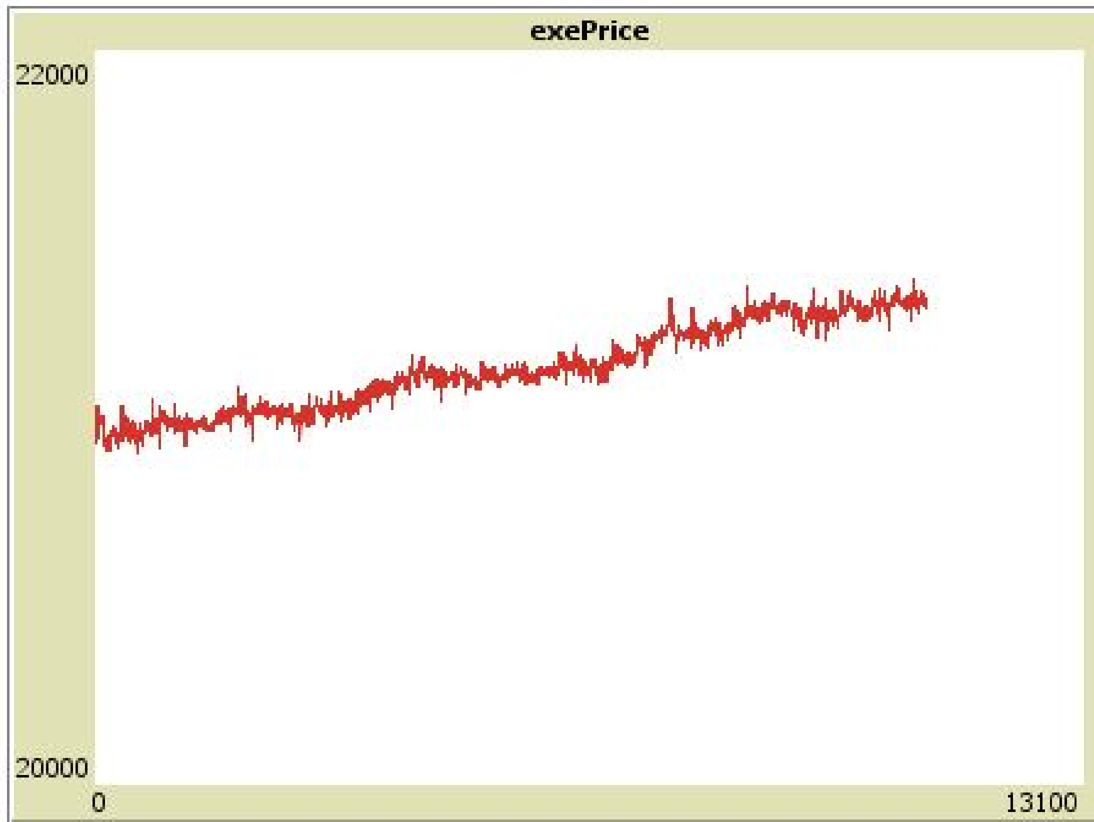

Figure 2.18: *CoveredAgents* : slightly bull market; plot of the variable *exeprice* after one-hundred ticks, seed = 4.

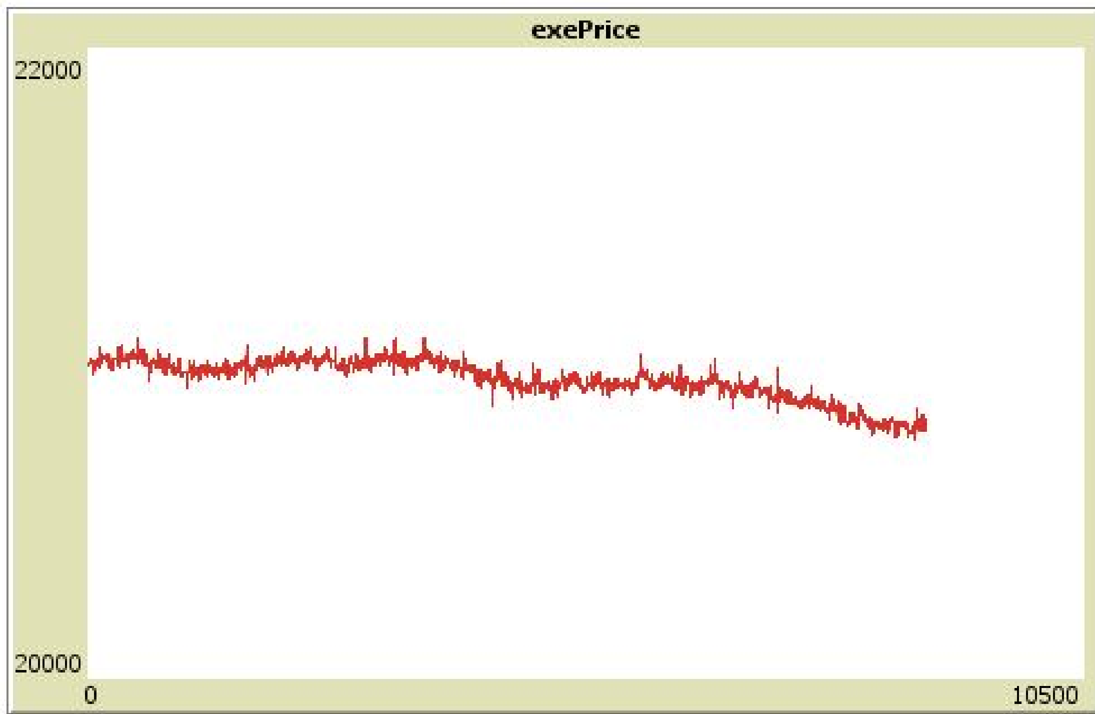

Figure 2.19: *CoveredAgents* : slightly bear market; plot of the variable *exeprice* after eighty-five ticks, seed = 9.

- `Volatility_of_the_stochastic_part` = 30

Differently from other breeds, the effects on the price trend, of changing the values of the sliders characterizing the `CoveredAgents` trading strategy (i.e. `Tc` and `sampleStoch`) , are more strong, and can lead to quite different price trends.

Moreover, implementing several trials, obtained by changing the value of the input `seed`, (and then pressing the `go` button), we can note that it is easier to have a smoothed bull market (sell the OTM call), instead of a smoothed bear market (sell the OTM the put); this is probably due to the fact that on average `CoveredAgents` are more inclined to buy than to sell, but it is not clear why.

One possible explanation can be the fact that in general the market price trend, generated by `RandomAgents` during the first ticks is often increasing, consequently forcing `CoveredAgents` to sell the call option and to buy the underlying.

In fact if we increase the sample of the moving average (decreasing the slider `sampleStoch`), and if we enlarge the maturities of the options (increasing the slider `Tc`), the price movements seem to become more random.

Note that this happens only in a framework in which the price trend is mainly driven by `RandomAgents` ; if we add the `arbitrageur` breed, we can do not care about that problem.

2.4 Simulations

The objective of this Section is to show some results, in terms of market price trend, obtained by implementing the `go` procedure of the programs created so far. They are:

- those explained in Section 2.3.1, i.e. characterised by a total of three breeds of agents: *AV.nlogo*, *AT.nlogo*, *AB.nlogo*, *ASL.nlogo*, *AC.nlogo*;
- those explained in Section 2.3.2, that are composed by more than three breeds of agents overall: *ABT.nlogo*, *A-C-SL.nlogo*, *A-B-SL-T-V.nlogo*.

For this purpose I have to define a set of custom values for the parameters (shared to all breeds) that affect the price formation (the fixed and variable parts of the price and the sliders value) : those custom values will work as a benchmark, in the major part of the further analyses (The possible range of values of such

sliders, will be discussed in detail in next Section.) . In this way it should be possible to exploit comparisons between similar frameworks.

These guidelines are the following:

1. As I have stressed in Sections 1 and 2, each breed of agents own a variable `price`, that corresponds to the price that they can put in the auction mechanism. This variable consists of two components: a fixed part, that is initialised in the `setup` procedure, and a variable part, that changes periodically in the `go` procedure. The fixed part is the same among agents, while the variable part is different for any agent (note that the definition of the price in such way is necessary for the functioning of the auction mechanism).

   Since my study is based on the comparison between an artificial market and a real one, (the first realigned to the last through an arbitrageur), I have to ensure a price formation consistent with the real price.

   Following this objective, all simulations consider a fixed part of price (initialised as `exeprice` in the `setup` procedure) equal to 21300 (that is close to the mean of the real price time series, that is about 21100); the variable part is set equal to a Normal random variable with mean zero and variance thirty (following a rule of thumb).

   Section 5.1 contains a detailed analysis with *BehaviorSearch* about the structure of the price used by each agent.

2. Since the program *g1_CDA_basic_model*, (explained in Section 1.1) creates a market with realistic price movements (Section 3.3), it is fundamental to maintain a great number of `RandomAgents` ,with respect to the number of the other breeds. In fact `RandomAgents` represent the main vehicle for the formation of the artificial market: for this reason their number (determined by the slider `nRandomAgents`) is kept fixed to one-hundred.

   While according to a rule of thumb, the number allowed for each other agent breed cannot be greater than the five percent of `nRandomAgents`. For the simulations of this Section, this number will be set equal exactly to five.

3. The slider `passLevel` (as stressed in Sections 2.1.1 and 2.3.1), represent the probability of `RandomAgents` of do not participate in the artificial price formation; this probability should be quite low, for the reasons explained in point 2. In next simulations the value of this parameter will be assumed fixed at 0.2 .

The value of the sliders, that are breed specific (discussed in Section 3) will be set up case by case.

2.4.1 Comparisons between artificial and real market : AV.nlogo

The program *AV.nlogo* has been discussed in section 2.3.1 (point 1) .

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `VolumeAgents` (5); this breed bases its trading strategy on the sliders:
    - `VolumeAgentStep` (Section 2.3.1 point 1). It determines the frequency of the strategy: it is kept fixed at 10 .

2.4.2 Comparisons between artificial and real market : AT.nlogo

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `trendAgents` (5); this breed bases its trading strategy on the sliders:
    - `StartingMA` (Section 2.3.1 point 2). It affects the magnitude of the sample used to calculate the moving average: it is kept fixed at 100.

2.4.3 Comparisons between artificial and real market : AB.nlogo

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `BBAgents` (5); this breed bases its trading strategy on the sliders:
    - `nMovingAverage` (Section 2.3.1 point 3). It determines the exact value of the sample used to compute the moving average : it is kept fixed at 20.
    - `Bandwidth` (Section 2.3.1 point 3). It affects the width of the *Bollinger bands* (explained in Section 2.1.7):: it is kept fixed at 2.

Figure 2.20: *AV.nlogo* : plot of the variable exeprice after 97 ticks

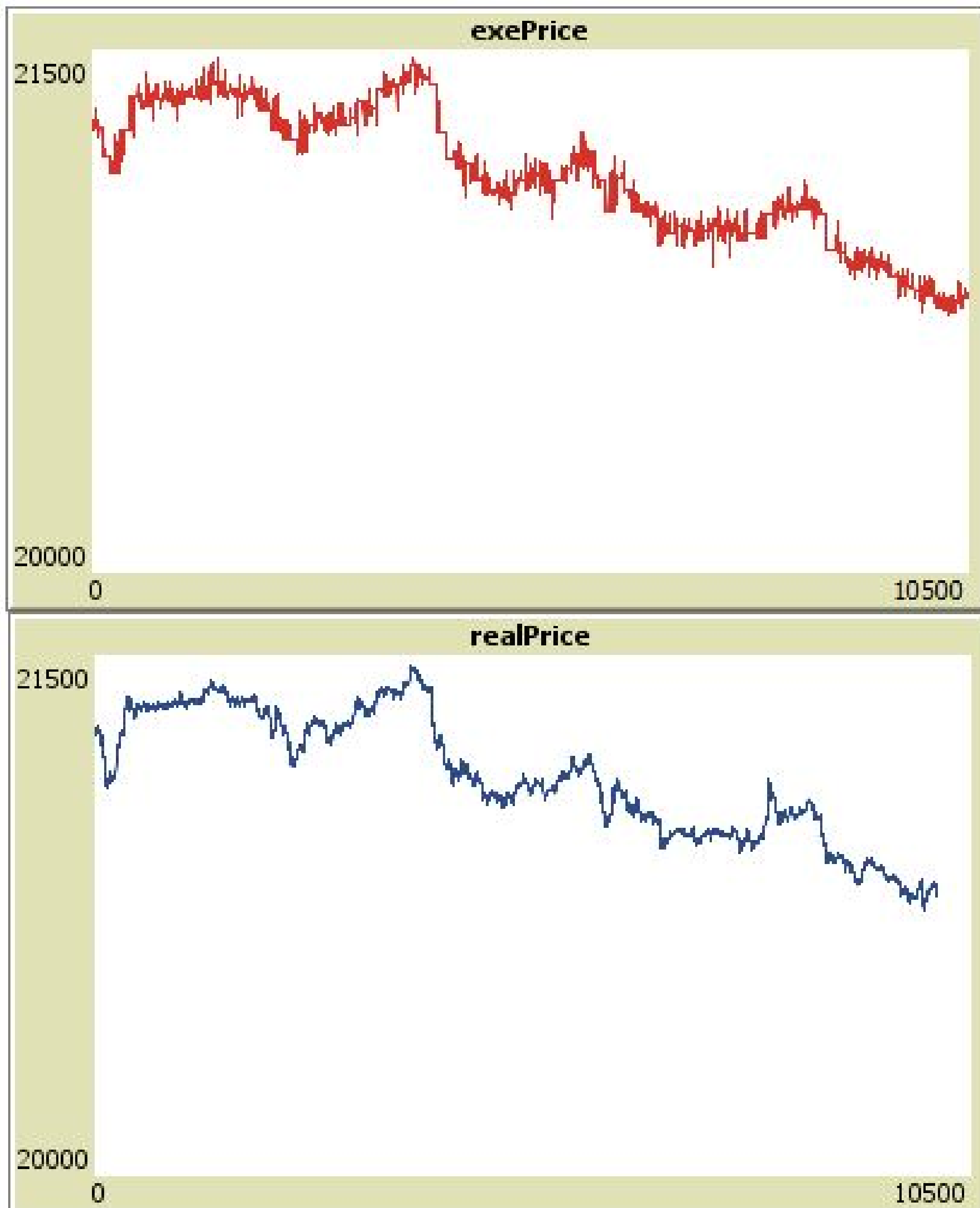

Figure 2.21: *AT.nlogo* : plot of the variable exeprice after 97 ticks

Figure 2.22: *AB.nlogo* : plot of the variable exeprice after 97 ticks.

2.4.4 Comparisons between artificial and real market : ASL.nlogo

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `SLAgents` (5); this breed bases its trading strategy on the sliders:
   - `risk-free` (Section 2.3.1 point 4). It represents the risk-less rate of return of the market, necessary to calculate the B-S formula (explained in Section 2.1.5 point 1) : it is kept fixed to 0.05
   - `Tsl` (Section 2.3.1 point 4). Multiplied by 700, it is the step after which the option expires, and the strategy is checked: it is kept fixed at 2.

2.4.5 Comparisons between artificial and real market : AC.nlogo

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `CoveredAgents` (5); this breed bases its trading strategy on the sliders:
   - `risk-free` (Section 2.3.1 point 5). It is the risk-less rate of return of the market, this slider is shared with `SLAgents`.
   - `Tc` (Section 2.3.1 point 5). It affects the expire date of the options traded by `CoveredAgents` : it is kept fixed at 10. `sampleStoch` (Section 3.1 point 5). It affects the magnitude of the sample considered to calculate the moving average of `CoveredAgents`, and the starting point of their trading strategy : it is kept fixed at 50.

Figure 2.23: *ASL.nlogo* : plot of the variable exeprice after 97 ticks.

Figure 2.24: *AC.nlogo* : plot of the variable exeprice after 97 ticks.

2.4.6 Comparisons between artificial and real market : ABT.nlogo

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `BBAgents` (5); with sliders:
   - `nMovingAverage` = 20.
   - `Bandwidth` = 2
4. `trendAgents` (5); with sliders:
   - `StartingMA` = 100.

2.4.7 Comparisons between artificial and real market : A-C-SL.nlogo

It contains the breeds :

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `CoveredAgents` (5); with sliders:
   - `Tc` = 10.
   - `SampleStoch` = 50
4. `SLAgents` (5); with sliders:
   - `Tsl` = 2.

