UNIVERSITÀ DEGLI STUDI DI TORINO
School of Management and Economics

SIMULATION MODELS FOR ECONOMICS

Final report

Delta hedging strategy

by Andrea Chiamenti, Simone Gallo, Filmon Teclai

Professor: Pietro Terna

Summary

1. Introduction

Our work aims at implementing, using the NetLogo software, a hedging strategy for a portfolio of stocks and options based on its Delta value. In order to do so, we exploit agent-based simulation to recreate two parallel markets. In the first one, stocks are exchanged following the bid and ask principle, thus to obtain asset prices reflecting agents' preferences. In the second one, the objects of trading are specific derivative instruments: options on the stocks traded in the former market. Inserted in this environment, one or more arbitrageurs will apply a Delta hedging strategy based on the Black-Scholes-Merton theory.

Before proceeding with the illustration of the code, let's take a brief but deeper look at the financial theory at the basis of this work.

Derivative instruments, like options, when properly used can be powerful risk managing tools. What we are specifically interested in right now are options, which are contracts giving their buyers the right, but not the obligation, to buy (call option) or sell (put option) an underlying asset (in our case, stocks) at a predetermined strike price and at a given point in time.

When talking about derivatives, we often refer to a series of values know as Greeks, which are able to capture the sensitivity of the derivative's price to a change in specific underlying parameters (such as the underlying asset price or volatility).

For the purpose of our work, the value we are interested in is the so called Delta, which is nothing but a measure of how the option price changes when the underlying price moves. We can give it a simple and intuitive mathematical representation as:

$$\Delta = \frac{\partial c}{\partial S}$$

where: c = option value and S = stock price.

This formula, however, can be expanded when we consider the option pricing theory developed by Fischer Black, Myron Scholes and Robert Merton. Their model gives us this formula for a European call option price:

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

where:

$$d_1 = \frac{ln\frac{S}{K} + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

with:

- S = spot price of the underlying asset;
- K = strike price;
- t = time in which the price is computed;
- T = option's maturity;
- r = continuously compounded annual risk-free rate;
- σ = underlying asset's return volatility.

From here, the first order derivative of the call option's price with respect to the underlying price (i.e. the Delta) is easily computed as:

$$\Delta = \frac{\partial c}{\partial S} = N(d_1)$$

Significant is the fact that the Delta is also commonly known as Hedge ratio, since a portfolio containing one option and an amount of stocks exactly equal to -δ is insensitive to movements in the stock price (and so is said to be delta-hedged). In formula, computed in time t:

$$P(t) = -c(t) + \Delta(t)S(t) + B(t)$$

where P = portfolio and B = money market account.

This is exactly the main idea at the basis of the delta hedging technique, which consists in continuously computing the delta of our portfolio in order to rebalance its composition, thus to achieve the above-mentioned condition. Of course, applying this rebalancing with a frequency which is *actually* continuous arises a lot of technical difficulties, so we can try to approximate the whole process through a discrete hedging.

In the next section, we are going to analyze in detail the NetLogo program.

2. The model

2.1 Interface

Before proceeding with the description of the code, we take a brief look at the user's interface and its functions.

The `setup` button runs all the preliminary operations and creates a graphical representation of the agents inside the "world" box.
The `go` button starts the simulation, that will continuously run until the go button is pressed again.
We have then a series of sliders, each of which allows the user to set the value of a specific variable:

- `nRandomAgents`: controls the number of agents inside the stock market;
- `nOptionAgents`: controls the number of agents inside the option market;
- `passLevel`: is a threshold that each agent has to exceed in order to not pass the deal;
- `out-of-marketLevel`: another threshold, controlling the portion of agents not entering the markets;
- `strike`: the call option's strike price;
- `levelOfEntry`: controls the number of `optionAgents` entering the market;
- `risk-free`: the values of the continuously compounded annual risk-free rate;
- `nArbAgents`: controls the number of arbitrageurs in the model;
- `sigma`: allows to set the value for the underlying price volatility.

Last, the `optionOrNot` switch activates both the `optionAgents` and the `arbAgents`.

