University of Turin

SCHOOL OF MANAGEMENT AND ECONOMICS
Master's degree in Quantitative Finance and Insurance

An Agent-based Market Maker Model using Genetic Algorithms

Supervisor:
Prof. Pietro Terna
Opponent:
Prof. Sergio Margarita

Candidate:
Manuel Mannara

Academic Year
2014/2015

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Terna, this work would never have materialized without his constant guidance and willingness to assist me. He gave me unending encouragement and suggestions, allowing me to learn something new in every step of the way.

I also want to acknowledge Professor Margarita for his helpful contribute.

On a personal note, I want to give a huge shout-out to my parents and my grandmother, who laid the foundation of my education, constantly believing in me.

Additionally I thank my friends and classmates for their unconditional support and for always being by my side.

Special thanks to my girlfriend for being so understanding and her essential love.

Last but not least I want to thank my twin brother, for continuously inspiring me to achieve my dreams and goals.

Contents

1 Agent-based models **11**
- 1.1 Main characteristics . . . 12
- 1.2 Why to use Agent-based models . . . 13
- 1.3 Uses and weaknesses . . . 15

2 Genetic Algorithms **17**
- 2.0.1 Limitations . . . 19

3 BehaviorSearch **21**
- 3.0.2 What is BehaviorSearch? . . . 22
- 3.0.3 Goals and features . . . 22
- 3.0.4 How it works? . . . 24

4 NetLogo model **35**
- 4.1 NetLogo program . . . 35
  - 4.1.1 Variables . . . 36
  - 4.1.2 NetLogo structure . . . 37
  - 4.1.3 Interface . . . 38
  - 4.1.4 Code . . . 40
  - 4.1.5 Info . . . 41
- 4.2 CDA basic model . . . 42
  - 4.2.1 Code . . . 43
  - 4.2.2 Interface . . . 48
- 4.3 Market maker model 0 . . . 49
  - 4.3.1 Code . . . 50
  - 4.3.2 Interface . . . 57
- 4.4 Market maker model 01 . . . 58
  - 4.4.1 Code . . . 58
  - 4.4.2 Interface . . . 67
- 4.5 Market maker model 02 . . . 68
  - 4.5.1 Code . . . 68

4.5.2 Interface . . . 78
4.6 Market maker model 03 . . . 79
4.6.1 Code . . . 79
4.6.2 Interface . . . 92
4.7 Market maker model 04 . . . 93
4.7.1 Code . . . 93
4.7.2 Interface . . . 104
4.7.3 Market maker model 04.01 . . . 105
4.7.4 Market maker model 04.02 . . . 106
4.8 Market maker model 05 . . . 107
4.8.1 Code . . . 107
4.8.2 Interface . . . 119
4.8.3 Market maker model 05.01 . . . 120
4.8.4 Market maker model 05.02 . . . 121
4.9 Market maker model 06 . . . 122
4.9.1 Code . . . 122
4.9.2 Interface . . . 136
4.9.3 Market maker model 06.01 . . . 137
4.9.4 Market maker model 06.02 . . . 138

5 NetLogo and R **139**
5.1 NetLogo-Rserve-Extension . . . 139
5.1.1 Primitives . . . 140
5.2 NetLogo market maker model 03_R . . . 141
5.2.1 Code . . . 141
5.2.2 Interface . . . 153
5.3 NetLogo market maker model 05_R . . . 154
5.3.1 Code . . . 154
5.3.2 Interface . . . 166
5.4 Results . . . 167

6 Simulation and BehaviorSearch experiments **171**
6.1 Maximizing market-maker-profit . . . 172
6.1.1 Experiment 1: Initial-nStocks and update-bid-ask? . . . 172
6.1.2 Experiment 2: Initial-Cash, Initial-nStocks and update-bid-ask? . 176
6.1.3 Experiment 3: Initial-Cash, Initial-nStocks update-bid-ask? and bid-ask-spread . . . 179
6.2 Bid-ask spread: market-maker-profit vs bid-ask-profit . . . 182
6.2.1 Experiment 1: nRandomInvestors . . . 183
6.2.2 Experiment 2: nRandomInvestors p-of-acting . . . 188

6.2.3 Experiment 3: market-maker-profit,bid-ask-spread and 'update-bid-ask?' 191
6.2.4 Experiment 3.1: market-maker profit, bid-ask-spread and 'update-bid-ask?' 195
6.2.5 Experiment 4: bid-ask-profit vs market-maker-profit 197
6.2.6 Experiment 5: maximizing bid-ask-profit, bid-ask-spread and standard deviation 201
6.2.7 Experiment 6: bid-ask-spread for market-maker-profit and bid-ask-profit 208

Introduction

Market-makers serve a crucial role in financial markets by providing liquidity to facilitate market efficiency and functioning, a market maker is a dealer firm that takes risk of holding a certain number of shares of a particular security in order to facilitate trading in that security. Market makers play an important role in financial markets, as they provide to decrease liquidity risk and ease the frequency at which participants can enter or exit the market.

Market makers are willing to buy at bid price and sell at ask price, they can make profits by buying low and selling high through bid-ask spreads, the amount by which the ask price exceeds the bid.

The market maker spread is the difference between the bid and ask price posted by the market maker for a security. It represents the potential profit that the market maker can make from this activity, compensate it for the risk of market making. While the spread between the bid and ask is only a few cents, market makers can make sizable profits by executing thousands of trades in a day. However, these profits can be nullified by volatile markets if the market maker is caught on the wrong side of the trade. Market maker spreads widen during volatile market periods because of the increased risk of loss. The wider spreads are a way to discourage investors from trading during such periods.

In this work we develop an agent-based market maker model through NetLogo program and we will explore the optimal decisions using genetic algorithms

In the first chapters (chapter 1 and 2) we introduce the main issues of this thesis, agent-based models and genetic algorithms.

Chapter 3 deals with BehaviorSearch, a software tool to help with automating the exploration of agent-based model, by using genetic algorithms and other heuristic techniques to search the parameter-space.

Chapter 4 is focused on the development of the NetLogo model, we briefly introduce this program and then we implement the market maker model, starting from the simplest version we evolve the model making it more detailed and sophisticated. In this part of the thesis we illustrate carefully the NetLogo code and the logic of the model.

In chapter 5 we interface NetLogo program with the statistical analysis software R through NetLogo R-serve extension, in this way we can create R-variables with values from NetLogo variables.

The last chapter is regarding experiments with the tool BehaviorSearch, the use of genetic algorithms allow us to find the optimal values for parameters whenever the search space is significantly large, and we can explore what variables cause specific outputs and behaviors.

The main reason that led me to explore these fields is the strong interest for the interaction between agent-based modeling and financial markets, especially with the use of genetic algorithms to find optimal solutions.

Chapter 1

Agent-based models

An agent-based model is a formal tool for scientific inquiry. An agent-based model is one of a class of computational models for simulating the actions and interactions of autonomous agents, both individual or collective entities such as organizations or groups. Agent-based Model are typically implemented as computer simulations to test how changes in individual behaviors will affect the system behavior.

Agent-based model combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming.

Citing Axtell and Epstein (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, that is, arise 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 agents heterogeneity, boundedly rational behavior, nonequilibrium dynamics, and spatial processes. A particularly natural way to implement agent-based models is through so call object-oriented programming.

1.1 Main characteristics

In agent-based modeling, a system is developed as a collection of autonomous decision-making entities called agents. Agents' behaviors are determined by rules and the system evolves over time.

Models in social sciences are simplified representation of reality, there are two methods to develop them.

- verbal argumentation
- mathematical equations, with statistics and econometrics.

We have also a third method (Gilbert and Terna, 2000) to build models: computer simulation, mainly agent based. This method can mix the description skills of verbal argumentation and the ability to calculate the effects of different situations and hypotheses.

We can consider agent-based models as useful tools to produce knowledge, citing Axelrod and Tesfatsion (2005):

> Simulation in general, and ABM in particular, is a third way of doing science in addition to deduction and induction. Scientists use deduction to derive theorems from assumptions, and induction to find patterns in empirical data. Simulation, like deduction, starts with a set of explicit assumptions. But unlike dedu ction, simulation does not prove theorems with generality. Instead, simulati on generates data suitable for analysis by induction. Nevertheless, unlike typical i nduction, the simulated data come from a rigorously specified set of assu mptions regarding an actual or proposed system of interest rather than direct measurements of the real world. Consequently, simulation differs from standard deduction and induction in both it s implementation and its goals. Simulation permits increased understanding of systems through controlled computational experiments

Agent-based modeling is seen as another approach to scientific analysis, it is an intermediary way lying between induction and deduction.

1.2 Why to use Agent-based models

Agent-based Models can improve economic analyses by solving some of the most important limitations of the traditional modeling tools used in the discipline.

One critique of the common practice of research in economics is the separation of the approaches to it in theoretical and empirical works. In economics is common to find evidence of separation between theory and data. A over-strict separation leads theoretical developments based mainly on theoretical consistency. while evidence from empirical studies is often not deeply studied because it can not be harmonize with pre-existing theory. The lack of flexibility in the tools typically used in economics is one of the many causes of this phenomenon. The main theoretical contributions take the form of equations when formalized. This formalism is often inadequate because of the complexity of the topics in question and the empirical data. The most common example of the separation between theory and data in economics is the evidence about the unrealistic nature of maximizing agents. Equation-based model are unable to include the algorithmic nature of behavioral data. Even if empirical economist observe data that is inconsistent with theoretical models of behavior and there is agreement on the unrealistic nature of some theoretical components, there is no substitute for them, at least no a substitute that can be used within the traditional approach.

The technical limitations of traditional tools in economics lead researchers who adopt them to give up a degree of realism, and to accept compromises. Agent-based Models avoid this trade-off, because they allow a large degree of integration between theoretical and empirical knowledge. Moreover, agent-based models allow to use empirical knowledge in theoretical analysis. The flexibility in modeling provided by agent-based models allows replication of the phenomenon of interest with a higher degree of realism than in other traditional models, the ability to realistically model the mechanism generating a phenomenon generates the ability to investigate which mechanism are responsible for a phenomenon. While on the contrary, when testing equation-based models over empirical data it is often difficult to distinguish causality from spurious correlation and to investigate endogeneity. The capability to model causal mechanism provided by agent-based models extend the capability to fully investigate uncertainty propagation in these models, so to trace the sources of the uncertainty systemic outcomes.

Another good reason to use Agent-Based Modeling is heterogeneity and interaction. In equation-based model, heterogeneity is bound to be very low, it is usual to separate the behavior and endowment of firms from those of consumers, but heterogeneity is not considered within each category of economic actors. Agent-based models do not impose any constraint on the degree of similarity between economic actors, economic agents in agent-based models could be modeled one by one, with completely different characteristics. Moreover social networks and any other kind of realistic interaction structure can be modeled in agent-based models. The capability of considering both the heterogeneity of agents and the interactions among them is one of the most powerful

advantages of agent-based models. The main reasons to use agent-based models is that they can provide more realism in modeling economic phenomena, moreover they provide the ability to model deeply the casual mechanisms determining economic phenomena. Finally agent-based models are a powerful tool to investigate hypothetical situation.

To sum up the main benefits of agent-based modeling are:

- they provide a natural description of a system.
- they are flexible.
- they capture emergent phenomena, deriving from the interactions of individual agents.
- the ability to investigate which mechanism are responsible for a phenomenon.
- they allow a large degree of integration between theoretical and empirical knowledge.
- heterogeneity and interaction.
- they are a powerful tool to investigate hypothetical situation.

1.3 Uses and weaknesses

Uses

According to Axtell (2000) we can classify three possible uses of Agent-based models.

- agent-based models as classical simulation, in order to make *what if* consideration.
- partially soluble models The second distinct use of agent-based computational models occurs when it is not possible to completely solve a mathematical model analytically. Here, theory yields mathematical relationships, perhaps even equations, but these are not directly soluble.
- models provably insoluble: agent computing as a substitute for analysis. When a model of a social process is completely intractable.

Weaknesses

There are some problems related to the application of agent-based models to the social, political, and economic sciences. they usually take into consideration human agents, with potentially irrational behavior, subjective choices, and complex psychology. so factors, difficult to quantify and analyze.

Another problem is that you must not make decisions on the basis of the quantitative outcome of a simulation that should be interpreted purely at the qualitative level.This is due to the varying degree of accuracy and completeness in the input to the model. Finally since agent-based models look at a system not at the aggregate level but at the level of its constituent units, simulating the behavior of all of the units can be extremely computation intensive and so time consuming.

Other weaknesses of agent-based models are:

- the necessity to study the program used to run the simulations in order to fully understand the model
- the necessity of checking the computer code to avoid the generation of inaccurate results due to coding errors.
- it is very difficult to explore the whole set of possible hypothesis in order to provide the best explanation. This is mainly due to the presence of behavioral rules for the agents within the hypothesis. This difficulty is determined by the uncontrolled dimension of the space of possibilities.

Chapter 2

Genetic Algorithms

In the 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 provide solutions to optimization problems using techniques driven by natural evolution, such as inheritance, mutation, selection, and crossover.

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 changed and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.

The evolution usually begins 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. 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.

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators.

The population size is related to the nature of the problem, but usually contains several hundreds or thousands of possible solutions. Often, the initial 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 more likely to be found.

Genetic algorithms are easy to implement, but their behavior is complicated to comprehend. Especially 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:

A description of a heuristic that performs adaptation by identifying and recombining 'building blocks'. 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.

2.0.1 Limitations

There are limitations of the use of a genetic algorithm compared to alternative optimization algorithms:

- Repeated fitness function evaluation for complex problems is usually the most prohibitive and limiting segment of artificial evolutionary algorithms. Reaching 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, a single function evaluation may need 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 evident 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 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, allowing 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 crucial 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 permits 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 be more efficient than genetic algorithms in terms of speed of convergence. Alternative and complementary algorithms include evolution strategies, evolutionary programming, simulated annealing, Gaussian adaptation, hill climbing, and swarm intelligence and methods based on integer linear programming. The suitability of genetic algorithms is dependent on the amount of knowledge of the problem; well known problems often have better, more specialized approaches.

Chapter 3

BehaviorSearch

The practice of designing and building new tools is essential to computer science. Compilers are an example of a software tool that basically changed the landscape of computer science. There are, naturally, many other examples of the success of tool building, including the NetLogo [Wilensky, 1999] platform that BehaviorSearch interfaces with.

Agent-based modeling lies at the intersection of computer science and many other disciplines, and as it is a growing field, there are many chances for building useful tools to serve this community.

By giving your model enough parameters, your model can show a wide range of behavior. So if you want to show the world a model that displays elephanttrunk-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 exists somewhere in the parameter space.

Other questions must be taken into consideration: are the parameter assignments that caused this behavior reasonable? what effect does each parameter have on the outcome, and are those trends reasonable?. However, it is the responsibility of the model author to carry out a critical analysis of the model.

3.0.2 What is BehaviorSearch?

BehaviorSearch is an open-source cross-platform tool that covers several search algorithms and search-space representations/encodings, and it is useful to explore the parameter space of any agent-based model, written in the NetLogo language. BehaviorSearch is a software tool to help with automating the exploration of agent-based models, by using genetic algorithms and other heuristic techniques to search the parameter-space.

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

Model exploration consist in four steps:

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

3.0.3 Goals and features

BehaviorSearch as NetLogo and Logo before it, pursue the twin design goals of 'low threshold' and 'high ceiling'. Hence BehaviorSearch tool should be both easy for beginners to learn and use ('low threshold'), while also providing advanced features that will allow expert modelers to engage in cutting-edge research and analysis ('high ceiling').

The features that support these design goals according to Stonedahl and Adviser-Wilensky (2011) are:

- ***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.
- ***Search method variety:*** BehaviorSearch covers 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 steadystate GA). This flexibility is important since different approaches can be more or less effective for exploring different models.

- ***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).

- ***Multi-resolution data output:***. BehaviorSearch can collect and store data at various levels of detail: recording each model 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.

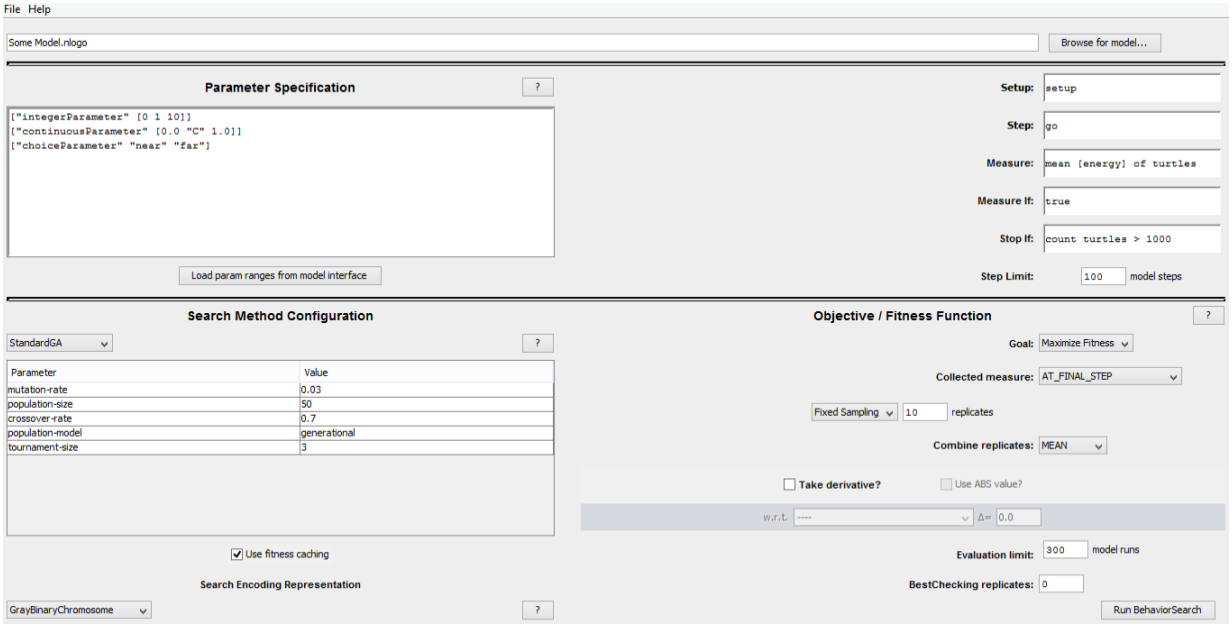
- ***Parametric derivatives:***. Built-in support for approximating derivatives of a behavioral objective function with respect to a specified parameter. This is useful for detecting phase transitions and critical points in the parameter space.

- ***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.

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

3.0.4 How it works?

When you open BehaviorSearch, the window that appears is the BehaviorSearch Experiment Editor.



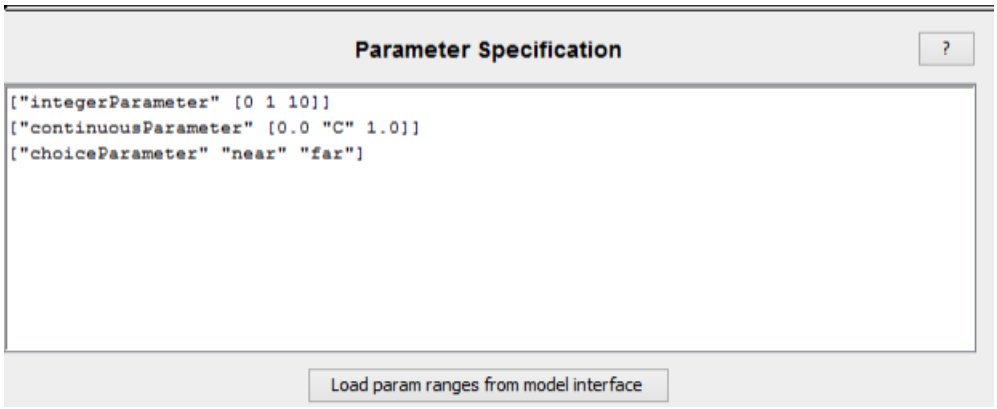
In order to compute an experiment on a NetLogo program, the procedure is composed by the following six steps.

1. Load a NetLogo program.
2. Parameter specification.
3. Options for model running.
4. Objective fitness function.
5. Search method configuration.
6. Run BehaviorSearch.

1. ***LOAD A NETLOGO PROGRAM***

First step to do is to load the NetLogo model you are interested into analyze, we can do this through the "Browse for model" button.

2. ***PARAMETER SPECIFICATION***



The next step is to define settings, or ranges of settings, for each of the model's parameters. The easiest way is to click the "Load param ranges from model interface" button, which will automatically report the specific parameters/ranges of your model's interface tab (*sliders, choosers,*and *switches).*

We can find:

- **['parameter-name' ['parameter-range']]**. 'parameter-name' represents the name assign to the variable. The 'parameter-range' is defined as **[starting-point increment ending-point]]** and it represents the space in which variable can takes value and the size of each possible increment. In this case the value of the slider *'parameter-name'* is not fixed and can change according to its range. In *BehaviorSearch* is possible to specify a continuous range for a parameter, by using 'C' for the increment, for istance **['parametername' [starting-point C ending-point]]**.
- **['parameter-name' parameter-value]**. In this case the value of the slider *'parameter-name'* is fixed and can not change according to its range during the *BehaviorSearch* analysis.
- **['parameter-name' true false]**. In this case we deal with Boolean parameters, this kind of parameter can assume as value true or false. Can be set fixed or can range during the analysis.

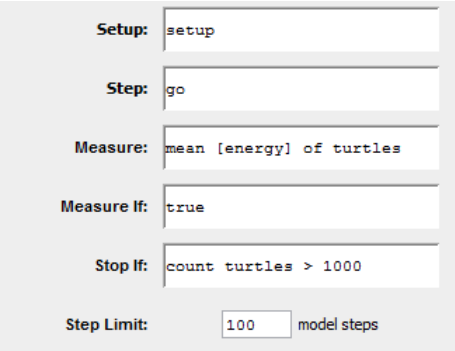
- **['parameter-name' 'choice 1' 'choice 2' ...]** for discrete-choice parameters. These parameters are a generalization of the Boolean parameters, we can have multiple situations that can be satisfied. Also in this case we can choose if consider them fixed or not.

The variable parameters loaded determine the size of the search space. This space is a multidimensional space in which each dimension is represented by a variable parameter.

To compute the size of the search space you need to calculate the total range of each parameter and then multiply these numbers.

BehaviorSearch is useful 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.

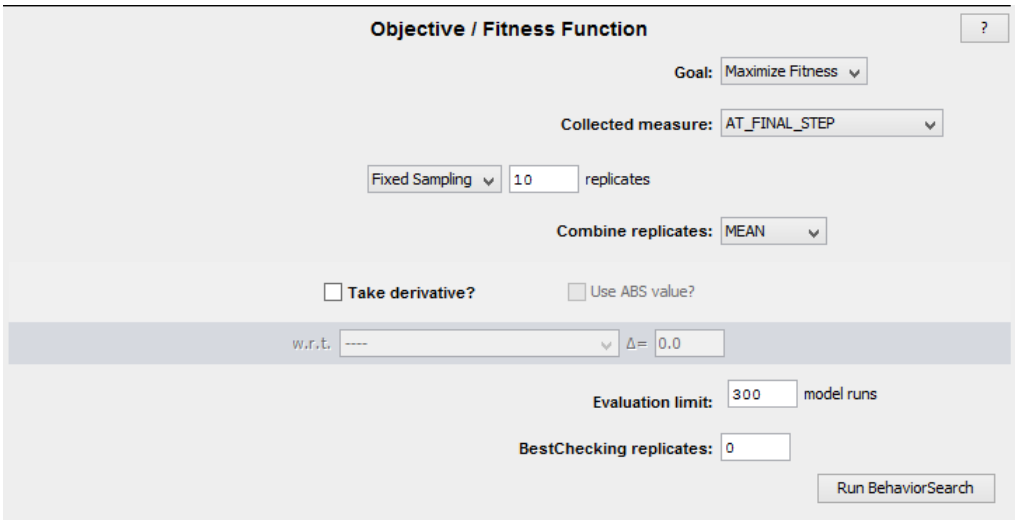
3. *OPTIONS FOR MODEL RUNNING*



After having specified the parameters, we have to define which variable we want to measure, as a function of specific parameters.

We must specify conditions, regarding how the model should be run.

- **Setup**: Identifies NetLogo commands that create the basic framework in which simulation occur (to setup the model). Usually this is defined as SETUP procedure.
- **Step**: Identifies NetLogo commands to be run over and over again in order to run simulations. Usually this is defined as GO procedure, if it includes *tick* command, one step refers to one *tick*.
- **Measure:** this 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's important is that the measure is correlated with the behavior we would like to elicit from the model.
- **Measure if**: it is an optional condition controlling on which steps the measure takes place.
- **Stop if**: it is an optional condition, a stop condition for the model.
- **Step limit**: a limit on the number of times the step commands will be run. We can not use the option 'forever' as in NetLogo program.



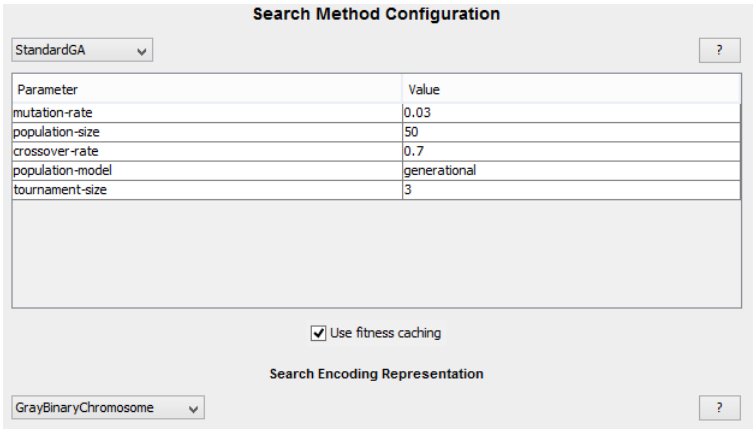
Above, we specified how to collect the data we need from the model in order to perform our search. We know how to collect the data, but now we need to turn it into an objective function ('fitness function').