   The slider `risk-free` is the same for both `CoveredAgents` and `SLAgents`, it is equal to 0.05

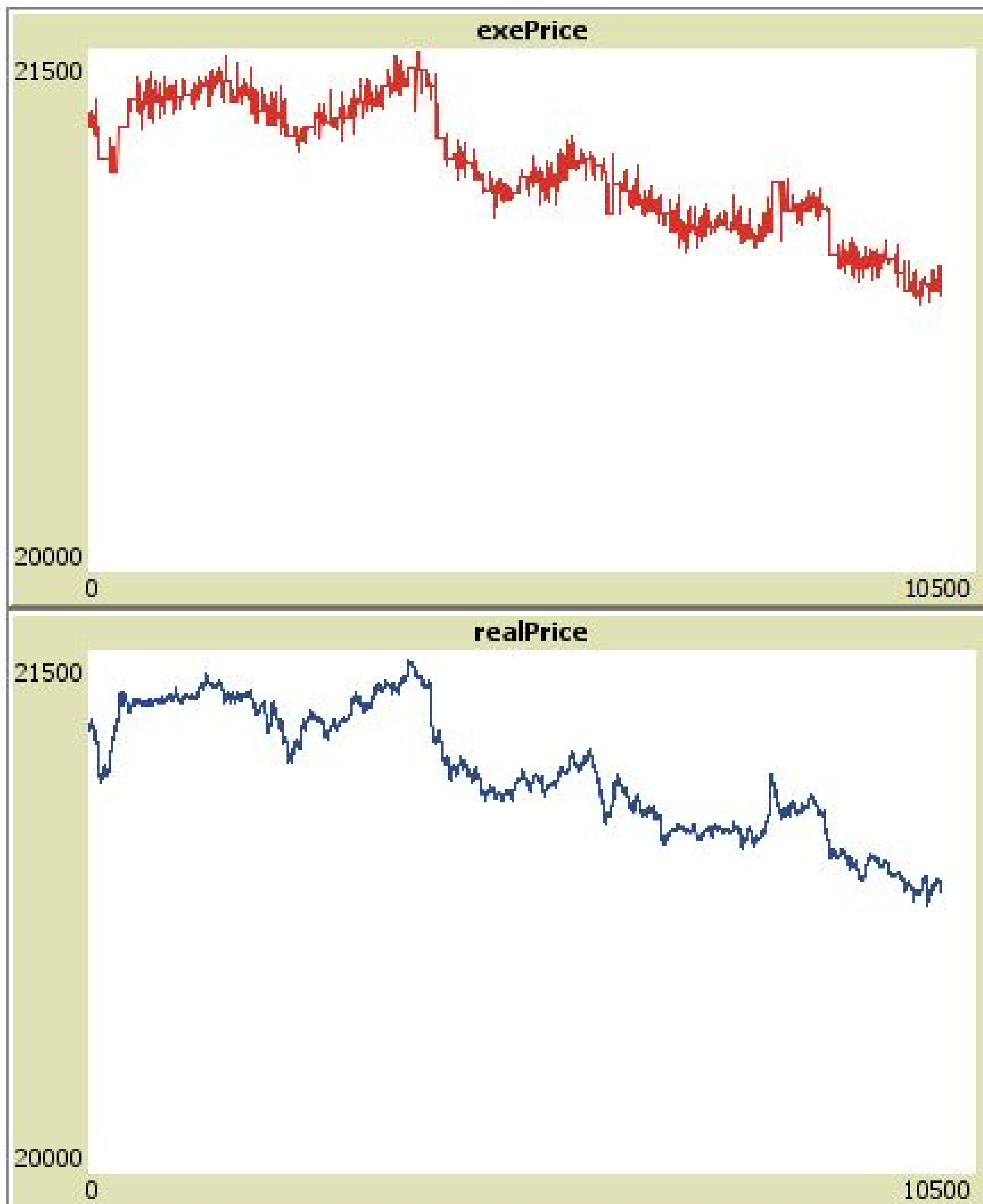

Figure 2.25: *ABT.nlogo* : plot of the variable *exeprice* after 90 ticks.

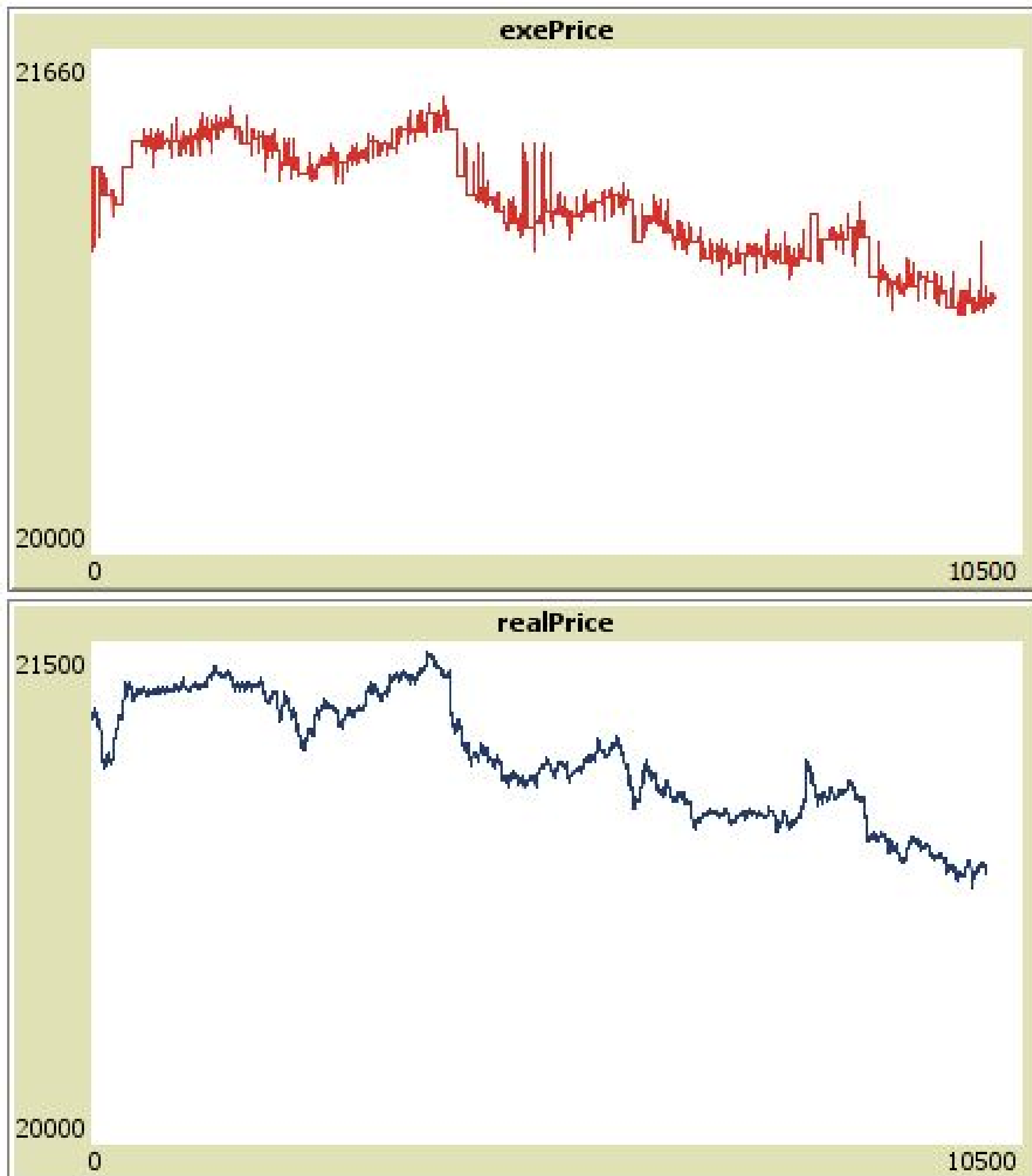

Figure 2.26: *A-C-SL.nlogo* : plot of the variable *exeprice* after 90 ticks.

2.4.8 Comparisons between artificial and real market : A-B-C-T-SL-V.nlogo

As explained in Section 2.3.2, the program *A-B-C-T-SL-V.nlogo* contains all breeds of agents included in my thesis.

1. `RandomAgents` (100)
2. `arbitrageur` (1)
3. `BBAgents` (5); with sliders:
   - `nMovingAverage` = 20.
   - `Bandwidth` = 2
4. `CoveredAgents` (5); with sliders:
   - `Tc` = 10.
   - `SampleStoch` = 50
5. `trendAgents` (5); with sliders:
   - `StartingMA` = 100.
6. `SLAgents` (5); with sliders:
   - `Tsl` = 2.
7. `VolumeAgents` (5); with sliders:
   - `VolumeAgentStep` = 10 .

   As for *A-C-SL.nlogo*, the slider `risk-free` is shared from both `CoveredAgents` and `SLAgents` ; it is equal to 0.05

The graphs related to Figures 2.20, 2.21, 2.22, 2.23, 2.24, 2.25, 2.26, 2.27, can be compared to the graph generated by the market of Figure 2.3, that is composed by only one-hundred `RandomAgents` and one `arbitrageur` (the basic framework described in Section 2.2).

But to quantify with more precision the differences between the eight artificial markets created so far, we need to use a search algorithm, and we need to define a fitness function to be minimized or maximized; this is the topic of the next Section.

Figure 2.27: *A*-*B*-*C*-*T*-*SL*-*V* : plot of the variable *exeprice* after 80 ticks.

2.5 Optimize Agents Behavior

The categories of trading agents, can influence the market price in different ways, and generates different price trends; the main question is: how can their strategies drive the market price out of the main benchmark trend, i.e. the *Ftse All Share* trend ?

As introduced in Section 2 the variable `artificialVSreal`, memorizes the average distance between the artificial and the real markets, in absolute value.

This variable is defined through the *NetLogo* command

```
if j < 10095
[ set artificialVSreal artificialVSreal
+ ( abs(item 0 SBoxP3 - item 0 realPvec3)) / 10095
  set realPvec3 but-first realPvec3]
```

where 10095 is the length of the vector of real prices, and represents also the interval of calculation of `rtificialVSreal` ; `realPvec3` is a vector that contains in the position 0 always the current real price.

`artificialVSreal` can be used as fitness function, to be minimized by a search algorithm.

For this purpose I introduce the *BehaviorSearch* software tool, that can interact with *NetLogo* and can try to solve optimization problems using various search algorithms (their characteristics are described in Chapter); to implement the analysis of my thesis about the distance between the artificial and the real market, I will use the most popular and flexible search algorithm of *Behaviorsearch* : the *Genetic Algorithm*, (named *StandardGA* in *BehaviorSearch*).

But what does it means 'trying' to solve an optimization problem in that context, i.e. for the programs described in Sections 2.3.1, and 2.3.2?

(Section 1.3.1, 1.3.2, 1.3.3, contains a detailed description of all the steps for the interaction between *NetLogo* and *BehaviorSearch*) briefly it means to:

1. define the objective of the analysis, i.e. the fitness function; in our case is the variable `artificialVSreal`. We can consider it as our dependent variable.

2. Define the value of the sliders that are not allowed to variate, and have a fixed value.

3. Define the range of the sliders that are allowed to variate, and to influence the output of the search; they are our independent variables. For this scope I will define such range case by case.

4. Analyse the results, in terms of dependent variables (the sliders characterising the trading strategy of each 'intelligent' agent)

In general, variables used in an experiment or modelling can be divided into three types: 'dependent variable', 'independent variable', or other. The 'dependent variable' represents the output or effect, or is tested to see if it is the effect. The 'independent variables' represent the inputs or causes, or are tested to see if they are the cause. Other variables may also be observed for various reasons (in our case they are the sliders with fixed value, and are useful to characterise the analysed market structure).

The independent variable can be considered the function for which we want to solve the optimization problem. In calculus, a function is a map whose action is specified on variables. Take x and y to be two variables. A function f may map x to some expression in x. Assigning $y = f(x)$ gives a relation between y and x. If there is some relation specifying y in terms of x, then y is known as a dependent variable (and x is an independent variable).

In our case, we are considering a multidimensional space, where the number of independent variables $x_1, x_2, ..., x_n$ (the sliders), depends on the program chosen to interact with *BehaviorSearch* , while the variable `artificialVS real` is our independent variable, or our function $y = f(x_1, x_2, ..., x_n)$. The function f is quite complex, and there are not clear relations between x and y ; for this reason we cannot find analytically the value of `artificialVSreal` for a certain composition of sliders: a possible solution can be given through an optimal search with *genetic algorithm.*

In order to exploit comparisons between the different market structures defined in Sections 3.1 and 3.2, I will recreate any framework (of the programs *AB.nlogo*, *AC.nlogo*, *AV.nlogo*, *AT.nlogo*, *ASL.nlogo*, *ABT.nlogo*, *A-C-SL.nlogo*) using the program *A-B-C-SL-T-V.nlogo* (it contains all trading agents categories described in my thesis): in this way it is possible to change the value of the sliders, without modify the value of the `seed`.

2.5.1 Preliminary BehaviorSearch test: the price structure

This section contains a preliminary analysis with *BehaviorSearch*: the objective is to study in the basic framework defined in Section 2, the optimal value of the fixed and the floating components of the variable `price`, being `artificialVSreal` the fitness function .

We have assumed until now a fixed price part equal to 21300 and a floating price part equal to a Normal random variable with mean 0 and variance 30.

(Note that defining the floating part in a different ways can lead to different results).

Remember that the basic framework contains only two breeds of agents: `RandomAgents` and `arbitrageur`.

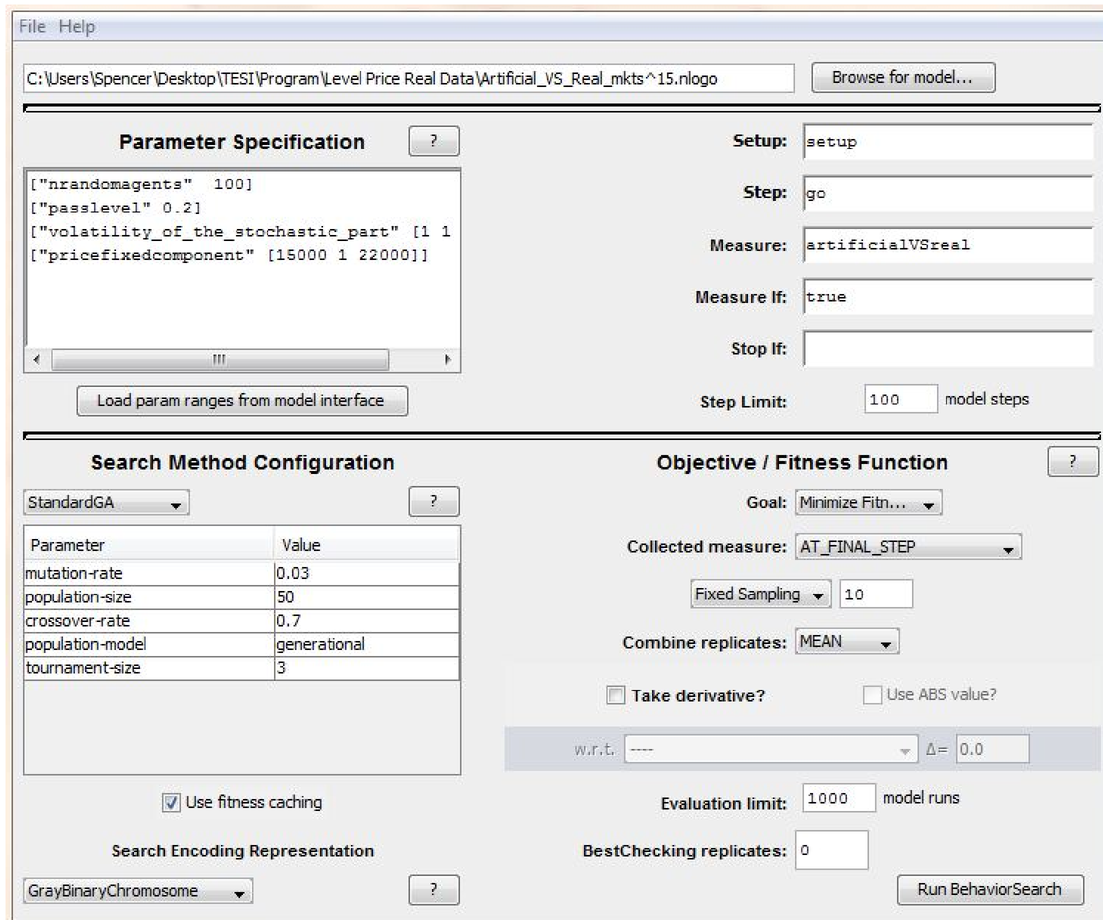

Figure 2.28: *BehaviorSearch Experiment Editor* : search of the optimal values for the fixed and the variable parts of the price.

The appearance of the *BehaviorSearch Experiment Editor* can be seen in Figure 2.28.

For what concern the `Parameter Specification`, we have two fixed values realted to the sliders `nRandomAgents` (= 100) and `pass-level` (= 0.2).

While the proper search space is defined by the variables:

1. `Volatility_of_the_stochastic_part`, it is the floating part of the price with range [1 , 150] and step 1.

2. `pricefixedcomponent`, it is the fixed part of the price; it is allowed to variate in the interval [15000 , 22000] with step 1.

So the magnitude of the search space is equal to $150 \times 7001 = 105150$.

Since I do not know the characteristics of the search space (it is smoothed or not?), the `Search Method Configuration` is completed with the *StandardGA* custom input parameters.

Another important thing to notice is the fact that the real price is completely simulated after about 100 ticks; for this reason the `Step Limit` is set equal to 100 (note that a big value of step runs enlarges the time due for the analysis).

The `Evaluation Limit` is set equal to 1000.

After ten trials(*BehaviorSearch* analyses with different seed) I get the following range of optimal values:

1. `Volatility_of_the_stochastic_part`, interval [1 , 7], and average about 3;

2. `pricefixedcomponent`, interval [18400 , 21000], with mean of about 19500.

In this context, the final fitness is always of about 75. It means that the distance per-price between the artificial and real markets is of 75 points in the best situation.

If we consider that the mean of the time series of real prices is exactly 21151.576, then the `artificialVSreal` variable is about 0.0035 percent of this value.

Moreover the value of `artificialVSreal` in the worst situation is about 90 units.

After the results of this analysis, I will assume in next Sections, a `pricefixed-component` = 19500, and a
`Volatility_of_the_stochastic_part` = 3
(these values will be kept fixed in next searches).

2.5.2 BehaviorSearch test number 1: VolumeAgents

Market structure composed by

- `RandomAgents` 100

- `arbitrageur` 1

- `VolumeAgents` [1 , 5]

The first analysis on the trading agents behavior in the market regards `VolumeAg-ents`.

The market analysed in this section is the one explained in Section 3.1 point 1. This breed invest on the market depending on trading volumes, i.e. the difference between the number of buyers (it is the length of the vector `logB`) and the sellers (it is the length of the vector `logS`) at a given auction mechanism step.

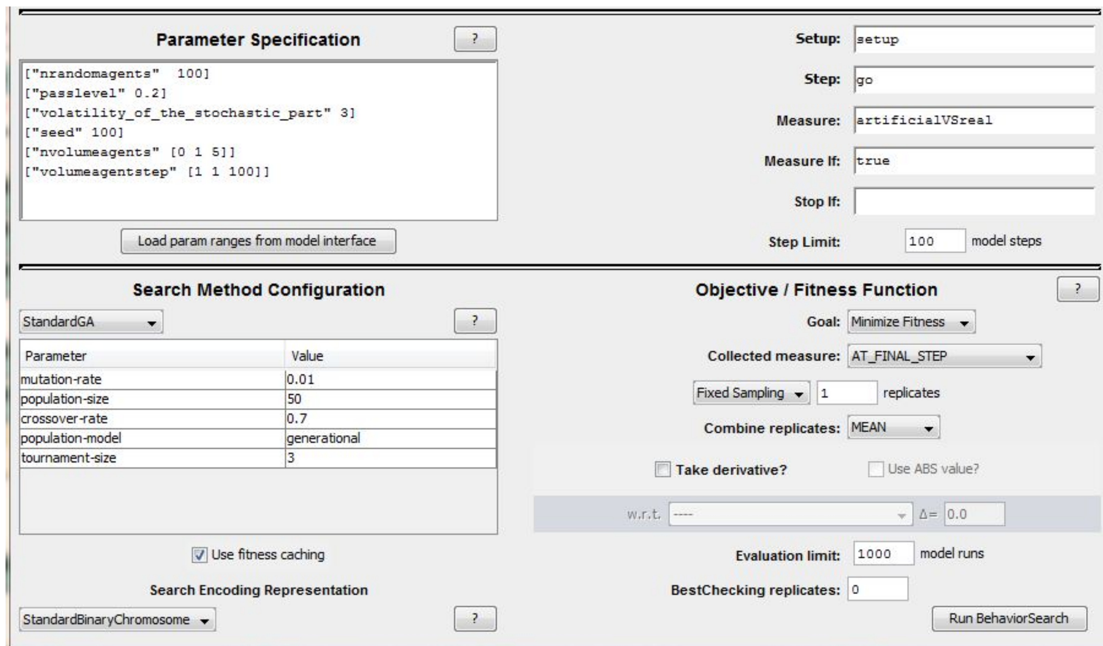

Figure 2.29: *BehaviorSearch Experiment Editor* : search of the optimal behavior of volume agents.