The interface is completed by three graphs, showing to the user the real time variations of the most relevant variables involved in the simulation: `exePrice` (the execution price of the stock), `callExePrice` (the execution price of the call option) and `arbGain` (the gain that the arbitrageur obtains applying the Delta hedging strategy).

2.2 Code

Before starting with the presentation of the code, we want to highlight that the basis of the program, simulating the stock market, is taken from a previous work, seen during our lessons: *h_CDA_basic_model*. Nonetheless, for the sake of completeness, and to let the reader have a fully understanding of the procedures involved, we will discuss it as well.

The very first thing we need to do is to define all the breeds and variables necessary throughout the whole code. Therefore, we associate the three agentsets of interest with the respective breeds, we assign to them exclusive variables via the `<breed>-own` function and, lastly, we identify the sets of variables which are common to all the agentsets thanks to the `globals` command.

To setup

This piece of code is mainly devoted to the graphical representation of the turtles in the "world" box, which is of minor interest for the purposes of this paper. However, this is not the only function of the setup phase: we manage the in/out-of-market condition of both `randomAgents` and `optionAgents` with:

```
set out-of-market False
```

and a similar action is done for the `arbAgents`:

```
set finish false
```

Later in the construction of the program we will create a procedure to define an execution price for the stocks, for the moment we just assign to it a starting fixed value of 1000:

```
set exePrice 1000
```

Finally, as usual in NetLogo programs, we exploit the setup phase to reset any previous action through the commands:

```
clear-all
reset-ticks
```

Moreover, we use the `to-report` command to define the procedure `erfcc`, useful to compute the standard normal cumulative distribution function, N(·):

```
to-report erfcc [x]
   let z abs x
   let q 1.0 / (1.0 + 0.5 * z)
      let r q *  exp (- z * z - 1.26551223 + q *
      (1.00002368 + q * (0.37409196 + q * (0.09678418 +
```

```
            q * (- 0.18628806 + q * (0.27886807 + q * (-
            1.13520398 + q * (1.48851587 + q * (- 0.82215223 +
            q * 0.17087277))))))))
        ifelse (x >= 0) [report r] [report 2.0 - r]
    end
```

To go

In this portion of the command we define the *modus operandi* of all the agentsets involved in the simulation. With this regard, the code is not linear, with several nested commands. With the aim of clarifying the actions executed by each agentset, we will analyze their cases separately.

randomAgents

The preliminary phase is devoted to (randomly) decide whether each agent will buy or sell stocks. First of all, we exploit the `ifelse` function in order to isolate only those agents that are not out-of-the-market. For this sub-group, a second `ifelse` condition is imposed: only if a pseudo-random value exceeds an arbitrary pass level (that can be modified by the user by means of a slider) the agent will actually exchange stocks; otherwise, it will skip the deal:

```
ask randomAgents
  [ifelse out-of-market
        [set color white]
        [ifelse random-float 1 < passLevel
              [set pass True][set pass False]
     ifelse not pass
       [ifelse random-float 1 < 0.5 [set buy  True set sell False]
                                    [set sell True set buy  False]]
       [set buy False set sell False]

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

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

Together with the action to execute, a color will be assigned to each `randomAgent`: white if he is out-of-the-market, grey if he passes, red if he buys, green if he sells. Lastly, the starting execution price is increased by a pseudo-random value ranged between 0 and 100, and associated to the variable named `price`.

The second set of requirements deals with the creation of the book of prices referred to the stock market. First of all, we exclude any `randomAgent` that is not executing a deal:

```
        if not pass and not out-of-market
```

Now, we create the temporary vector `tmp`, in which we put the prices created previously, together with the variable `who`, identifying which agent has set such price. Now, each turtle has two possibilities, created with the `if` command:

- if it is a buyer, the price to which it is associated will be inserted into the vector of bid prices, `logB`, and such vector will be reordered to make the list decreasing:

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

- if it is a seller, the same action will be done, creating the ask prices vector, logS, but organized in an increasing order.

What is left to do it to create the market price, matching the bid and the ask. Any time the two corresponding vectors are not empty, their first items (meaning the highest bid and the lowest ask) are compared:

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

Whenever the "greater than or equal to" condition is satisfied, the deal is executed at the ask price:

```
set exePrice item 0 (item 0 logS)
```

Which is assigned to both agents by putting it into a specific vector, thus to allow the agents avoiding redundant operations. At the same time, the prices are removed from the market book:

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

The last task referred to this agentset is to manage the entrance and the exit of the agents from the market: the program generates a uniform pseudo-random number between 0 and 1 and confront it to the out-of-marketLevel value. Whenever this threshold is not passed, the program checks whether one out of two conditions is verified: if the execution price exceeds 1500, the agent exits the market; if the execution price is lower than 500, the agent comes back into the market.