- **Goal**: specify your objective, according to your goal, to minimize or to maximize the fitness function.
- **Collected measure**: During one model run, we may have collected the measure multiple times. We can condense those value in the following ways:
  - `AT_FINAL_STEP`:
    it reports the last measure calculated as the final result of the analysis, it is useful if you are only interested in the last measure that was recorded.
  - `MEAN_ACROSS_STEP`:
    it reports the mean of the multiple measures implemented.
  - `MEDIAN_ACROSS_STEP`:
    it reports the median across steps
  - `MIN_ACROSS_STEP`:
    it reports the minimum value measured across steps
  - `MAX_ACROSS_STEP`:
    it reports the maximum value measured across steps
  - `VARIANCE_ACROSS_STEP`:
    it reports the variance of the value obtained across steps
  - `SUM_ACROSS_STEP`:
    it reports the sum of all values calculated across steps

- **Sampling**: you have to set 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 rise the lasting of the BehaviorSearch analysis.
- **Combine replicates**: If you are doing multiple replicate runs of the model you have to combine those results, to get a single number for our objective function. It can computed as:
  - **mean:** simple average of the replicates.
  - **median:** may be a better choice if your measure occasionally yields extremely high or low outlier values, which you do not want consider.
  - **min/max:** may be useful choices if you want to search for parameters that cause extreme behavior, while ignoring average behavior.
  - **variance:** may be useful for finding parameters for which there is volatility in whether the model exhibits a behavior or not. Such volatility might show a phase transition between two regimes of model behavior.
  - **stdev:** the fitness function values will be in the same units of the original parameter, which make easier 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 and a specified delta (change amount).
- **If Use ABS value?** if it is checked, then the reported difference is always positive.
- **Evaluation limit**: number of total model runs should the search process perform, before stopping,
- **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 agent-based models are usually stochastic, and when sampling a measure a small/finite number of times, 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.

5. *SEARCH METHOD CONFIGURATION*



Once the objective function is fully specified, the final choice is about how the parameter space should be represented and explored. This involves choosing a search encoding representation (for the search space), as well as choosing a search algorithm and setting the necessary parameters for that search algorithm.

Search Algorithms and Options

The search algorithm determines what order the different parameter settings will be tried in, in order to find the behavior that you quantified above.

BehaviorSearch currently includes the following 4 search algorithms:

- **RandomSearch:** This is a naive baseline search algorithm, which simply randomly generates one set of parameters after another, computing the objective function for each in turn, and at the end it returns the best settings it found. RandomSearch is unlikely to be the most efficient search choice, but it is very straightforward method, and it performs an unbiased exploration of the search space.
- **MutationHillClimber:** This search algorithm starts with a random location in the search space, and then repeatedly generates a neighboring location (using a mutation operator) and moves to that new location only if it is better than the old location.

  Its parameters are:

  - *mutation-rate:* controls the probability of mutation.
  - *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.
- **SimualtedAnnealing:** 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. Simulated annealing is inspired by the physical annealing process in metallurgy: heating followed by the controlled cooling of a material in order to increase crystal size.

  Its parameters are:

  - *mutation-rate:* 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.
  - *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.)

- **StandardGA:** This is a simple classic Genetic Algorithm, which can operate on any of the available search representations.

  Its parameters are:

  - *mutation-rate:* controls the probability of mutation
  - *crossover-rate:* the probability of using two parents when creating a child (otherwise the child is created asexually)
  - *population-size:* the number of individuals in each generation.
  - *tournament-size:* the number of individuals that compete to be selected for reproduction via tournament selection. Higher values cause more selection pressure which will push the GA population to converge more quickly. Usually 2 or 3 is a good value.
  - *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.

Fitness Caching

There is also a checkbox 'Use fitness caching'. This controls whether the search algorithm caches (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 agent-based model 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 agent-based models 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.

Search Space Encoding Representation

The search space consists of all allowable combinations of settings for the model parameters, as you specified above. BehaviorSearch currently supports four different search space encodings:

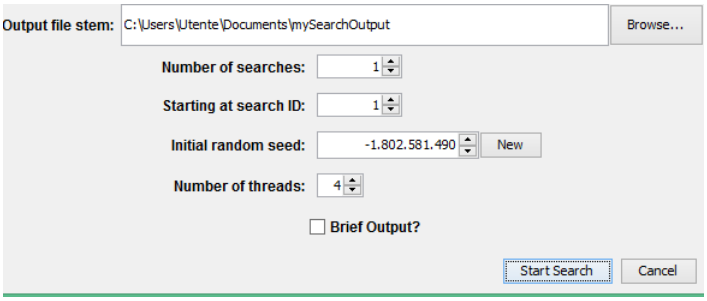
- ***MixedTypeChromosome:*** This encoding most closely matches the way that one commonly thinks of the agent-based model parameters. Each parameter is stored separately with its own data type (discrete numeric, continuous numeric, categorical, boolean, etc). Mutation applies to each parameter separately.
- ***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.
- ***GrayBinaryChromosome:*** 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.
- ***RealHypercubeChromosome***: In this encoding, every parameter (numeric or not) is represented by a 'real-valued' continuous variable. This encoding exists mainly to facilitate the (future) use of algorithms that assume a continuous numeric space, and allow them to be applied even when some of the model parameters are not numeric.

6. RUNNING THE SEARCH

After setting all of those options for how to perform the search, we can proceed with 'Run BehaviorSearch' button

Edit BehaviorSearch run options:

Clicking 'Run BehaviorSearch' brings up a dialog for choosing some additional options relating to the search running configuration.



- **Output file stem:** where to save the output data from the search. Specifically, a number of files will be created, each starting with this same filename 'stem'.
- **Number of searches:**: Specify how many times should the whole search process be repeated. A single search may not find the best parameter; additional searches improve confidence.
- **Starting at search ID:** This option only affects the ID numbering in the output files.
- **Initial random seed:** Random seed for the pseudo-random number generator, to start the first search (additional searches will be seeded with following consecutive numbers). This is useful for reproducing the exact same search later.
- **Number of threads:** Using multiple threads can significantly speed up the search process (on multi-processor/multi-core computers). Different numbers of threads should only affect running-time, and not the results obtained.
- **Brief Output?:** BehaviorSearch's default behavior is to create a variety of output data files, some of which can be quite large (containig the results of all model runs and all objective function evaluations). Selecting this checkbox suppresses the creation of the two largest output files.

Chapter 4

NetLogo model

4.1 NetLogo program

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 has 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 by Uri Wilensky, director of Northwestern University's Center for Connected Learning and Computer-Based Modeling.

NetLogo 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. NetLogo lets students open simulations and 'play' with them, exploring their behavior under various conditions.

The NetLogo world is made up of agents. Agents are beings that can follow instructions. In NetLogo, there are four types of agents: turtles, patches, links, and the

observer. Turtles are agents that move around in the world. Links are agents that connect two turtles. The observer doesn't have a location – you can imagine it as looking out over the world of turtles and patches. The observer doesn't observe passively – it gives instructions to the other agents. When NetLogo starts up, there are no turtles. The observer can make new turtles. Patches have coordinates. The patch at coordinates (0, 0) is called the origin and the coordinates of the other patches are the horizontal and vertical distances from this one. We call the patch's coordinates pxcor and pycor. The total number of patches is determined by the settings Links do not have coordinates. Every link has two ends, and each end is a turtle. If either turtle dies, the link dies too. A link is represented visually as a line connecting the two turtles.

In many NetLogo models, time passes in discrete steps, called 'ticks'. NetLogo includes a built-in tick counter so you can keep track of how many ticks have passed. The current value of the tick counter is shown above the view. In code, to retrieve the current value of the tick counter, use the ticks reporter. The tick command advances the tick counter by 1. The *clear-all* command clears the tick counter along with everything else. Use the *reset-ticks* command when your model is done setting up, to start the tick counter.

In NetLogo, commands and reporters tell agents what to do. A command is an action for an agent to carry out, resulting in some effect. A reporter is instructions for computing a value, which the agent then 'reports' to whoever asked it. Commands and reporters built into NetLogo are called primitives.

The NetLogo Dictionary has a complete list of built-in commands and reporters. Commands and reporters you define yourself are called procedures. Each procedure has a name, preceded by the keyword *to* or *to-report*, depending on whether it is a command procedure or a reporter procedure. The keyword *end* marks the end of the commands in the procedure. Once you define a procedure, you can use it elsewhere in your program.

4.1.1 Variables

Agent variables are places to store values (such as numbers) in an agent. An agent variable can be:

- *global variable*
- *turtle variable*
- *patch variable*
- *link variable*

If a variable is a global variable, there is only one value for the variable, and every agent can access it. You can think of global variables as belonging to the observer.

Turtle, patch, and link variables are different. Each turtle has its own value for every turtle variable. The same goes for patches and links. Some variables are built into NetLogo. For example, all turtles and links have a color variable, and all patches have a pcolor variable. (The patch variable begins with 'p' so it doesn't get confused with the turtle variable, since turtles have direct access to patch variables.) If you set the variable, the turtle or patch changes color.

You can also define your own variables. You can make a global variable by adding a *switch*, *slider*, *chooser*, or *input* box to your model, or by using the *globals* keyword at the beginning of your code, like this:

```
globals [score]
```

You can also define new turtle, patch and link variables using the *turtles-own* , *patches-own* and *links-own* keywords, like this:

```
turtles-own [energy speed]
patches-own [friction]
links-own [strength]
```

These variables can then be used freely in your model. Use the set command to set them. (Any variable you don't set has a starting value of zero.) Global variables can be read and set at any time by any agent.

A local variable is defined and used only in the context of a particular procedure or part of a procedure. To create a local variable, use the let command. If you use let at the top of a procedure, the variable will exist throughout the procedure.

4.1.2 NetLogo structure

NetLogo is structured as follows.

- *Interface*
- *Info*
- *Code*

4.1.3 Interface

The Interface tab is where you watch your model run. It also has tools you can use to inspect and alter what's going on inside the model.

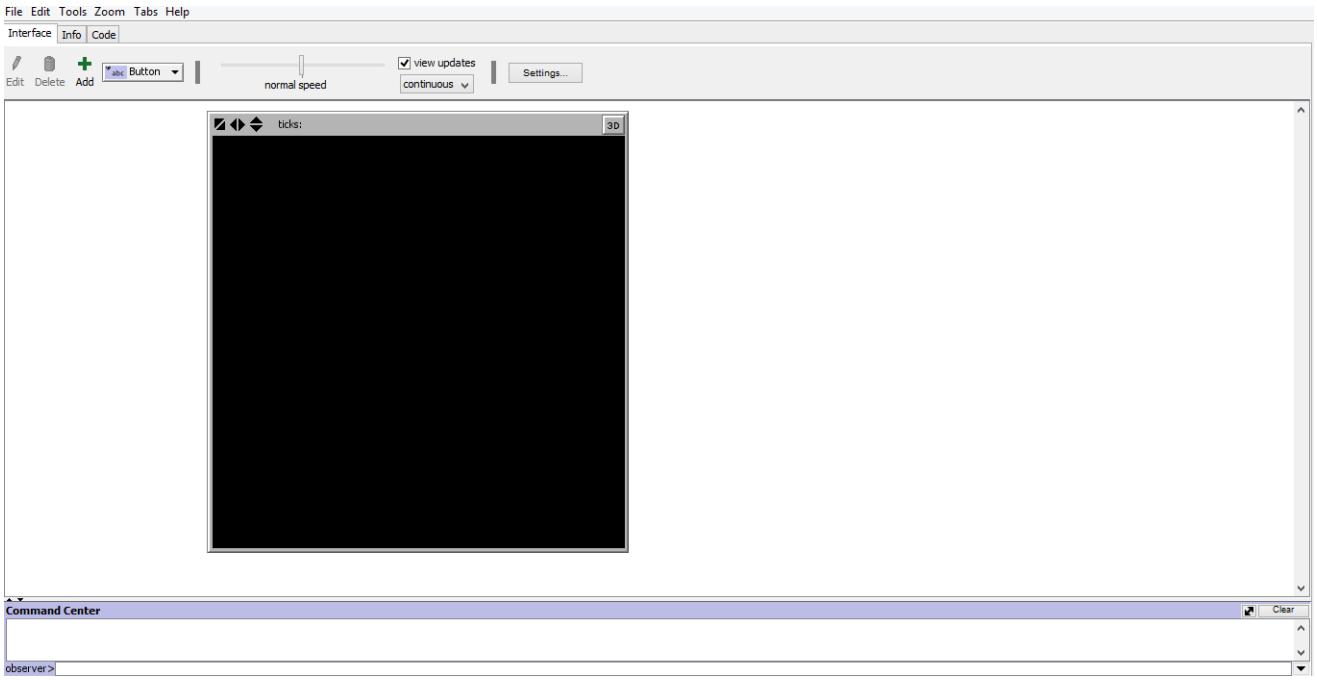
In the Interface we can find:

- *World*: The world is two dimensional and is divided up into a grid of patches. Each patch is a square piece of "ground" over which turtles can move. Links are agents that connect two turtles. The observer doesn't have a location.

- *Interface builder:* The toolbar on the Interface tab contains buttons that let you edit, delete, and create items in the Interface tab and a menu that lets you select different interface item.

  - *Button*: A button is either once or forever. When you click on a once button, it executes its instructions once. The forever button executes the instructions over and over, until you click on the button again to stop the action.
  - *Slider*: Sliders are global variables, which are accessible by all agents. They are used in models as a quick way to change a variable without having to recode the procedure every time. Instead, the user moves the slider to a value and observes what happens in the model.
  - *Switch*: Switches are a visual representation for a true/false global variable. You may set the variable to either on (true) or off (false) by flipping the switch.
  - *Chooser*: Choosers let you choose a value for a global variable from a list of choices, presented in a drop down menu. The choices may be strings, numbers, Booleans, or lists.
  - *Input*: Input Boxes are global variables that contain strings or numbers. The model author chooses what types of values you can enter. Input boxes can be set to check the syntax of a string for commands or reporters. Number input boxes read any type of constant number reporter which allows a more open way to express numbers than a slider.
  - *Monitor*: Monitors display the value of any reporter. Monitors automatically update several times per second.
  - *Plot*: Plots show data the model is generating.
  - *Output*: The output area is a scrolling area of text which can be used to create a log of activity in the model. A model may only have one output area.
  - *Note*: Notes lets you add informative text labels to the Interface tab. The contents of notes do not change as the model runs.

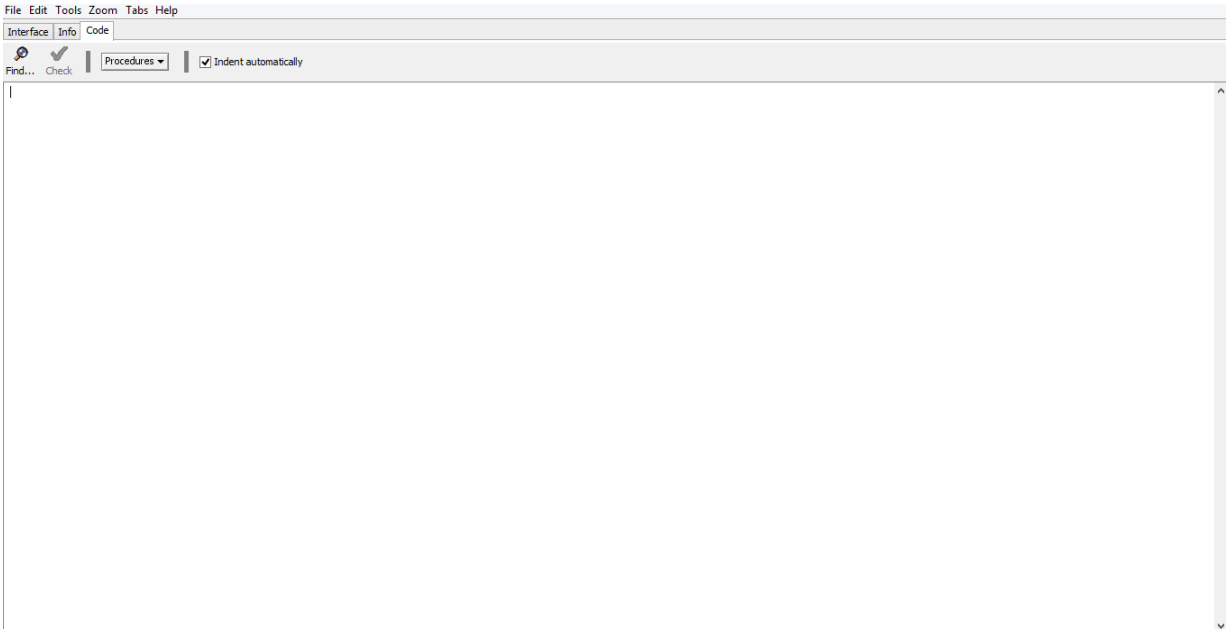
- *Speed-slider*: The slider lets you control how fast the model runs.
- *Settings*: The 'Settings' button allows you to change model settings
- *Command-center*: The Command Center allows you to issue commands directly, without adding them to the model's procedures. This is useful for inspecting and manipulating agents on the fly. The smaller box, below the large box, is where you type a command. After typing it press the Return or Enter key to run it.



4.1.4 Code

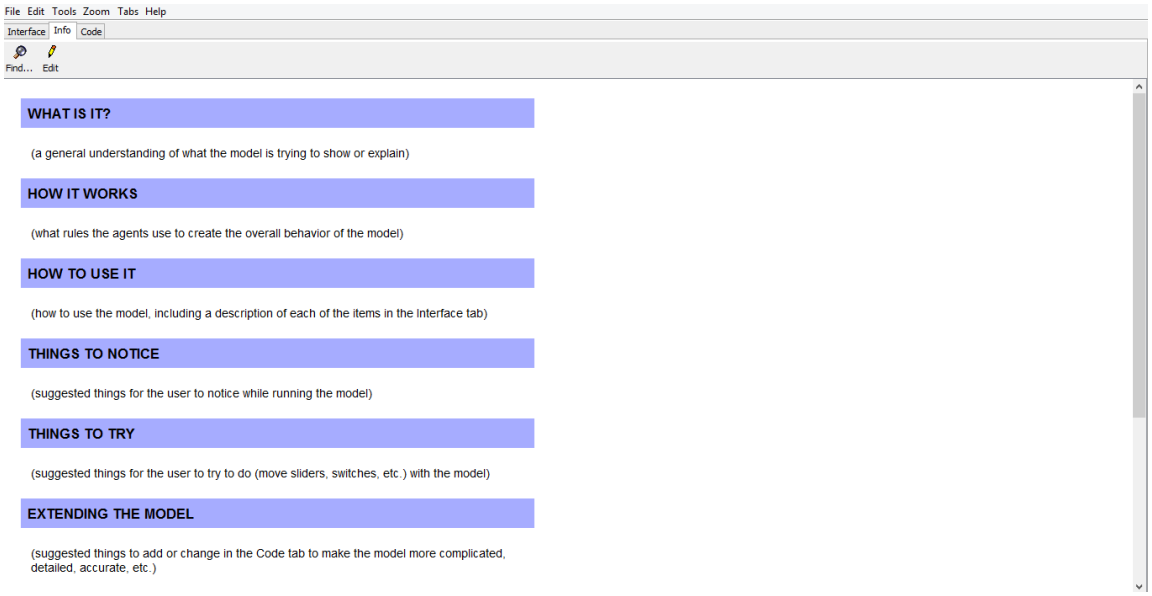
In the Code tab the author will insert the code lines. At the very beginning the modeler can create new *breeds*, assign globals or own variables. Then through *setup* procedure you can create agents and place them in the *world*, through go procedure simulations will occur. The different components of the code have different colors according their function.

To control if the code is right you can use the *Check* button, It provides a description of the error in order to correct it. Finally we have a *Procedures* button, reporting the name of all procedures written by the user.



4.1.5 Info

The Info tab provides an introduction to a model. It explains what system is being modeled, how the model was created, and and how to use it. It may also suggest things to explore and ways to extend the model, or call your attention to particular NetLogo features the model uses.



4.2 CDA basic model

My NetLogo simulation begins with the implementation of a simple CDA model, a continuous double auction model. In this program agents act in the single stock market, buying and selling, determining the stock price. In this model transactions are processed one at a time by a mechanism known as continuous double auction, buyers and sellers place at any time their orders, specifying at what price they are willing to buy or sell. Ordered prices are stored in an electronic order book, in which the lowest selling price and the highest buying price are kept on top and compared, if the buying one is greater or equal than the selling one, transaction is done.

We have a certain number of agents, determined by the slider *nrandomInvestors*, that decide with equal probability to be buyer or seller and with a certain probability, determined by the slider *passLevel*, to pass and do not operate in the market. Once agents set their position, they fix a certain price at which they are able to buy or sell the stock, this price is determined by a fix number (*exePrice*) plus a random number. Then they trade in the market, making transactions every time they find a counterpart.

Stock price floats according to the last transaction price.

4.2.1 Code

At the very beginning of the code we create *randomInvestors* breed, representing our agents in the market, and we assign them exclusive variables via *randomInvestors-own* keyword.

```
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice]
```

Finally we allocate the set of variables which are common to the all agentsets using the globals; command. It defines new global variables. Global variables are 'global' because they are accessible by all agents and can be used anywhere in a model.

GLOBAL VARIABLES

- ***nRandomInvestors***: defined by the slider, sets the number of investors in the market, fixed at 100.
- ***passlevel***: defined by the slider, represents the probability of passing, probability for an agent do not operate. Determines the percentage of inactive investors.
- ***logB***: a list storing all prices at which agents are willing to buy, bid prices. Sorted in decreasing order.
- ***logS***: a list storing all prices at which agents are willing to buy, ask prices. Sorted in increasing order.
- ***exePrice***: represents the stock price.

AGENTS VARIABLES

- ***buy***: a Boolean variable assuming value *true* when the investor is a buyer, *false* otherwise.
- ***sell***: a Boolean variable assuming value *true* when the investor is a seller, *false* otherwise.
- ***pass***: a Boolean variable assuming value *true* when the investor decide do not trade, *false* otherwise.
- ***price***: a floating point number specifying the investor's bid/ask price.
- ***cash***: a floating point number specifying investor's cash, fixed at zero at the beginning of the simulation.
- ***stocks***: number of stocks held by the investor, fixed at zero at the beginning of the simulation.

The code is composed by a setup procedure and a go procedure.

Setup procedure initialize the model, creates *nRandomInvestors* agents and places them orderly.

Moreover creates two empty lists , *logB* and *logS*, and set price of the stock equal 1000.

```
to setup

clear-all
set exePrice 1000
set logB []
set logS []
reset-ticks

create-randomInvestor nRandomInvestors

let side sqrt nRandomInvestors
let step max-pxcor / side

ask randomInvestors

[set shape "person"
 set size 2
 set stocks 0
 set cash 0]

let an 0

let x 0
let y 0
while [an < nRandomInvestors]
[if x > (side - 1) * step
[set y y + step
     set x 0]

ask randomInvestor an
[setxy x y]
 set x x + step
 set an an + 1]
end
```

Go procedure runs all transaction in a trading day, represented by a tick.

Agents with a certain probability will trade, and they will sell or buy with equal probability.

Then the code assign different colors to agents according their decision.

```
ask randomInvestors

[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]
```

Then to each agent is assigned a random price, representing at which price they are willing to buy or sell.

```
set price exePrice + (random-normal 0 100)
```

A temponary vector tmp, containing all agents prices and the relative agent number, is build.

Prices are added to this list through *lput* command while command *who* allow us to link agents to prices, without it we would lose agent identity (command *ask* shuffles the order of the agents).

```
ask randomInvestors

[if not pass
[let tmp[]
 set tmp lput price tmp
 set tmp lput who tmp
```

From this vector are created two different vectors *logB* and *logS*, respectively for buyer and seller.

Buyer prices are order increasing, while seller prices are ordered decreasing respectively through the commands reverse sort-by and sort-by.

```
[set logB lput tmp logB]
 set logB reverse sort-by [item 0 ?1 < item 0 ?2]logB

[set logS lput tmp logS]
 set logS sort-by [item 0 ?1 < item 0 ?2] logS
```

Then the first prices of this two list are compared, if the buyer price is greater or equal than the seller price the transction is set.

The price is determined by the order arrival, if the buyer enter the market and finds a counterpart seller with a price lower than hims, he buys at the seller price, while if a seller finds a counterpart with a price higher than hims, he sells at the buyer price.

If they buy:

```
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)
```

if they sell:

```
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)
```

Once transaction is done, cash and stocks variables are updated and the first elements are removed by the list through command *but-first*, *agB* and *agS* represent the agents identity of the buyer and the seller.

```
let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash -exePrice]

ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]

set logB but-first logB
set logS but-first logS
```

Into go procedure we have graph procedure that allows us to analyze the stock price path in the interface through a plot.

```
to graph

set-current-plot "exePrice"
plot exePrice

end
```

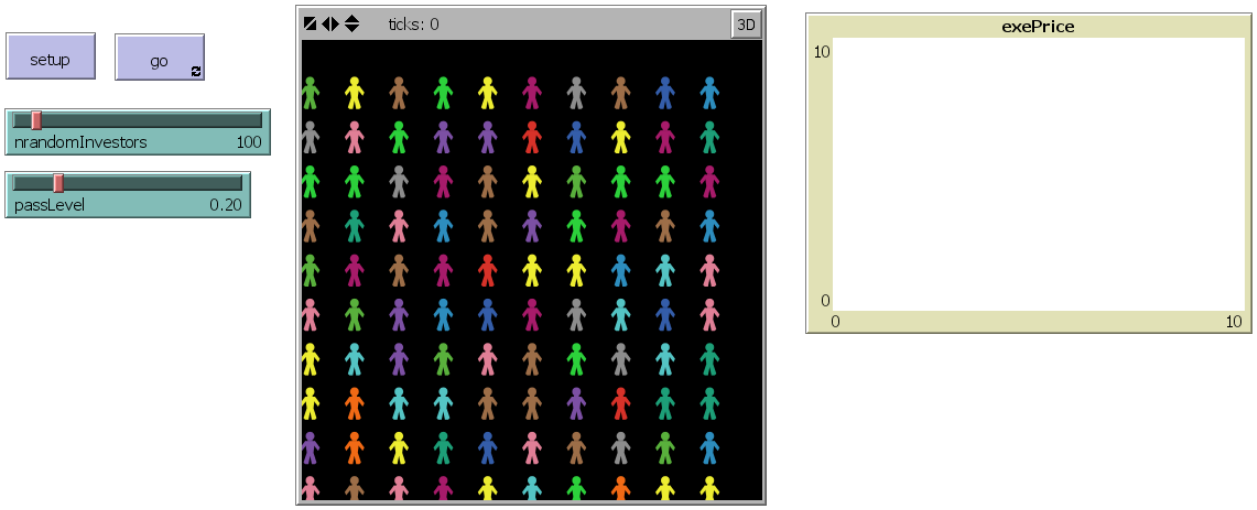
4.2.2 Interface

The interface is characterized by the presence of the world, buttons, sliders and a plot.

We have a setup and a go button to run the relative procedures.

Sliders to define global variables (nRandomInvestors, passLevel) and a plot of the evolution of *exePrice* in time.

In the world simulations occur.



4.3 Market maker model 0

I developed this market-maker model implementing the CDA model, introducing the presence of a market-maker.

The aim of this very first model is to insert the market maker in a continuous double auction model.

In this simple model the market marker acts offering himself to buy or sell whenever an agent is unable to find a counterpart. Agents first search a counterpart in the market between others agents, if they could not find it marker maker acts.

In this model the market maker does not set bid and ask price, simply buy and sell stocks at the price investors submit.

We can notice that with a market maker of this kind, acting every trading day, every tick in the model , the market falls down as market marker is always able to buy at a higher price with respect to the selling price, moreover is always willing to buy or sell at agents prices without setting a own ask and bid price.