So their strategy is affected mostly by randomness (i.e. by the probabilities of buy and sell of `RandomAgents`); the only parameter of the `VolumeAgents` strategy that can be used as independent variable to minimize the fitness, is the frequency of their investments, governed by the slider `VolumeAgentStep`. The interval of variation is [1 , 100].

Another slider that is allowed to variate is `nVolumeAgents`, but only in the interval [1 , 5].

The total search space size is equal to $101 \times 5 = 505$.

In this framework, the *BehaviorSearch Experiment Editor* can be seen in Figure 2.29 .

After the implementation of ten trials, we can collect the following results.

Seed	nVolumeAgents	VolumeAgentStep	Fitness
100	3	15	70.7
101	2	14	72
102	1	7	68.5
103	4	8	73.4
104	4	7	70.8
105	3	35	76.9
106	4	19	76
107	3	3	72.1
108	3	2	67.9
109	2	21	72.6

Where:

- the average fitness is about 72: it is lower than that observed for the basic framework, that is about 75 (see Section 5.1);
- the average value of `VolumeAgentStep` is about 13, but with a big variance; in fact the interval of values lies in the interval [2 , 35];
- the average number of `VolumeAgents` is 3.

2.5.3 BehaviorSearch test number 2: trendAgents

Market structure composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `trendAgents` [1 , 5]

This Section treats the *BehaviorSearch* analysis of `trendAgents` investments, in a the basic framework defined in Section 2 .

`trendAgents` invest in the market according to the value of the current price with respect to a moving average, calculated on a sample that is different for any `trendAgent` (see Section 2.3.1 point 2).

The investment strategy of this breed is affected only by the slider `StartingMA` : it is proportional to the magnitude of the sample used for the moving average calculation. it is allowed to variate in the interval [45 , 300].

The other slider that is allowed to variate is `nTrendAgents`. Its range is defined by the interval [1 , 5].

Indeed the search space size is equal to $254 \times 5 = 1270$.

The *BehaviorSearch Experiment Editor* can be seen in Figure 2.30.

Figure 2.30: *BehaviorSearch Experiment Editor* : search of the optimal behavior of trend agents.

This breed of agents, seems to have the strongest effect on the variable `artificialVSreal`. After ten trials we have:

- The average fitness is about 67: it is the lowest with respect to that observed in the market structures composed by only the basic framework plus one breed of trading agents;
- the average value of `StartingMA` is about 160;
- the average number of `trendAgents` is 4.5, but with a small variance: in fact the possible values of the slider `nTrendAgents` are 4 or 5.

The following table shows the results of the *BehaviorSearch* trials, considering seeds from 100 to 109.

Seed	nTrendAgents	StartingMA	Fitness
100	4	136	62.6
101	5	274	59.9
102	4	166	66.3
103	5	84	70.1
104	5	208	71.3
105	4	171	66.9
106	4	94	72
107	4	262	69.1
108	5	104	61.9
109	5	95	70.8

2.5.4 BehaviorSearch test number 3: BBAgents

Market structure composed by

- `RandomAgents` 100
- `arbitrageur` 1
- `BBAgents` [1 , 5]

The objective of the analysis of this Section is the search of the optimal `BBAgents` sliders values, being `artificialVSreal` the fitness function. This market structure is explained in Section 2.3.1 point 3.

The independent variables that want to consider are the ones characterising the `BBAgents` behavior:

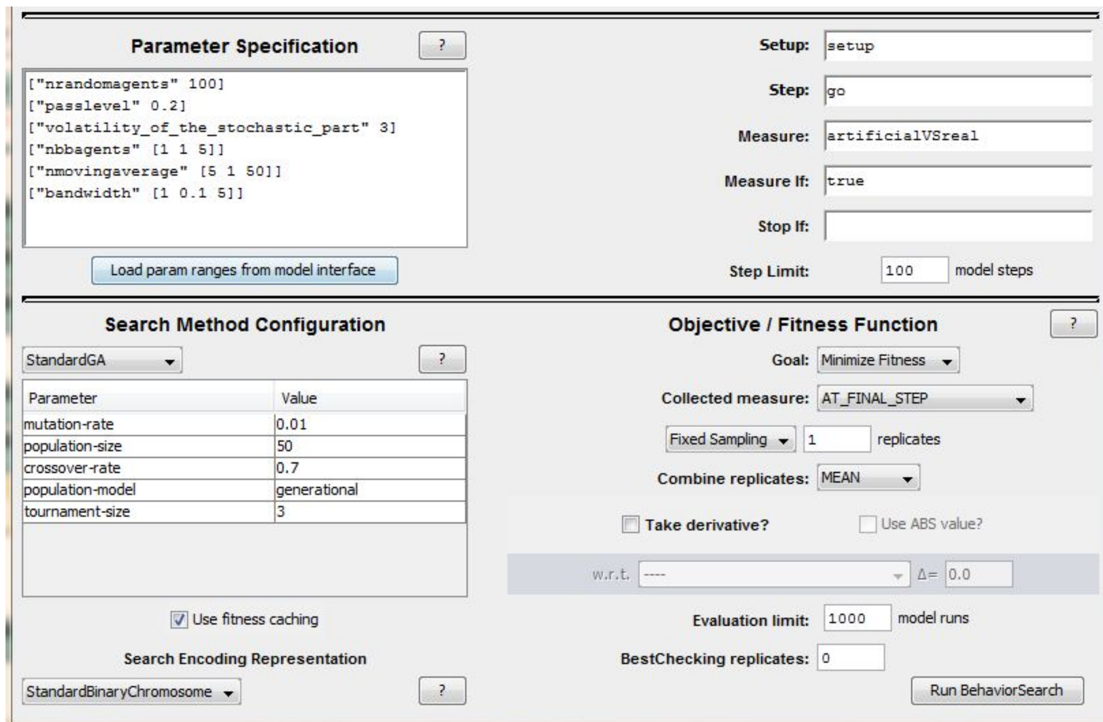

Figure 2.31: *BehaviorSearch Experiment Editor* : analysis of the optimal parameters of Bollinger bands agents .

1. the slider `nMovingAverage` ; it is the exact sample that each `BBAgent` uses to calculate the moving average. Its range is [5 , 50] with step 1.

2. `Bandwidth` ; it is necessary to calculate the *Bollinger bands*, and to define their width. It ranges from 1 to 5 with step 0.1 .

The last slider that is allowed to variate is `nBBAgents`, that identifies the number of Bollinger agents in the market; its range of variability is [1 , 5] with step 0.1 .

In this framework the size of the search space is equal to
$46 \times 41 \times 5 = 9430$.

Figure 2.31 shows the complete *BehaviorSearch Experiment Editor.*

BehaviorSearch results very efficient, and can find an optimal solution always within the first 200 model runs.

Considering ten different random seeds, (in the *NetLogo* program), the fitness ranges in the interval [68 , 78] with mean of about 73: it seems that adding only

the `BBAgents` breed, the final fitness is slightly lower with respect to that observed for the basic framework (that was of about 75).

The range of the optimal values of the sliders is:

[11 , 49] with mean of about 30 for `nMovingAverage`;

[1.1 , 4.2] with mean of about 2.5 for `Bandwidth`;

On average the number of `BBAgents` that participate to the market is 3; but the most frequent is 2.

Instead, for what concerns the possible combinations of `nMovingaverage` and `Bandwidth` values, we can have the following cases:

- Quite big value of `nMovingaverage`, and quite big value of `Bandwidth`, for example 31 and 3.3: this case is rare, but it represents a quite stable combination. In fact in reality the two parameter should be positively correlated.

- Big value of `nMovingaverage`, and small value of `Bandwidth`, for example 41 and 1.5: this case is the most frequent in the simulation, but unrealistic. A low width of the bands should be associated to a very sensitive moving average, i.e. calculated on a small sample. However these combinations are very risky, and produce often strongly negative cash flows.

- Small value of `nMovingaverage`, and big value of `Bandwidth`, for example 13 and 3.1. This association is quite frequent, and is quite realistic.

- Small value of both sliders. It is coherent but quite risky. In the simulation this optimal combination is rare.

However, if we consider the mean (on ten search trials) of these two sliders, we obtain a rational Bollinger strategy.

The following table shows the data collected from *BehaviorSearch* , related to the optimal value of `BBAgents` sliders, and the corresponding fitness.

Seed	nBBAgents	MovingAverage	Bandwidth	Fitness
100	2	11	1.1	73.2
101	4	33	2.4	73.9
102	2	39	1.5	71.2
103	4	18	2.5	73.7
104	2	38	2.8	68.1
105	3	49	1.6	73.3
106	5	31	3.3	76.5
107	1	13	3.1	70.5
108	5	41	1.5	78.3
109	3	23	4.2	70.2

2.5.5 BehaviorSearch test number 4: SLAgents

Market structure composed by

- `RandomAgents` 100
- `arbitrageur` 1
- `SLAgents` [1 , 5]

This market structure (described in Section 2.3.1) has been built to study the investment behavior of `SLAgents`, in the the basic framework (defined in Section 2.2).

This breed of agents uses a stop loss strategy, that involves the calculation of a call option with the B-S formula (explained in Section 2.1.5); the sliders that affect the `SLAgents` trading choices are:

1. `Tsl`; it determines the frequency of the strategy; one unit of the slider is equivalent to one trading day (500 prices with one minute frequency). Its range is defined by the interval [1 , 10], with step 1.
2. `risk-free`; it is the risk-less rate of return of the market. Its interval of variability is [0 , 0.5] with step 0.01 .

Also the number of `SLAgents` is allowed to variate in the interval [0 , 5], and it is governed by the slider `nSLAgents`.

After the definition of our independent variables, we are able to calculate the search space size in this framework; it is equal to $10 \times 501 \times 5 = 25050$.

The *BehaviorSearch Experiment Editor*, related to the analysis of the market structure of this Section, can be seen in Figure 2.32.

The analyses related to ten different frameworks, created by considering seeds from 100 to 109, lead to the following average values of the sliders involved:

- the average fitness is about 74, so it is slightly less than the fitness reached for the basic framework (that is equal to 75), computed considering the same seeds;
- the mean of the `Tsl` values is about 5 (remember that one unit of this value corresponds to one trading day, i.e. 500 realisations of the artificial market price);

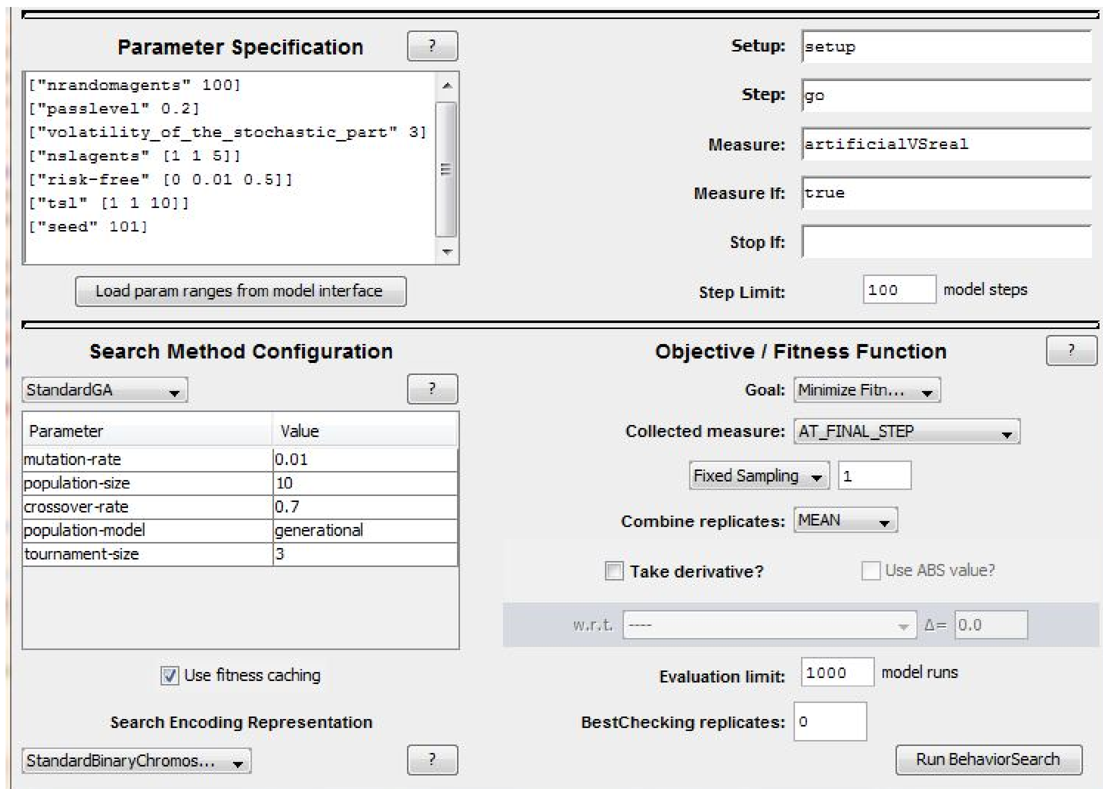

Figure 2.32: *BehaviorSearch Experiment Editor* : analysis of the optimal parameters of Stop Loss agents .

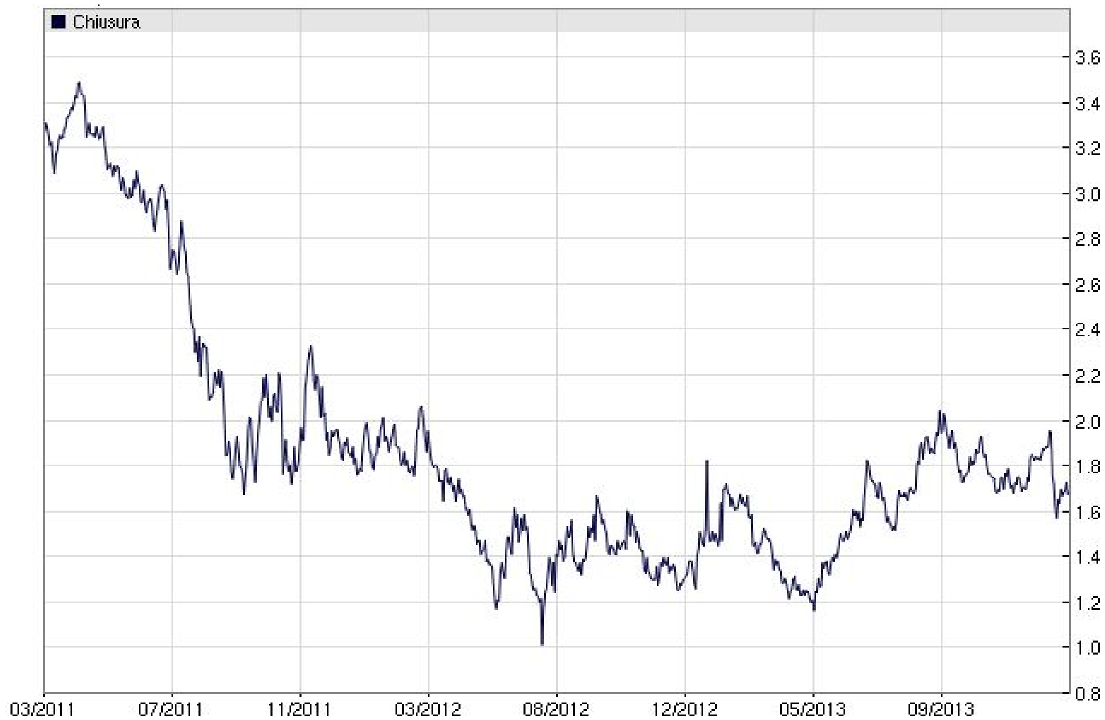

Figure 2.33: German bund return in percentage.

- the average risk-less rate is about 0.2: this value seems too high; I expected a value included in the interval [0.01 % , 0.05 %].

  Risk-free interest rate is the theoretical rate of return of an investment with no risk of financial loss.

  In practice to work out the risk-free interest rate in a particular situation, a risk-free bond is usually chosen that is issued by a government or agency where the risks of default are so low as to be negligible.

  An example the bund return trend (from 03/2011 until 17/02/2014) can be seen in Figure 2.33.

- the average number of `SLAgets` is 3.4 .

The averages discussed above are related to the data collected in the following table.

Seed	nSLAgents	Tsl	risk-free	Fitness
100	4	8	0.32	76.4
101	4	2	0.09	78.1
102	1	2	0.25	73.9
103	4	10	0.08	74.5
104	5	9	0.41	75.5
105	5	9	0.07	70.3
106	2	2	0.12	74.5
107	3	2	0.47	72.8
108	4	5	0.02	72.9
109	2	4	0.15	71.5

2.5.6 BehaviorSearch test number 5: CoveredAgents

Market structure composed by

- `RandomAgents` 100
- `arbitrageur` 1
- `CoveredAgents` [1 , 5]

The characteristics of this framework are the ones explained for the program *AC.nlogo* (Section 2.3.1 point 5). This breed of agents has a quite complex trading strategy (explained in Section 2.1.6), based on options. They can be considered as a combination of `trendAgents` and `SLAgents` (since they base their investment choices on both, the value of moving average that is different for any `trendAgent` and the trade of an option, calculated through the B-S formula).