```
if random-float 1 < out-of-marketLevel
   [if exePrice > 1500 [set out-of-market False]
    if exePrice <  500 [set out-of-market True] ]
```

This condition is necessary to ensure the non-negativity of the stock price, since as the price falls below the desired level, the transactions will stop.

optionAgents

The first requirements for this agentset are the same seen for the `randomAgents` one: we randomly decide if each agent is out-of-the-market, if passes, if buys or if sells options, and we assign to it the corresponding color (white, gray, yellow or blue).

The following instructions are executed at the user's discretion: the interface, indeed, shows a switch allowing to put in action the `optionAgents` as well as putting them "to sleep", and a corresponding `if` condition is present in the code. After this, we make use of a new command, the `n-of` one. The purpose here is to make act only a portion of the whole agentset, and this would be impossible with the `ask` function only. Thanks to `n-of` and to a dedicated slider, we can achieve our goal:

```
ask n-of levelOfEntry optionAgents
```

Now, as for the stock market, we want to exclude all the agents that are not going to execute a deal:

```
if not pass and not out-of-market
```

and we are ready to set the mechanism that will create the option prices. Again, the procedure is similar to the one saw for the stock market. Indeed, we make use of another temporary vector, `x`, this time filled with the call option prices. Those are set as follows:

```
set exePrice exePrice + (random-normal 0 50)
set callPrice exePrice - strike
set callPrice callPrice + premium
```

Now, provided that the price is non-negative, we match the bid and the ask. Again, we create the corresponding price vectors, `logBC` and `logSC`. The former is ordered decreasingly, the latter is ordered increasingly.

```
if buy [set logBC lput x logBC]
set logBC reverse sort-by [item 0 ?1 < item 0 ?2] logBC

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

We have now all the elements needed to match the options supply and demand: when we have non-empty vectors, the highest bid is matched with the lowest ask, and the latter value is set equal to the call execution price:

```
if (not empty? logBC and not empty? logSC) and
item 0 (item 0 logBC) >= item 0 (item 0 logSC)

set callExePrice item 0 (item 0 logSC)
```

Once again, to avoid redundant operations, we store the execution prices into dedicated vectors, `agBC` and `agSC`, eliminating them from the book.

arbAgents

The very first requirement in order to start this agentset action is to have a non-zero `sigma`. The reason is trivial: we will face formulas in which the only denominator of a ratio is `sigma`.

```
if sigma > 0 and not finish
```

Remember that the arbitrageur bases its activity on the Black-Scholes-Merton theory, so we need to set up the list of all the variables and formulas involved in the related computations:

```
set d1 (ln (exePrice / strike) + (risk-free +
           ((sigma ^ 2) / 2)))  / (sigma)
set d2 d1 - sigma * sqrt 1
let a d1 / (2 ^ 0.5)
let b d2 / (2 ^ 0.5)
let c -d1 / (2 ^ 0.5)
let d -d2 / (2 ^ 0.5)
set Nd1 1 - 0.5 * erfcc a
set Nd2 1 - 0.5 * erfcc b
set N-d1 1 - Nd1
set N-d2 1 - Nd2
set DeltaC Nd1
set DeltaVec1 fput DeltaC DeltaVec1
set BScall exePrice * Nd1 - strike *
           exp(- risk-free) * Nd2
```

Notice that, since the arbitrageur will operate a Delta hedging strategy, we create the vector `DeltaVec1`, containing the options' Deltas, `DeltaC`.