4.3.1 Code

At the very beginning of the code we have:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
market-maker-own[cash stocks]
globals [logB logS exePrice]
```

I added a new *breed*, to insert the market-maker, assigning him variables such cash and stocks through *market-maker-own* keyword.

```
breed [market-maker]
market-maker-own[cash stocks]
```

PROCEDURES

to setup:

```
to setup

clear-all
set exePrice 1000
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-market-maker 1

let side sqrt (nrandomInvestors + 1)
let step max-pxcor / side

ask randomInvestors
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]

ask market-maker
[set size 4
```

```
 set stocks 0
 set cash 0
 set color blue
 set shape "house" ]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + 1]
[if x > (side - 1) * step
[set y y + step
 set x 0 ]
 ask turtle an
[setxy x y]
 set x x + step
 set an an + 1 ]

end
```

In particular adds the presence of market maker, cash and stocks are fixed at 0 at the beginning of the simulation.

```
create-market-maker 1
ask market-maker

[set size 4
 set color blue
 set shape "house"
 set stocks 0
 set cash 0]
```

to go:

```
to go

initialize
trade
market-maker-acting

end
```

Composed by 3 procedures, *initialize, trade* and *market-maker-acting*

to initialize:

```
ask randomInvestors

[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 buy false set sell true ]]

[set buy false set sell false]

set price exeprice + (random-normal 0 100)

if exeprice < 400
[if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]


set logB []
set logS []

tick
```

This procedure is necessary to determining agents position in the trading day, they can be buyer seller or do not operate at all.

To recognize them it assigns them different colors, green for seller red for buyer and grey for inactive agents.

After positions are determined, agents decide at which price they are willing to trade.

An interesting part of this procedure is

```
if exeprice < 400
[if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]
```

through this command is avoided the possibility to have negative prices, when stock price will be lower than 400 all agents with some probability *p-of-buying* will be buyers, so increasing the number of agents willing to buy will increase the stock price.

to trade:

```
to trade

ask randomInvestors

[if not pass
[ let tmp []
  set tmp lput price tmp
  set tmp lput who tmp
 ;show tmp


if buy [set logB lput tmp logB]
        set logB reverse sort-by [item 0 ?1 < item 0 ?2] logB
       ;show logB

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS]
```

```
if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs
         ;show logS

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS]

graph]]

end
```

Agents enter the market, set an order with a certain price at which they are willing to trade and they search a counterpart.

If they are able to find it they make the transaction and agent variables such *cash* and *stocks* are updated.

to market-maker-acting:

```
to market-maker-acting

if market-maker?
[if ticks mod 100 = 0

[ask randomInvestors

[if buy
[if (not empty? logB and not empty? logS)
     and item 0 (item 0 logB) < item 0 (item 0 logS)
     and price > 0

[set exePrice item 0 (item 0 logB)
 let agB item 1 ( item 0 logB)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask market-maker       [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB]

if (empty? logS and not empty? logB)  and price > 0

[set exePrice item 0 (item 0 logB)
 let agB item 1 (item 0 logB)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask market-maker       [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB]]


if sell
[if (empty? logB and not empty? logS)
     and price > 0

[set exePrice item 0 (item 0 logS)
 let agS item 1 (item 0 logS)
```

```
ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
ask market-maker       [set stocks stocks + 1
                        set cash cash - exePrice]

set logS but-first logS]

if (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) < item 0 (item 0 logS)
    and price > 0

[set exePrice item 0 (item 0 logS)
 let agS item 1 (item 0 logS)

 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 ask market-maker       [set stocks stocks + 1
                         set cash cash - exePrice]

 set logS but-first logS]]]
graph]]

end
```

Through the switcher *market-maker?* we can include in the simulation the presence or not of the market maker.

In this model the market maker acts every 100 ticks (trading days).

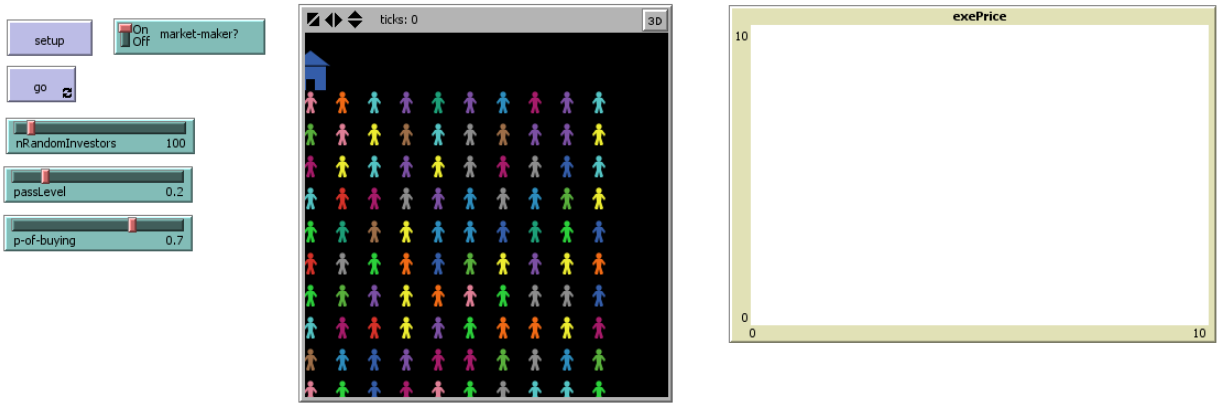
If traders were unable to find a counterpart, because there is no trader with opposite position or their order prices are incompatible, they can make the transaction with the market maker to their own price conditions.

As in the previous model into go procedure we have graph procedure that allows us to analyze the stock price path in the interface through a plot.

```
to graph

set-current-plot "exePrice"
plot exePrice

end
```

4.3.2 Interface

In the interface of this model with respect to the CDA basic model, we have a new slider *p-of-buying*, determining the probability for an agent to be a buyer when *exePrice* is lower than 400.



4.4 Market maker model 01

Implementing *market maker model 0* i developed a new model *market maker model 01* where the market maker does not trade at agents condition, but it sets an ask price and a bid price, at which it is willing to buy or sell.

The market maker goal is to earn profits and to achieve them buying low and selling high, so the ask price always exceeds the bid price. I assume market maker operate more times in a trading day, with a certain probability *p-of-acting*, when it is active on the market investors can deal at its prices.

4.4.1 Code

At the very beginning of the code we have:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
market-maker-own[cash stocks]
globals [logB logS exePrice market-maker-profit ask-price bid-price]
```

I added at the very beginning of the code through *globals*[ ] 3 globals variables.

- *market-maker-profit*: It represents the market maker variable *cash*, considering the number of stocks (if positive) that it owns.
- *ask-price*: price at which market maker is willing to sell.
- *bid-price*: price at which market maker is willing to buy.

in particular

```
globals [ market-maker-profit ask-price bid-price ]
```

PROCEDURES

to setup:

```
to setup

clear-all
set exePrice 1000
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-market-maker 1

let side sqrt (nrandomInvestors + 1)
let step max-pxcor / side

ask randomInvestors
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]

ask market-maker
[set size 4
 set shape "house"
 set stocks 0
 set cash 0
 set color blue]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + 1]
[ if x > (side - 1) * step
[set y y + step
 set x 0 ]
ask turtle an
[setxy x y]
```

```
 set x x + step
 set an an + 1 ]

end
```

this procedure remains invariant, creates the initial world, with investors and the market maker, where simulations will occur.

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[trade]

end
```

Composed by 3 procedures, *initialize, trade* and *book-market-maker*. We have a switcher *market-maker?*, if it is *on* go procedure will consist in *initialize* and *book-market-maker* procedures, otherwise instead of *book-market-maker* we will have *trade* procedure, and so a simple CDA basic model. ***to initialize***: in this procedure with respect to the previous model i simply added 3 global variables to allow the market maker to set ask and bid price, we have *ask-price* exceeding *bid-price* of a certain value defined by the global variable *bid-ask-spread*. The ask price is higher than the stock price while the bid price is lower.

```
to initialize


ask market-maker[ set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2 ]

ask randomInvestors
[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 buy false set sell true]]
```

```
[set buy false set sell false]

set price exePrice + (random-normal 0 100)

if exePrice < 400 [if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]

set logB []
set logS []

tick

end
```

to book-market-maker:

```
to book-market-maker

ask randomInvestors

[ifelse random-float 1 < p-of-acting
[trade-with-market-maker][trade]]

end
```

It is made by an *ifelse* command, so agents with a certain probability *p-of-acting* will be able to trade also with the market maker through *trade-with-market-maker* procedure or will trade among themselves through *trade* command.

to trade-with-market-maker:

```
to trade-with-market-maker

if not pass
[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

ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
         or (not empty? logB and empty? logS)
         and item 0 (item 0 logB) >= ask-price

[set exePrice ask-price
 let agB item 1 (item 0 logB)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask market-maker       [set stocks stocks - 1
                         ifelse stocks > 0
                         [set market-maker-profit
                              cash + (stocks * exePrice)]
                         [set market-maker-profit cash]]
 set logB but-first logB graph][buyer-trade-with-investors]

if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs

ifelse (not empty? logB and not empty? logS)
        and bid-price > item 0 ( item 0 logB)
        and bid-price >=  item 0 (item 0 logS)
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)

[set exePrice  bid-price
 let agS item 1 (item 0 logS)

 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 ask market-maker       [set stocks stocks + 1
                         set cash cash - exePrice
                         ifelse stocks > 0
                         [set market-maker-profit
```

```
                              cash + stocks * exePrice]
                          [set market-maker-profit cash]]
 set logS but-first logS graph][seller-trade-with-investors]
 graph]

end
```

In this procedure agents can trade both with the market maker and the other investors. Agents when they are active, compare the price at which they are willing to deal with market maker ask and bid price according their position, then if the prices are suitable they compare market maker price with other traders prices, if the market maker is offering the best price they will make the transaction. If it is not the case they will behave according their position through two different procedures:

- *buyer-trade-with-investors*
- *seller-trade-with-investors*

After any transaction own-variables,*cash, stocks*, are updated, also the global variable *market-maker-profit* depending by *cash*and *stocks* of the agent *market-maker* and the global variable *exePrice*.

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS]


end
```

The buyer investor if is not able to deal with the market maker since their price don't fix or other investor prices are more convenient, decide to trade with other investors. Whenever she will be able to find a seller counterpart, the transaction will be made at the best price for the buyer.

to seller-trade-with-investors:

```
to seller-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS]


end
```

It works as the previous procedure, with the only difference that we have a seller agent trying to find a buyer counterpart and transaction will be made at the best price for the seller.

to trade:

```
to trade

if not pass
[let tmp []
 set tmp lput price tmp
 set tmp lput who tmp
;show tmp

if buy [set logB lput tmp logB]
        set logB reverse sort-by [item 0 ?1 < item 0 ?2] logB
       ;show logB

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
set logB but-first logB
set logS but-first logS]

if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs
        ;show logS

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)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
```

```
ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB
set logS but-first logS]
graph]

end
```

same procedure of *market maker model 0*, where agents are trading by themselves without the market maker.

to graph:

```
to graph

set-current-plot "exePrice"
plot exePrice

end
```

same procedure of the previous model, allowing us to analyze stock price path in time.

4.4.2 Interface

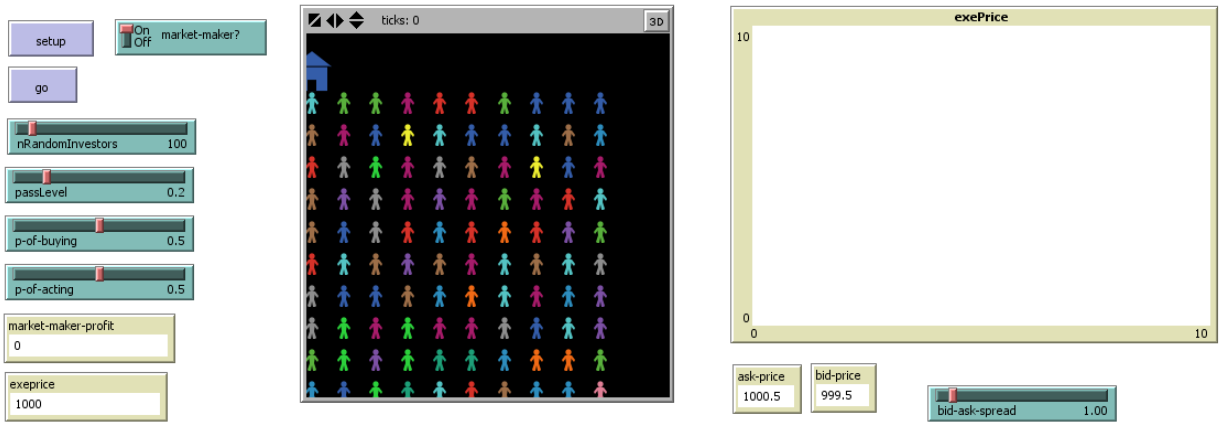
I added in this interface 2 sliders and 3 monitors.

sliders:

- *p-of-acting*: the probability that the market maker acts in the market.
- *bid-ask-spread*: it defines the difference between bid and ask price, how much ask price exceeds bid price.

monitors:

- *bid-price*: reporting the value of the bid price.
- *ask-price*: reporting the value of the ask price.
- *market-maker-profit*: reporting the value of the *market-maker* variable *cash* if $stocks < 0$ or the value $cash + (stocks * exePrice)$ if $stocks > 0$



4.5 Market maker model 02

In this new model *market maker 02*, starting from the model *market maker 01* i introduce the presence of an initial budget constrain and a given number of stocks. In the previous models the market maker begin simulation with zero cash and zero stocks, assuming no constraints it could sell and buy with no limits. In this model, in order to create a more realistic settings we assign to market maker an initial availability of sources, in terms of cash and stocks, and it will be able to trade according to them.

4.5.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit ask-price
         bid-price market-maker-stocks market-maker-cash]
```

In this model with respect the previous one we have no more *market-maker-own* in order to assign variables to it. The agent variables, *cash* and *stocks*, are replaced by two global variables representing market maker availability of cash and stocks, respectively *market-maker-cash* and *market-maker-stocks*. In particular i added:

```
globals [market-maker-stocks market-maker-cash]
```

PROCEDURES

to setup:

```
to setup

clear-all
set exePrice 1000
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-market-maker 1
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash

let side sqrt nrandomInvestors + 1
let step max-pxcor / side

ask randomInvestors
   [set shape "person"
    set size 2
    set stocks 0
    set cash 0]

ask market-maker
   [set size 4
    set color blue
    set shape "house"
    set ask-price exePrice + bid-ask-spread / 2
    set bid-price exePrice - bid-ask-spread / 2]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + 1]
[if x > (side - 1) * step
[set y y + step
```

```
 set x 0 ]
ask turtle an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

In this procedure i set two new global variables:

```
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash
```

representing respectively the market maker availability of stocks and cash, i can decide and modify this variables according to the sliders.

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[ask randomInvestors [trade]]

end
```

Go procedure, as before composed by the following procedure:

- *initialize*
- *book-market-maker*
- *trade*

to initialize:

```
to initialize

ask market-maker[ set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2 ]

ask randomInvestors
[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 buy false set sell true]]
[set buy false set sell false]

set price exePrice + (random-normal 0 100)

if exePrice < 400 [if random-float 1 < p-of-buying
[ set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]

set logB []
set logS []

tick

end
```

Procedure used to set initial variables in order to run the simulation.

to book-market-maker:

```
to book-market-maker

ask randomInvestors
[ifelse random-float 1 < p-of-acting
[trade-with-market-maker][trade]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*.

to trade:

```
to trade

if not pass
[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 (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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS]
```

```
if sell [set logS lput tmp logS]
         set logS sort-by [item 0 ?1 < item 0 ?2] logS
        ;show logS

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS
]

graph
]

end
```

This procedure is the same as before, whereas there are new instructions in the procedure *to trade-with-market-maker*

to trade-with-market-maker:

```
to trade-with-market-maker

if not pass
[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

ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0
        or (not empty? logB and empty? logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0

[set exePrice ask-price
 let agB item 1 (item 0 logB)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask market-maker       [set market-maker-stocks market-maker-stocks - 1
                         set market-maker-cash market-maker-cash + exePrice
                         ifelse market-maker-stocks > 0
                         [set market-maker-profit
                          market-maker-cash + (market-maker-stocks * exePrice)]
                         [set market-maker-profit market-maker-cash]]
 set logB but-first logB][ buyer-trade-with-investors]


if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs

ifelse (not empty? logB and not empty? logS)
        and bid-price > item 0 ( item 0 logB)
```

```
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price

[set exePrice  bid-price
 let agS item 1 (item 0 logS)

 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 ask market-maker [set market-maker-stocks market-maker-stocks + 1
                   set market-maker-cash market-maker-cash - exePrice
                   ifelse market-maker-stocks > 0
                  [set market-maker-profit
                       market-maker-cash + (market-maker-stocks * exePrice)]
                  [set market-maker-profit market-maker-cash]]
 set logS but-first logS ][seller-trade-with-investors]
 graph]
end
```

In order to develop a more realistic model we have the presence of constraints for the market maker. In the previous model it was able to sell and buy how many stocks with no limits, assuming initial level of stocks and cash equal to zero.

In this model it is able to trade according its sources, the simulation begins with given values of cash and stocks. Through this command it could sell stocks if

```
market-maker-stocks > 0
```

and it could buy stock if

```
market-maker-cash >= bid-price
```

If the agent could not trade with the market maker, she will try to trade with the other investors through:

- *to buyer-trade-with-investors*
- *to seller-trade-with-investors*

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

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)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB
set logS but-first logS]

end
```

to seller-trade-with-investors:

```
to seller-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS]

end
```

to graph:

```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

end
```

In the graph procedure i added:

```
set-current-plot "market-maker-stocks"
plot market-maker-stocks
```

in order to analyze in a graph the number of stocks owned by the market maker.

4.5.2 Interface

I added in this interface two sliders two monitors and one graph. ***sliders***:

- *Initial-Cash*: representing the value of the global variable *market-maker-cash* at the beginning of the simulation.
- *Initial-nStocks*: representing the value of the global variable *market-maker-stocks* at the beginning of the simulation.

monitors:

- *market-maker-cash*: representing the available cash for the market maker.
- *market-maker-stocks*: representing number of stocks owned by the market maker.
- *market-maker-profit*: representing the profit of the market maker equal to the sum of market-maker cash plus the number of stocks valuated at the current market value.

graph:

- *market-maker-stocks*: plots the number of stocks owned by the market maker.

4.6 Market maker model 03

In this model with respect the previous one, the market maker is able to update the bid and ask price during the trading day (one tick), whenever stock price changes.

In *market maker model 02*, the market maker sets bid and ask price at the beginning of every trading day, without updating them even if stock price changes according to transactions. In this model in order to be more real we allow to the market maker the instantaneous update of bid and ask, according to new prices.

While in the previous models the initial stock price was fixed at 1000, an implementation of this model is the possibility to choose the initial stock price according to a new slider: *Initial-StockPrice*.

4.6.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit ask-price
         bid-price market-maker-stocks market-maker-cash]
```

PROCEDURES

to setup:

```
to setup

clear-all
set exePrice Initial-StockPrice
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-market-maker 1
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash
```

```
let side sqrt (nrandomInvestors + 1)
let step max-pxcor / side

ask randomInvestors
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]

ask market-maker
[set size 4
 set color blue
 set shape "house"
 set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2 ]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + 1]
[if x > (side - 1) * step
[set y y + step
 set x 0 ]
ask turtle an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

In this procedure I set the initial value of *exePrice* equal to the value of the slider *Initial-StockPrice*, while in the previous models was fixed at 1000. In this way we can change and analyze the effect of different initial stock price.

```
set exePrice Initial-StockPrice
```

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[ask randomInvestors [trade]]

end
```

Go procedure, as before composed by the following procedure:

- *initialize*
- *book-market-maker*
- *trade*

to initialize:

```
to initialize

if market-maker?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2 ]]

ask randomInvestors
[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 buy false set sell true]]
[set buy false set sell false]
set price
    exePrice + (random-normal 0 Initial-StockPrice * 0.05)
if exePrice < Initial-StockPrice * 0.2
[if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]

set logB []
set logS []

tick

end
```

Procedure used to set initial variables in order to run the simulation. With respect investors set their expectation on price, the price at which they are willing to trade according to a fixed element (*exePrice*) and a random element. In *market-maker-model 02* the random part is represented by a random normal number with mean 0 and standard deviation 100, now the standard deviation is expressed in function of the initial stock price, it is equal to a percentage of it.

```
set price exePrice + (random-normal 0 Initial-StockPrice * 0.05)
```

Also the level at which agents are willing to buy in order to don't allow price to be negative is expressed according to *Initial-StockPrice*.

```
if exePrice < Initial-StockPrice * 0.2
[if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]
```

to book-market-maker:

```
to book-market-maker

ask randomInvestors [
ifelse random-float 1 < p-of-acting [ trade-with-market-maker][trade]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*.

to trade:

```
to trade

if not pass
[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 (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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
```

```
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

if( market-maker? and update-bid-ask?)
 [ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]


if sell [set logS lput tmp logS]
         set logS sort-by [item 0 ?1 < item 0 ?2] logS
        ;show logS

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

 if( market-maker? and update-bid-ask?)
 [ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]

graph
]
end
```

In this procedure the market maker updates bid and ask price at every price changing, through this command:

```
if( market-maker? and update-bid-ask?)
```

```
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]
```

This command will run only if market maker presence and update skill are assumed according the respective switchers *market-maker?* and *update-bid-ask?*.

to trade-with-market-maker:

```
to trade-with-market-maker

if not pass
[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

ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0
        or (not empty? logB and empty? logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0

[set exePrice ask-price
 let agB item 1 (item 0 logB)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask market-maker [set market-maker-stocks market-maker-stocks - 1
                   set market-maker-cash market-maker-cash + exePrice
                   set market-maker-profit
                       market-maker-cash + (market-maker-stocks * exePrice)
                       - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
                         if update-bid-ask?
                         [set ask-price exePrice + bid-ask-spread / 2
                          set bid-price exePrice - bid-ask-spread / 2]]
 set logB but-first logB][ buyer-trade-with-investors]


if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs
```

```
;show logS
ifelse (not empty? logB and not empty? logS)
        and bid-price > item 0 ( item 0 logB)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price

[set exePrice  bid-price
 let agS item 1 (item 0 logS)

ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
ask market-maker  [set market-maker-stocks market-maker-stocks + 1
                   set market-maker-cash market-maker-cash - exePrice
                   set market-maker-profit
                       market-maker-cash + (market-maker-stocks * exePrice)
                       - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
                  if update-bid-ask?
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2 ]]

 set logS but-first logS][seller-trade-with-investors]

graph]

end
```

In order to develop a more realistic model the market maker updates bid and ask prices every price changing through this command:

```
if update-bid-ask?
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]
```

In this model the value of variable *market-maker-profit* is expressed considering the amount of initial investment given by:

```
Initial-Cash + (Initial-nStocks * Initial-StockPrice)
```

in this way we have net profit when the variable is positive and we have a loss when is negative. The following formula represents *market-maker-profit*:

```
set market-maker-profit
market-maker-cash + (market-maker-stocks * exePrice)
- Initial-Cash - (Initial-nStocks * Initial-StockPrice)
```

equal to the sum of market maker cash and the number of owned stocks valued at the current price subtracting the amount of the initial investment.

If the agent could not trade with the market maker, she will try to trade with the other investors through:

- *to buyer-trade-with-investors*
- *to seller-trade-with-investors*

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS
if update-bid-ask?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]
]

end
```

to seller-trade-with-investors:

```
to seller-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

if update-bid-ask?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]
]

end
```

I added the update price even when the transaction occurs among investors, the market maker notice the price change and immediately adjust its prices.

```
if update-bid-ask? [
ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                 set bid-price exePrice - bid-ask-spread / 2]]]
```

to graph:

```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

set-current-plot "market-maker-profit"
plot market-maker-profit

end
```

In the graph procedure I added:

```
set-current-plot "market-maker-profit"
plot market-maker-profit
```

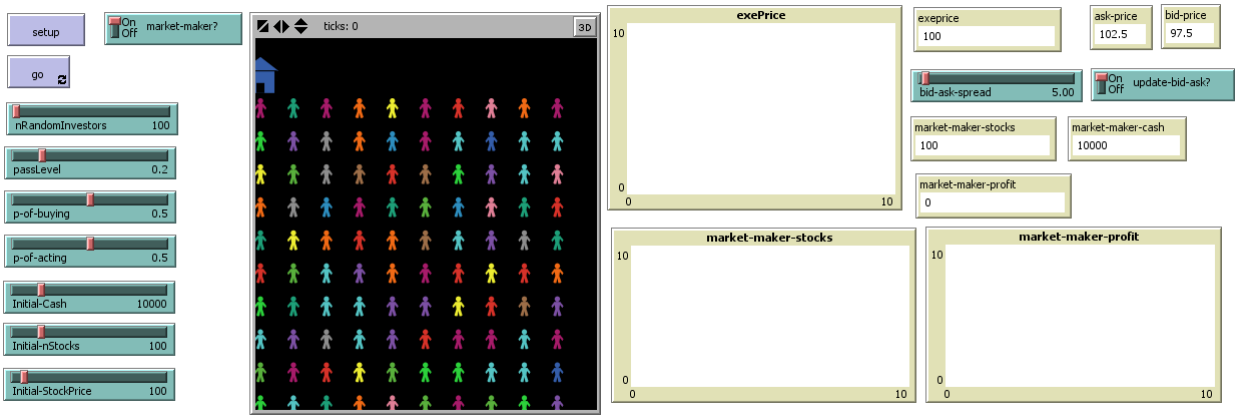
in order to analyze in a graph the path of market maker profits, when is positive or negative.

4.6.2 Interface

I added in this interface one slider one switch and one graph.

- *Initial-StockPrice*: slider representing the value of the global variable *exePrice* at the beginning of the simulation.
- *update-bid-ask?*: switch, allowing the model to include the market maker skill to update bid and ask prices whenever stock price changes.
- *market-maker-profit*: plots the value of the market maker net profit.

Moreover I set the maximum value of sliders *Initial-Cash* in function of *nRandom-Investors* and *Initial-StockPrice* and the value of slider *Initial-nStocks* in function of *nRandomInvestors*.



4.7 Market maker model 04

As in the previous model the market maker is able to adjust its decision according to the price, in this model also investors will be able to react at changing during the trading day. In this model we have a new switch *update-price?* allowing investors to fix new prices at which they are willing to buy or sell. While before market maker adjusted bid and ask in every change, now agents' prices and bid ask will be update with a certain probability, respectively *p-of-update-price* and *p-of-update-bid-ask*.