The sliders that are allowed to variate , and that identify the `CoveredAgents` trading strategy, are:

- `sampleStoch`; it affects the magnitude of the sample used to calculate the moving average, in particular its magnitude is inversely proportional to the magnitude of this sample. It is allowed to variate in the interval [10 , 90] .
- `risk-free`; it is the risk-less rate of return of the market. Its interval of variability is [0 , 0.5] with step 0.01 .
- `Tc`; it determines the frequency of the trading strategy of `CoveredAgents`. Its range is [1 , 50].

The last independent variable of this framework, is represented by the slider `nCoveredAgents`, that can range in the interval [1 , 5].

The search space size for this market structure is equal to

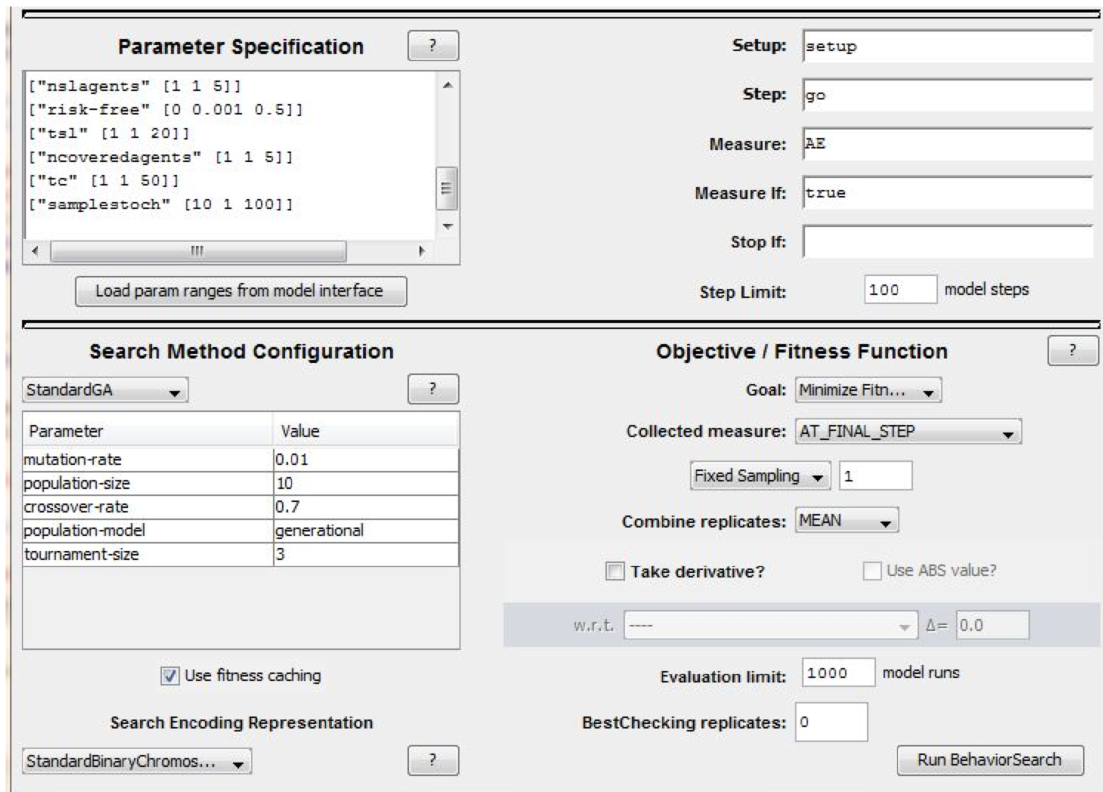

Figure 2.34: *BehaviorSearch Experiment Editor* : analysis of the optimal parameters of Covered agents .

$81 \times 501 \times 50 \times 5 = 10145250.$

The related *BehaviorSearch Experiment Editor* can be seen in Figure 2.34 .

The data collected from the *BehaviorSearch* analyses are reported in the following table.

Seed	nCoveredAgents	Tc	SampleStoch	risk-free	Fitness
100	5	15	13	0.11	75.3
101	3	18	39	0.37	63.2
102	5	20	35	0.24	68.7
103	4	28	56	0.48	71.2
104	3	47	25	0.47	72
105	3	10	34	0.5	68
106	4	22	84	0.4	74.3
107	3	18	29	0.47	71.5
108	2	35	15	0.45	70.6
109	4	15	12	0.15	70.9

Where:

- the average fitness is about 71;
- the average frequency of the `CoveredAgents` strategy, measured through `Tc`, is 23;
- the means of the ten risk-free realisations is 0.36: this number is too high and unrealistic, as discussed in section 5.5 .
- the average value of `SampleStoch` is equal to about 34 (remember that this value is inversely proportional to the magnitude of the sample used to calculate the moving average).
- the average number of `CoveredAgents` participating to the market is equal to 4.

2.5.7 BehaviorSearch test number 6: trendAgents + BBAgents

In this Section I consider a market structure composed by the basic framework plus both `trendAgents` and `CoveredAgents`.

The main objective is to study the aggregate effect on the artificial market, of the breeds of agents that base their strategy on *technical analysis*.

Each *BehaviorSearch* analysis is implemented considering the breeds of trading agents individually.

It means that a first analysis is made on `trendAgents` optimal sliders value, keeping fixed the values of `BBAgents` parameters; a second analysis is made vice-versa.

The value that are kept fixed for `BBAgents` and `trendAgents` respectively, are those obtained with previous analyses of Sections 2.5.3 and 2.5.4 (from seed 100 to seed 109).

Analysis on trendAgents parameters keeping fixed BBAgents parameters

Market structure composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `BBAgents` are set up with the sliders values reported in the table of Section 5.4 (from seed 100 to 109).
- `trendAgents` [1 , 5]

The details on the market structure created with the description of the trading agents categories can be found in Section 3.2 point 1.

The analysis implemented for `trendAgents` will be then compared to that of Sections 5.3, 5.4 (reported in the corresponding table).

The *BehaviorSearch Experiment Editor* maintains almost the same characteristics of those showed in previous Sections (Figures 2.28, 2.29, 2.30, 2.31, 2.32, 2.34); the only part that changes is the `Parameter Specification`, that variates according to the seed considered.

An example can be seen in Figure 2.35 , that shows the `Parameter Specification` for seed = 100.

The results of the analysis on `trendAgents`, with fixed parameters values of `BBAgents`, are reported in the following table.

Seed	nTrendAgents	StartingMA	Fitness
100	4	193	68
101	5	233	69.5
102	4	189	64.8
103	4	125	73.8
104	2	92	68.9
105	2	78	67.8
106	3	136	70.1
107	5	62	70.3
108	3	97	57.6
109	2	206	67.4

An important thing to notice is the fact that the analyses of this Section start from the optimal framework generated by `BBAgents`, whose optimal parameters have been collected in the table of Section 5.4 .

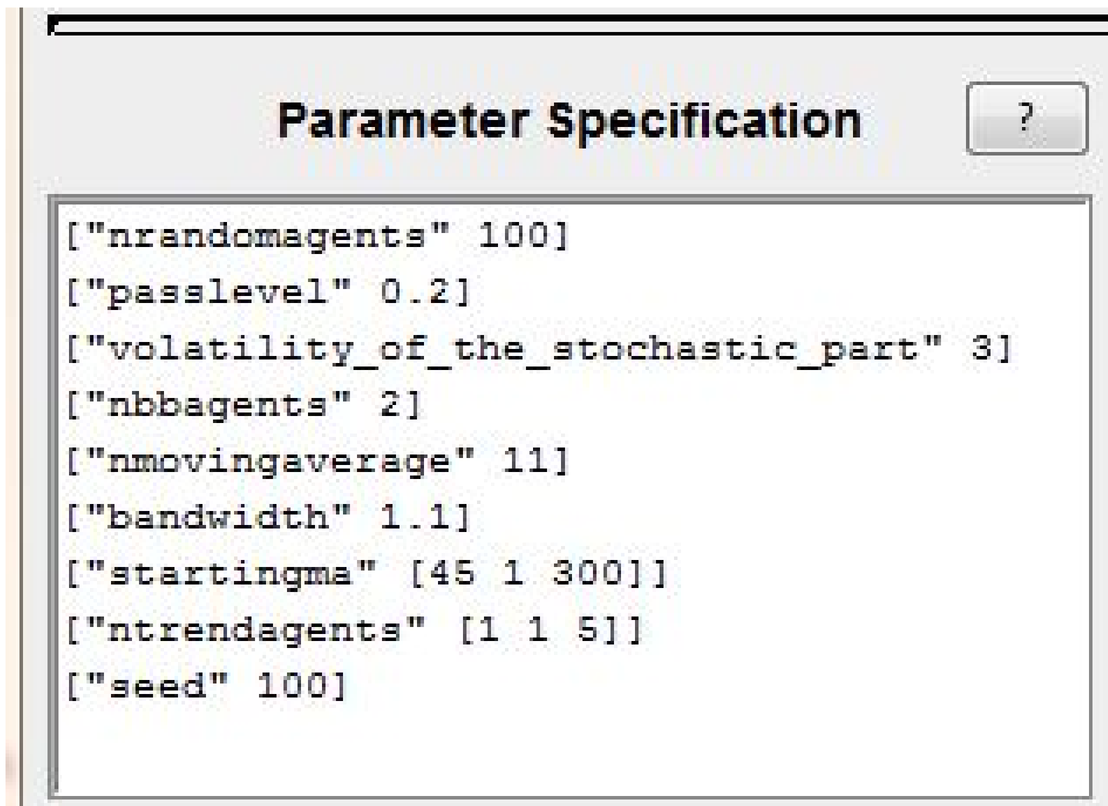

Figure 2.35: *Parameters Specification* : search of the optimal parameters of trend agents, with Bollinger agents parameters fixed .

For this purpose it is interesting to compare the new optimal parameters of `trendAgents` with the previous ones, i.e. those obtained without `BBAgents` constraints.

While the new values of fitness can be compared to those of both tables of Sections 2.5.3, 2.5.4 .

The new optimal parameters of `trendAgents`, have the following expected values:

- average fitness equal to 67.8: it is lower than the one observed in Section 5.3 (72), that considers the framework created by the program *AB.nlogo*; it is slightly bigger than the one observed in Section 2.5.4 (67.1), that is obtained analysing the framework created by the program *AT.nlogo*.

- the average number of `trendAgents` is 3.5; it is less than in the case without `BBAgents`,(Section 2.5.4, with average number of `trendAgents` equal to 4.5).

- the mean of the collected values of `StartingMA` is 141; it is less than the case of Section 2.5.3 that presented an average of about 160.

Analysis on BBAgents parameters keeping fixed trendAgents parameters

Market structure composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `trendAgents` are set up with the sliders values reported in the table of Section 2.5.3 (from seed 100 to 109).
- `BBAgents` [1 , 5]

The details on the market structure created with the description of the trading agents categories can be found in Section 2.3.2 point 2.

The table that shows the results of the `BBAgents` sliders values, constrained to `trendAgents` (with sliders value collected in the table of Section 2.5.3), is the following.

Seed	nBBAgents	MovingAverage	Bandwidth	Fitness
100	2	31	1.6	65.1
101	4	19	1.6	65.5
102	3	39	1	67.3
103	4	43	5.1	68
104	1	10	1.2	70.2
105	3	29	3	67
106	2	46	3.9	69.4
107	4	30	2.5	72.2
108	4	28	2.4	70.2
109	4	37	3.9	67.5

Comparing the table above with the table of Section 5.4, we can note:

- the average fitness is about 68.2; so it is lower than that observed without `trendAgents` constraints (that is about 73).

  However it is bigger than the one obtained in a market composed by only `trendAgents`, in which the average fitness measures 67.1.

- The average values of the sliders `nMovingAverage` and `Bandwidth` are very close to those observed analysing the framework created by the program *AB.nlogo* (in some trials they have exactly the same): they measure 31 and 2.6 respectively, while those observed in Section 2.5.4 are 30 and 2.5 respectively .

- The average number of `BBAgents` is 3, it is the same of that observed in the case of Section 2.5.4 .

2.5.8 BehaviorSearch test number 7: SLAgents + CoveredAgents

This Section follows the same guidelines of Section 2.5.7: the optimal behavior of `SLAgents` and `CoveredAgents` is analysed individually.

The values of the trading agents sliders that are kept fixed during the optimal search can be seen in the tables of Sections 2.5.5, 2.5.6 .

The slider `risk-free` will be allowed to variate in both cases described in Sections 2.5.8 (points 1 and 2) because the major part of results collected for this slider in Sections 2.5.5, 2.5.6 were very unrealistic.

Analysis on SLAgents parameters keeping fixed CoveredAgents parameters

This Section analyses the optimal behavior of `SLAgents`, in the framework characterising the program `A-C-SL.nlogo` : the parameter kept fixed for `CoveredAgents` are the ones collected in the table of Section 2.5.6 .

Briefly this market structure contains the following breeds:

- `RandomAgents` 100
- `arbitrageur` 1
- `CoveredAgents` : the number variates depending on the value of `nSLAgents` in the table of Section 2.5.6.
- `SLAgents` [1 , 5]

The results of the *BehaviorSearch* analysis can be seen in the following table.

Seed	nSLAgents	Tsl	risk-free	Fitness
100	4	4	0.41	79.9
101	1	2	0.01	71.7
102	3	3	0.28	81.2
103	4	3	0.16	68.7
104	3	6	0.35	78.2
105	1	3	0.2	78.2
106	3	5	0.16	79.2
107	4	3	0.46	72.2
108	5	3	0.41	71.4
109	1	2	0.04	74.3

The expected value of the sliders collected above is:

- about 3 for `Tsl` values ; so the strategy, in this case, is more frequent than in the case of Section 2.5.5, whose average is 5.
- about 3 for `nSLAgents` ; this average is exactly equal to that observed in Section 2.5.5.

the average fitness measures 75.5: it is bigger than both the cases of Sections 2.5.1 and 2.5.5, that are related to markets composed by respectively no trading agents, and by only `SLAgents` (with average finesses of 75 and 74).

Analysis on CoveredAgents parameters keeping fixed SLAgents parameters

This Sections analyses the opposite situation with respect to Section 2.5.8 point 1 : here we are searching the optimal parameters of `CoveredAgents`, having those of `SLAgents` fixed.

The values of the sliders characterising the investment strategy of `SLAgents` are the optimal ones, collected in the table of Section 2.5.5 .

In this way we get the following combination of agents in the market.

- `RandomAgents` 100
- `arbitrageur` 1
- `SLAgents` the number variates depending on the value of `nSLAgents` in the table of Section 2.5.6.
- `CoveredAgents` [1 , 5]

The *BehaviorSearch* analysis reached the following results.

Seed	nCoveredAgents	Tc	SampleStoch	risk-free	Fitness
100	2	16	95	0.42	71.3
101	4	17	26	0.01	70.5
102	1	8	32	0.32	69.4
103	1	5	40	0.1	73.3
104	2	39	74	0.21	71.2
105	1	17	82	0.35	71
106	2	12	58	0.15	71.9
107	2	40	97	0.06	71.6
108	2	15	20	0.12	67.7
109	2	16	67	0.49	74.4

Comparing the table above with that of Section 2.5.6 we can note:

- the average fitness is about 71 as measured in the case characterised by only the `CoveredAgents` breed.

  It is also less than both cases of Sections 2.5.5 (with only `SLAgents` and fitness equal to about 74), and 2.5.8 point 1 (with `SLAgents` constrained to `CoveredAgents` optimal parameters, fitness equal to 75.5).

- Inserting the `SLAgents` category, the average frequency of the strategy is smaller: it is equal to about 19 instead of 23.

- The average value of the slider `SampleStoch` is bigger with respect to the one reached in Section 2.5.6 (59 instead of 34). It means that on average, the time required to start the trading strategy of each `CoveredAgents` is lower if there are the `SLAgents` constraints.

- The mean of the slider `nCoveredAgents` is 2 instead of 4.

2.5.9 BehaviorSearch test number 8: all trading agents breeds

The objective of this Section is to study the investment behavior of each trading agent breed individually, in the framework described by the program *A-B-C-SL-T-V.nlogo*.

For this purpose I will consider as fixed parameters for the sliders of the agents, that are out of the focus of the search, those collected in the tables of Sections 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6 .

The following table contains all data collected in previous analyses, related to the optimal sliders values of each trading agents.

SEED	**100**	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**
nBBAgents	2	4	2	4	2	3	5	1	5	3
nMovingAverage	11	33	39	18	38	49	31	13	41	23
Bandwidth	1.1	2.4	1.5	2.5	2.8	1.6	3.3	3.1	1.5	4.2
StartingMA	136	274	166	84	208	171	94	262	104	95
nTrendAgents	4	5	4	5	5	4	4	4	5	5
nVolumeAgents	3	2	1	4	4	3	4	3	3	2
VolumeAgentStep	15	14	7	8	7	35	19	3	2	21
nSLAgents	4	4	1	4	5	5	2	3	4	2
Tsl	8	2	2	10	9	9	2	2	5	4
nCoveredAgents	5	3	5	4	3	3	4	3	2	4
Tc	15	18	20	28	47	10	22	18	35	15
SampleStoch	13	39	35	56	25	34	84	29	15	12

These values are the optimal ones of each breed, obtained analysing the frameworks defined by the programs *AV.nlogo*, *AT.nlogo*, *AB.nlogo*, *AC.nlogo*, *A-SL.nlogo* (described in Sections 2.3.1, 2.3.2).

Moreover, to be consistent with reality I will not consider as variable parameter, the value of `risk-free`, because previous *BehaviorSearch* analyses lead to too big values of the slider. For this reason its value will be kept fixed at 0.01 (to be consistent with the graph of Figure 2.33).

The ranges of variability of each slider, that define the search space conformation, are overall showed in Figure 2.36 .

Next Sections are organised in the following way:

1. summary of the breeds participating in the market analysed, and their number;
2. table containing the value of the sliders optimized, and the fitness: each table contains ten trials, implemented for seeds from 100 to 109;
3. calculation of the expected value for each parameter reported in the tables, and for the fitness (case by case).

The results of point 3 together with those of Sections 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, will be then compared in Chapter 3 (Conclusions).