It's time now to define the Delta hedging strategy. This can be broke into two different cases, by means of an `ifelse` command:

- the call market price exceeds the theoretical Black-Scholes price: the arbitrageur will go long on the asset and short on the option:

  ```
  set assetP DeltaC * exePrice
  set optionP (- callExePrice)
  ```

  Moreover, we calculate the money market account:

  ```
  set mma (- assetP / exp( risk-free))
  ```

  The gain for the agent is computed as:

  ```
  set gainP assetP - mma - optionP
  ```

- the theoretical Black-Scholes price exceeds the call market price: the arbitrageur will go short on the asset and long on the option:

```
set assetP (- DeltaC * exePrice)
set optionP (- callExePrice)
```

Moreover, we calculate the money market account:

```
set mma (- assetP / exp( risk-free ))
```

The gain for the agent is computed as:

```
set gainP (mma - assetP - optionP)
```

Graphs

After each cycle is completed by the agents, and after all the variables of interest are generated, we want to obtain a graphical representation of the results:

```
to graph

set-current-plot "exePrice"
plot exePrice

set-current-plot "callExePrice"
plot callExePrice

set-current-plot "arbGain"
plot gainP

end
```

In this way, the execution prices for the stocks and the options will be displayed, together with the arbitrageur's gain.

3. Simulations

We will now illustrate the results of several simulations whit different variables values. Our ultimate goal is to study the performances of the arbitrageur and, in particular, the gains (if any) he obtains applying the Delta hedging strategy. To do so, of course, we will also need to analyze the behavior of the prices in the stock and option markets.

The section is organized to compare two states of the world differing only by one variable, which is the object of study. The titles of the following paragraph intuitively indicate the object of the simulation contained in it. For each simulation we will present three graphs: `exePrice` for the execution price of the stock, `callExePrice` for the execution price of the call option, `arbGain` for the gain of the arbitrageur.

Notice also that, in order to run those simulations, we need to make two simplifying assumptions:

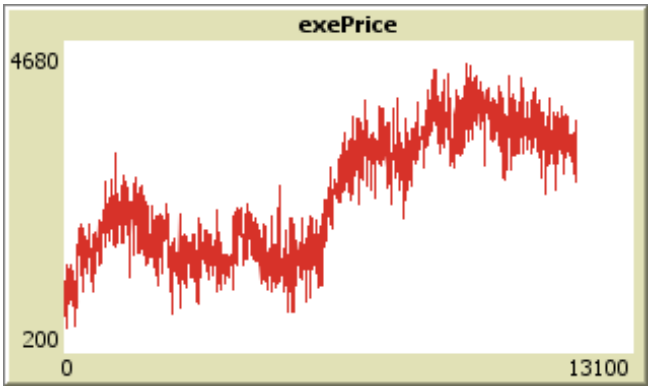
- the time to maturity for any call option is 1;
- the only variables that vary over time are the prices of the stock and of the call option, while factors such as the volatility of the stock price, the strike price and the risk-free interest rate are assumed to be constant.

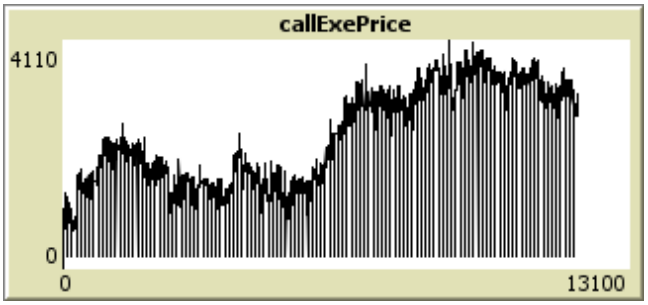
Strike price

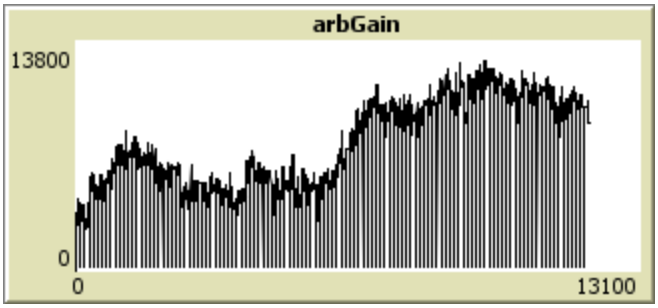
We set two extreme values for the call option strike price: one close to the lowest possible, the other one close to the highest possible.