Starting from this model I developed two implemented models, adding more and more details, respectively:

- ***market maker model 04.01***
- ***market maker model 04.02***

4.7.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit
         ask-price bid-price market-maker-stocks market-maker-cash]
```

PROCEDURES

to setup:

```
to setup

clear-all
set exePrice Initial-StockPrice
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-market-maker 1
```

```
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash

let side sqrt (nrandomInvestors + 1)
let step max-pxcor / side

ask randomInvestors
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]

ask market-maker
[set size 4
 set color blue
 set shape "house"
 set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + 1]
[if x > (side - 1) * step
[set y y + step
 set x 0 ]
ask turtle an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

This procedure is the same with respect the previous model.

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[ask randomInvestors [trade]]

end
```

Go procedure, as before composed by the following procedure:

- *initialize*
- *book-market-maker*
- *trade*

to initialize:

```
to initialize

if market-maker?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]

ask randomInvestors
[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 buy false set sell true]]
[set buy false set sell false]

set price exePrice + (random-normal 0 Initial-StockPrice * 0.05)

if exePrice < Initial-StockPrice * 0.2
[if random-float 1 < p-of-buying
```

```
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]

set logB []
set logS []

tick

end
```

Procedure used to set initial variables in order to run the simulation.

to book-market-maker:

```
to book-market-maker

ask randomInvestors [ifelse random-float 1 < p-of-acting
[trade-with-market-maker][trade]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*.

to trade:

```
to trade

if not pass
[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 (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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

 if( market-maker? and update-bid-ask?)
 [ask market-maker[if random-float 1 < p-of-update-bid-ask
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]
 if update-price?
 [ask randomInvestors [if random-float 1 < p-of-update-price
                       [set price
                            exeprice + (random-normal 0 exePrice * 0.05)]]]
]


 if sell [set logS lput tmp logS]
          set logS sort-by [item 0 ?1 < item 0 ?2] logS
```

```
         ;show logS

 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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

 if( market-maker? and update-bid-ask?)
 [ask market-maker[if random-float 1 < p-of-update-bid-ask
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]
 if update-price?
 [ask randomInvestors [if random-float 1 < p-of-update-price
                       [set price
                            exeprice + (random-normal 0 exePrice * 0.05)]]]
]
graph
]

end
```

In this procedure also investors according to the switch *update-price?* will update their price at every price changing with a certain probability defined by the slider *p-of-update-price*, through this command:

```
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]
```

In this model, according to the respective switchers *market-maker?* and *update-bid-ask?*, the market maker will update bid and ask not with probability one, but with a certain probability defined by the slider *p-of-update-bid-ask*.

```
if( market-maker? and update-bid-ask?)
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                 [set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]]
```

to trade-with-market-maker:

```
to trade-with-market-maker

if not pass
[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
 ifelse (not empty? logB and not empty? logS)
         and ask-price < item 0 (item 0 logS)
         and item 0 (item 0 logB) >= ask-price
         and market-maker-stocks > 0
         or (not empty? logB and empty? logS)
         and item 0 (item 0 logB) >= ask-price
         and market-maker-stocks > 0

[set exePrice ask-price
 let agB item 1 (item 0 logB)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask market-maker [set market-maker-stocks market-maker-stocks - 1
                   set market-maker-cash market-maker-cash + exePrice
                   set market-maker-profit
                       market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
```

```
                  if update-bid-ask? [if random-float 1 < p-of-update-bid-ask
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]

 if update-price?
[ask randomInvestors
[if random-float 1 < p-of-update-price
[set price exeprice + (random-normal 0 exePrice * 0.05)]]]

 set logB but-first logB][ buyer-trade-with-investors]

 if sell [set logS lput tmp logS]
          set logS sort-by [ item 0 ?1 < item 0 ?2] logs
         ;show logS
 ifelse (not empty? logB and not empty? logS)
         and bid-price > item 0 ( item 0 logB)
         and bid-price >=  item 0 (item 0 logS)
         and market-maker-cash >= bid-price
         or (empty? logB and  not empty? logS)
         and bid-price >=  item 0 (item 0 logS)
         and market-maker-cash >= bid-price

 [set exePrice  bid-price
  let agS item 1 (item 0 logS)

ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks + 1
                  set market-maker-cash market-maker-cash - exePrice
                  set market-maker-profit
                      market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks Initial-StockPrice)
                      if update-bid-ask?
                       [if random-float 1 < p-of-update-bid-ask
                       [set ask-price exePrice +  bid-ask-spread / 2
                        set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]]
```

```
set logS but-first logS][seller-trade-with-investors]
graph]

end
```

In order to develop a more realistic model the market maker and agents update their conditions to trade every price changing with a certain probability, through this command:

```
if update-bid-ask? [ if random-float 1 < p-of-update-bid-ask
                   [set ask-price exePrice + bid-ask-spread / 2
                    set bid-price exePrice - bid-ask-spread / 2]]]

if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]
```

If the agent could not trade with the market maker, she will try to trade with the other investors through:

- *to buyer-trade-with-investors*
- *to seller-trade-with-investors*

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
```

```
 set logB but-first logB
 set logS but-first logS




 if update-bid-ask?
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                 [set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]]

 if update-price?
 [ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]
]

end
```

to seller-trade-with-investors:

```
to seller-trade-with-investors

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)

 ask randomInvestor agB [set stocks stocks + 1
                         set cash cash - exePrice]
 ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

if update-bid-ask?
[ask market-maker[if random-float 1 < p-of-update-bid-ask
```

```
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]
 ]end
```

I added the update price even when the transaction occurs among investors, the market maker and investors notice the price change and immediately adjust their prices with some probability.

```
if update-bid-ask? [if random-float 1 < p-of-update-bid-ask
                    [set ask-price exePrice + bid-ask-spread / 2
                     set bid-price exePrice - bid-ask-spread / 2]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]
```

to graph:

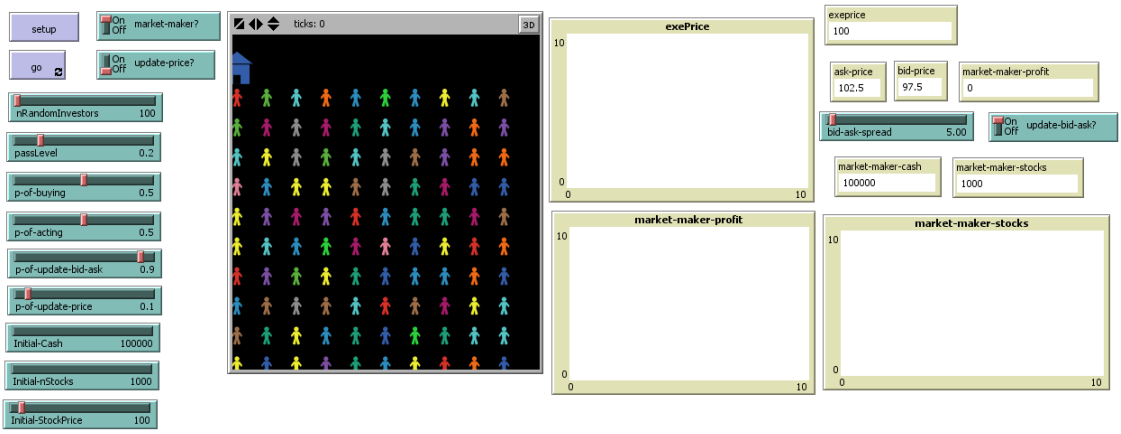
```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

set-current-plot "market-maker-profit"
plot market-maker-profit

end
```

The same procedure as before, in order to plot stock price. number of stocks owned by the market maker and market maker profit.

4.7.2 Interface

I added in this interface two sliders and one switch.

- *p-of-update-bid-ask*: slider representing the probability that the market maker will adjust bid and ask price when stock price changes.
- *p-of-update-price*: slider representing the probability that the investor will adjust her condition to trade (price at which she is willing to buy or sell) when stock price changes.
- *update-price?*: switch, allowing the model to include the investors skill to update the variable *price* whenever stock price changes.



4.7.3 Market maker model 04.01

In this implementation of the model *market maker model 04*, we allow to *p-of-acting* to change during the simulation. The value of the variable *p-of-acting* represents the probability that the market maker will be able to trade with the random investors.

In the previous models this value is fixed at 0.5, in this development this value is fixed at that value at the beginning of the simulation, then it moves according to price movements. If the price goes up *p-of-acting* will increase, while if it goes down the probability will descrease. In order to get this I added the following part of code

```
if exePrice < bid-price[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > bid-price[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

and

```
if exePrice < ask-price[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > ask-price[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

in the procedure *to trade-with-market-maker*
then

```
if exePrice < item 0 (item 0 logS)
[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > item 0 (item 0 logS)
[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

in procedures *to buyer-trade-with-investors* and *to trade* and

```
if exePrice < item 0 (item 0 logB)
[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > item 0 (item 0 logB)
[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

in procedures *to seller-trade-with-investors* and *to trade*.

In this model we can analyze in the interface how *p-of-acting* changes when simulation occur.

4.7.4 Market maker model 04.02

In this additional implementation we consider investors set their condition price on the basic of a fixed and a variable part as before, but we decrease the size of the variable part. In the first models we consider the fixed part to be equal to the stock price (*exePrice*=1000) and the variable part equal to a number of a random normal distribution with zero mean and standard deviation equal to 100. This allow market maker to set bid and ask price with a too high spread, continuing to trade. This is not very likely to be. In order to analyze a more realistic environment we have to decrease the variable part.

In *market maker model 03* we introduce a new way of measure the variable part: equal to 5% of *Initial-StockPrice*. Now in this model we decrease further the variable part setting it equal to 1% of the initial stock price. The way of measuring the variable part is also changed when investors update *price*.

The value of variable *price* now is given by:

```
[set price exeprice + (random-normal 0 exePrice * 0.01)]
```

4.8 Market maker model 05

This model is characterized by the presence of a new 'breed' of investors.

In the previous model we had market maker updating its bid and ask price with a very high probability, and it deals with investors who react to price changing with low probability. In this environment the market will be populated by a market maker and two different types of investors.

- *nRandomInvestors:* agents that set their condition price at the beginning of each trading day, and rarely they react to price movements.
- *nActiveInvestors:* this kind of investor is active, meaning that she will follows market trends, they immediately react to price changing adjusting their price at which they are willing to trade.

Starting from this model as before I developed two implemented models, adding more and more details, respectively:

- ***market maker model 05.01***
- ***market maker model 05.02***

4.8.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
breed [ActiveInvestors ActiveInvestor]
randomInvestors-own[buy sell pass price cash stocks]
ActiveInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit
         ask-price bid-price market-maker-stocks market-maker-cash]
```

In order to add a new 'breed' I used the following command

```
breed [ActiveInvestors ActiveInvestor]
```

assigning them exclusive variables through *ActiveInvestors-own* keyword.

```
ActiveInvestors-own[buy sell pass price cash stocks]
```

Procedures

to setup:

```
to setup

clear-all
set exePrice Initial-StockPrice
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-ActiveInvestors nActiveInvestors
create-market-maker 1
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash

let side sqrt (nrandomInvestors + nActiveInvestors + 1)
let step max-pxcor / side

ask turtles[if(breed = randomInvestors or breed = ActiveInvestors)
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]

ask market-maker
[set size 4
 set color blue
 set shape "house"
 set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2 ]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + nActiveInvestors + 1]
[if x > (side - 1) * step
```

```
[set y y + step
 set x 0 ]
ask turtle an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

In this procedure we added the presence of *ActiveInvestors* through the following instruction, the number of *ActiveInvestors* that will be present in the simulation is defined according to the slider *nActiveInvestors*.

```
create-ActiveInvestors nActiveInvestors
```

Then we assign specific values to investors exclusive variable, since they share the same variable we can define them with just one command.

```
ask turtles[if(breed = randomInvestors or breed = ActiveInvestors)
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]
```

The remaining part of this procedure is the same, creates a market maker with its own variables and then place orderly in the 'world' every agent, hence the program is ready to run simulations.

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[ask turtles [if (breed = randomInvestors or breed = ActiveInvestors)[trade]]]

end
```

Go procedure, as before composed by the following procedure:

- *initialize*
- *book-market-maker*
- *trade*

All procedure are the same as before, the only difference in the code is that rather than use the command

```
ask randomInvestors
```

we will use

```
ask turtles [if (breed = randomInvestors or breed = ActiveInvestors)
```

, in this way all traders operate randomly regardless their 'breed'.

to initialize:

```
to initialize

if market-maker?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]

ask turtles
[if(breed = randomInvestors or breed = ActiveInvestors)
[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 buy false set sell true]]
[set buy false set sell false]

set price
    exePrice + (random-normal 0 Initial-StockPrice * 0.05)

if exePrice < Initial-StockPrice * 0.2 [if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]]

set logB []
set logS []

tick

end
```

Procedure used to set initial variables in order to run the simulation.

to book-market-maker:

```
to book-market-maker

ask turtles[if(breed = randomInvestors or breed = ActiveInvestors)
[ifelse random-float 1 < p-of-acting [ trade-with-market-maker][trade]]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*.

to trade:

```
to trade

if not pass
[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 (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)

 ask turtle agB [set stocks stocks + 1
                 set cash cash - exePrice]
 ask turtle agS [set stocks stocks - 1
                 set cash cash + exePrice]
set logB but-first logB
set logS but-first logS
```

```
if( market-maker? and update-bid-ask?)
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                 [set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                     [set price exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors  [if random-float 1 < p-of-Active-update-price
                     [set price exeprice + (random-normal 0 exePrice * 0.05)]]
]]

if sell [set logS lput tmp logS]
         set logS sort-by [item 0 ?1 < item 0 ?2] logS
        ;show logS

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)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if( market-maker? and update-bid-ask?)
[ask market-maker [if random-float 1 < p-of-update-bid-ask
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
```

```
]]graph]
end
```

In this procedure also *ActiveInvestors* according to the switch *update-price?* will update their price at every price changing with a certain probability defined by the slider *p-of-active-update-price*, through this command:

```
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
```

to trade-with-market-maker:

```
to trade-with-market-maker

if not pass
[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
ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0
        or (not empty? logB and empty? logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0

[set exePrice ask-price
 let agB item 1 (item 0 logB)
```

```
ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks - 1
                  set market-maker-cash market-maker-cash + exePrice

                  set market-maker-profit
                      market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks * Initial-StockPrice)

       if update-bid-ask? [if random-float 1 < p-of-update-bid-ask
                           [set ask-price exePrice + bid-ask-spread / 2
                            set bid-price exePrice - bid-ask-spread / 2]]]
 if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]

set logB but-first logB][ buyer-trade-with-investors]

if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs
        ;show logS
ifelse (not empty? logB and not empty? logS)
        and bid-price > item 0 ( item 0 logB)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price

[set exePrice  bid-price
 let agS item 1 (item 0 logS)

ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks + 1
                  set market-maker-cash market-maker-cash - exePrice
```

```
                    set market-maker-profit
                        market-maker-cash + (market-maker-stocks * exePrice)
                        - Initial-Cash - (Initial-nStocks * Initial-StockPrice)



    if update-bid-ask? [ if random-float 1 < p-of-update-bid-ask
                        [set ask-price exePrice + bid-ask-spread / 2
                         set bid-price exePrice - bid-ask-spread / 2]]]

 if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]]

 set logS but-first logS][seller-trade-with-investors]

graph]

end
```

If the agent could not trade with the market maker, she will try to trade with the other investors through:

- *to buyer-trade-with-investors*
- *to seller-trade-with-investors*

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

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)
```

```
 ask turtle agB [set stocks stocks + 1
                 set cash cash - exePrice]
 ask turtle agS [set stocks stocks - 1
                 set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

 if update-bid-ask?
 [ask market-maker[if random-float 1 < p-of-update-bid-ask
                  [set ask-price
                           exePrice + bid-ask-spread / 2
                   set bid-price
                       exePrice - bid-ask-spread / 2]]]

 if update-price?
 [ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
  ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]]

end
```

to seller-trade-with-investors:

```
to seller-trade-with-investors

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)

 ask turtle agB [set stocks stocks + 1
                 set cash cash - exePrice]
```

```
 ask turtle agS [set stocks stocks - 1
                 set cash cash + exePrice]
 set logB but-first logB
 set logS but-first logS

 if update-bid-ask?
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                 [set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]]

 if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                     [set price exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price exeprice + (random-normal 0 exePrice * 0.05)]]]
end
```

to graph:

```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

set-current-plot "market-maker-profit"
plot market-maker-profit

end
```

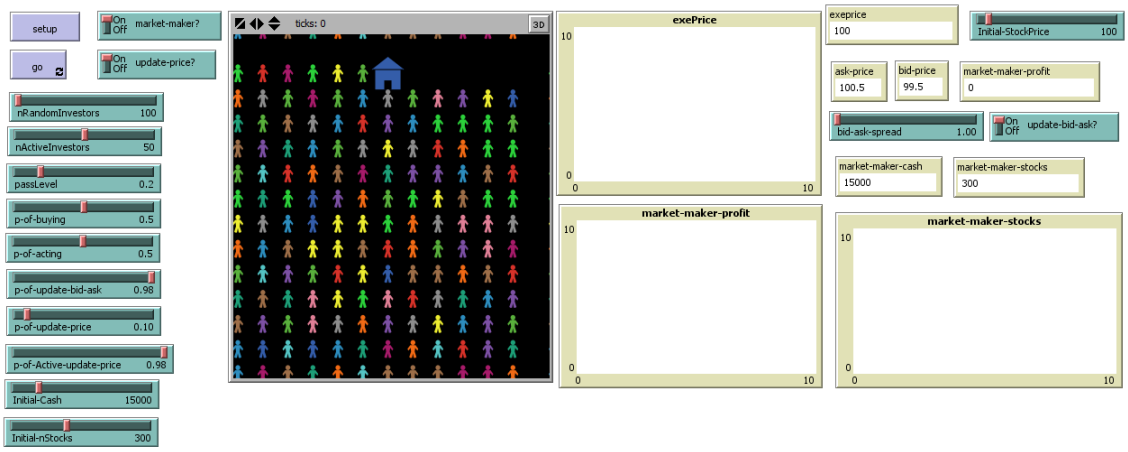
The same procedure as before, in order to plot stock price. number of stocks owned by the market maker and market maker profit.

Finally the difference with the previous model is the presence of two different kind of investors, one that does not react to market changes and another ready to adjust behavior according price movements. The market maker as before update its bid and ask whenever a transaction make price going up or down.

4.8.2 Interface

I added in this interface two sliders.

- *nActiveInvestors*: slider representing the number of *ActiveInvestors* in the market.
- *p-of-active-update-price*: slider representing the probability that *ActiveInvestors* will adjust their condition to trade when stock price changes.



4.8.3 Market maker model 05.01

In this implementation of the model *market maker model 04*, we allow to *p-of-acting* to change during the simulation. The value of the variable *p-of-acting* represents the probability that the market maker will be able to trade with the random investors.

In the previous models this value is fixed at 0.5, in this development this value is fixed at that value at the beginning of the simulation, then it moves according to price movements. If the price goes up *p-of-acting* will increase, while if it goes down the probability will descrease. In order to get this I added the following part of code

```
if exePrice < bid-price[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > bid-price[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

and

```
if exePrice < ask-price[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > ask-price[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

in the procedure *to trade-with-market-maker*
then

```
if exePrice < item 0 (item 0 logS)
[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > item 0 (item 0 logS)
[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

in procedures *to buyer-trade-with-investors* and *to trade* and

```
if exePrice < item 0 (item 0 logB)
[set p-of-acting p-of-acting + 1 / nRandomInvestors]
if exePrice > item 0 (item 0 logB)
[set p-of-acting p-of-acting - 1 / nRandominvestors]
```

in procedures *to seller-trade-with-investors* and *to trade*.

In this model we can analyze in the interface how *p-of-acting* changes when simulation occur.

4.8.4 Market maker model 05.02

In this additional implementation we consider investors set their condition price on the basic of a fixed and a variable part as before, but we decrease the size of the variable part. In the first models we consider the fixed part to be equal to the stock price (*exePrice*=1000) and the variable part equal to a number of a random normal distribution with zero mean and standard deviation equal to 100. This allow market maker to set bid and ask price with a too high spread, continuing to trade. This is not very likely to be. In order to analyze a more realistic environment we have to decrease the variable part.

In *market maker model 03* we introduce a new way of measure the variable part: equal to 5% of *Initial-StockPrice*. Now in this model we decrease further the variable part setting it equal to 1% of the initial stock price. The way of measuring the variable part is also changed when investors update *price*.

The value of variable *price* now is given by:

```
[set price exeprice + (random-normal 0 exePrice * 0.01)]
```

4.9 Market maker model 06

In the previous models we consider as profit

```
market-maker-cash + (market-maker-stocks * exePrice)
  - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
```

The amount of cash, the number of stocks owned valued at the current stock price, subtracting the total amount of the initial investment. In this way market maker profit could derive from a very high number of transactions, or from price movements. Suppose that the initial number of stocks owned by the market maker is 300, and it closes zero transactions, if the stock price increase for instance from 100 to 150 the market maker profit will be equal 50$\dot{3}$00, since the value of its stocks is increased by 50 for each security. In this work we want to analyze market maker optimal behavior achieving profits through a very high number of transactions, regardless price movements. Hence we developed this last model. Starting from this model as before I developed two variation of the model, in order to analyze more accurately this final market.

- ***market maker model 06.01***
- ***market maker model 06.02***

4.9.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
breed [ActiveInvestors ActiveInvestor]
randomInvestors-own[buy sell pass price cash stocks]
ActiveInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit ask-price bid-price
         market-maker-stocks market-maker-cash market-maker-transaction-b
         market-maker-transaction-s bid-ask-profit Initial-investment
         market-maker-profits net-position daily-bid-ask-profit
         daily-net-position daily-market-maker-transaction-b
         daily-market-maker-transaction-s daily-Initial-cash]
```

We added new global variables

- *Initial-investment*: necessary amount of money for the market maker to operate in the market.
- *daily-Initial-cash*: amount of cash owned by the market maker at the beginning of each trading day.
- *market-maker-transaction-b*: total number of buy transactions.
- *daily-market-maker-transaction-b*: number of buy transaction during the current trading day.
- *market-maker-transaction-s*: total number of sell transactions.
- *daily-market-maker-transaction-s*: number of sell transaction during the current trading day.
- *net-position*: equal to the difference between *market-maker-transaction-b* and *market-maker-transaction-s*.
- *daily-net-position*: equal to the difference between *daily-market-maker-transaction-b* and *daily-market-maker-transaction-s*.
- *bid-ask-profit*: equal to the lowest number between *market-maker-transaction-s* and *market-maker-transaction-b* multiplied by the bid-ask spread.
- *daily-bid-ask-profit*: equal to the lowest number between *daily-market-maker-transaction-s* and *daily-market-maker-transaction-b* multiplied by the bid-ask spread.
- *market-maker-profits*: market maker profits equal to

```
ifelse daily-net-position >= 0
[set market-maker-profits
 daily-bid-ask-profit - Initial-investment
 - (daily-net-position * bid-price)
 + (market-maker-stocks * exePrice) + daily-Initial-cash]
[set market-maker-profits
     daily-bid-ask-profit - Initial-investment
     - (daily-net-position * ask-price)
     + (market-maker-stocks * exePrice) + daily-Initial-cash]
```

The value of the global variable *market-maker-profits* is equivalent to *market-maker-profit*, variable already defined in the previous models.