Analysis on VolumeAgents optimal parameters keeping fixed all the remaining trading agents parameters

The market structure is composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `Volumeagents` [1 , 5]
- `trendAgents` the number variates depending on the value of `nTrendAgents` in the table of Section 2.5.3.
- `BBAgents` the number variates depending on the value of `nBBAgents` in the table of Section 2.5.4.
- `CoveredAgents` the number variates depending on the value of `nCoveredAgents` in the table of Section 2.5.5.
- `SLAgents` the number variates depending on the value of `nSLAgents` in the table of Section 2.5.6.

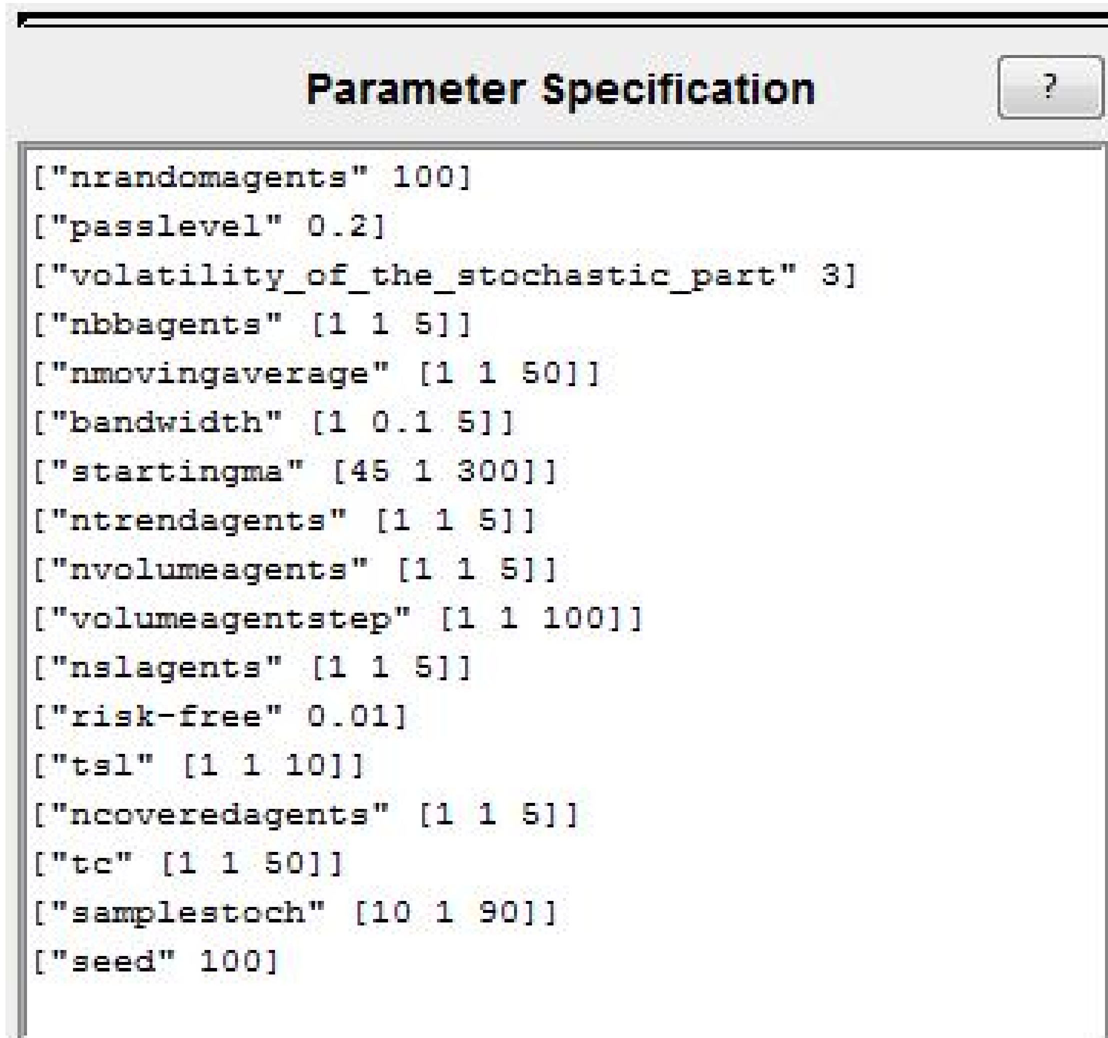

Figure 2.36: *Parameters Specification* : ranges of all parameters involved in the simulation; they will be analysed individually according to the trading agents breed, keeping all the remaining values fixed.

The *BehaviorSearch* results about `VolumeAgents` investment behavior in this market structure, are reported in the following table.

Seed	nVolumeAgents	VolumeAgentStep	Fitness
100	2	11	67.4
101	1	15	74.9
102	2	50	69.2
103	4	6	77.9
104	3	36	75.9
105	3	30	80.4
106	1	19	80.4
107	4	6	73.8
108	5	17	74.5
109	1	9	79.6
Mean	**3**	**20**	**75.4**

Analysis on trendAgents optimal parameters keeping fixed all the remaining trading agents parameters

The market structure is composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `Volumeagents` the number variates depending on the value of `nVolumeAgents` in the table of Section 2.5.2 .
- `trendAgents` [1 , 5]
- `BBAgents` the number variates depending on the value of `nBBAgents` in the table of Section 2.5.4 .
- `CoveredAgents` the number variates depending on the value of `nCoveredAgents` in the table of Section 2.5.5 .
- `SLAgents` the number variates depending on the value of `nSLAgents` in the table of Section 2.5.6 .

The *BehaviorSearch* results about `trendAgents` investment behavior in this market structure, are reported in the following table.

Seed	nTrendAgents	StartingMA	Fitness
100	1	71	72.6
101	5	63	72.9
102	1	113	73.2
103	4	195	77.6
104	1	140	74.3
105	3	144	77
106	3	116	73.8
107	4	104	71.2
108	2	70	71.9
109	4	299	76
Mean	**3**	**132**	**74**

Analysis on BBAgents optimal parameters keeping fixed all the remaining trading agents parameters

The market structure is composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `Volumeagents` the number variates depending on the value of `nVolumeAgents` in the table of Section 2.5.2 .
- `trendAgents` the number variates depending on the value of `nTrendAgents` in the table of Section 2.5.3.
- `BBAgents` [1 , 5]
- `CoveredAgents` the number variates depending on the value of `nCoveredAgents` in the table of Section 2.5.5.
- `SLAgents` the number variates depending on the value of `nSLAgents` in the table of Section 2.5.6.

The *BehaviorSearch* results about `BBAgents` investment behavior in this market structure, are reported in the following table.

Seed	nBBAgents	MovingAverage	Bandwidth	Fitness
100	4	9	4	72.5
101	3	41	4.6	73.9
102	3	21	1.7	68.9
103	3	47	2.6	76.1
104	1	46	3.9	75.8
105	3	37	1.5	75.2
106	2	45	2.8	70.3
107	2	23	2.4	69.4
108	4	19	3.2	72.2
109	3	43	2.4	72.7
Mean	**3**	**33**	**2.9**	**72**

Analysis on SLAgents optimal parameters keeping fixed all the remaining trading agents parameters

The market structure is composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `Volumeagents` the number variates depending on the value of `nSLAgents` in the table of Section 2.5.2 .
- `trendAgents` the number variates depending on the value of `nTrendAgents` in the table of Section 2.5.3 .
- `BBAgents` the number variates depending on the value of `nBBAgents` in the table of Section 2.5.4 .
- `CoveredAgents` the number variates depending on the value of `nCoveredAgents` in the table of Section 2.5.5 .
- `SLAgents` [1 , 5]

The *BehaviorSearch* results about `SLAgents` investment behavior in this market structure, are reported in the following table.

Seed	nSLAgents	Tsl	Fitness
100	1	9	79.7
101	4	1	81
102	2	3	77.3
103	4	10	82.5
104	4	6	80
105	5	6	79.3
106	5	7	78.8
107	2	2	70.6
108	5	3	80.1
109	4	2	76.3
Mean	**4**	**4**	**78.5**

Analysis on CoveredAgents optimal parameters keeping fixed all the remaining trading agents parameters

The market structure is composed by:

- `RandomAgents` 100
- `arbitrageur` 1
- `Volumeagents` the number variates depending on the value of `nVolumeAgents` in the table of Section 2.5.2.
- `trendAgents` the number variates depending on the value of `nTrendAgents` in the table of Section 2.5.3.
- `BBAgents` the number variates depending on the value of `nBBAgents` in the table of Section 2.5.4.
- `CoveredAgents` [1 , 5]
- `SLAgents` the number variates depending on the value of `nSLAgents` in the table of Section 2.5.6.

The *BehaviorSearch* results about `CoveredAgents` investment behavior in this market structure, are reported in the following table.

Seed	nCoveredAgents	Tc	SampleStoch	Fitness
100	1	9	29	72.1
101	5	26	13	71.8
102	1	26	13	69.8
103	4	21	30	77
104	1	14	55	73.2
105	2	16	41	77.6
106	2	18	26	70.7
107	3	15	17	71.5
108	3	10	29	75
109	4	17	58	69.3
Mean	**3**	**16**	**37**	**72.8**

2.6 Statistical tests with R upon results

BehaviorSearch allows a very big range of possible analyses, if we consider the artificial market structures defined so far.

In particular, my scope is that of studying the difference in term of average fitness (`artificialVSreal`), between the different breeds, and to analyse the possible changes in the investment strategy of any category of trading agents in various market conditions.

So we can say that my work is mainly focused on the individual effect of the breed, instead of analysing the optimal behavior of all classes of agents together; this because it is more difficult to interpret the results in a complex multivariate space, with a lot of independent variables.

To pursuit my objective and comment the fitness results reached in Section 5, I will use the following statistical tests:

1. One-way ANOVA test to compare the average fitnesses of the market structures with only one trading agent breed ; it means to compare the average distance between artificial and real markets for the five groups:

   (a) `VolumeAgents`, in the framework defined by the program *AV.nlogo*.

   (b) `trendAgents`, in the framework defined by the program *AT.nlogo*.

   (c) `BBAgents`, in the framework defined by the program *AB.nlogo*.

   (d) `SLAgents`, in the framework defined by the program *A-SL.nlogo*.

   (e) `CoveredAgents`, in the framework defined by the program *AC.nlogo*.

2. Linear regression to study the relationships (correlations) between the trading behavior of any breed alone in the basic framework (`Randomagents` +

arbitrageur) and the one of the same breeds, constrained (framework defined by the programs *ABT.nlogo*, *A-C-SL.nlogo*, *A-B-C-SL-T-V.nlogo*).

The statistical tests are related to the following data.

- Collection of the fitnesses observed in Sections 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6 (unconstrained market).

TABLE-1

Seed	VolumeAgents	trendAgents	BBAgents	SLAgents	CoveredAgents
100	70.7	62.6	73.2	76.4	75.3
101	72	59.9	73.9	78.1	63.2
102	68.5	66.3	71.2	73.9	68.7
103	73.4	70.1	73.7	74.5	71.2
104	70.8	71.3	68.1	75.5	72
105	76.9	66.9	73.3	70.3	68
106	76	72	76.6	74.5	74.3
107	72.1	69.1	70.5	72.8	71.5
108	67.9	61.9	78.3	72.9	70.6
109	72.6	70.8	70.2	71.5	70.9
Mean	**72.1**	**67.1**	**72.9**	**74**	**70.6**

- Collection of the fitnesses observed in Sections 2.5.9, points 1 to 5 (constrained to all breeds market) .

TABLE-2

Seed	VolumeAgents	trendAgents	BBAgents	SLAgents	CoveredAgents
100	67.4	72.6	72.5	79.7	72.1
101	74.9	72.9	71.1	81	71.8
102	69.2	73.2	68.9	77.3	69.8
103	77.7	77.6	76.1	82.5	77
104	75.9	74.3	75.8	80	73.2
105	80.4	77	75.2	79.3	77.6
106	80.4	73.8	70.3	78.8	70.7
107	73.8	71.2	69.4	70.6	71.5
108	74.5	71.9	72.2	80.1	75
109	79.6	76	72.7	76.3	69.3
Mean	**75.5**	**74.1**	**72.4**	**78.6**	**72.8**

- Collection of the fitnesses observed in Sections 2.5.7, 2.5.8 (constrained to one breed markets).

TABLE-3

Seed	trendAgents	BBAgents	SLAgents	CoveredAgents
100	68	65.1	79.9	71.3
101	69.5	65.5	71.7	70.5
102	64.8	67.3	81.2	69.4
103	73.8	68	68.7	73.3
104	68.9	70.2	78.2	71.2
105	67.8	67	78.2	71
106	70.1	69.4	79.2	71.9
107	70.3	72.2	72.2	71.6
108	57.6	70.2	71.4	67.7
109	64.4	67.5	74.3	74.4
Mean	**67.8**	**68.2**	**75.5**	**71.2**

2.6.1 ANOVA test on the fitness results of agents breeds

Analysis of variance (ANOVA) is a collection of statistical models used to analyse the differences between group means and their associated procedures (such as "variation" among and between groups). In ANOVA setting, the observed variance in a particular variable is partitioned into components attributable to different sources of variation. In its simplest form, ANOVA provides a statistical test of whether or not the means of several groups are equal, and therefore generalizes the t-test to more than two groups. Doing multiple two-sample t-tests would result in an increased chance of committing a type I error. A type I error, also known as an error of the first kind, occurs when the null hypothesis (H0) is true, but is rejected. It is asserting something that is absent, a false hit. A type I error may be compared with a so-called false positive (a result that indicates that a given condition is present when it actually is not present) in tests where a single condition is tested for.

For this reason, ANOVAs are useful in comparing (testing) three or more means (groups or variables) for statistical significance.

The normal-model based ANOVA analysis assumes the Independence of observations, Normality of residuals, and the homogeneity of variances.

ANOVA is a particular form of statistical hypothesis testing heavily used in the analysis of experimental data. A statistical hypothesis test is a method of making decisions using data. A test result (calculated from the null hypothesis and the sample) is called statistically significant if it is deemed unlikely to have occurred

by chance, assuming the truth of the null hypothesis. A statistically significant result (when a probability (p-value) is less than a threshold (significance level)) justifies the rejection of the null hypothesis, but only if the a priori probability of the null hypothesis is not high. In the typical application of ANOVA, the null hypothesis is that all groups are simply random samples of the same population. This implies that all treatments have the same effect (perhaps none). Rejecting the null hypothesis implies that different treatments result in altered effects.

In our context the ANOVA test is implemented on the samples collected in TABLE-1 (I will consider that all the assumptions to implement the test are met) : the null hypothesis, denoted H0, for the overall F-test for this experiment would be that all five levels of the factor (fitnesses related to `VolumeAgents`, `trendAgents`, `BBAgents`, `SLAgents`, `CoveredAgents`) produce the same response, on average. To calculate the F-ratio we must follow the steps:

1. Calculate the mean within each group.

   $E(Fit_{volume}) = 72.1$

   $E(Fit_{trend}) = 67.1$

   $E(Fit_{BB}) = 72.9$

   $E(Fit_{SL}) = 74$

   $E(Fit_{covered}) = 70.6$

   where:

   - $E(Fit_{volume})$ is the average fitness analysed the framework defined by the program *AV.nlogo*
   - $E(Fit_{trend})$ is the average fitness analysed the framework defined by the program *AT.nlogo*
   - $E(Fit_{BB})$ is the average fitness analysed in the framework defined by the program *AB.nlogo*
   - $E(Fit_{SL})$ is the average fitness in analysed the framework defined by the program *ASL.nlogo*
   - $E(Fit_{covered})$ is the average fitness analysed the framework defined by the program *AC.nlogo*

2. Calculate the overall mean.

   $E(Fit_{TOT}) = \frac{1}{5} \times (72.1 + 67.1 + 72.9 + 74 + 70.6) = 71.3$

3. Calculate the "between-group" sum of squares:

   $S_B = 10 \times [(E(Fit_{volume}) - E(Fit_{TOT}))^2 + (E(Fit_{trend}) - E(Fit_{TOT}))^2 + (E(Fit_{BB}) - E(Fit_{TOT}))^2 + (E(Fit_{SL}) - E(Fit_{TOT}))^2 + (E(Fit_{covered}) - E(Fit_{TOT}))^2] =$

   $10 \times [(71.2 - 71.3)^2 + (67.1 - 71.3)^2 + (72.9 - 71.3)^2 + (74 - 71.3)^2 + (70.6 - 71.3)^2] = 279.9$

   The between-group degrees of freedom is one less than the number of groups.

   $f_B = 5 - 1 = 4$

   The between-group mean square value is equal to

   $MS_B = \frac{S_B}{f_B} = \frac{279.9}{4} \cong 70$

4. Calculate the "within-group" sum of squares.

   The within-group sum of squares is the sum of squares of all 50 values of TABLE-1.

   $S_W = 478.3$

   The within-group degrees of freedom is

   $f_W =$ number of samples $\times$ (number of values in each sample - one)

   $=$ 5 (10 - 1) $=$ 45

   Thus the within-group mean square value is equal to

   $MS_W = \frac{S_W}{f_W} = \frac{478.3}{45} = 10.6$

5. The F-ratio is:

   $F = \frac{MS_B}{MS_W} = \frac{70}{10.6} \cong 6.6$

The critical value is the number that the test statistic must exceed to reject the test. In this case, $F_{crit}(f_b, f_W) = F(4, 45) \cong 2.65$ at $\alpha = 0.05$. The p-value is about 0.0003.

Since F $=$ 6.6 $>$ 2.65, the results are significant at the 5% significance level. One would reject the null hypothesis, concluding that there is strong evidence that the expected values in the three groups differ.

After performing the F-test, it is possible to carry out some "post-hoc" analysis of the group means. In this case the standard error is equal to

$\sqrt{\frac{10.6}{10} + \frac{10.6}{10} + \frac{10.6}{10} + \frac{10.6}{10}} = 2.06;$

it means that there is no evidence that results produced by `VolumeAgents`, `BBAgents` and `SLAgents` have different population means from each other, as their mean difference is comparable to the standard error. The same happens within `VolumeAgents`, `BBAgents` and `CoveredAgents`.

2.6.2 Linear regressions between unconstrained fitness of breeds and constrained fitness of breeds

In statistics, linear regression is an approach to modeling the relationship between a scalar dependent variable y and one or more explanatory variables denoted X. The case of one explanatory variable is called simple linear regression.

Simple linear regression is the least squares estimator of a linear regression model with a single explanatory variable. In other words, simple linear regression fits a straight line through the set of n points in such a way that makes the sum of squared residuals of the model (that is, vertical distances between the points of the data set and the fitted line) as small as possible.