Parameters of the first simulation:

Number of agents trading stocks: 100
Number of agents trading call options: 100
Call option strike price: 100
Risk-free interest rate: 1%
Volatility: 0.2
Pass level: 0.3

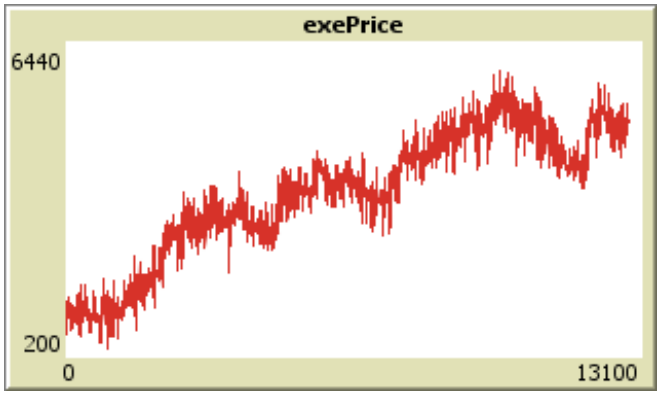


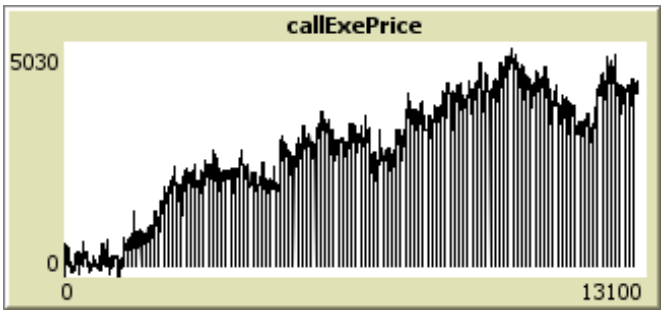


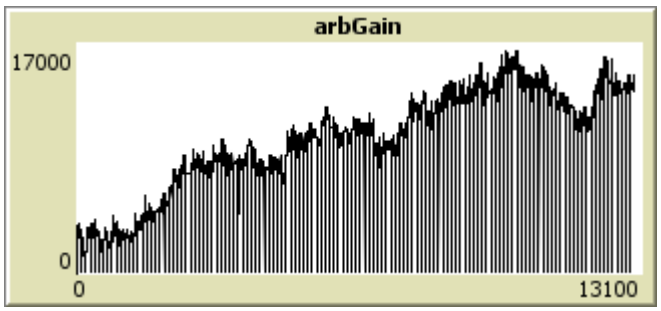


Parameters of the second simulation:

Number of agents trading stocks: 100
Number of agents trading call options: 100
Call option strike price: 750
Risk-free interest rate: 1%
Volatility: 0.2
Pass level: 0.3







First of all, since `exePrice` enters the formula that the `optionAgents` apply to create the call option price, it will follow that the change in value affects the `callExePrice` results. Remember the dedicated line of code:

```
set callPrice exePrice - strike
```

In particular, for equal values of exePrice we observe an lowering in the price of the corresponding option.

This leads to a second effect: an increment of the arbitrageur's gain. Indeed, all others equal, a lower option price means a higher arbitrage profit:

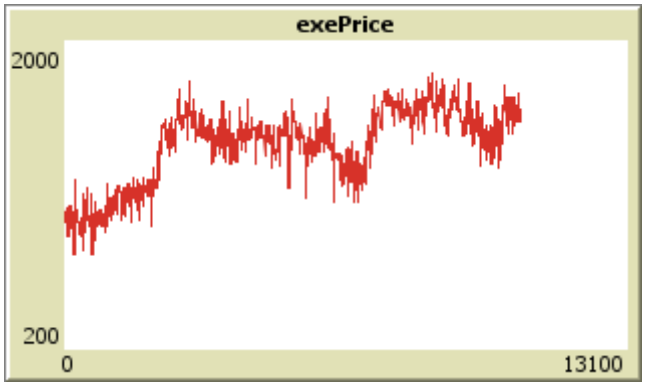
```
set gainP assetP - mma – optionP
set gainP (mma - assetP - optionP)
```