We added these variables through this command.

```
globals[daily-market-maker-transaction-b
        daily-market-maker-transaction-s
        market-maker-transaction-b market-maker-transaction-s
        Initial-investment market-maker-profits net-position
        daily-net-position daily-bid-ask-profit bid-ask-profit
        daily-Initial-cash]
```

Procedures

to setup:

```
to setup

clear-all
set exePrice Initial-StockPrice
set logB []
set logS []
reset-ticks

create-randomInvestors nRandomInvestors
create-ActiveInvestors nActiveInvestors
create-market-maker 1
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash
set Initial-investment (Initial-nStocks * Initial-StockPrice) + Initial-Cash
set daily-Initial-cash market-maker-cash

let side sqrt (nrandomInvestors + nActiveInvestors + 1)
let step max-pxcor / side

ask turtles[if(breed = randomInvestors or breed = ActiveInvestors)
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]
```

```
ask market-maker
[set size 4
 set color blue
 set shape "house"
 set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2 ]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + nActiveInvestors + 1]
[if x > (side - 1) * step
[set y y + step
 set x 0 ]
ask turtle an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

In this procedure we added

```
set Initial-investment
(Initial-nStocks * Initial-StockPrice) + Initial-Cash

set daily-Initial-cash Initial-cash
```

Setting the new global variable *Initial-investment* equal to the sum of *Initial-cash* and the amount of money needed to buy *Initial-nStocks*. Then we set at the beginning of the simulation *daily-Initial-cash*, representing the amount of money owned by the market maker at the beginning of each trading day, equal to *Initial-cash*.

The remaining part of this procedure is the same, creates a market maker with its own variables and then place orderly in the 'world' every agent, hence the program is ready to run simulations.

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[ask turtles[if(breed = randomInvestors or breed = ActiveInvestors)[trade]]]

mm-profits

end
```

Go procedure, now is composed by the following 4 procedure:

- *initialize*
- *book-market-maker*
- *trade*
- *mm-profits*

to initialize:

```
to initialize

set daily-Initial-cash market-maker-cash
set daily-market-maker-transaction-s 0
set daily-market-maker-transaction-b 0
set daily-net-position 0
set daily-bid-ask-profit 0

if market-maker?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
set bid-price exePrice - bid-ask-spread / 2]]

ask turtles
[if (breed = randomInvestors or breed = ActiveInvestors)
[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 buy false set sell true]]
[set buy false set sell false]

set price
    exePrice + (random-normal 0 Initial-StockPrice * 0.05)

if exePrice < Initial-StockPrice * 0.2 [if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]
if pass [set color gray]
if buy  [set color red]
if sell [set color green]]]

set logB []
set logS []
tick

end
```

Procedure used to set initial variables in order to run the simulation.

to book-market-maker:

```
to book-market-maker

ask turtles[if(breed = randomInvestors or breed = ActiveInvestors)[
ifelse random-float 1 < p-of-acting [ trade-with-market-maker][trade]]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*.

to trade:

```
to trade

if not pass
[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 (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)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if( market-maker? and update-bid-ask?)
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors  [if random-float 1 < p-of-Active-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]]

if sell [set logS lput tmp logS]
         set logS sort-by [item 0 ?1 < item 0 ?2] logS

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)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if( market-maker? and update-bid-ask?)
[ask market-maker [if random-float 1 < p-of-update-bid-ask
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]]
                     graph]
end
```

to trade-with-market-maker:

```
to trade-with-market-maker

if not pass
[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
ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0
```

```
        or (not empty? logB and empty? logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0

[set exePrice ask-price
 set daily-market-maker-transaction-s daily-market-maker-transaction-s + 1
 let agB item 1 (item 0 logB)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks - 1
                  set market-maker-cash market-maker-cash + exePrice

if update-bid-ask? [if random-float 1 < p-of-update-bid-ask
                    [set ask-price exePrice + bid-ask-spread / 2
                     set bid-price exePrice - bid-ask-spread / 2]]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]

set logB but-first logB][ buyer-trade-with-investors]

if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs
        ;show logS
ifelse (not empty? logB and not empty? logS)
        and bid-price > item 0 ( item 0 logB)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price

[set exePrice  bid-price
 set daily-market-maker-transaction-b daily-market-maker-transaction-b + 1
 let agS item 1 (item 0 logS)
```

```
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks + 1
                  set market-maker-cash market-maker-cash - exePrice

if update-bid-ask? [ if random-float 1 < p-of-update-bid-ask
                    [set ask-price exePrice + bid-ask-spread / 2
                     set bid-price exePrice - bid-ask-spread / 2]]]

if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
 ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]]

set logS but-first logS][seller-trade-with-investors]

graph]

end
```

In this procedure I added

```
set dayly-market-maker-transaction-s dayly-market-maker-transaction-s + 1

set dayly-market-maker-transaction-b dayly-market-maker-transaction-b + 1
```

If the agent could not trade with the market maker, she will try to trade with the other investors through:

- *to buyer-trade-with-investors*
- *to seller-trade-with-investors*

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

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)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if update-bid-ask?
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                  [set ask-price exePrice + bid-ask-spread / 2
                   set bid-price exePrice - bid-ask-spread / 2]]]

if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]]]

end
```

to seller-trade-with-investors:

```
to seller-trade-with-investors

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)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if update-bid-ask?
[ask market-maker[if random-float 1 < p-of-update-bid-ask
                 [set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]]

if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]]
end
```

to mm-profits:

```
to mm-profits

set market-maker-profit
market-maker-cash + (market-maker-stocks * exePrice)
- Initial-Cash - (Initial-nStocks * Initial-StockPrice)

ifelse dayly-market-maker-transaction-s <= dayly-market-maker-transaction-b
[set dayly-bid-ask-profit
 dayly-bid-ask-profit + (dayly-market-maker-transaction-s * bid-ask-spread)]
[set dayly-bid-ask-profit
 dayly-bid-ask-profit + (dayly-market-maker-transaction-b * bid-ask-spread)]

set dayly-net-position
 (dayly-market-maker-transaction-b - dayly-market-maker-transaction-s)

ifelse dayly-net-position >= 0
[set market-maker-profits
 daily-bid-ask-profit - Initial-investment - (daily-net-position * bid-price)
 + (market-maker-stocks * exePrice) + daily-Initial-cash]
[set market-maker-profits
daily-bid-ask-profit - Initial-investment - (daily-net-position * ask-price)
 + (market-maker-stocks * exePrice) + daily-Initial-cash]


set net-position net-position + daily-net-position
set market-maker-transaction-b
    market-maker-transaction-b + daily-market-maker-transaction-b
set market-maker-transaction-s
     market-maker-transaction-s + daily-market-maker-transaction-s
set bid-ask-profit bid-ask-profit + daily-bid-ask-profit

end
```

This new procedure is necessary to compute the market maker profit, To do this we define *market-maker-profits* as:

```
[set market-maker-profits
daily-bid-ask-profit - Initial-investment - (daily-net-position * bid-price)
 + (market-maker-stocks * exePrice) + daily-Initial-cash]
```

if *daily-net-position* is greater or equal than zero, otherwise

```
[set market-maker-profits
daily-bid-ask-profit - Initial-investment - (daily-net-position * ask-price)
+ (market-maker-stocks * exePrice) + daily-Initial-cash]
```

Finally the market maker profit is influenced positively by *daily-bid-ask-profit*, and we can have a positive or negative impact according to *daily-net-position* and price movements.

to graph:

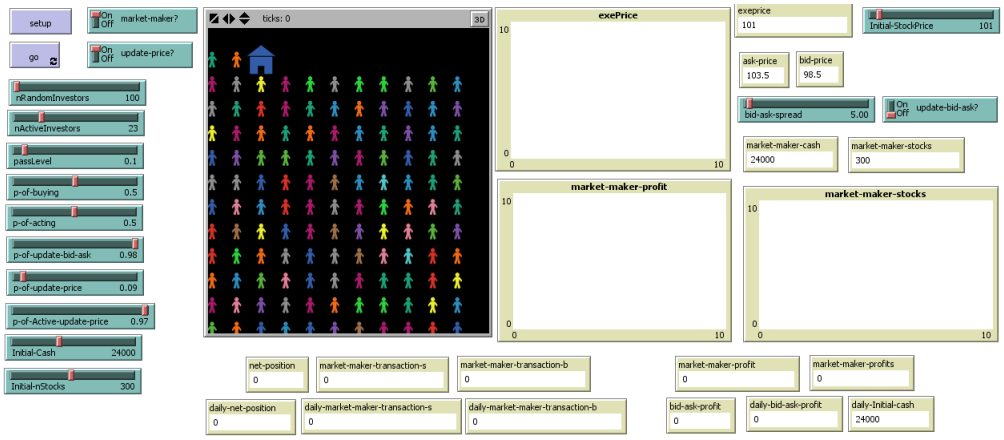
```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

set-current-plot "market-maker-profit"
plot market-maker-profit

end
```

The same procedure as before, in order to plot stock price. number of stocks owned by the market maker and market maker profit.

4.9.2 Interface

I added in this interface the following monitors:

- *daily-Initial-cash*
- *market-maker-transaction-b*
- *daily-market-maker-transaction-b*
- *market-maker-transaction-s*
- *daily-market-maker-transaction-s*
- *net-position*
- *daily-net-position*
- *bid-ask-profit*
- *daily-bid-ask-profit*
- *market-maker-profits*



4.9.3 Market maker model 06.01

In this model we include three new global variables, with the respective monitors in the interface. These variables are necessary to understand how many transaction occur in the simulation, splitting them in buy transaction and sell transaction, so finally we could analyze the exact percentage of transactions that market maker is able to achieve with a certain bid-ask spread. In detail the variable are the following:

- *transaction-b*: number of buy transaction, transaction closed at the best conditions for the buyer. This agent was able to find a counterpart when she enter the market.
- *transaction-s*: number of seller transaction, transaction closed at the best conditions for the seller. This agent was able to find a counterpart when she enter the market.
- *transaction*: Total number of transactions.

In order to define this variable initially we create them through *globals* command.

```
globals [transaction-b transaction-s transaction]
```

then we added this part of code whenever a deal occurs, if the agent is a buyer

```
set transaction-b transaction-b + 1
set transaction transaction + 1
```

otherwise

```
set transaction-s transaction-s + 1
set transaction transaction + 1
```

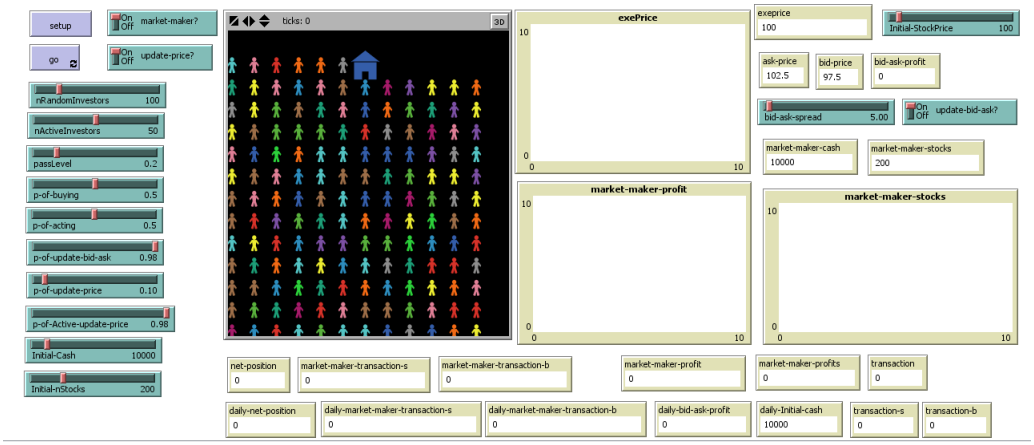
In order to find the percentage we have to compare the value of *market-maker-transaction-s* with *transaction-b*, and *market-maker-transaction-b* with *transaction-s*. A difference with respect *market maker model 06* is the standard deviation of the variable component of *price*, before was defined as

```
set price exePrice + (random-normal 0 Initial-StockPrice * 0.05)
```

while in this model we increased the percentage of *Initial-StockPrice* from 5% up to 10%,

```
set price exePrice + (random-normal 0 Initial-StockPrice * 0.05)
```

in order to analyze possible effects during the experiments with BehaviorSearch, to see how the value of the standard deviation affects the optimal bid-ask spread. In the interface we simply added the respective monitors for the added global variables, in order to identify the values.



4.9.4 Market maker model 06.02

In this variation we simply change the value of standard deviation, before we increased up to 10%, now we decreased the value to 2.5%,

```
set price exePrice + (random-normal 0 Initial-StockPrice * 0.025)
```

this model will be useful during the BehaviorSearch experiments procedure.

Chapter 5

NetLogo and R

5.1 NetLogo-Rserve-Extension

The NetLogo-Rserve-Extension connects the simulation platform NetLogo with the statistical analysis software R via Rserve. It adds some new primitives to NetLogo, which offers the interchange of data with R and the call of R functions from NetLogo.

Launching Rserve

Rserve comes now as an R package, so one way to start Rserve is from within R, just type:

```
library(Rserve)
Rserve()
```

That command knows how to find Rserve, how to setup the environment and how to start it, regardless of your platform.

```
> library(Rserve)
> Rserve()
Starting Rserve...
 "C:\Users\Utente\DOCUME~1\R\WIN-LI~1\3.2\Rserve\libs\x64\Rserve.exe"
```

5.1.1 Primitives

The primitives I used to develop my model *market maker model 03_ R* are the following:

- *rserve:init* Initializes a connection to an Rserve server. There can only one connection at a time.
- *rserve:close*

  It closes the connection to Rserve server, if there is one.
- *rserve:isConnected*

  Reports if there is a connection established.
- *rserve:eval*

  rserve:eval R-command It evaluates the submitted R command. The R command shouldn't return a value.
- *rserve:put*

  rserve:put name value. Creates a new variable in R with the name 'name'. The value can be a String, Number, Boolean or List. NetLogo Lists are converted to R vectors, if all entries are of the same data type. If a NetLogo list contains different data types (mixed Strings, Numbers of Booleans), it will be converted into an R list.

5.2 NetLogo market maker model 03_R

In this NetLogo model we are able to connect it to the statistical software R through Rserve. In this way we can create R-Variables with values from NetLogo variables or agents and others to evaluate commands in R with and without return values

5.2.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
randomInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit ask-price bid-price
         market-maker-stocks market-maker-cash tsprice t0-price]
extensions [Rserve]
```

In order to use the extension in the model I added this command at the beginning of the code. We have new globals variable:

- tsprice: a vector storing all prices changes, expressed in terms of return.
- t0-price: representing the price before the movement, is used to compute the return.

```
extensions [Rserve]
globals    [tsprice t0-price]
```

Procedures:

to init:

```
to init

rserve:init 6311 "localhost"

end
```

Initializes a connection to an Rserve server.

to r-idle:

```
to r-idle ;; a "forever" method

rserve:eval "Sys.sleep(0.01)"

end
```

In order to avoid that the plot frame will be locked, we have to give some cpu time to R by executing rserve:eval "Sys.sleep(0.01)" with a forever button.

to setup:

```
to setup

clear-all
set exePrice Initial-StockPrice
set logB []
set logS []
set tsprice[]

reset-ticks

create-randomInvestors nRandomInvestors
create-market-maker 1
set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash

let side sqrt (nrandomInvestors + 1)
let step max-pxcor / side

ask randomInvestors

[set shape "person"
 set size 2
 set stocks 0
 set cash 0]
```

```
ask market-maker
[set size 4
 set color blue
 set shape "house"
 set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]

let an 0

let x 0
let y 0
while [an < nRandomInvestors +1]
[if x > (side - 1) * step
[set y y + step
 set x 0 ]
ask turtles an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

The same procedure as in *market maker model 03*. I simply added the new empty list *tsprice* in the following way.

```
set tsprice[]
```

to go:

```
to go

initialize

if market-maker?
[book-market-maker]
if not market-maker?
[ask randomInvestors [trade]]
if rserve:isConnected
[send-to-R
if ticks = 100 [stop]]

end
```

Now this procedure is composed by 4 different procedures:

- initialize
- book-market-maker
- trader
- send-to-R

to initialize:

```
to initialize

if market-maker?[
ask market-maker[set ask-price exePrice + bid-ask-spread / 2
set bid-price exePrice - bid-ask-spread / 2]]

ask randomInvestors
[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 buy false set sell true]]
[set buy false set sell false]

set price
    exePrice + (random-normal 0 Initial-StockPrice * 0.05)

if exePrice < Initial-StockPrice * 0.2 [if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]

set logB []
set logS []

tick

end
```

Same procedure used to set initial variables in order to run the simulation.

to book-market-maker:

```
to book-market-maker

ask randomInvestors [
ifelse random-float 1 < p-of-acting [ trade-with-market-maker][trade]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*. In these instrunctions we have new elements in order to fill the vector *tsprice*. This vector will be analyzed through R.

```
to trade-with-market-maker

if not pass
[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
ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0
        or (not empty? logB and empty? logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0

[set t0-price exePrice
 set exePrice ask-price
 set tsprice lput ((exePrice - t0-price)/ t0-price)tsprice

let agB item 1 (item 0 logB)
```

```
ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks - 1
                  set market-maker-cash market-maker-cash + exePrice
                  set market-maker-profit
                      market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
if update-spread?
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]
 set logB but-first logB][ buyer-trade-with-investors]

if sell [set logS lput tmp logS]
           set logS sort-by [ item 0 ?1 < item 0 ?2] logs
        ;show logS

ifelse (not empty? logB and not empty? logS)
        and bid-price > item 0 ( item 0 logB)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price

[set  t0-price exePrice
 set exePrice  bid-price
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

let agS item 1 (item 0 logS)

ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks + 1
                  set market-maker-cash market-maker-cash - exePrice
                  set market-maker-profit
                      market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
if update-spread?
```

```
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]
 set logS but-first logS][seller-trade-with-investors]
graph]end
```

We added before the price of new transaction is fixed this instruction

```
set  t0-price exePrice
```

In this way we can save the last price, that we need to compute the return according the new price generated by the transaction, Then we fill the vector *tsprice* with the return through this command.

```
set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice
```

We added this part of code in every procedure including the possibility to make transactions.

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

if (not empty? logB and not empty? logS) and
    item 0 (item 0 logB) >= item 0 (item 0 logS)

[set  t0-price exePrice
 set exePrice item 0 (item 0 logS)
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB
set logS but-first logS
if update-spread? [ask market-maker
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2 ]]
```

```
]

end
```

to seller-trade-with-investors

```
to seller-trade-with-investors

if (not empty? logB and not empty? logS) and
    item 0 (item 0 logB) >= item 0 (item 0 logS)

[set  t0-price exePrice
 set exePrice item 0 (item 0 logB)
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice
let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if update-spread?
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2 ]]
]
end
```

to trade:

```
to trade

if not pass
[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 (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) >= item 0 (item 0 logS)
```

```
[set t0-price exePrice
 set exePrice item 0 (item 0 logS)
 set tsprice lput ((exePrice - t0-price)/ t0-price)tsprice

let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask randomInvestor agS [set stocks stocks - 1
                         set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if( market-maker? and update-spread?)
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2]]

if sell [set logS lput tmp logS]
           set logS sort-by [item 0 ?1 < item 0 ?2] logS
        ;show logS

if (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) >= item 0 (item 0 logS)

[set  t0-price exePrice
 set exePrice item 0 (item 0 logB)
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

 let agB item 1 (item 0 logB)
 let agS item 1 (item 0 logS)

ask randomInvestor agB [set stocks stocks + 1
                        set cash cash - exePrice]
ask randomInvestor agS [set stocks stocks - 1
                        set cash cash + exePrice]
set logB but-first logB
set logS but-first logS
```

```
if( market-maker? and update-spread?)
[ask market-maker[set ask-price exePrice + bid-ask-spread / 2
                  set bid-price exePrice - bid-ask-spread / 2 ]]]
graph]
end
```

to send-to-R:

```
to send-to-R

if rserve:isConnected [
if ticks = 100 [
(rserve:put "stock_price" tsprice)
rserve:eval "ts.plot(stock_price)"]]

end
```

This procedure works only if there is a connection established, you can check this through the following primitive: print rserve:isConnected. The function of this procedure is to create a new variable in R with the name 'stock-price'. In this case values are numbers. NetLogo List *tsprice* is converted to R vector, since all entries are of the same data type. To get this we use rserve:put primitive. Then we use R command through rserve:eval, in particular

```
rserve:eval "ts.plot(stock_price)"
```

In this way we obtain several time series on a common plot.

to acf

```
to acf

if rserve:isConnected [
if ticks = 100 [
rserve:eval "acf(stock_price)" ]]

end
```

Procedure called through the respective button, allow us to compute the autocorrelation function of *stock-price* through R command acf.

to pacf

```
to pacf

if rserve:isConnected [
if ticks = 100 [rserve:eval "pacf(stock_price)"]]

end
```

Procedure called through the respective button, allow us to compute the partial autocorrelations of *stock-price* through R command pacf.

to graph

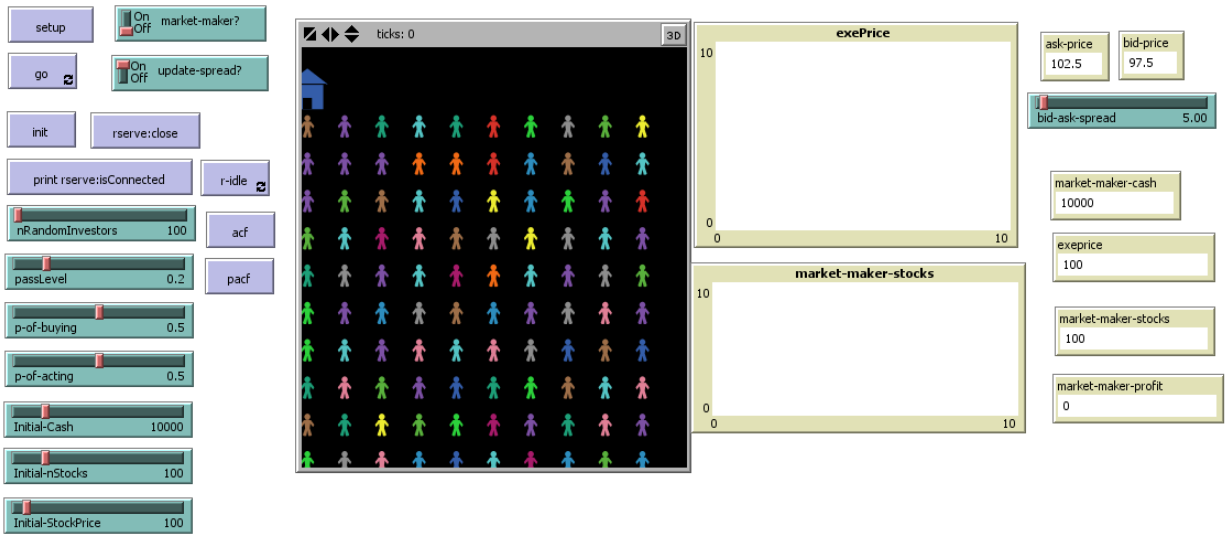
```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

end
```

The same procedure as in *market maker model 03*.

5.2.2 Interface

In the interface we added 6 *buttons*.

- *init*: initializes a connection to an Rserve server.
- *rserve:close*: it closes the connection to Rserve server, if there is one.
- *print rserve:isConnected*: reports if there is a connection established.
- *r-idle*: this 'forever' button gives some cpu time to R, so the plot frame will not be locked.
- *acf*: execute R command acf
- *pacf*: execute R command pacf



5.3 NetLogo market maker model 05_R

We computed the same procedure with *market maker model 05*, in order to analyze possible different results.

5.3.1 Code

At the very beginning we have this initial instructions:

```
breed [market-maker]
breed [randomInvestors randomInvestor]
breed [ActiveInvestors ActiveInvestor]
randomInvestors-own[buy sell pass price cash stocks]
ActiveInvestors-own[buy sell pass price cash stocks]
globals [logB logS exePrice market-maker-profit ask-price
         bid-price market-maker-stocks market-maker-cash tsprice t0-price]
extensions [Rserve]
```

Procedures:

to init:

```
to init

rserve:init 6311 "localhost"

end
```

Initializes a connection to an Rserve server.

to r-idle:

```
to r-idle

rserve:eval "Sys.sleep(0.01)"

end
```

In order to avoid that the plot frame will be locked, we have to give some cpu time to R by executing rserve:eval "Sys.sleep(0.01)" with a forever button.

to setup:

```
to setup

clear-all
set exePrice Initial-StockPrice
set logB []
set logS []
set tsprice[]

reset-ticks

create-randomInvestors nRandomInvestors
create-ActiveInvestors nActiveInvestors
create-market-maker 1

set market-maker-stocks Initial-nStocks
set market-maker-cash Initial-Cash

let side sqrt (nrandomInvestors + nActiveInvestors + 1)
let step max-pxcor / side

ask turtles [ if ( breed = randomInvestors or breed = ActiveInvestors)
[set shape "person"
 set size 2
 set stocks 0
 set cash 0]]
```

```
ask market-maker
[set size 4
 set color blue
 set shape "house"
 set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]

let an 0

let x 0
let y 0
while [an < nRandomInvestors + nActiveInvestors + 1]
[if x > (side - 1) * step
[set y y + step
 set x 0 ]
ask turtle an
[setxy x y]
 set x x + step
 set an an + 1]

end
```

The same procedure as in *market maker model 05*. I simply added the new empty list *tsprice* in the following way.

```
set tsprice[]
```

to go:

```
to go

initialize

if market-maker?
[book-market-maker]

if not market-maker?
[ask turtles [ if ( breed = randomInvestors or breed = ActiveInvestors)[trade]]]

if rserve:isConnected
[send-to-R if ticks = 100 [stop]]end
```

This procedure is composed by 4 different procedures:

- initialize
- book-market-maker
- trader
- send-to-R

to initialize:

```
to initialize

if market-maker?[
ask market-maker[set ask-price exePrice + bid-ask-spread / 2
set bid-price exePrice - bid-ask-spread / 2 ]]

ask turtles [ if ( breed = randomInvestors or breed = ActiveInvestors)
[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 buy false set sell true]]
[set buy false set sell false]

set price
    exePrice + (random-normal 0 Initial-StockPrice * 0.05)

if exePrice < Initial-StockPrice * 0.2 [if random-float 1 < p-of-buying
[set buy true set sell false set pass false]]

if pass [set color gray]
if buy  [set color red]
if sell [set color green]]]

set logB []
set logS []

tick

end
```

Same procedure used to set initial variables in order to run the simulation.

to book-market-maker:

```
to book-market-maker

ask turtles [if (breed = randomInvestors or breed = ActiveInvestors)
[ifelse random-float 1 < p-of-acting [trade-with-market-maker][trade]]]

end
```

Agents will follow with some probabilty (defined by the slider p-of-acting) the instruction of the procedure *trade-with-market-maker* or the procedure *trade*. In these instrunctions we have new elements in order to fill the vector *tsprice*. This vector will be analyzed through R.

```
to trade-with-market-maker

if not pass
[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
ifelse (not empty? logB and not empty? logS)
        and ask-price < item 0 (item 0 logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0
        or (not empty? logB and empty? logS)
        and item 0 (item 0 logB) >= ask-price
        and market-maker-stocks > 0

[set t0-price exePrice
 set exePrice ask-price
 set tsprice lput ((exePrice - t0-price)/ t0-price)tsprice

let agB item 1 (item 0 logB)

ask turtle agB  [set stocks stocks + 1
                 set cash cash - exePrice]
```

```
ask market-maker [set market-maker-stocks market-maker-stocks - 1
                  set market-maker-cash market-maker-cash + exePrice
                  set market-maker-profit
                      market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
if update-spread?
[if random-float 1 < p-of-update-bid-ask
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]]
 set logB but-first logB][ buyer-trade-with-investors]


if sell [set logS lput tmp logS]
         set logS sort-by [ item 0 ?1 < item 0 ?2] logs
        ;show logS
ifelse (not empty? logB and not empty? logS)
         and bid-price > item 0 ( item 0 logB)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price
        or (empty? logB and  not empty? logS)
        and bid-price >=  item 0 (item 0 logS)
        and market-maker-cash >= bid-price

[set t0-price exePrice
   set exePrice  bid-price
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

let agS item 1 (item 0 logS)

ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
ask market-maker [set market-maker-stocks market-maker-stocks + 1
                  set market-maker-cash market-maker-cash - exePrice
                  set market-maker-profit
                      market-maker-cash + (market-maker-stocks * exePrice)
                      - Initial-Cash - (Initial-nStocks * Initial-StockPrice)
if update-spread?
[if random-float 1 < p-of-update-bid-ask
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]]
```

```
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
]
set logS but-first logS][seller-trade-with-investors]
graph]

end
```

We added before the price of new transaction is fixed this instrunction

```
set  t0-price exePrice
```

In this way we can save the last price, that we need to compute the return according the new price generated by the transaction, Then we fill the vector *tsprice* with the return through this command.

```
set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice
```

We added this part of code in every procedure including the possibity to make transactions.

to buyer-trade-with-investors:

```
to buyer-trade-with-investors

if (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) >= item 0 (item 0 logS)

[set t0-price exePrice
 set exePrice item 0 (item 0 logS)
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
```

```
set logB but-first logB
set logS but-first logS
if update-spread? [if random-float 1 < p-of-update-bid-ask
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
]]

end
```

to seller-trade-with-investors

```
to seller-trade-with-investors

if (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) >= item 0 (item 0 logS)

[set  t0-price exePrice
 set exePrice item 0 (item 0 logB)
     set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if update-spread?  [if random-float 1 < p-of-update-bid-ask
                    [set ask-price exePrice + bid-ask-spread / 2
                     set bid-price exePrice - bid-ask-spread / 2 ]]
if update-price?
```

```
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]]]
end
```

to trade:

```
to trade

if not pass
[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 (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) >= item 0 (item 0 logS)

[set t0-price exePrice
 set exePrice item 0 (item 0 logS)
 set tsprice lput ((exePrice - t0-price)/ t0-price)tsprice

let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if( market-maker? and update-spread?)
[if random-float 1 < p-of-update-bid-ask
[set ask-price exePrice + bid-ask-spread / 2
```

```
 set bid-price exePrice - bid-ask-spread / 2 ]]

if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
                      [set price
                           exeprice + (random-normal 0 exePrice * 0.05)]]
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                     [set price
                          exeprice + (random-normal 0 exePrice * 0.05)]]
]]

if sell [set logS lput tmp logS]
             set logS sort-by [item 0 ?1 < item 0 ?2] logS
        ;show logS

if (not empty? logB and not empty? logS)
    and item 0 (item 0 logB) >= item 0 (item 0 logS)

[set t0-price exePrice
 set exePrice item 0 (item 0 logB)
 set tsprice lput ((exePrice - t0-price)/ t0-price) tsprice

let agB item 1 (item 0 logB)
let agS item 1 (item 0 logS)

ask turtle agB [set stocks stocks + 1
                set cash cash - exePrice]
ask turtle agS [set stocks stocks - 1
                set cash cash + exePrice]
set logB but-first logB
set logS but-first logS

if( market-maker? and update-spread?)
[if random-float 1 < p-of-update-bid-ask
[set ask-price exePrice + bid-ask-spread / 2
 set bid-price exePrice - bid-ask-spread / 2]]
if update-price?
[ask randomInvestors [if random-float 1 < p-of-update-price
[set price
     exeprice + (random-normal 0 exePrice * 0.05)]]
```

```
ask ActiveInvestors [if random-float 1 < p-of-Active-update-price
                    [set price
                         exeprice + (random-normal 0 exePrice * 0.05)]]
]]
graph]
end
```

to send-to-R:

```
to send-to-R

if rserve:isConnected [
if ticks = 100 [
(rserve:put "stock_price" tsprice)
 rserve:eval "ts.plot(stock_price)"]]

end
```