The slope of the fitted line is equal to the correlation between y and x corrected by the ratio of standard deviations of these variables. The intercept of the fitted line is such that it passes through the center of mass (x, y) of the data points.

In this Section, linear regression is implemented on the different fitnesses of the same breeds of agents, obtained considering different frameworks: for this purpose the data considered are those collected in TABLE-1, TABLE-2, and TABLE-3.

Linear regressions on fitnesses collected in TABLE-1 and TABLE-2

In this Section the frameworks composed by only one breed of intelligent agents is compared to those composed by all categories of trading agents.

Briefly they are:

- `VolumeAgents` : comparison between frameworks defined by programs *AV.nlogo* and *A-B-C-SL-T-V.nlogo*; the linear regression is implemented with the data of the first column of TABLE-1 and the first column of TABLE-2.

- `trendAgents` : comparison between frameworks defined by programs *AT.nlogo* and *A-B-C-SL-T-V.nlogo*; the linear regression is implemented with the data of the second column of TABLE-1 and the second column of TABLE-2.

- `BBAgents` : comparison between frameworks defined by programs *AB.nlogo* and *A-B-C-SL-T-V.nlogo*; the linear regression is implemented with the data of the third column of TABLE-1 and the third column of TABLE-2.

- `SLAgents` : comparison between frameworks defined by programs *ASL.nlogo* and *A-B-C-SL-T-V.nlogo*; the linear regression is implemented with the data of the fourth column of TABLE-1 and the fourth column of TABLE-2

- `CoveredAgents` : comparison between frameworks defined by programs *AC.nlogo* and *A-B-C-SL-T-V.nlogo*; the linear regression is implemented with the data of the fifth column of TABLE-1 and the fifth column of TABLE-2.

For each case a linear regression, a t-test and an F-test (the same necessary for the ANOVA), are implemented with *R*.

R is a free software programming language and software environment for statistical computing and graphics.

The t-test and the F-test are useful to check the statistical consistency of the coefficient characterising the regression, i.e. the intercept and the slope of the line respectively.

The generic code that permits these analyses in *R* is the following.

```
x = c()
y = c()
plot(x, y, main="tradingAgents", xlab=" xaxis",ylab="yaxis",
volumes ", pch=19)
abline(lm(x~y), col="red")
model = lm(formula = x ~ y, x=TRUE, y=TRUE)
model
summary(model)
```

Where the commands `c()` defines the vector (in this case related to the independent and dependent variables); `abline()` draws the line of the regression; `lm` implements the linear regression; `summary` reports the results of the Fisher test.

CASE 1 : **VolumeAgents**

The results on the *R* console are can be seen in Figure 2.37.

The correlation coefficient is about 0.74: the two fitnesse are strongly correlated.

The coefficient of determination, denoted R^2 is equal to 0.54: it indicates how well data points fit the statistical model; in this case it is quite big.

The results of the F-statistics and the t-statistic, are significant in a confidence interval of 95%, since the p-value is equal to 0.014, that is less than 0.05; it means that the intercept and the slope are consistent with the model (and they are statistically different from zero).

Figure 2.38 reports the line of the regression.

```
Residuals:
    Min      1Q  Median      3Q     Max
-3.7713 -1.2859  0.1722  1.3314  2.4213

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  36.2210    11.6027   3.122   0.0142 *
volume2       0.4758     0.1537   3.096   0.0147 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.057 on 8 degrees of freedom
Multiple R-squared: 0.5451,     Adjusted R-squared: 0.4883
F-statistic: 9.587 on 1 and 8 DF,  p-value: 0.01475
```

Figure 2.37: VolumeAgents: results of the regression

CASE 2 : **trendAgents**

The results of the linear regression on fitness values of `trendAgents` can be seen in Figure 2.39.

The correlation coefficient is about 0.45: there is a quite strong dependence between the two fitnesses.

The results, of the F-test and the t-test, provide a p-value of 0.1894 and 0.997 respectively: it means that both the intercept and the slope of the line, are not statistically different from zero, considering a confidence interval of 95% (moreover the standard error is big with respect to the intercept and the slope of the curve).

Indeed the model does not fit well on a linear regression, and the R-squared is of about 0.2 (it is quite small).

The information showed in Figure 2.39, if traduced into plot, produce a line that almost coincides with the x-axis.

CASE 3 : **BBAgents**

The results on `BBAgents`, reported in the *R* console, can be seen in Figure 2.40.

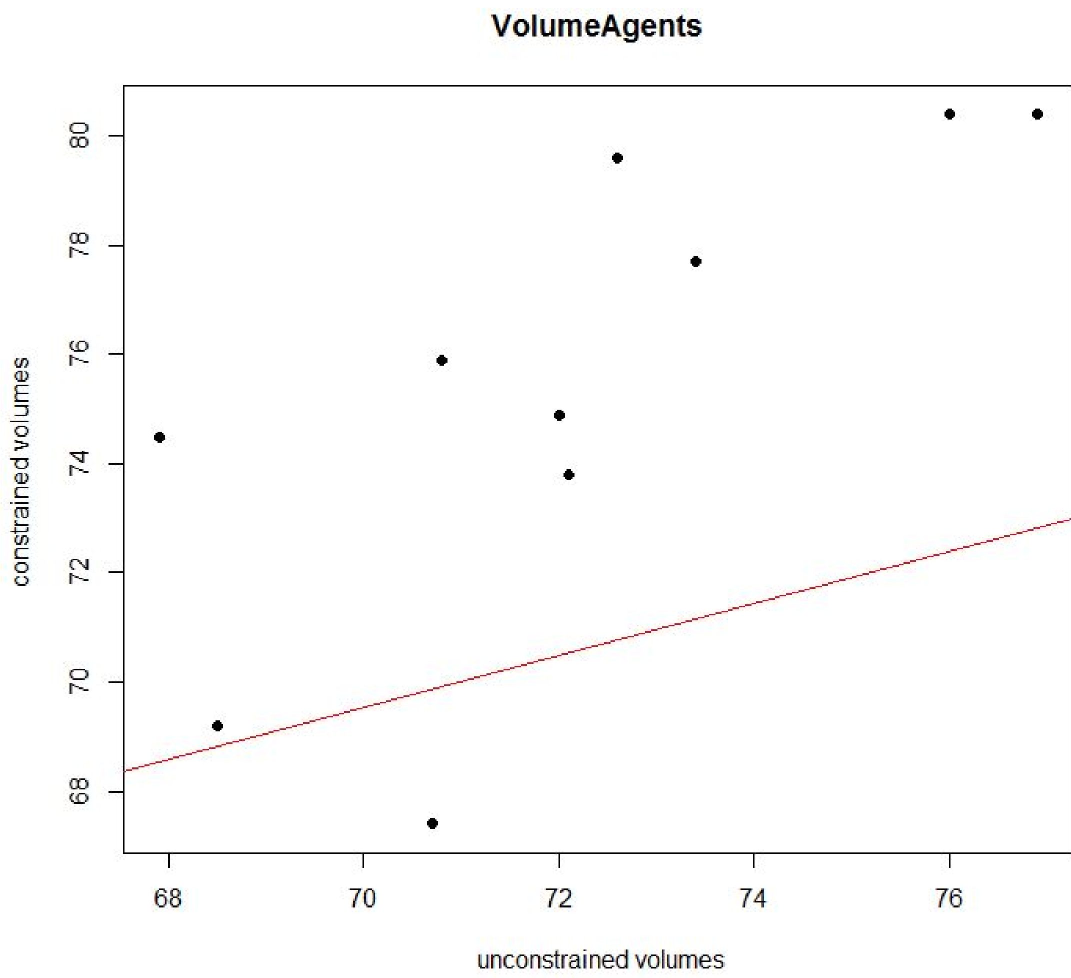

Figure 2.38: VolumeAgents: graph of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max 
-6.1508 -3.0987 -0.1099  3.4751  5.1359 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.1769    46.6690   0.004    0.997
trend2        0.9036     0.6300   1.434    0.189

Residual standard error: 4.089 on 8 degrees of freedom
Multiple R-squared: 0.2046,     Adjusted R-squared: 0.1051 
F-statistic: 2.057 on 1 and 8 DF,  p-value: 0.1894 
```

Figure 2.39: trendAgents: results of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max 
-4.2409 -2.5608  0.5474  1.2715  5.3636 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)  
(Intercept)  84.8791    29.8185   2.847   0.0216 *
BB2          -0.1654     0.4115  -0.402   0.6982  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 3.203 on 8 degrees of freedom
Multiple R-squared: 0.0198,     Adjusted R-squared: -0.1027 
F-statistic: 0.1616 on 1 and 8 DF,  p-value: 0.6982 
```

Figure 2.40: BBAgents: results of the regression

The correlation coefficient is about - 0.14: it is a low negative correlation between the two fitnesses considered for `BBAgnets`.

The t-statistics on the intercept of the curve, leads to reject the null hypothesis of an equal to zero intercept, in a confidence interval of 95% : in fact the p-value is equal to 0.02.

Instead, the F-statistics provides a curve slope not statistically different from zero: so the slope value reported (0.6982) is not significant (moreover it is lower than the standard error, equal to 3.203).

The percentage of model explained by the regression, is equal to -0.0198: it is very small.

The plot of the regression on `BBAgents` different levels of fitness, is showed in Figure 2.41.

The linear regression does not work particularly well on `BBAgents` fitness; moreover the correlation between the two fitnesses is quite small.

CASE 4 : **SLAgents**

The *R* console produced for `SLAgents` case, the results reported in Figure 2.42.

The correlation coefficient is equal to about 0.4 (quite strong dependence of the two fitnesses).

The R-squared is equal to 0.1658: indeed the percentage of model explained is 16.58%.

As in `BBAgents` case, the t-statistic provides a significantly different from zero intercept (p-value $= 0.019 < 0.05$); but the F-statistic leads to a non statistically significant slope (lower than the standard error).

The plot of the regression can be seen in Figure 2.43.

However, differently from Figure 2.43, the correct representation of the linear regression curve is not clear, given the fact that the value of the slope is not significant.

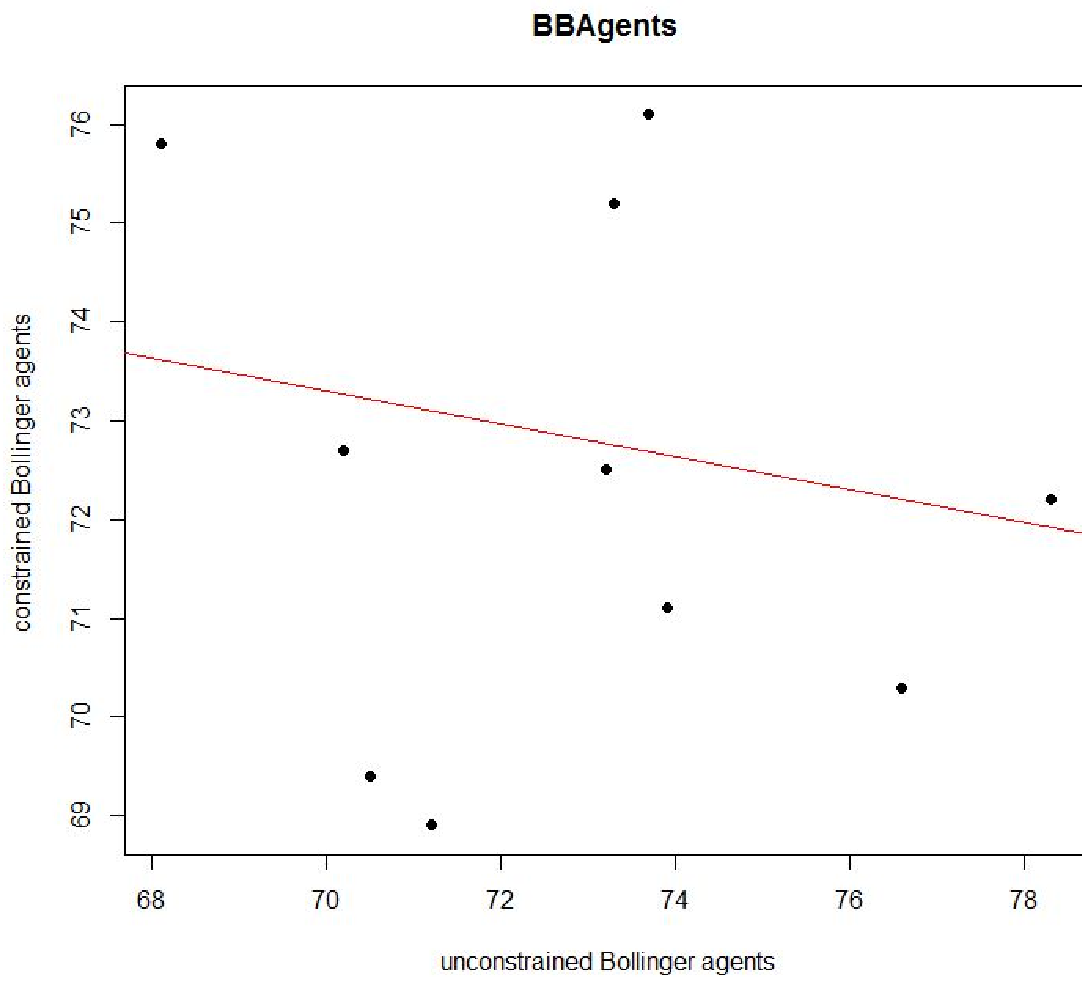

Figure 2.41: BBAgents: plot of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max 
-3.9508 -1.3497  0.3053  1.0442  3.3649 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)  
(Intercept)  51.6592    17.7605   2.909   0.0196 *
SL2           0.2849     0.2259   1.261   0.2428  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 2.235 on 8 degrees of freedom
Multiple R-squared: 0.1658,     Adjusted R-squared: 0.06157 
F-statistic:  1.59 on 1 and 8 DF,  p-value: 0.2428 
```

Figure 2.42: SLAgents: results of the regression

CASE 5 : **CoveredAgents**

The results reached through the *R* analysis are reported in Figure 2.44.

The correlation coefficient is equal to about -0.11 : it is quite low, indeed there is a low degree of dependence between the two fitnesses of `CoveredAgents`.

The intercept is statistically different from zero: the t-test provides a confidence level of 1 - 0.0297 = 0.9703 > 0.95.

The slope of the curve is not statistically different from zero: the F-statistic has a confidence interval of 1 - 0.7649 = 0.2351 < 0.95.

The R-squared is equal to 0.01163: less than the 2% of the total variation of outcomes is explained by the model.

The plot of the regression is showed in Figure 2.45.

Case 1 fits the best on linear regression, while case 2 fits the worst. The remaining cases, i.e. case 3, case 4 and case 5, have non statistically significant slopes, meaning that the lines of the regression are almost parallel to the x-axis.

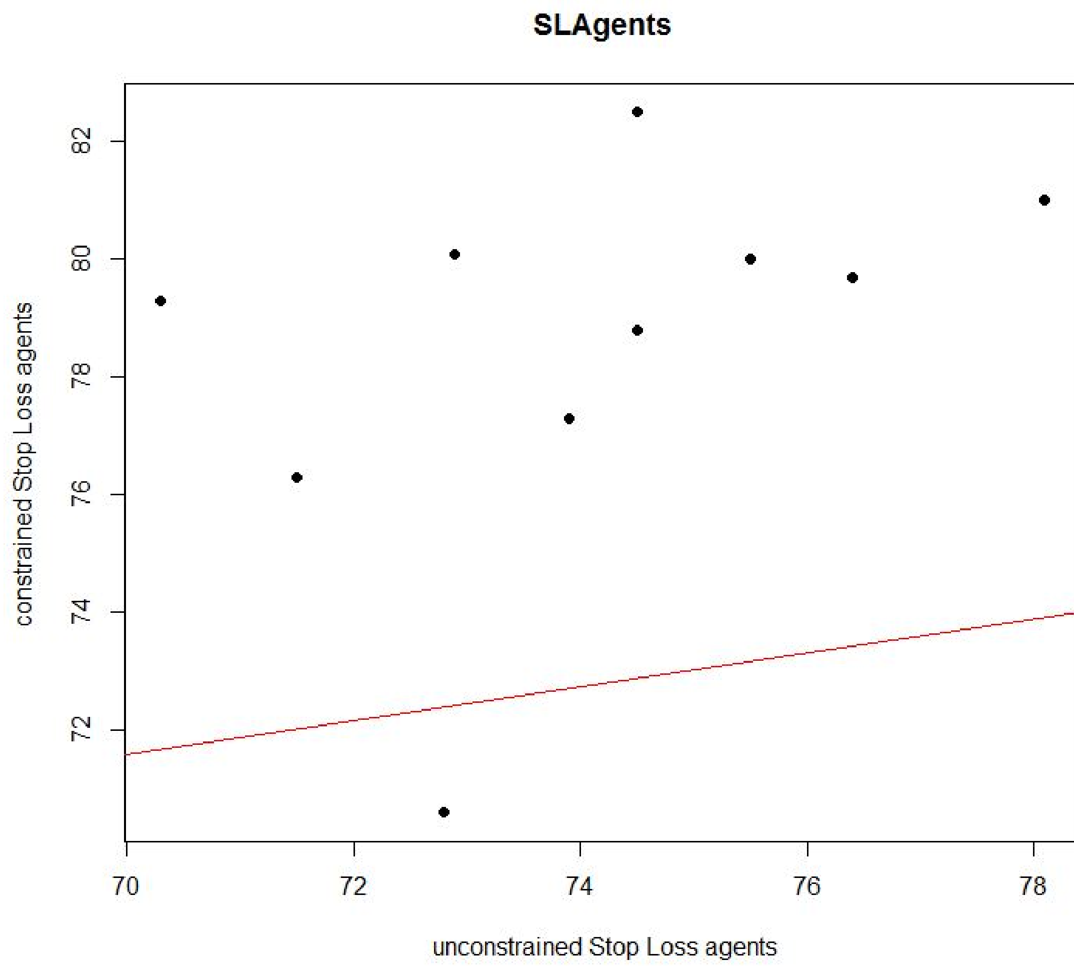

Figure 2.43: SLAgents: plot of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max
-7.4986 -1.4947  0.5379  1.4036  4.6400

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  79.9289    30.2650   2.641   0.0297 *
C2           -0.1286     0.4154  -0.309   0.7649
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.586 on 8 degrees of freedom
Multiple R-squared: 0.01183,    Adjusted R-squared: -0.1117
F-statistic: 0.09576 on 1 and 8 DF,  p-value: 0.7649
```

Figure 2.44: CoveredAgents: results of the regression

Linear regressions on fitnesses collected in TABLE-1 and TABLE-3

In this Section the frameworks composed by only one breed of intelligent agents is compared to those composed by two categories of trading agents.

Briefly they are:

- `trendAgents` : comparison between frameworks defined by programs *AT.nlogo* and *ABT.nlogo*; the linear regression is implemented with the data of the first column of TABLE-1 and the second column of TABLE-3.
- `BBAgents` : comparison between frameworks defined by programs *AB.nlogo* and *ABT.nlogo*; the linear regression is implemented with the data of the third column of TABLE-1 and the second column of TABLE-3.
- `SLAgents` : comparison between frameworks defined by programs *ASL.nlogo* and *A-C-SL.nlogo*; the linear regression is implemented with the data of the fourth column of TABLE-1 and the third column of TABLE-3.
- `CoveredAgents` : comparison between frameworks defined by programs *AC.nlogo* and *A-C-SL.nlogo*; the linear regression is implemented with the data of the fifth column of TABLE-1 and the fourth column of TABLE-3.

The statistical analyses of this Section are implemented in *R*, as happened in previous Section.

CASE 6.1 : **trendAgents constrained to BBAgents**

The *R* console results are reported in Figure 2.46; while the plot of the line of the regression is showed in Figure 2.47

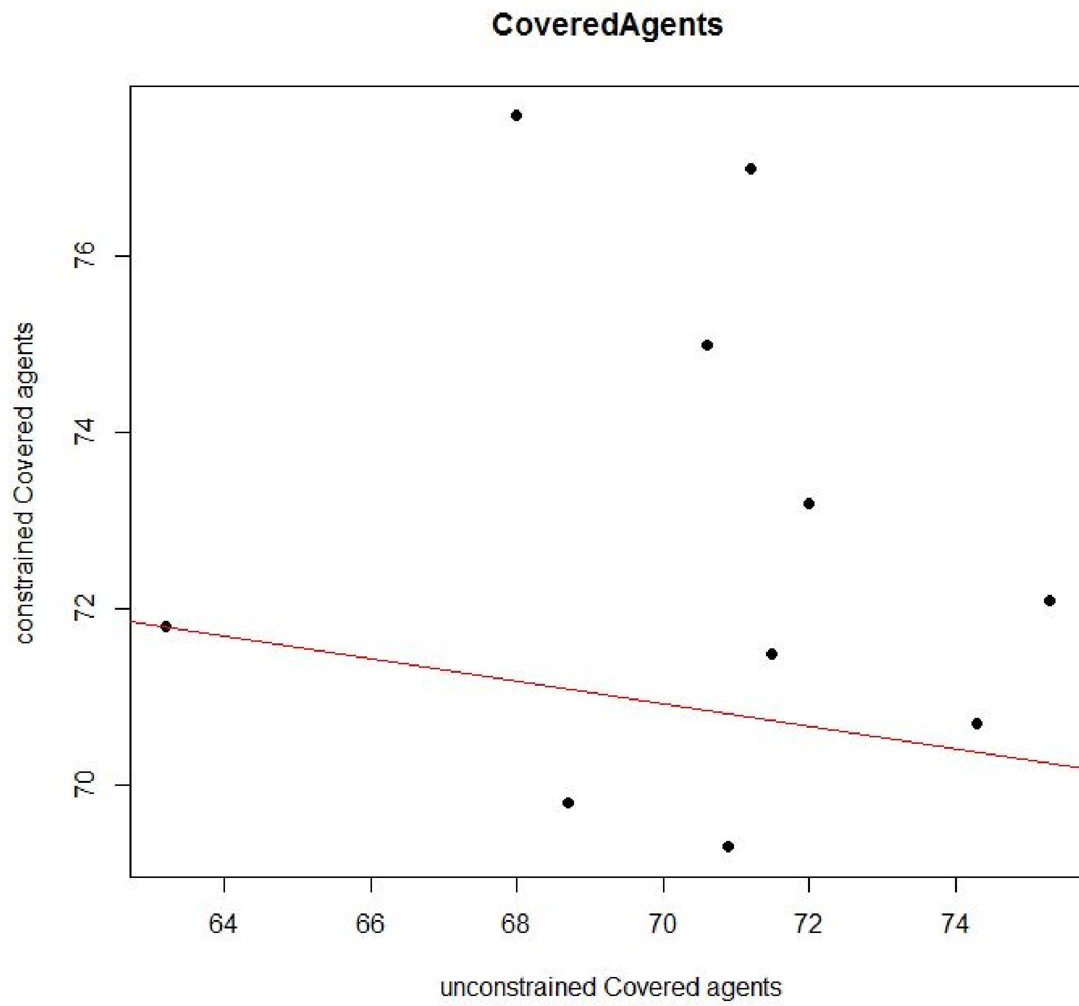

Figure 2.45: CoveredAgents: plot of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max 
-8.0022 -0.2318  0.3945  2.9686  3.9131 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  34.3013    21.2977   1.611    0.146
trend2        0.4835     0.3135   1.542    0.162

Residual standard error: 4.025 on 8 degrees of freedom
Multiple R-squared: 0.2292,     Adjusted R-squared: 0.1328 
F-statistic: 2.379 on 1 and 8 DF,  p-value: 0.1616 
```

Figure 2.46: trendAgents constrained to BBAgents: results of the regression

Looking at Figure 2.46, we can note that both intercept and slope are not statistically different from zero, since both t-test and F-test provide p-values bigger than 0.05 (0.146 and 0.162 respectively).

As happened in CASE 2, the linear regression is not a consistent model for such kind of data, since the standard error is too big.

The two fitnesses considered are positively correlated, with correlation coefficient equal to -0.4787.

CASE 6.2 : **BBAgents constrained to trendAgents**

The *R* console results are reported in Figure 2.48; while the plot of the line of the regression is showed in Figure 2.49

The correlation coefficient is equal to 0.078: it means that the two fitnesses are weakly dependent.

The intercept is statistically different from zero (t-test with p-value 0.0402), while the slope is not statistically different from zero (F-test with p-value 0.8298).

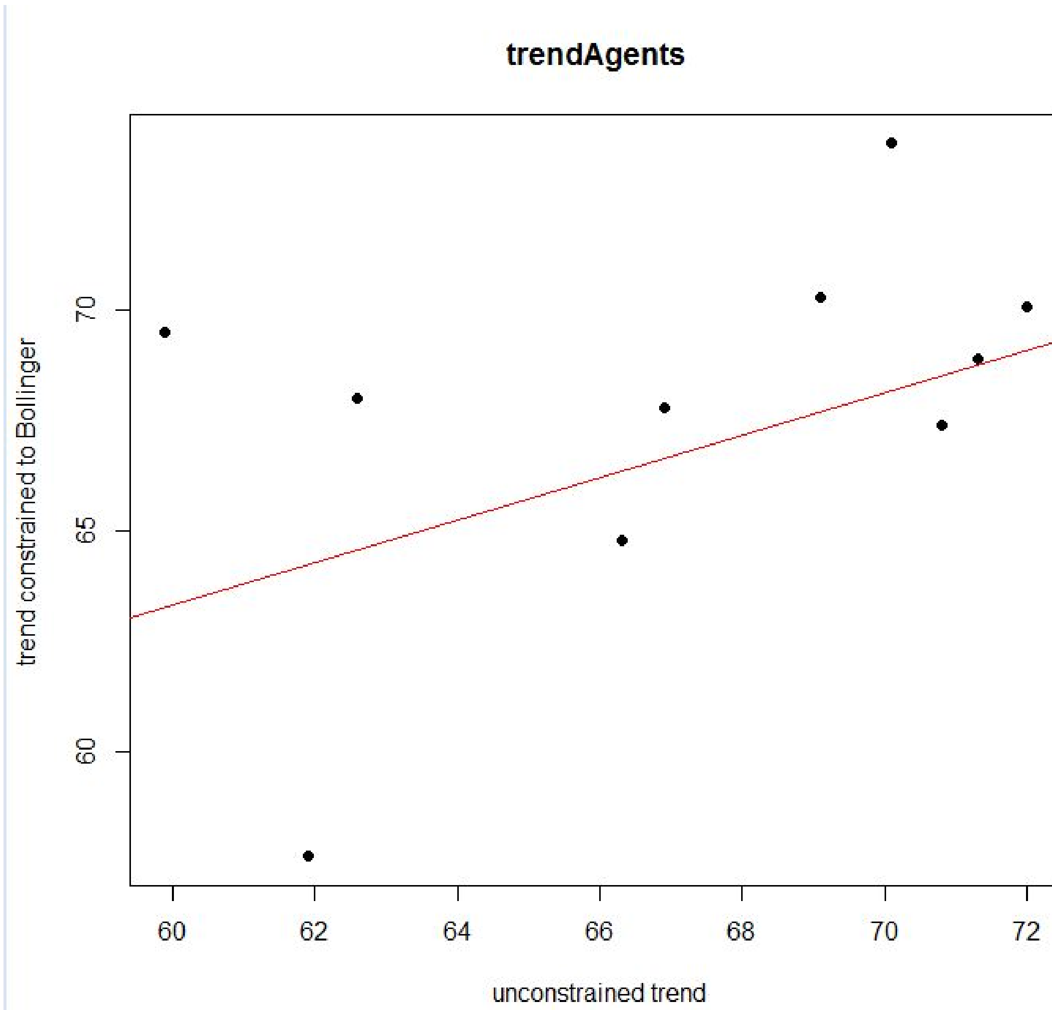

Figure 2.47: trendAgents constrained to BBAgents: plot of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max
-4.5910 -1.9333  0.1165  0.7578  5.6090

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  80.1773    32.7851   2.446   0.0402 *
BB2          -0.1066     0.4802  -0.222   0.8298
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.225 on 8 degrees of freedom
Multiple R-squared: 0.006127,   Adjusted R-squared: -0.1181
F-statistic: 0.04932 on 1 and 8 DF,  p-value: 0.8298
```

Figure 2.48: BBAgents constrained to trendAgents: results of the regression

CASE 7.1 : **SLAgents constrained to CoveredAgents**

The analysis with *R* reached the results reported in Figure 2.50 ; while the plot of the regression can be seen in Figure 2.51 .

The two fitnesses are almost independent, since the correlation coefficient is equal to 0.0178 .

The intercept is significant, as proved by the t-test with a p-value of 0.000757; the slope is very close to zero, and not statistically different from zero, since the F-statistic reached a p-value of 0.960979.

As showed in Figure 2.51, the line is almost parallel to the x-axis.

CASE 7.2 : **CoveredAgents constrained to SLAgents**

The results related to this case, given by *R*, are showed in Figure 2.52 .

The plot of the regression can be seen in Figure 2.53 .

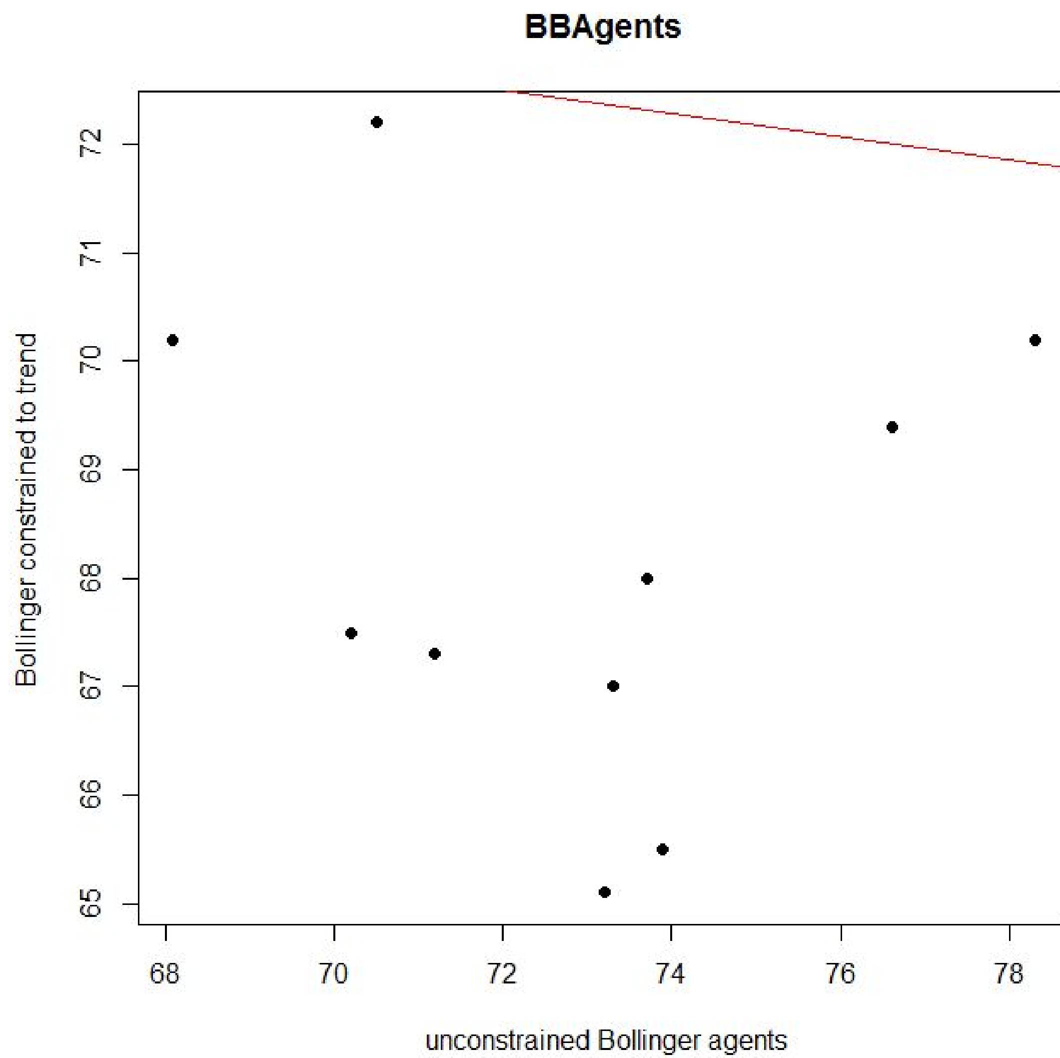

Figure 2.49: BBAgents constrained to trendAgents: plot of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max 
-3.7144 -1.2482  0.1548  1.2379  4.0240 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 74.755282  14.191141   5.268 0.000757 ***
SL2         -0.009474   0.187683  -0.050 0.960979    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 2.447 on 8 degrees of freedom
Multiple R-squared: 0.0003184,  Adjusted R-squared: -0.1246 
F-statistic: 0.002548 on 1 and 8 DF,  p-value: 0.961
```

Figure 2.50: SLAgents constrained to CoveredAgents: results of the regression

The coefficient of the regression is equal to about 0.2586.

Both the t-test and the F-test, provide a not statistically different from zero coefficients of the regression (p-values of 0.429 and 0.471 respectively).

It means that the plot of Figure 2.53, is not consistent, since the line should coincide with the x-axis.

2.7 Technical achievements

Commenting upon results of Sections 2.6.1 and 2.6.2, we can say :

1. `trendAgents` are the breed with the lowest fitness if considered alone in the market; but their influence is much less effective if we consider them together with other breeds.

   Their fitnesses in different frameworks have a quite big correlation (of about 0.45), but the linear regression seems not to be the right model to explain their variation (maybe higher degree models fits better for the fitness of this breed).

2. `VolumeAgents` : they are the only breed whose effect on different market structures, can be approximatively predicted, using a linear regression (moreover they have the greatest correlation coefficient).

3. The constrained fitnesses are always bigger than the unconstrained ones, except in the case of `BBAgents` constrained by `trendAgents`.

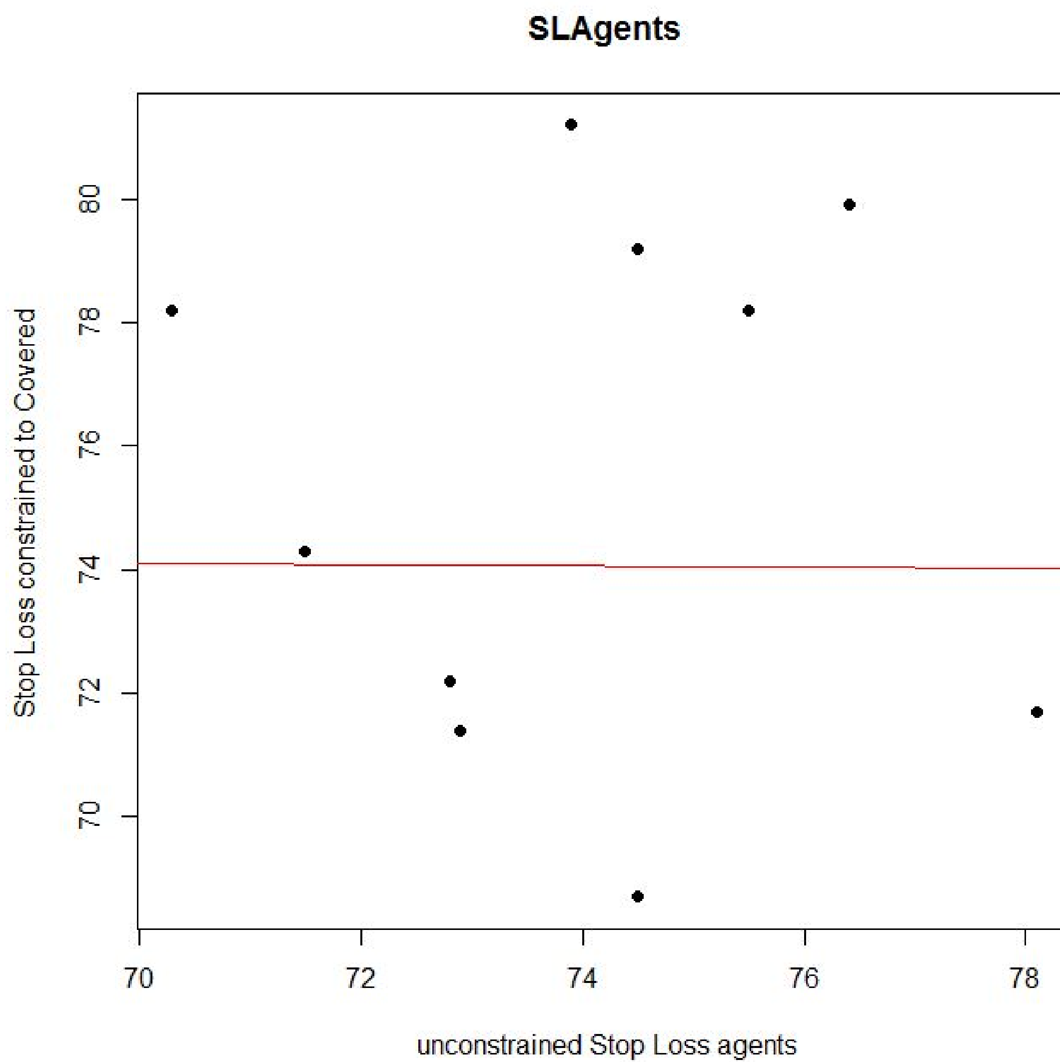

Figure 2.51: CoveredAgents constrained to SLAgents: plot of the regression