This procedure works only if there is a connection established, you can check this through the following primitive: print rserve:isConnected. The function of this procedure is to create a new variable in R with the name 'stock-price'. In this case values are numbers. NetLogo List *tsprice* is converted to R vector, since all entries are of the same data type. To get this we use rserve:put primitive. Then we use R command through rserve:eval, in particular

```
rserve:eval "ts.plot(stock_price)"
```

In this way we obtain several time series on a common plot.

to acf

```
to acf

if rserve:isConnected [
if ticks = 100 [
rserve:eval "acf(stock_price)"]]

end
```

Procedure called through the respective button, allow us to compute the autocorrelation function of *stock-price* throug R command acf.

to pacf

```
to pacf

if rserve:isConnected [
if ticks = 100 [rserve:eval "pacf(stock_price)"]]

end
```

Procedure called through the respective button, allow us to compute the partial autocorrelations of *stock-price* throug R command pacf.

to graph

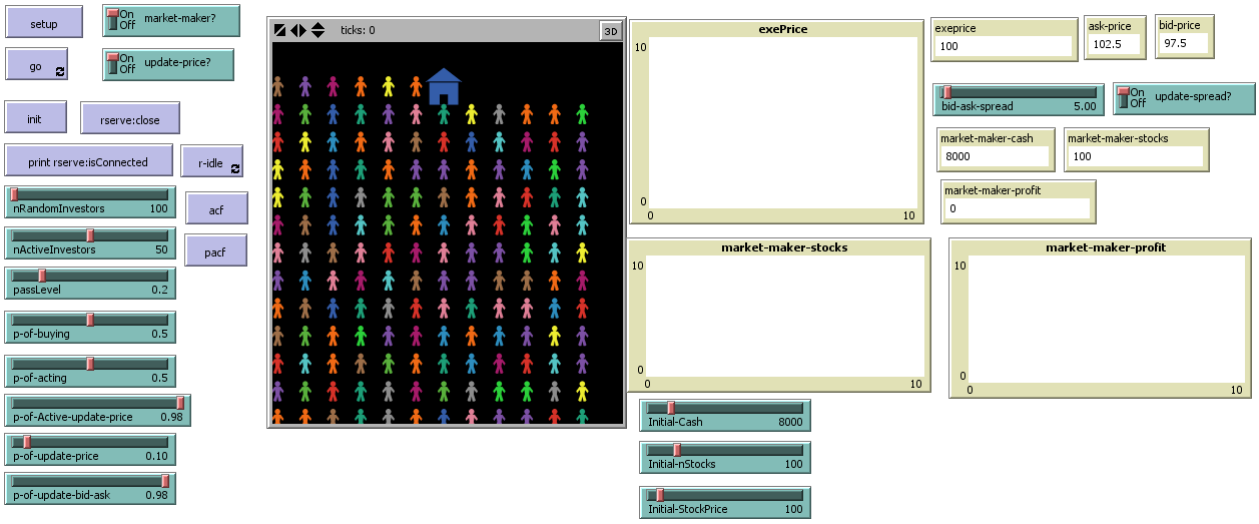
```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "market-maker-stocks"
plot market-maker-stocks

set-current-plot "market-maker-profit"
plot market-maker-profit

end
```

The same procedure as in *market maker model 05*.

5.3.2 Interface

As in the previous case we added the following 6 *buttons*:

- *init*: initializes a connection to an Rserve server.
- *rserve:close*: it closes the connection to Rserve server, if there is one.
- *print rserve:isConnected*: reports if there is a connection established.
- *r-idle*: this 'forever' button gives some cpu time to R, so the plot frame will not be locked.
- *acf*: execute R command acf
- *pacf*: execute R command pacf

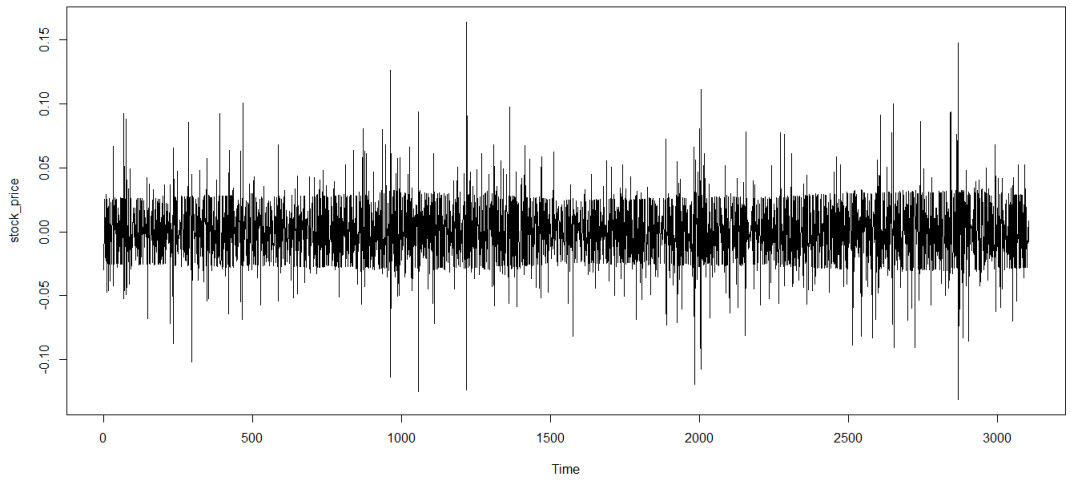


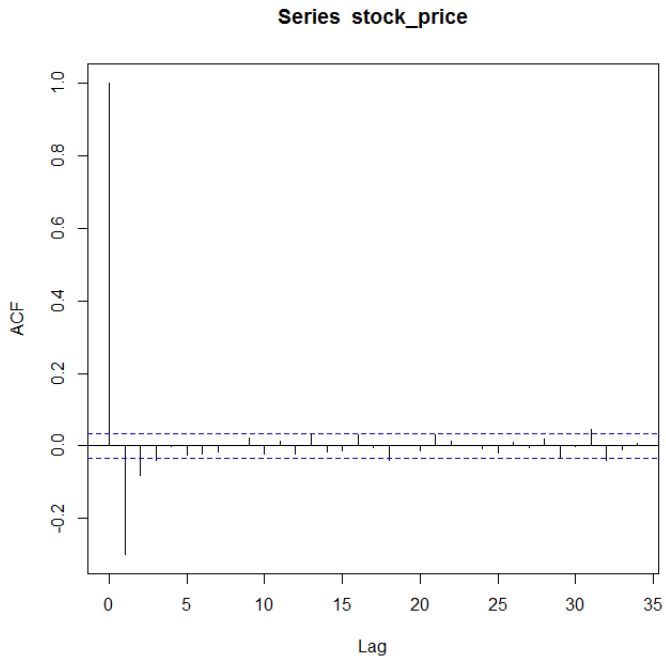
5.4 Results

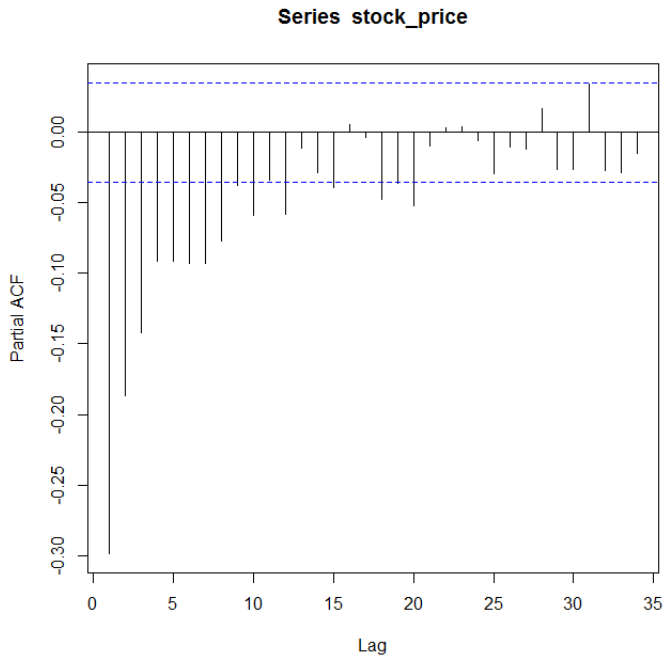
After the development of these models, we obtained through R the following results.

Market maker model 03_R

Running the simulation we obtain the following graphs through R, relative to price time series generated by NetLogo model considering 100 trading days. (*ticks*). In these simulations we considered both markets with and without market maker, the results are not influenced by this factor.





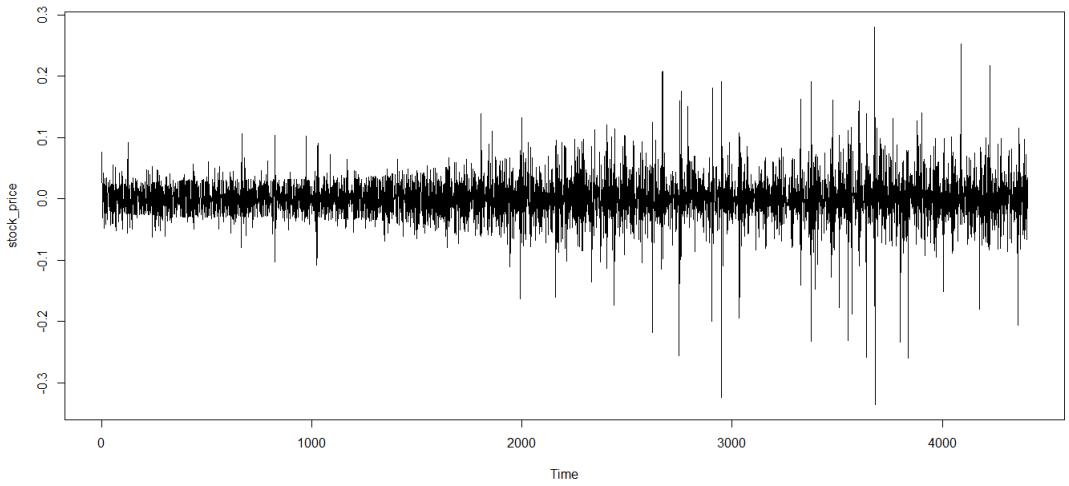


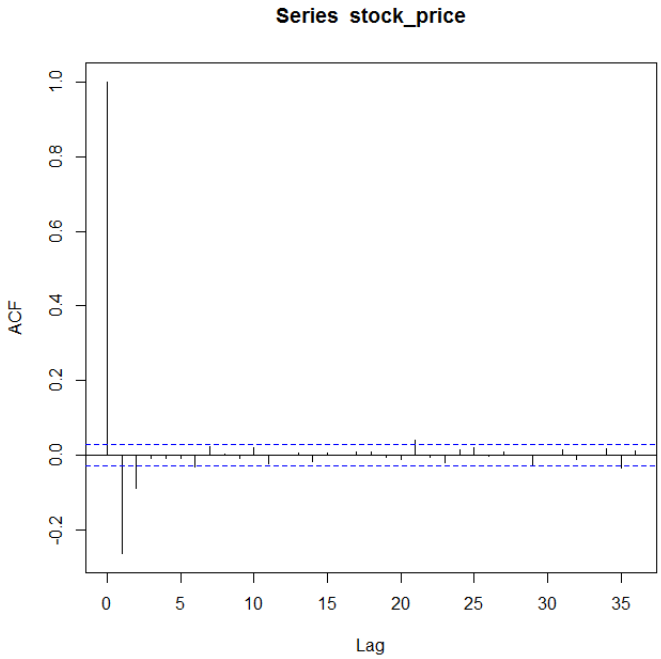
In these simulations we considered both markets with and without market maker, the results are not influenced by this factor. As we can notice in the ACF function we have significant values different of zero just in Lag 1, while the PACF function exponentially

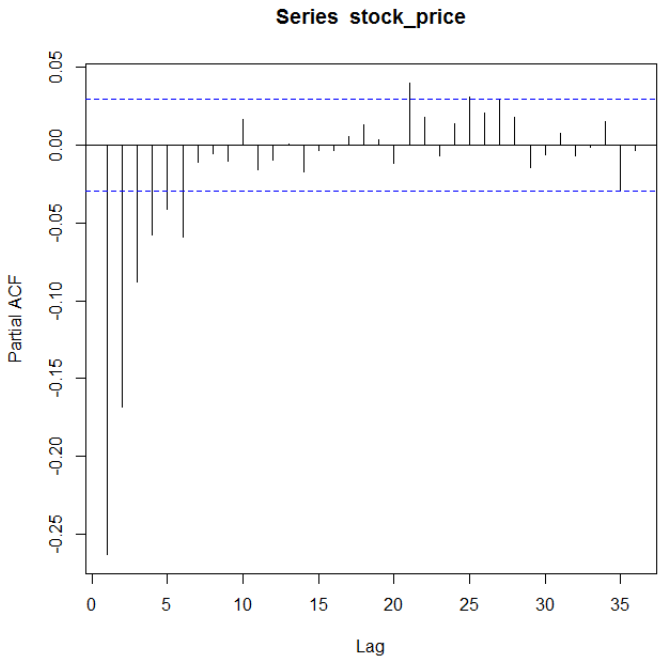
decrease to zero, we can recognize that the price of our model follows a moving average of the first order.

Market maker model 05_R

We computed the same procedure in this implemented model in order to... As we can see we obtain identical results, price following a moving average of the first order.






Series stock_price
Partial ACF
0.05
0.00
-0.05
-0.10
-0.15
-0.20
-0.25
0
5
10
15
20
25
30
35
Lag

Chapter 6

Simulation and BehaviorSearch experiments

After the development of the models we analyze the market behavior through many simulation and experiments via BehaviorSearch tool.

This is necessary to compute analysis on NetLogo models, in these experiments we will try to solve optimization problems using the most flexible search algorithm: the *Genetic algorithm* (StandardGA). To solve an optimization problem you need to compute the following steps:

1. define the *fitness function* (objective function), this will be the dependent variable.
2. define the variables that will be fixed and not allowed to range, specifying their values.
3. define the variables that will be allowed to range, affecting the final value of the objective function, specifying the exact range and the size of the increments. These will be considered as our independent variables.
4. Analyze the result.

Hence in these experiments we will have three types of variables:

- *independent variable:* function for which we want to solve the optimization problem. Represents the inputs. We want to see what values lead to a specific output.
- *dependent variable:* variable affected by independent variables, represent the outputs.
- *fixed variable:* variable not allowed to variate, they characterize the environment in which simulation occur.

In the next experiments we have more than one independent variable, we work on a multidimensional space.

6.1 Maximizing market-maker-profit

In the first part of the experiments we focus on the maximization problem of the market maker profits in the NetLogo model *market maker model 05*, we define as the *fitness function* (dependent variable) the value of *market-maker-profit*, equal to:

```
market-maker-cash + (market-maker-stocks * exePrice)
- Initial-Cash - (Initial-nStocks * Initial-StockPrice)
```

according to NetLogo model code.

We define the market contest, defining which variable will be fixed and which will be allowed to range, representing our independent variable.

6.1.1 Experiment 1: Initial-nStocks and update-bid-ask?

Independent variables

In the first experiment we define as independent variables:

- *Initial-nStocks*: representing the initial number of stocks owned by the market-maker. Defined through a *slider*. We specify as range

  ```
  [0 10 1000]
  ```

- *update-bid-ask?*: it is a Boolean variable, can assume value 'true' or 'false', if is true the market will update its bid and ask price following every stock price movement. Defined through a *switch*. The NetLogo instruction is the following:

  ```
  if update-bid-ask?
  [ask market-maker[if random-float 1 < p-of-update-bid-ask
                   [set ask-price exePrice + bid-ask-spread / 2
                    set bid-price exePrice - bid-ask-spread / 2]]]
  ```

Fixed variables

We define the value of fixed variables characterizing the market enviroment.

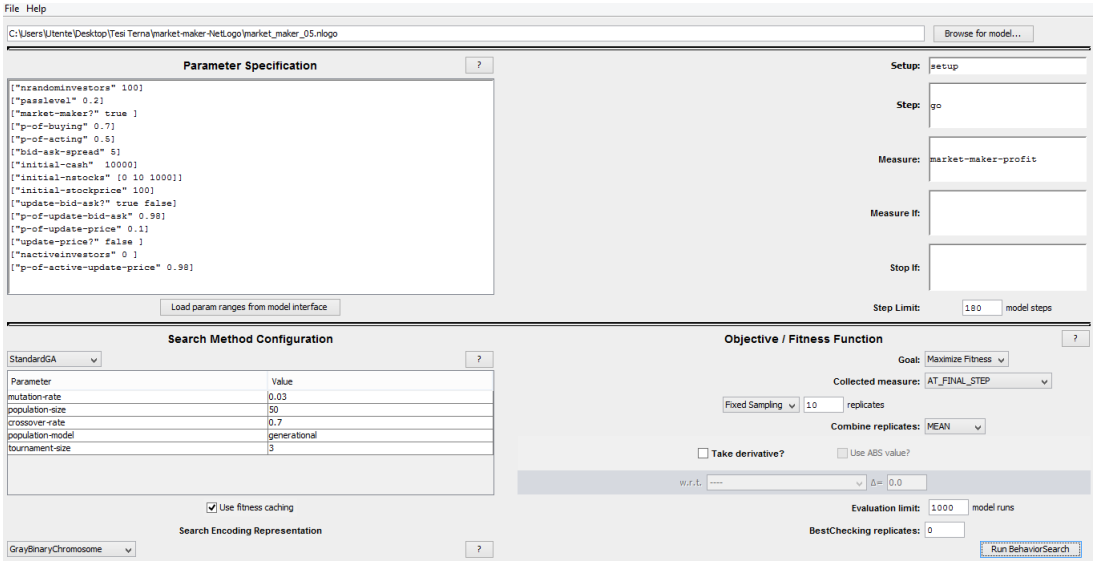
- *nRandomInvestors*: 100

- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-acting*: 0.5
- *bid-ask-spread*: 5, equal to the value of the standard devition of variable part of *price*

  ```
  (random-normal 0 Initial-StockPrice * 0.05)
  ```

  , *Initial-StockPrice* set equal to 100.
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000
- *Initial-StockPrice*: 100

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 180 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.1: experiment 1 results

Seed	Initial-nStocks	update-bid-ask?	Fitness
1	890	false	10574.3
2	820	false	14567.6
3	790	true	17279.9
4	980	true	11900
5	860	false	12201.2
6	800	false	13492.1
7	970	false	11377.4
8	630	false	8058.9
9	970	false	11693.8
10	900	false	16022.7

We can notice that:

- the average *Initial-nStocks* is 861, and the interval of values that it takes is [630 , 980]. On ten trials it results that nine times it takes value greater or equal to 790, in just one case it has a lower value and this leads to a lower fitness function. In this market we have just 100 investors, so *market-maker-profit* is driven more by price fluctuacions rather than the number of transactions closed. Hence in this enviroment the market maker pursue profits according to stock trends, this is a risky strategy, unlikely to be implemented.

- the average fitness is 12716.19 and lies in the range [8058.9 , 17279.9].

- just two times the value of *update-bid-ask?* is 'true'. In this kind of market with agents setting their expectation of price at the beginning of the trading day, the most of the times is not convenient for the market maker to update its bid-ask price, since the investors will not change price condition until the next day.

6.1.2 Experiment 2: Initial-Cash, Initial-nStocks and update-bid-ask?

Independent variables

In this experiment we define as independent variable also *Initial-cash*, the initial amount of cash owned by the market maker. We have three independent variable:

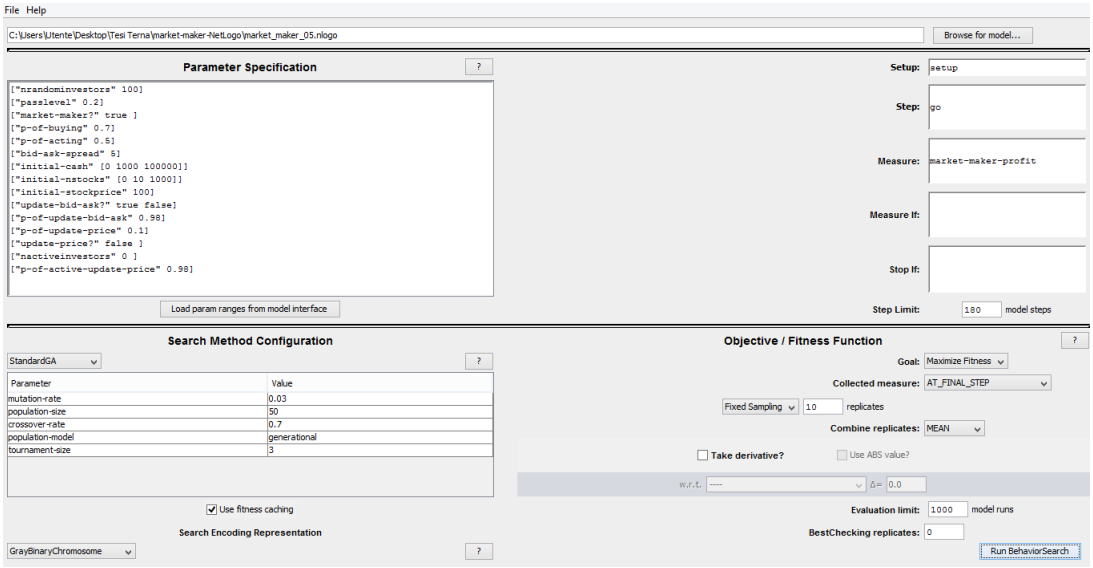
- *Initial-Cash*: the parameter range is specified as [0 1000 100000], defined through a *slider*.
- *Initial-nStocks*: the parameter range is specified as [0 10 1000]
- *update-bid-ask?*: can assume as value 'true' or 'false'

Fixed variables

We define the value of fixed variables characterizing the market enviroment.

- *nRandomInvestors*: 100
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-acting*: 0.5
- *bid-ask-spread*: 5
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-StockPrice*: 100

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 180 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.2: experiment 2 results

Seed	Initial-Cash	Initial-nStocks	update-bid-ask?	Fitness
1	79000	570	false	7434.93
2	14000	600	false	15474.3
3	84000	740	false	14148.2
4	62000	650	false	6824.72
5	64000	810	true	14197
6	55000	790	false	13210.8
7	19000	780	false	21391.1
8	27000	830	false	10360.6
9	91000	990	false	20790.6
10	99000	790	false	7625.18

we can notice that:

- the average value of *Initial-cash* is 59400, it takes value in the interval [14000 , 99000], we can conclude that the value of this variable does not influence the dependent variable *market-maker-profit*, and so we can assign to the market maker the necessary value of *Initial-cash* in order to operate in the market.
- the average value of *Initial-nStocks* is 755 and range in the interval [570 , 990].
- the average value of *Fitness* is 13145.74, in a range of [6824.72 , 21391.1], this is due to the fact that the value of the objective function in this kind of market is strongly affect by price movements, in order to make profits the stock price has to increase with respect the intial value at the beginning of the simulation at day 1.
- as before we can see that in such enviroment is convenient for the market-maker do not update bid and ask price during the trading day.

6.1.3 Experiment 3: Initial-Cash, Initial-nStocks update-bid-ask? and bid-ask-spread

Independent variables

In this experiment we add another independent variable, *bid-ask-spread*, the difference between the ask and the bid price, the market maker is willing to buy at the bid price and to sell at the ask, in order to make profits the latter has to exceed the former. The NetLogo model loaded is always *market-maker-model 05*, with a change in the code, we reduced the standard deviation of the variable component of *price* as follows:

```
set price exePrice + (random-normal 0 Initial-StockPrice * 0.01)
```

In this way the agents price condition will be distributed in a narrower interval.
We have four independent variable:

- *Initial-Cash*: the parameter range is specified as [0 1000 100000]
- *Initial-nStocks*: the parameter range is specified [0 10 1000]
- *update-bid-ask?*: can assume as value 'true' or 'false'
- *bid-ask-spread*: the parameter range is specified as [0 0.01 10], defined through a *slider*.

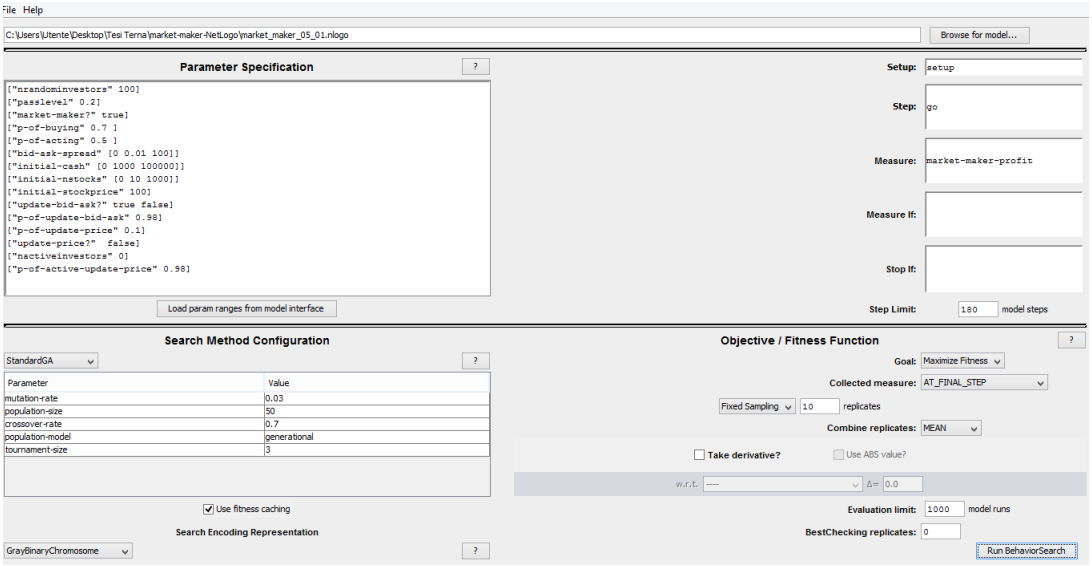
Fixed variables

We define the value of fixed variables characterizing the market enviroment.