```
Residuals:
    Min      1Q  Median      3Q     Max
-7.0258 -1.1254  0.2047  1.6319  4.6970

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  36.9824    44.3681   0.834    0.429
C2            0.4715     0.6227   0.757    0.471

Residual standard error: 3.485 on 8 degrees of freedom
Multiple R-squared: 0.06688,    Adjusted R-squared: -0.04975
F-statistic: 0.5734 on 1 and 8 DF,  p-value: 0.4706
```

Figure 2.52: CoveredAgents constrained to SLAgents: results of the regression

4. The difference in the average fitness (tested with ANOVA) of the samples of the different breeds, is statistically significant: between `trendAgents` and `VolumeAgents, BBAgents, SLAgents` ; between `CoveredAgents` and `SLAgents`.

5. The fitness of `SLAgents` is the biggest compared to that of the other breeds in the same conditions; the correlation between fitnesses of the programs *ASL.nlogo* and *A-B-C-SL-T-V.nlogo* (with `SLAgents` optimal parameters search), is quite big (about 0.4); but the linear regression does not work on those data.

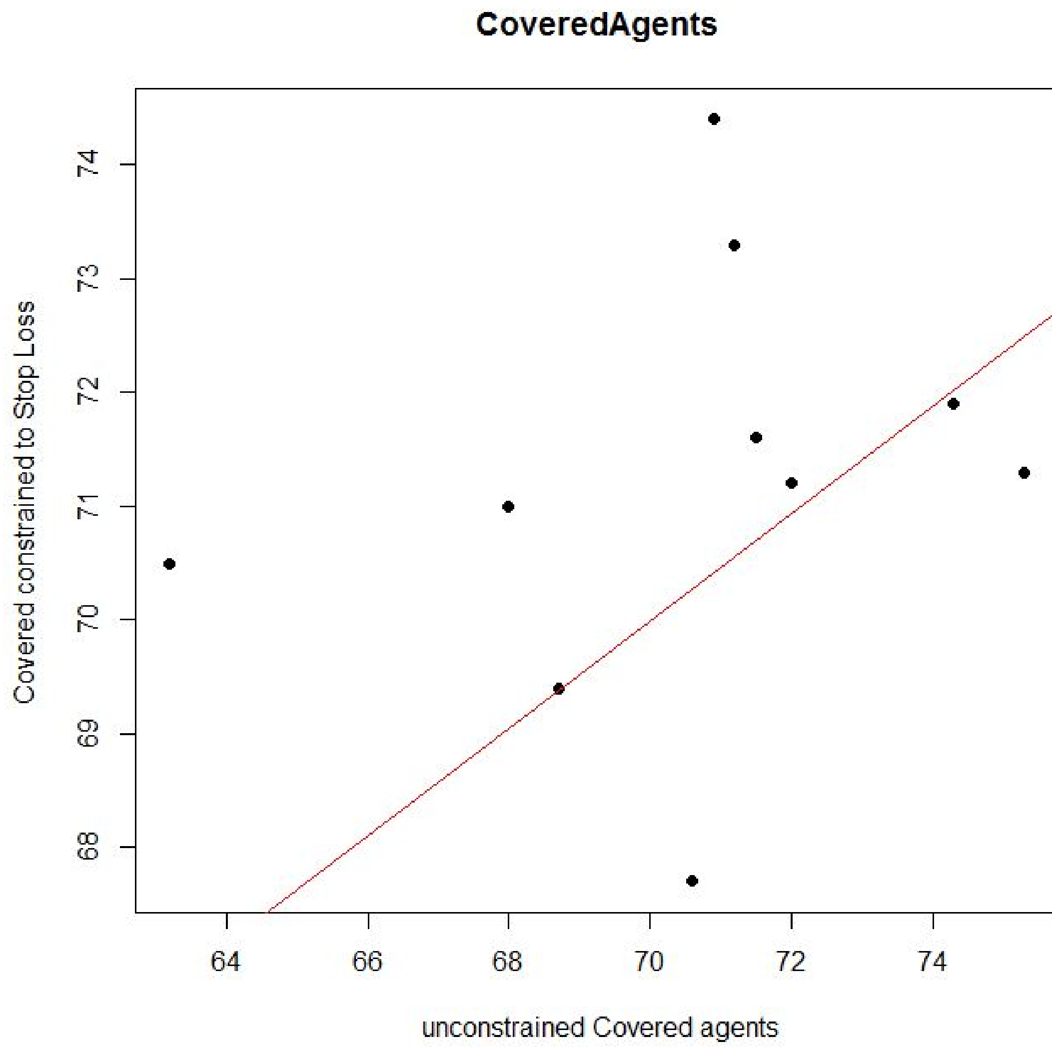

Figure 2.53: CoveredAgents constrained to SLAgents: plot of the regression

Chapter 3

Conclusions

Summarizing the work presented in the two parts:

About the first part of the thesis: the 'methodological' one.

The objective of the first chapter is to study the *BehaviorSearch* characteristics, through the analyses of some *NetLogo* programs.

For this purpose it has been given a preliminary description of the parts composing the *BehaviorSearch Experiment Editor*; then several searches have been implemented, in order to compare the effectiveness of the algorithms available in *BehaviorSearch*: they are *StandardGA* (genetic algorithm), *RandomSearch*, *MutationHillClimber*, and *SimulatedAnnealing*.

The fitness function is defined as the distance between the value of `heigh` founded during the search (can be considered as the third dimension), and the maximum value of `heigh`, present in the search space.

The fitness results reported (mean fitness values, intervals), are related to one-hundred trials, where each trial determines a different search, characterised by a seed.

The *NetLogo* programs used to test *BehaviorSearch* algorithms, are:

- *localH,localH2, localH3.* They create simple three-dimensional spaces of size 16810000, they can be represented graphically as surfaces.

  1. For what concern *localH*: it is a completely random and non smoothed surface; to complete the analysis *BehaviorSearch*, lasts 3-4 minutes for every algorithm..

     StandardGA is not efficient with custom parameters (fitness is often lower than -0.2), while it is significantly bigger increasing the `population-size` input parameter (for example setting it to 200 instead of 50), or set-

ting `steady-state-replace-worst` as `population-model` (fitness often bigger than -0.1, sometimes equal to zero).

The *RandomSearch* algorithm, works well on *localH*, providing a fitness always included in the interval [-0.04 , 0].

The `SimulatedAnnealing` and the `MutationHillClimber` algorithms lead to fitnesses included in the intervals [-0.15 , 0] and [-0.1 , -0.01] respectively.

2. About *localH2*: it is a smoothed surface; *BehaviorSearch* lasts 3-4 seconds for every algorithm to complete the optimal search.

   The algorithms *StandardGA*, *MutationHillClimber* and *SimulatedAnnealing* fit well on this kind of search space, and the fitness is often zero. In particular, with the *StandardGA*, it is possible to get an always zero fitness by increasing both `population-size` and the number of model runs (but the software requires more time to complete the analyses).

   The `RandomSearch` algorithm is significantly less efficient than the others, and the fitnesses reached are included in the interval [-0.6 , -0.12] .

3. About *localH3*: it is a smoothed surface, created using the function sin().

   The algorithms effectivenesses are the same of those observed for *localH2*: only the *RandomSearch* provides a fitness that is rarely zero.

- *localH1.1, localH2.2*. They are constructed as *localH* and *localH2*, with the difference that the search space created is larger: size of 65610000 for *localH1.1*, size of 630010000 for *localH2.2* (instead of 16810000 for both *localH* and *localH2*).

  1. About *localH1.1*: the time due to complete the *BehaviorSearch* analysis is about four times greater than that of *localH*.

     The *StandardGA* with custom parameters is more effective than in *localH* case, and the average fitness is about -0.05. Increasing the population size leads to an increase in the fitness value, as observed for *localH* (for example with `population-size`= 200 the average fitness is about -0.02).

     The *RandomSearch* can lead to an average fitness of -0.02: for *localH1.1*, it is slightly more efficient than *MutationHillClimber* and *SimulatedAnnealing* algorithms, that can attain an average fitness of about -0.06.

2. About *localH2.2*: *BehaviorSearch* requires a time to complete the search, that is 30-40 times greater than that of its predecessor *localH2* (it is proportional to the size of the search space analysed).

   The algorithms *StandardGA*, *MutationHillClimber*, and *SimulatedAnnealing* produce an always zero fitness, within few model runs (300-500 out of 5000).

   The *RandomSearch* algorithm is little faster, but reaches a worse fitness, included in the interval [-0.2 , 0].

- *MultivariateLocalH.* It creates a six dimensional space; its shape is that of an hypersphere: although it cannot be represented graphically, it is quite simple to find its critical points, and it is smoothed. The search space size is about 2×10^{24} . The maximum is equal to 60010; the *BehaviorSearch* analysis lasts 3-4 seconds only.

  the average fitnesses observed are:

  1. -5000 for *StandardGA* with custom parameters;

     -3000 setting as population model `steady-state-replace-worst`;

     -6000 setting as population model `steady-state-replace-random`;

     -2000 by changing only `population-size`= 10

     -1000 setting `population-size`= 10 and population model equal to `steady-state-replace-worst`.

  2. -16000 for the *RandomSearch* algorithm.

  3. -1500 for the *MutationHillClimber* algorithm.

  4. -1600 for the *SimulatedAnnealing* algorithm.

More studies can be made implementing several search analyses on more complex (multidimensional) search spaces, in order to identify other relevant characteristics of the *BehaviorSearch* algorithms.

About the second part of the thesis: the 'empirical' one.

The second chapter exploits the genetic algorithm in an stock exchange agent based simulation, to investigate the investment behavior of some trading agents breeds, and their effect on the market.

The simulation is implemented using the *NetLogo* program, while the analysis on the agents optimal parameters is performed using *BehaviorSearch* software tool.

The core structure (basic framework) of the stock exchange simulation is composed by one-hundred `RandomAgents` and one `arbitrageur` : the first breed decides whether to buy or sell depending on randomness, while the second breed is necessary to realign the artificial market price to a real stock price, i.e. *Ftse All Share*; the real price, has one minute frequency, and is represented by a time series of 10095 realisations.

Then, there are five breeds of 'intelligent' agents that invest in the artificial market price following different strategies.

They are:

1. `VolumeAgents` : their strategy is based on trading volumes.

2. `trendAgents` : their strategy is based on the value of a moving average, different for any `trendAgent`.

3. `BBAgents` : they invest according to the market price level with respect to Bollinger bands.

4. `SLAgents` : their investment behavior is based on a Stop Loss strategy, that implies the use of call options, calculated through the *Black and Scholes* formula.

5. `CoveredAgents` : their strategy is focused on both the value of a moving average, and the trade of call or a put option, calculated through the *Black and Scholes* formula.

Different compositions of trading agents in the market, define different market structures; the *NetLogo* programs that define such market structures are:

- *AV.nlogo* : basic framework plus `VolumeAgents` breed.

- *AT.nlogo* : basic framework plus `trendAgents` breed.

- *AB.nlogo* : basic framework plus `BollingerAgents` breed.

- *ASL.nlogo* : basic framework plus `SLAgents` breed.

- *AC.nlogo* : basic framework plus `CoveredAgents` breed.

- *ABT.nlogo* : basic framework plus both `BBAgents` and `trendAgents` breeds.

- *A-C-SL.nlogo* : basic framework plus both `SLAgents` and `CoveredAgents` breeds.

- *A-B-C-SL-T-V.nlogo* : basic framework plus all categories of trading agents.

The *BehaviorSearch* analyses, using genetic algorithm (*StandardGA*), are implemented for each *NetLogo* program, in order to determine the optimal parameters (sliders), and the fitness value related to the single breed, whose effect is indeed tested in different frameworks (market structures).

The fitness is defined as the average difference per-price, between artificial and real markets, and it must be minimized.

For each market structure, ten fitness values, and ten values for the optimal parameters, have been collected per trading agent breed, distinguishing each trial by a different seed.

Such values are collected in the tables of Sections .

Then two statistical tests have been performed to compare the fitnesses collected:

1. *ANOVA* (one way). It is used to analyse the differences between group means: one way ANOVA provides a statistical test (Fisher test) of whether or not the means of several groups are equal.

   The different groups tested are represented by the fitnesses collected for `VolumeAgents`, `trendAgents`, `BBAgents`, `SLAgents`, `CoveredAgents` in the frameworks defined by the programs *AV.nlogo*, *AT.nlogo*, *AB.nlogo*, *ASL.nlogo*, *AC.nlogo*.

   The Fisher test has produced a statistically significant difference between the means of such groups, providing a confidence interval of 0.9997 (p-value of 0.0003). The standard error is equal to 2.06: there is evidence that results provided by `VolumeAgents` and `trendAgents`, `BBAgents` and `trendAgents`, `CoveredAgents` and `trendAgents`, `SLAgents` and `trendAgents`, `SLAgents` and `CoveredAgents` have different population means from each other.

2. *Linear regression* (simple). It is an approach to modeling the relationship between two scalar variables (explanatory on the x-axis, dependent on the y-axis), through the least squares estimator.

   For what concern the fitnesses collected, for each breed of agents is implemented a linear regression in which: the explanatory variable is represented by the unconstrained fitnesses, i.e. those reached from the frameworks created by the programs *AV.nlogo*, *AT.nlogo*, *AB.nlogo*, *ASL.nlogo*, *AC.nlogo*; the dependent variable is represented by the constrained fitnesses, i.e. those collected from the frameworks defined by the programs *ABT.nlogo*, *A-C-SL.nlogo*, *A-B-C-SL-T-V.nlogo*.

   The results can be summarised as follows;

(a) for `VolumeAgents` : with explanatory variable taken from *AV.nlogo*, and dependent variable taken from *A-B-C-SL-T-V.nlogo*, both the coefficients of the regression (intercept and slope) are statistically different from zero. This is the only case that shows a valid linear relationship between the variables considered;

(b) for `trendAgents` : considering the fitnesses from *AT.nlogo* framework as explanatory variable and those of *A-B-C-SL-T-V.nlogo* as dependent one, both coefficient of the regression are not statistically different from zero, meaning that the linear regression model does not work.

The same result hold, considering as dependent variable the group of fitnesses obtained from the framework defined by the program *ABT.nlogo*.

(c) for `BBAgents` : considering the fitnesses from *AB.nlogo* framework as explanatory variable and those of *A-B-C-SL-T-V.nlogo* as dependent one, the intercept is statistically different from zero but the slope is not statistically different from zero.

The same result hold, considering as dependent variable the group of fitnesses obtained from the framework defined by the program *ABT.nlogo*.

It means that in both cases the linear regression model does not work.

(d) for `SLAgents` : considering the fitnesses from *ASL.nlogo* framework as explanatory variable and those of *A-B-C-SL-T-V.nlogo* or *A-C-SL.nlogo* as dependent one, the results are equal to those observed for `BBAgents`.

(e) for `CoveredAgents` : considering the fitnesses from *AC.nlogo* framework as explanatory variables and those of *A-B-C-SL-T-V.nlogo* or *A-C-SL.nlogo* as dependent ones, the results are equal to those observed for `BBAgents` and `SLAgents`.

However regressions are strongly influenced by the magnitude of the samples analysed: considering bigger samples can lead to more interesting and significant results, compared to those reported in chapter 2.

In general it is very difficult to investigate the true behavior of agents in the market, knowing only the price trend, since there are a lot of random variables that are difficult to predict.

My work has the ambitious objective of approaching a realistic search of such investment behavior through an agent based simulation.

In this direction further analyses can be made by considering other market structures (for example with more agents), bigger samples for the statistical tests (that can give more statistical significance), and other statistical tools to analyse the data. In this way it should be possible to reach interesting results.

For example considering the market structure of the program *A-B-C-SL-T-V.nlogo*, we can investigate the optimal risk-free rate, and compare it with reality; moreover always in such structure, if we allow the number of `RandomAgents` to variate in the interval [10 1000] , we can get fitness values that are about one half of those obtained in my analyses.

While to improve my work on *NetLogo* it should be possible to put beside the artificial market, an artificial option market, through the creation of other breeds of trading agents that are able to negotiate Call and Put options, assuming long and short positions.

However increasing the degree of complexity of the simulation program, it is always more difficult to associate the correct statistics to the framework created, and to interpret results.

Further developments of my work might lead to a new kind of professional tool, useful to investigate the stock exchange strategies, using both *genetic algorithms* and *ABM*.

Bibliography

Axtell, R. L. (2006). *COORDINATION IN TRANSIENT SOCIAL NETWORKS: AN AGENT-BASED COMPUTATIONAL MODEL OF THE TIMING OF RETIREMENT ROBERT L. AXTELL AND JOSHUA M. EPSTEIN*. In «Generative social science: Studies in agent-based computational modeling», p. 146.

Ferraris, G. and Lamieri, M. (2004). *DRAFT ART Artificial Reasoning Toolkit How To Use*. In «Doctoral Dissertation».

Holland, J. H. (1992). *Genetic algorithms*. In «Scientific american», vol. 267(1), pp. 66–72.

Stonedahl, F. J. and Adviser-Wilensky, U. J. (2011). *Genetic algorithms for the exploration of parameter spaces in agent-based models*. In «Doctoral Dissertation».

Wright, A. H. (2011). *The exact schema theorem*. In «arXiv preprint arXiv:1105.3538».