- *nRandomInvestors*: 100
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-acting*: 0.5
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *p-of-update-price*: 0.1

- *p-of-active-update-price*: 0.98
- *Initial-StockPrice*: 100

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 180 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.3: experiment 3 results

Seed	Initial-Cash	Initial-nStocks	bid-ask-spread	update-bid-ask?	Fitness
1	15000	510	5.33	true	1391.21
2	76000	520	6.4	false	2445.01
3	65000	1000	5.98	false	2255
4	0	100	7.94	true	20.0874
5	43000	940	4.64	false	3803.07
6	49000	880	5.91	true	1737.87
7	19000	490	4.84	true	1899.28
8	28000	600	5.56	false	1405.18
9	56000	300	3.4	false	503.78
10	79000	420	4.54	false	7625.18

We can notice that:

- the average values for *Initial-cash*, *Initial-nStocks* and Fitness are respectively 43000, 676 and 20313.9, these values are not significant since they have high variance, they lie in wide intervals.

- the average *bid-ask-spread* is 5.45, it takes value in the interval [3.4 , 7.94].

- the value of *update-bid-ask-spread?* is not significant as in the previous experiments.

With respect the previous simulation we can see that the value of the *Fitness* function is significantly lower, this is due to the change in the standard deviation.

In this work we assume the standard deviation to be equal to a percentage of the *Initial-StockPrice*, in this simulation we set the value of stock price equal to 100, so in the first and second experiment we deal with a standard deviation equal to 5, while in this case we have a standard deviation equal to 1.

In such a market the movement price are reduced, so at the end of each simulation the price will be not so distant from its initial value, this means that the market maker profits deriving from price fluctuaction will be lower and it will be count just on earnings generated from transaction, which are not enough in a market populated by 100 investors. Morevover we will see that the resulted bid-ask-spreads are too high to guarantee a significant *bid-ask-profit* (profit deriving from market maker transactions in a day with a bid and ask price).

6.2 Bid-ask spread: market-maker-profit vs bid-ask-profit

In the experiments on the NetLogo model *market-maker model 05* we focused on the maximization problem of the variable 'market-maker-profit', these experiment have been useful in order to understand better the model and to develop a further model, *market-maker-model 06*. We know that market maker objective is to achieve profits through a high number of transactions, regardless price fluctuations, so we have to analyze what parameters values allow the market maker to maximize the profit deriving from the transactions. To get this we defined the variable *market-maker-profit* in an equivalent way, underlying profits deriving from deals with a bid and ask price, we refer to this profit as *bid-ask-profit*. In the NetLogo model we define *market-maker-profits* (equivalent measure of *market-maker-profit*) as:

```
ifelse daily-net-position >= 0
[set market-maker-profits
daily-bid-ask-profit - Initial-investment
- (daily-net-position * bid-price)
+ (market-maker-stocks * exePrice) + daily-Initial-cash]
[set market-maker-profits
daily-bid-ask-profit - Initial-investment
- (daily-net-position * ask-price)
+ (market-maker-stocks * exePrice) + daily-Initial-cash]
```

So we have that if the *daily-net-position* is positive, meaning that the number of stocks that the market bough is higher than the number of stocks it sold, the market maker profit will be equal to

```
[daily-bid-ask-profit - Initial-investment
- (daily-net-position * bid-price)
+ (market-maker-stocks * exePrice) + daily-Initial-cash]
```

while if it has a negative *daily-net-position* it will be

```
[ daily-bid-ask-profit - Initial-investment
- (daily-net-position * ask-price)
+ (market-maker-stocks * exePrice) + daily-Initial-cash]
```

with a *daily-net-position* equal to 0 we will have that the profit is simply

```
[ daily-bid-ask-profit - Initial-investment
+ (market-maker-stocks * exePrice) + daily-Initial-cash]
```

Dividing profits in these components we can notice that the main sources of earnings for the market maker are the number of stocks it owns valued at the current stock price (an increase in the price will have a positive effect), and the daily-bid-ask-profit. The real goal of a market maker is to maximize the 'bid-ask profit', in order to achieve earnings without market risk given by the stock price. The 'bid-ask profit' is simply the sum of all *daily-bid-ask-profit*.

```
set bid-ask-profit bid-ask-profit + daily-bid-ask-profit
```

In the previous experiments the market-maker operating in a market with few investors and an highly variable price, it pursed profits holding a high number of stocks, in order to be exposed to price movements, and setting bid-ask spread too high to permit a significant number of transaction, in these new experiments we will focus on *bid-ask-profit*, maximizing this kind of profits, surely in a market with just 100 investors and significantly floating price the *bid-ask-profit* will not always be enough to cover the potential losses, however considering a market-maker operating in a system with many investors, trading more than one single stock, is reasonable to expect enough profits regardless stock price movements in time.

6.2.1 Experiment 1: nRandomInvestors

Dependent variable

We assume as dependent variable *bid-ask-profit*. In this experiment our goal is to maximize this variable and understand which variables determine positive or negative effect.

Independent variables

In the first experiment we define just one independent variable:

- *nRandomInvestors*: representing the number of trading agents in the model, its parametric range is [0 10 100]

We expect to find result supporting positive effect of this variable, high number of investors leads to a high number of potential transaction for the market maker, increasing its capability of make profits.

Fixed variables

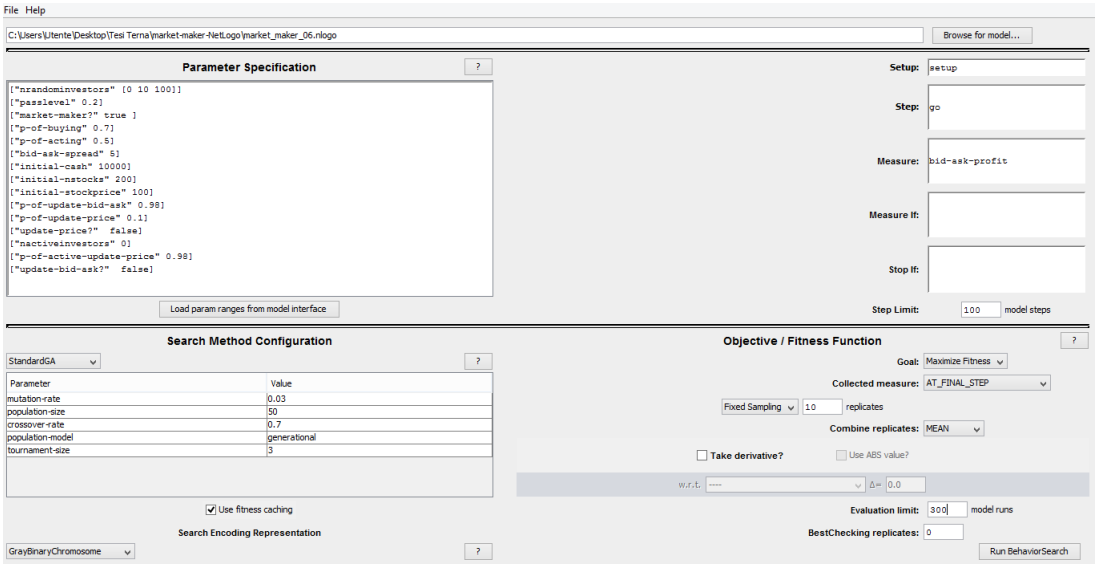
We define the value of fixed variables characterizing the market environment.

- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-acting*: 0.5
- *bid-ask-spread*: 5, equal to the value of the standard deviation of variable part of *price*

  ```
  (random-normal 0 Initial-StockPrice * 0.05)
  ```

  , *Initial-StockPrice* set equal to 100.
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *update-bid-ask?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000
- *Initial-StockPrice*: 100
- *Initial-nStocks*: 100

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 100, the value of *market-maker-profit* is considered after 100 ticks (trading days).
- *Evaluation Limit*: 300

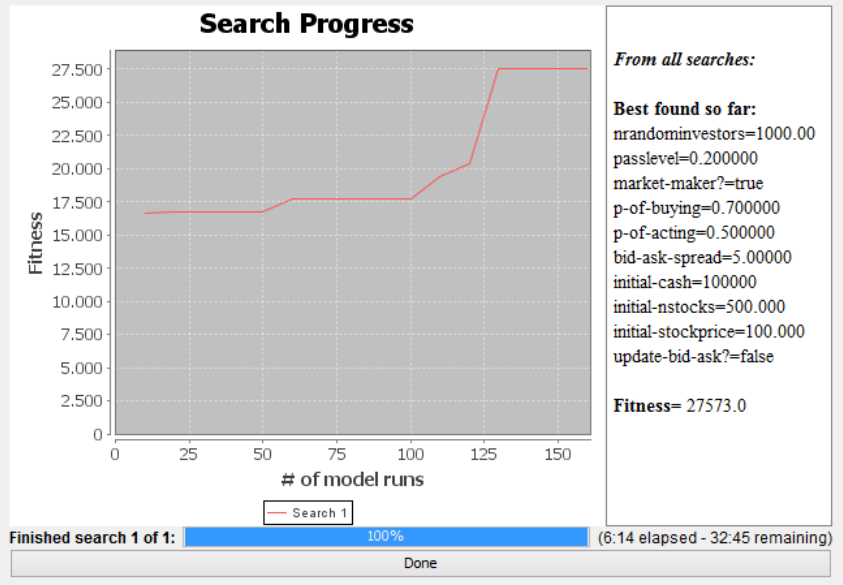
BehaviorSearch results

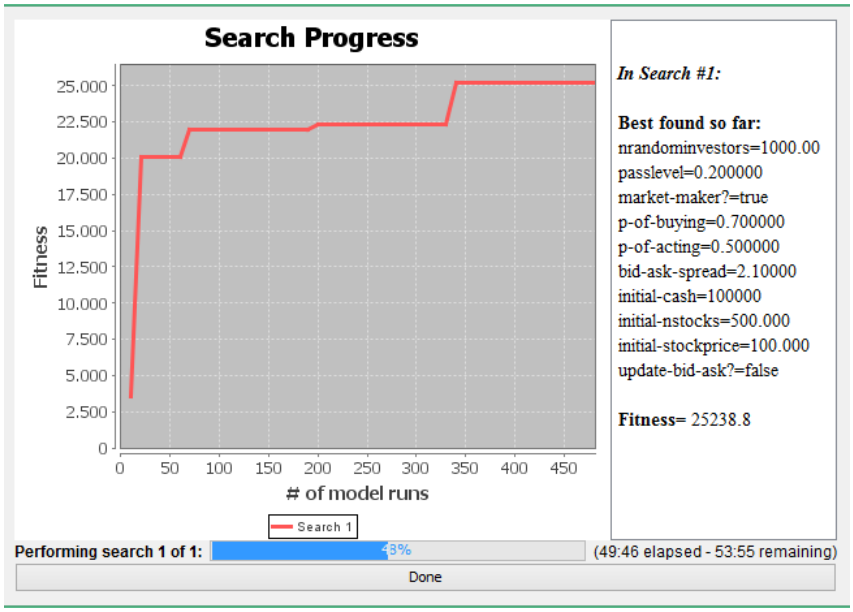
After 5 trials with different seeds, we obtain these results:

Table 6.4: experiment 1 results

Seed	nRandomInvestors	Fitness
1	100	867
2	100	883.5
3	100	845
4	100	906.5
5	100	901.5

As we expected the number of the dependent variable is always equal to its maximum value, 100. The average value of 'Fitness' is 880.7. Hence for the market maker more investor operate in the market more profits deriving from transactions can achieve. We compute also two times the same experiment allowing *nRandomInvestors* to reach a maximum value of 1000, the results give support to our assumption.




Search Progress
Fitness
25.000
22.500
20.000
17.500
15.000
12.500
10.000
7.500
5.000
2.500
0
0
50
100
150
200
250
300
350
400
450
# of model runs
Search 1
In Search #1:
Best found so far:
nrandominvestors=1000.00
passlevel=0.200000
market-maker?=true
p-of-buying=0.700000
p-of-acting=0.500000
bid-ask-spread=2.10000
initial-cash=100000
initial-nstocks=500.000
initial-stockprice=100.000
update-bid-ask?=false
Fitness= 25238.8
Performing search 1 of 1:
43%
(49:46 elapsed - 53:55 remaining)
Done

6.2.2 Experiment 2: nRandomInvestors p-of-acting

Dependent variable

We assume as dependent variable *bid-ask-profit*.

Independent variables

In this experiment we add another independent variable, so we have:

- *nRandomInvestors*: representing the number of trading agents in the model, its parametric range is [0 10 100]
- *p-of-acting*: representing the probability that the market maker will be able to trade with investors. It is a probability so it is bounded between zero and one, its parametric range is [0 0.1 1].

We expect to find result supporting positive effect of both variables, high number of investors leads to a high number of potential transaction for the market maker, increasing its capability of make profits and in such a market would be convenient for the market maker operate as much as possible.

Fixed variables

We define the value of fixed variables characterizing the market environment.

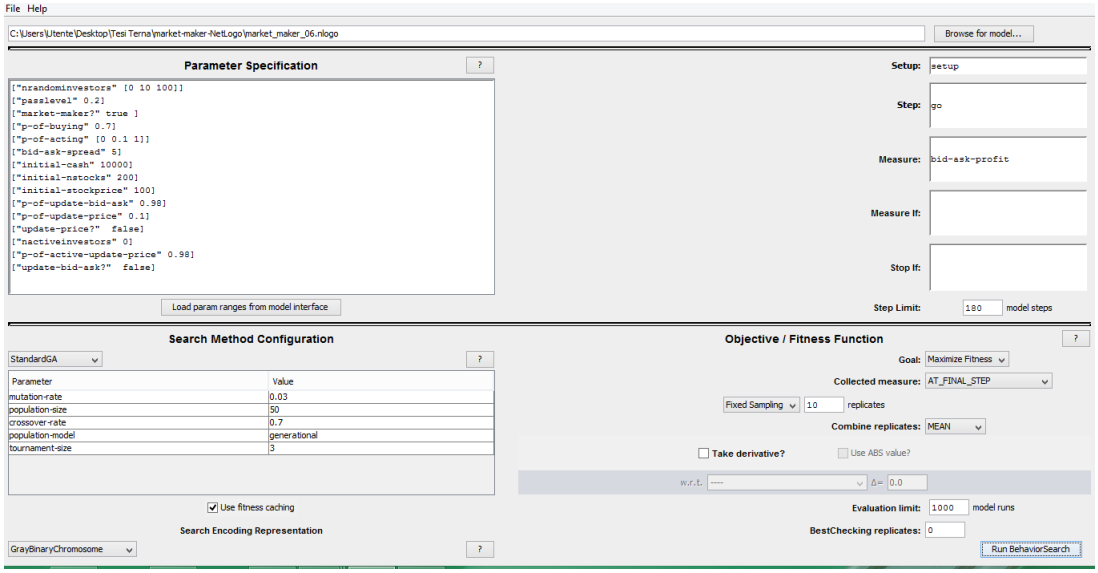
- *nActiveInvestors*: 0
- *market-maker?*: true
- *bid-ask-spread*: 5, equal to the value of the standard deviation of variable part of *price*

  ```
  (random-normal 0 Initial-StockPrice * 0.05)
  ```

  , *Initial-StockPrice* set equal to 100.
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false

- *update-bid-ask?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000
- *Initial-StockPrice*: 100
- *Initial-nStocks*: 100

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 100 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 5 trials with different seeds, we obtain these results:

Table 6.5: experiment2 results

Seed	nRandomInvestors	p-of-acting	Fitness
1	100	1	2975
2	100	0.9	2730
3	100	1	2911.5
4	100	1	2730
5	100	1	2908

as we expected even p-of-acting has a positive impact in *bid-ask-profit*, the only time that its value was lower than 1 (0.9) we report a lower 'Fitness' value. The number of investors as in the previous experiment is 100 in all cases. The average 'Fitness' is 2850.9.

6.2.3 Experiment 3: market-maker-profit,bid-ask-spread and 'update-bid-ask?'

NetLogo model loaded: *market-maker-model 05*

Dependent variable

Now we assume as dependent variable *market-maker-profit*.

Independent variables

The dependent variable will be:

- *bid-ask-spread*: The amount by which the ask price exceeds the bid. In the NetLogo model *market-maker-model 05* ,assuming that the value of the 'switch' *update-bid-ask?* is 'false', the market maker will set at the beginning of each trading day its bid and ask price according these commands:

  ```
  set ask-price exePrice + bid-ask-spread / 2
  set bid-price exePrice - bid-ask-spread / 2]
  ```

  If the possibility of adjust the bid and ask is included it will change these values even during the trading day. the parametric range will be [0 0.1 10], notice that the standard deviation of *price* is 5.

- *update-bid-ask?*: can assume as value 'true' or 'false'

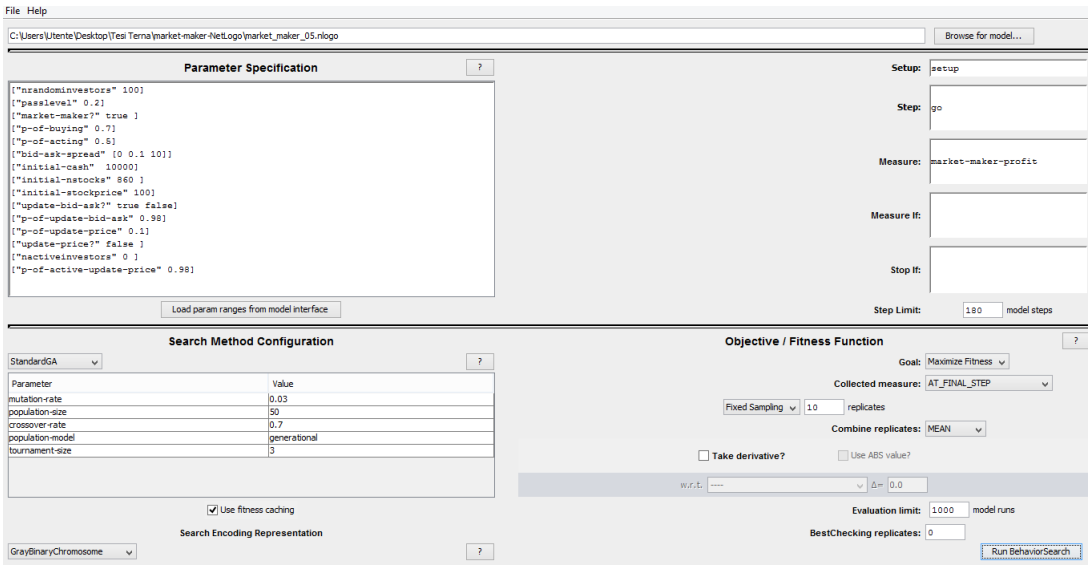
We will try to find the optimal bid-ask-spread in order to maximize market maker profits.

Fixed variables

We define the value of fixed variables characterizing the market environment.

- *nRandomInvestors*: 100
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000
- *Initial-StockPrice*: 100
- *Initial-nStocks*: 860

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 180 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.6: experiment 3 results

Seed	bid-ask-spread	update-bid.ask?	Fitness
1	9.7	true	13488.9
2	6.4	false	12681.6
3	9.1	true	18021.2
4	9.1	false	17266
5	9.4	false	26686.7
6	8	false	18254.3
7	9.7	false	15158.9
8	9	true	11976.7
9	8.2	false	13421.9
10	7.9	false	14368.9

we can notice that:

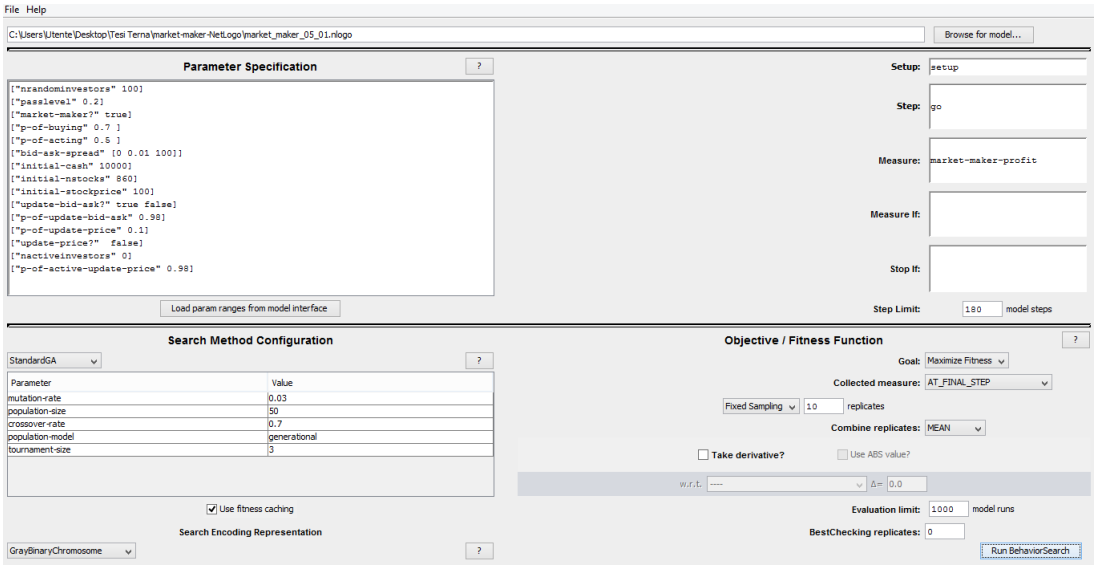
- the average bid-ask-spread is 8.65, the average 'Fitness' is 16132.41 and the effect of *update-bid-ask?* is not defined.
- The value of 'Fitness' is quite variable, it lies in the interval [11976.7 , 26686.7], so there are other variables affecting the final value of *market-maker-profit*. We can just observe that the value of the bid-ask-spread is close to the maximum value of its range, for the next experiments we will increase its parametric range.

6.2.4 Experiment 3.1: market-maker profit, bid-ask-spread and 'update-bid-ask?'

In this experiment with respect the previous one we have two differences:

- we increase the parametric range of 'bid-ask-spread', that will be equal to [0 0.01 100].
- the standard deviation will be equal to 1% of the 'Initial-StockPrice', so it will be equal to 1.

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 100 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.7: experiment 3.1 results

Seed	bid-ask-spread	update-bid.ask?	Fitness
1	5.23	false	2973.59
2	8.19	true	1630.13
3	6.71	true	2073
4	1.71	false	143.66
5	7.87	true	2589.29
6	4.73	false	3574.15
7	5.37	false	5264.01
8	2.72	true	3404.07
9	2.46	true	1441.31
10	6.36	true	3670.46

we can notice that:

- the value of the variable 'update-bid-ask?' is always uncertain, we can conclude that this is due to the fact that in general is not convenient for the market maker adjust price in a market where investors do not change trading condition during the day, moreover changing bid-ask price sometime can lead positive effects on the other hand can also lead to negative effects.

- the average value of the bid-ask spread is 5,13, this value is very high with respect to standard deviation (equal to 1), these kind of spread can not allow market maker a high number of transactions. The 'Fitness' value as before is variable and lies in a wide interval [143.66 , 5264]. The average value (2676,36) is much lower with respect to experiment 3, this is due to the lower value of the standard deviation, the price is more stable and market maker is not able to achieve consistent profits through increase of stock price.

In this experiment we understand that in a market with few investors, trying to maximize the 'market-maker-profit' as we defined it lead to incorrect results on the optimal choice of the bid-ask-spread. In the next experiments we will focus on the 'bid-ask-profit' in order to find the optimal values.

6.2.5 Experiment 4: bid-ask-profit vs market-maker-profit

In this experiment we compute the same procedure to find the optimal values of the bid-ask-profit in order to maximize 'bid-ask-profit' in one case and 'market-maker-profit' in the other.

Experiment 4.1 Maximizing market-maker.profit:

NetLogo model loaded: *market-maker-model 06*

Standard deviation: 5

Dependent variable

We assume as dependent variable *market-maker-profit*.

Independent variables

The Independent variable will be:

- *bid-ask-spread*: parametric range [0 0.1 10]

We will try to find the optimal bid-ask-spread in order to maximize market maker profits.

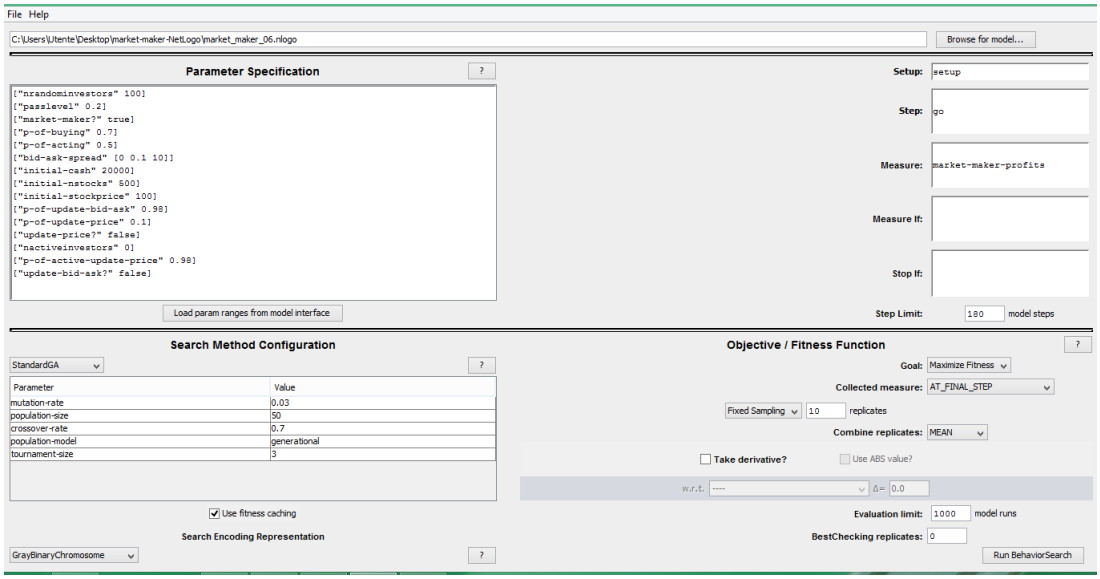
Fixed variables

We define the value of fixed variables characterizing the market environment.

- *nRandomInvestors*: 100
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *update-bid-ask?*: false

- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 20000
- *Initial-StockPrice*: 100
- *Initial-nStocks*: 500

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 100 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.8: experiment 4.1 results

Seed	bid-ask-spread	Fitness
1	6	12803.3
2	7.5	8532.6
3	6.6	12575.8
4	8.3	9182.32
5	8.2	11002.8
6	4.9	11637.5
7	7.9	13148.7
8	7.7	9039.95
9	9.3	8963.13
10	7.6	6853.73

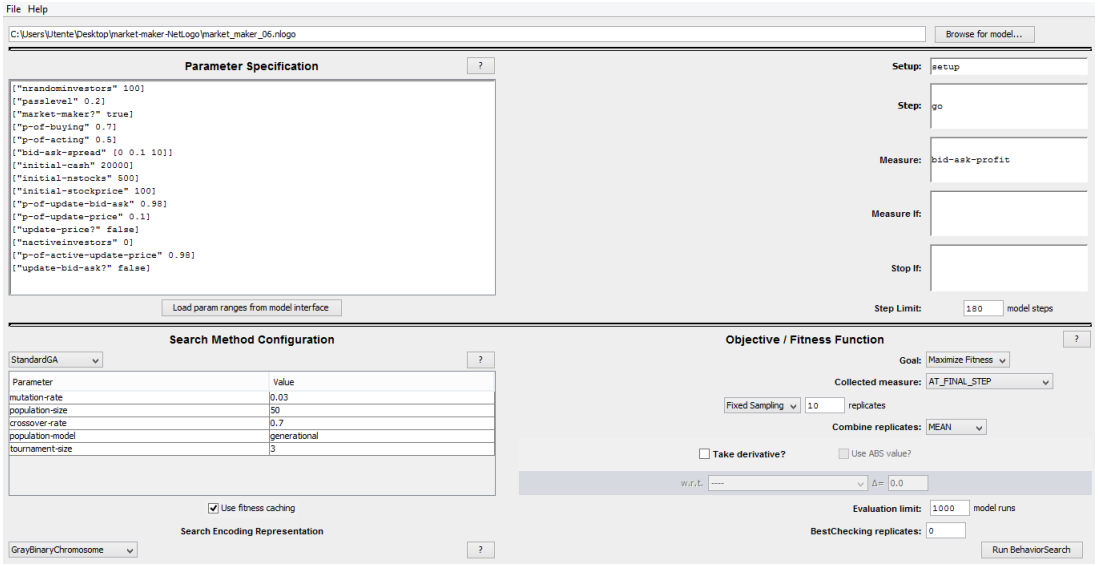
we can notice that that:

- the average 'Fitness' is 10374.07 , it lies in the interval [6853.73 , 13148.7].
- the average bid-ask-spread is 7.4, with interval [4.9 , 9.3]. Its value is quite variable and high with respect the standard deviation, we can not find a clear relation between the profit and the bid-ask-spread,

Experiment 4.2 Maximizing bid-ask-profit:

We compute the same experiment with the only difference in the definition of the objective function, in this case we will maximize the variable'bid-ask-profit'.

BehaviorSearch experiment editor



BehaviorSearch results

After 10 trials with different seeds, we obtain

Table 6.9: experiment 4.2 results

Seed	bid-ask-spread	Fitness
1	2.8	2063.32
2	2.6	2052.18
3	2.8	2035.2
4	2.8	2029.16
5	2.9	2059.29
6	2.9	2076.4
7	2.6	2051.92
8	3.3	2073.39
9	2.6	2088.84
10	3	2053.50

we can notice that in this case we have a clear correlation between the bid-ask-spread and the bid-ask-profit.

- the average 'Fitness' and bid-ask-spread respectively are 2058 and 2.83, and they lie in a small intervals.
- we can conclude that in such a market the optimal bid-ask-spread variate in a range of [2.6 , 3.3]. With respect the standard deviation is equal to about [55% , 65%] of its values, it would be interesting too see what results occur changing the value of the standard deviation, trying to find any kind of correlation between these two variables.

6.2.6 Experiment 5: maximizing bid-ask-profit, bid-ask-spread and standard deviation

In this experiment we compute the same procedure to find the optimal values of the bid-ask-profit in order to maximize 'bid-ask-profit' in one case and 'market-maker-profit' in the other.

Experiment 5.1

Standard deviation: 2.5

NetLogo model loaded: *market-maker-model 06*

Dependent variable

We assume as dependent variable *bid-ask-profit*.

Independent variables

The Independent variable will be:

- *bid-ask-spread*: parametric range [0 0.1 100]

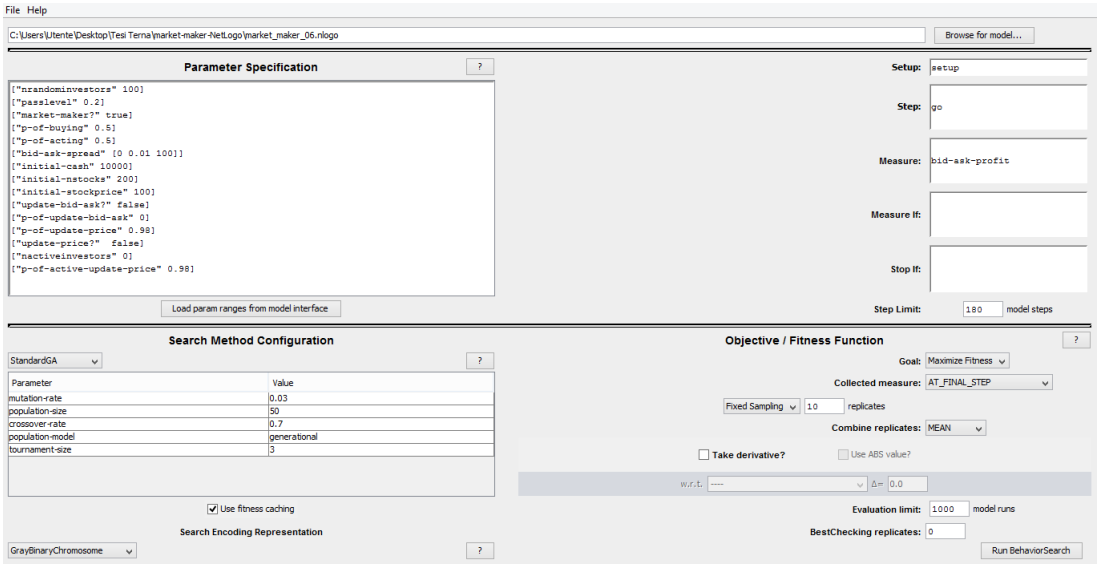
We will try to find the optimal bid-ask-spread in order to maximize 'bid-ask-profit'. With respect the previous experiment 4.2 we increased the range setting the maximum value of the spread equal 100 and we decreased the standard deviation from 5 to 2.5.

Fixed variables

We define the value of fixed variables characterizing the market environment.

- *nRandomInvestors*: 100
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *update-bid-ask?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000
- *Initial-StockPrice*: 100
- *Initial-nStocks*: 200

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 180 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.10: experiment 5.1 results

Seed	bid-ask-spread	Fitness
1	1.6	1009.44
2	1.6	1009.44
3	2	915.4
4	1.8	989.64
5	1.4	1029.56
6	1.4	1029.56
7	3.2	572.48
8	1.5	1032.15
9	2.4	825.12
10	1.8	1001.52

We can notice that:

- the average 'Fitness' and bid-ask-spread are 941.42 and 1.87, they lie in the respective intervals [572.48 , 1032.15] and [1.4 , 3.2].
- for values of bid-ask-spread between 1.4 and 1.8 we observe 'bid-ask-profits' higher than 1000, with similar values, the only time that the market maker set a too high bid-ask spread (3.2) the respective 'Fitness' value is significantly lower (572.48).

As we expected there is a correlation between the value of the standard deviation and the bid-ask-spread, in order to confirm these outputs in the next experiment we will increase the value from 5 to 10, our expectation is too observe values of the objective function and bid-ask spread about equal to times that reported in experiment 4.2.

Experiment 5.2

Standard deviation: 10

NetLogo model loaded: *market-maker-model 06*

Dependent variable

We assume as dependent variable *bid-ask-profit*.

Independent variables

The Independent variable will be:

- *bid-ask-spread*: parametric range [0 0.1 30]

We will try to find the optimal bid-ask-spread in order to maximize 'bid-ask-profit'. With respect the previous experiment 5.1 we decreased the range setting the maximum value of the spread equal 30 and we increased the standard deviation from 5 to 10.

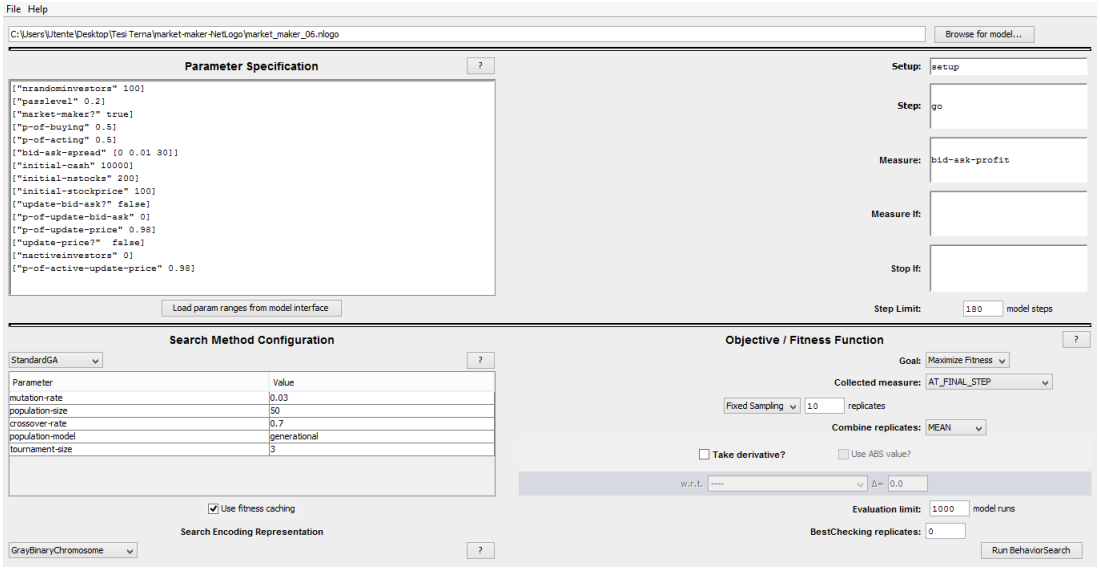
Fixed variables

We define the value of fixed variables characterizing the market environment.

- *nRandomInvestors*: 100
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *update-bid-ask?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000

- *Initial-StockPrice*: 100
- *Initial-nStocks*: 200

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 180, the value of *market-maker-profit* is considered after 180 ticks (trading days).
- *Evaluation Limit*: 1000

BehaviorSearch results

After 10 trials with different seeds, we obtain these results:

Table 6.11: experiment 5.2 results

Seed	bid-ask-spread	Fitness
1	5.5	4086.5
2	6	4113
3	5.9	4171.3
4	6.8	4097
5	5.8	4115.1
6	6.1	4088.83
7	6.1	4088.3
8	6.1	4071.14
9	6.7	3979.8
10	6.7	3979.8

we can notice that:

- the average value of 'Fitness' and bid-ask spread are 4079.2 and 5.56, they lie in small intervals [5.5 , 6.8] [3979.8 , 4115.1].

as we expected we observe values of the spread and 'bid-ask-profit' equal about two times the values obtained in experiment 4.2. We can conclude that there is a significant correlation between the optimal value of the bid-ask spread and the standard deviation of the 'variable' component of *price*.

```
set price exePrice + (random-normal 0 Initial-StockPrice * 0.05)
```

- *exePrice*: fixed component, equal the current value of the stock price.
- *(random-normal 0 Initial-StockPrice* $\dot{5}\%$*)*: variable component, reports a normally distributed random floating point number with mean zero and standard deviation equal to a percentage of the *Initial-StockPrice*.

The optimal value of the bid-ask spread in such a model, representing an artificial single stock market, is equal to a number bounded between the 55% and 65% of the standard deviation. According our analysis the value of the optimal bid-ask spread that maximize 'bid-ask-profit' should converge with the optimal one that maximize the 'market-maker-profit' in a market populated by a significant number of investors, in the last experiment of this work we will check if these values converge setting the number of investors equal to 1000 rather than 100.

Table 6.12: st.dv vs bid-ask-spread

standard deviation	optimal bid-ask-spread
2.5	[1.4 - 1.7]
5	[2.8 - 3.3]
10	[5.5 - 6.5]

6.2.7 Experiment 6: bid-ask-spread for market-maker-profit and bid-ask-profit

We want to check if the optimal value of the bid-ask spread for the 'bid-ask-profit' is the same for the 'market-maker-profit' considering a market populated by a consistent number of agents (1000).

Experiment

Standard deviation: 2.5

NetLogo model loaded: *market-maker-model 06*

Dependent variable

We assume as dependent variable *market-maker-profit*.

Independent variables

The Independent variable will be:

- *bid-ask-spread*: parametric range [1.5 0.5 10]

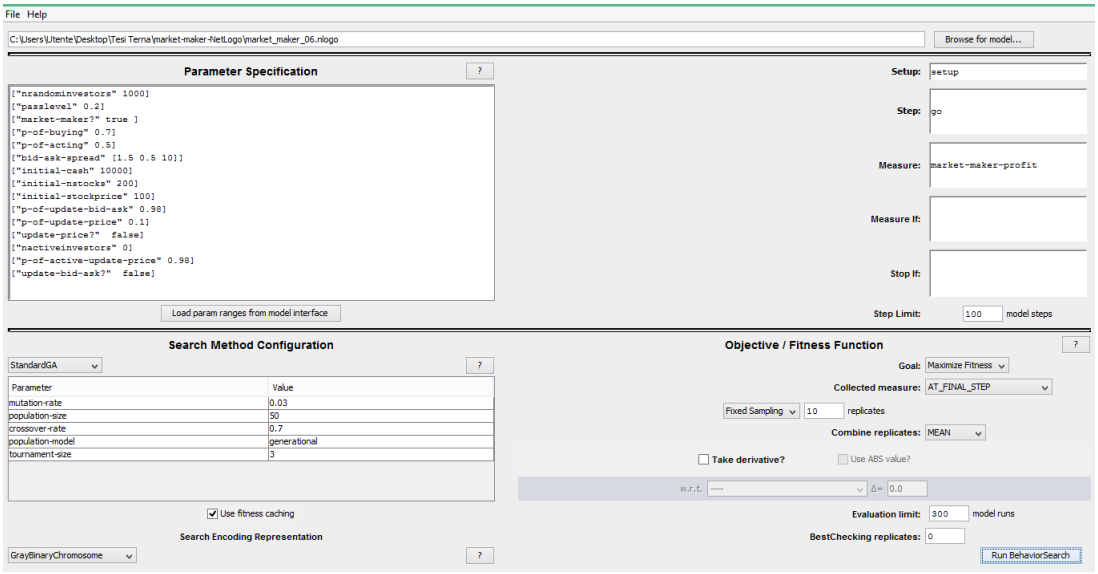
We will try to find the optimal bid-ask-spread in order to maximize 'market-maker-profit'. We will compare the results with the experiment 4.2.

Fixed variables

We define the value of fixed variables characterizing the market environment.

- *nRandomInvestors*: 1000
- *nActiveInvestors*: 0
- *market-maker?*: true
- *p-of-update-bid-ask*: 0.98
- *passLevel*: 0.2
- *p-of-buying*: 0.7
- *update-price?*: false
- *update-bid-ask?*: false
- *p-of-update-price*: 0.1
- *p-of-active-update-price*: 0.98
- *Initial-Cash*: 10000
- *Initial-StockPrice*: 100
- *Initial-nStocks*: 200

BehaviorSearch experiment editor



- *Search Method Configuration*: we insert standardGA custom input parameters.
  - *mutation-rate*: 0.03
  - *population-size*: 50
  - *crossover-rate*: 0.7
  - *population-model*: generational
  - *tournament-size*: 3
- *Step Limit*: 100, the value of *market-maker-profit* is considered after 100 ticks (trading days).
- *Evaluation Limit*: 300

BehaviorSearch results

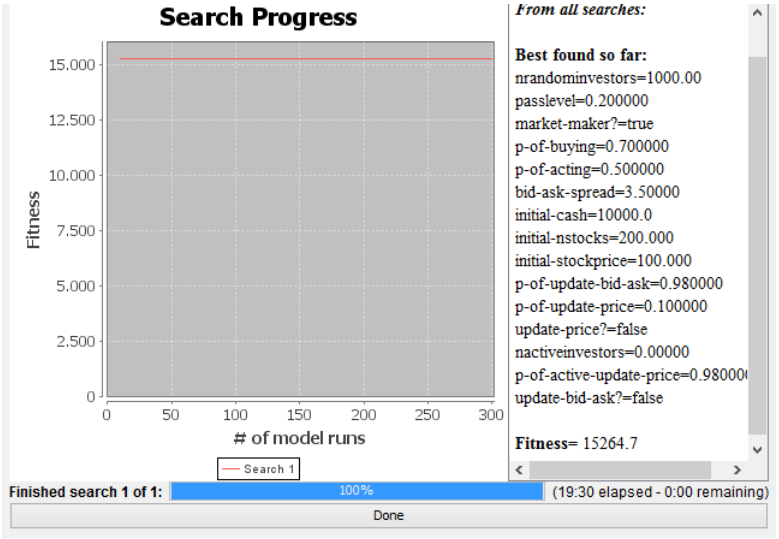
After 3 trials with different seeds, we obtain these results:

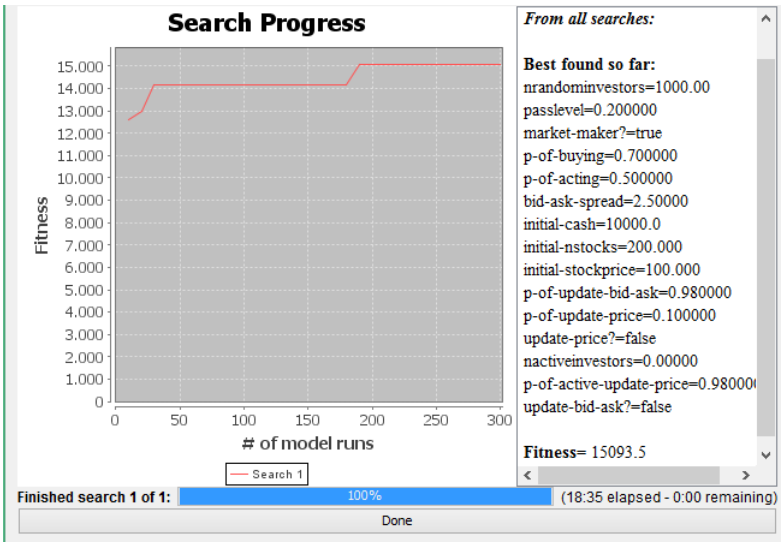
Table 6.13: experiment 6 results

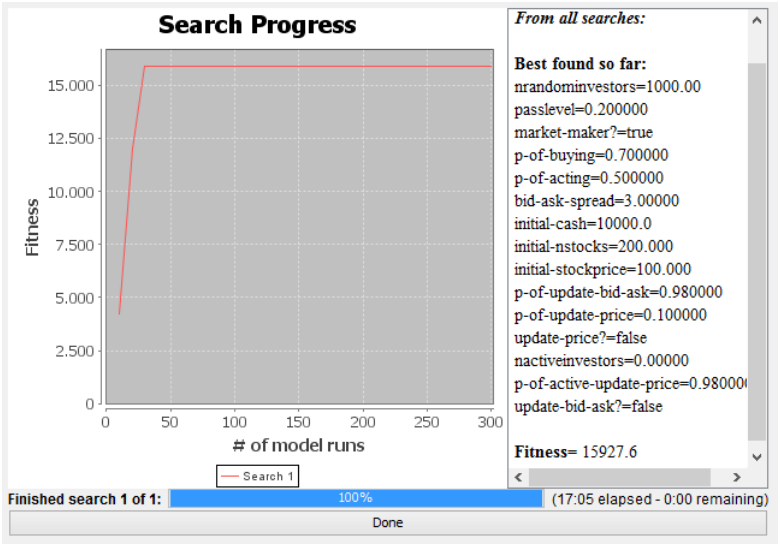
Seed	bid-ask-spread	Fitness
1	3.5	15264.7
2	2.5	15093.5
3	3	15927.6

These results confirm our expectation, in a stock market with many investors the market maker pursue profits thanks to a high number of transactions, so in such a environment maximize the 'bid-ask-profit' is equivalent to maximize the 'market-maker-profit'.

Experiment 6 results:






Search Progress
Fitness
15.000
12.500
10.000
7.500
5.000
2.500
0
0
50
100
150
200
250
300
# of model runs
Search 1
From all searches:
Best found so far:
nrandominvestors=1000.00
passlevel=0.200000
market-maker?=true
p-of-buying=0.700000
p-of-acting=0.500000
bid-ask-spread=3.00000
initial-cash=10000.0
initial-nstocks=200.000
initial-stockprice=100.000
p-of-update-bid-ask=0.980000
p-of-update-price=0.100000
update-price?=false
nactiveinvestors=0.00000
p-of-active-update-price=0.98000
update-bid-ask?=false
Fitness= 15927.6
Finished search 1 of 1:
100%
(17:05 elapsed - 0:00 remaining)
Done

Conclusion

To sum up this work can be divided into three parts:

1. In the first part (chapter 1, 2 and 3) we introduce the main concept on which we developed this thesis, necessary to fully understand the model and the experiments implemented in the following parts.

2. In the second part (chapter 4 and 5) we develop the market maker model through NetLogo and then we interface this model with R in order to analyze the stock price time series generated in NetLogo through R-commands, computing autocorrelation functions and partial autocorrelations.

   NetLogo Model

   Starting from a simple CDA model (continuous double auction), we construct a single stock market model with the presence of a market maker. We analyze different models, making them more real and complex. In the last model the market maker goal is to maximize profits, setting bid and ask price at which is willing to trade.

   The NetLogo models implemented are:

   - *CDA basic model:* simple continuous double auction model with agents trading among themselves, we have no market maker in this initial stock market.
   - *Market maker model 0:* we introduce the presence of the market maker, offering to buy and to sell whenever a trader is not able to find a counterpart. In this model it trades at investors condition, without aiming to make profits.
   - *Market maker model 01*: we include the setting of bid and ask price by the market maker, in this market its goal is to achieve profits through bid-ask spread.
   - *Market maker model 02*: In this model the market maker actions are constrained, we provide the market maker with a given initial number of stocks and availability of cash.

- *Market maker model 03*: In the previous models the market maker could set bid and ask once every trading day, now we consider the possibility of update these prices during the day according price movements.
- *Market maker model 04*: We also allow investors to change trade condition, with some probability, during the trading day.
- *Market maker model 05*: We divide the agents in 'random-investors' and 'active-investors', so we have two different kind of traders, the former will adjust price condition with a low probability , the latter will follow carefully the market promptly updating ,with high probability, the price at which is willing to buy or sell.
- *Market maker model 06*: This final model is necessary for the experiments via BehaviorSearch, in this model we split the variable 'market-maker-profit' in more components, underlying profits deriving from selling at the ask price and buying at the bid.

NetLogo and R

We explain how we connect these programs through R-serve extension and develop two model starting from the previous one, in order to store the stock price time series and computing analysis through R-command. We complete this procedure on the model 03 and 05 and we obtain the same results, the stock price follows a moving average of the first order.

3. In the third and last part we exploit the use of genetic algorithms in order to analyze the optimal choice of the bid-ask spread, initial availability of cash and stocks for the market maker in order to achieve maximum profit. In order to find the optimal values of these variables we compute experiments through BehaviorSearch.

   In the first experiments (exp.1, exp.2 and exp.3) we focus on the maximization of 'market-maker-profit'. leading to uncertain results, these outputs moved us to wonder if was the right choice to optimize this variable as we defined it, and what components affected it.

   In this way in the last experiments (exp.4 and exp.5) we focused on the 'bid-ask-profit', profit deriving from a high number of transactions regardless stock price component. These experiment allow us to find interesting results about the optimal level of the bid-ask-price in particular, we was able to find a direct correlation between spread and the nature of the price condition of agents in the model. In this artificial market investors are willing to buy or sell at a particular 'price', this 'price' is generated by a fixed and a variable term, the variable component is a normally distributed random floating point number with mean zero and standard

deviation equal to a percentage of the *Initial-StockPrice*. We found the optimal spread in relation of the standard deviation, equal to a range between 55% and 65% of the standard deviation.

In the last experiment (exp.6) we checked if the uncertain results obtained in the first experiments was partially due to the market contest (just 100 investors, low number to guarantee significant 'bid-ask-profit'), so we tried to maximize 'market maker profit' in a model populated by 1000 investors comparing these results with the optimal spread for the bid-ask-profit. As we expected it appears to be that the optimal spread maximizing 'market-maker-profit' converge to the one maximizing 'bid-ask-profit'.

Possible future developments

Potential further analysis could be made to deeply investigate how market makers interact in financial markets.

- We could in the model the presence of informed and uninformed traders, to make more real and sophisticated the decision process of buy or sell.
- It would be interesting search what values of spread minimize its 'net position', avoiding to be exposed to volatile markets
- develop a more complex model including possible shocks on the stock price.
- understand through genetic algorithms how and in which size 'active-investors' affect market maker behavior.

This work aim to investigate market makers behavior in financial markets, final results lead us to the optimal choice of the bid-ask spread, given many simplifying assumptions, however further analysis and developments could lead to significant findings in a real contest in which market makers operate.

Bibliography

Axelrod, R. and Tesfatsion, L. (2005). *A guide for newcomers to agent-based modeling in the social sciences*. In K.L. Judd and L. Tesfatsion, eds., *Handbook of Computational Economics*, vol. 2. North-Holland, pp. 1647-1658.
URL `http://www.econ.iastate.edu/tesfatsi/GuidetoABM.pdf`

Axtell, R. (2000). *Why Agents? On the Varied Motivations for Agent Computing in the Social Sciences*. In *Proceedings of the Workshop on Agent Simulation: Applications, Models and Tools*. Argonne National Laboratory, IL.
URL `http://www.brookings.edu/\textasciitilde/media/Files/rc/reports/2000/11technology\_axtell/agents.pdf`

Axtell, R. L. and Epstein, J. M. (2006). *Coordination in Transient Social Networks: an Agent-Based Computational Model of the Timing of Retirement*. In J.M. Epstein, ed., *Generative social science: Studies in agent-based computational modeling*. Princeton University Press, p. 146.

Boero, M., R. Morini and Terna, P. (2015). *Agent-based Models of the Economy from Theories to Applications*. Palgrave Macmillan.
URL `http://www.palgrave.com/page/detail/agentbased-models-of-the-economy-/?K=9781137339805`

Gilbert, N. and Terna, P. (2000). *How to build and use agent-based models in social science*. In «Mind & Society», vol. 1(1), pp. 57-72.

Stonedahl, F. J. and Adviser-Wilensky, U. J. (2011). *Genetic algorithms for the exploration of parameter spaces in agent-based models*. In .
URL `http://forrest.stonedahl.com/thesis/forrest_stonedahl_thesis.pdf`