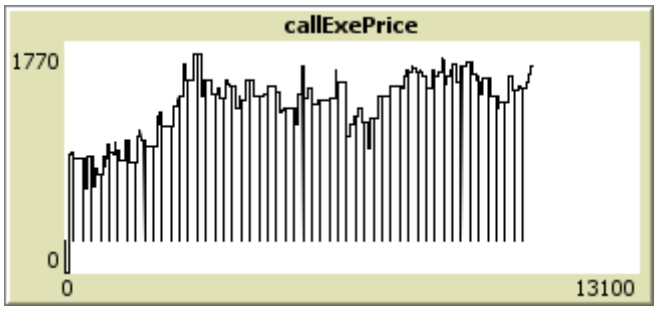
Risk-free interest rate

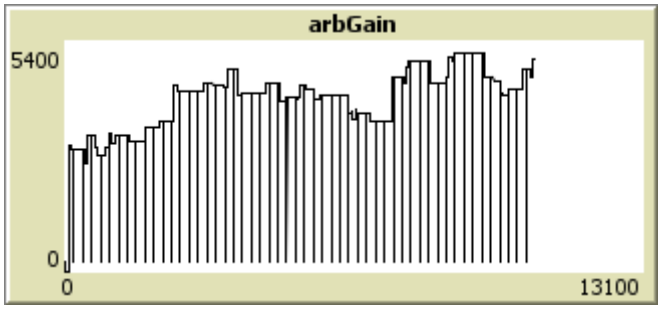
Again, the logic of the experiment is to set two extremely different values, thus to make any possible effect emerge clearly.

Parameters of the first simulation:

Number of agents trading stocks: 100
Number of agents trading call options: 100
Call option strike price: 100
Risk-free interest rate: 1%
Volatility: 0.2
Pass level: 0.3

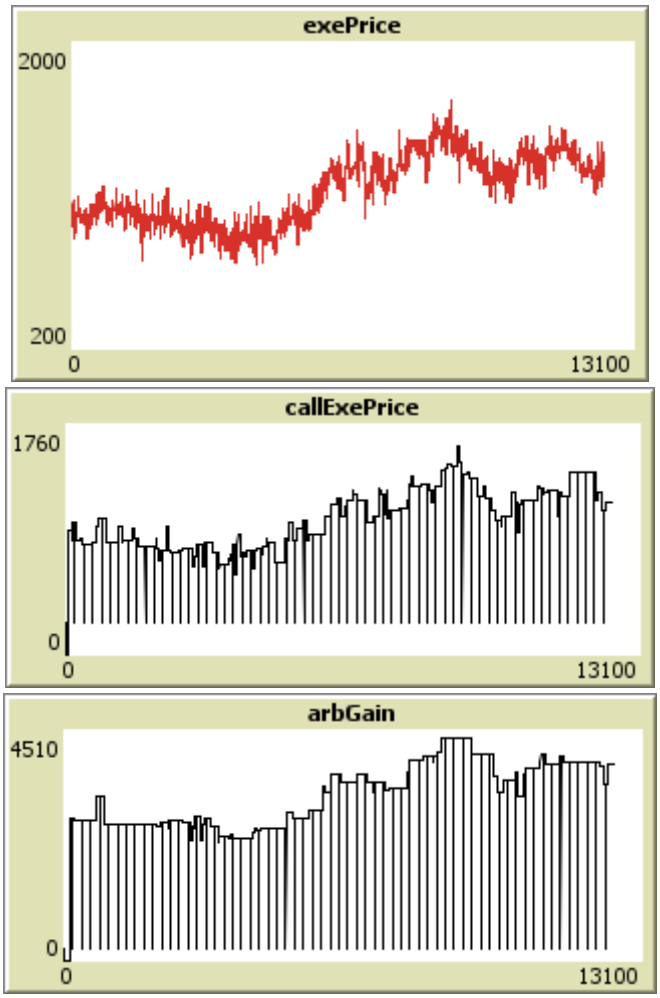






Parameters of the second simulation:

Number of agents trading stocks: 100
Number of agents trading call options: 100
Call option strike price: 750
Risk-free interest rate: 10%
Volatility: 0.2
Pass level: 0.3



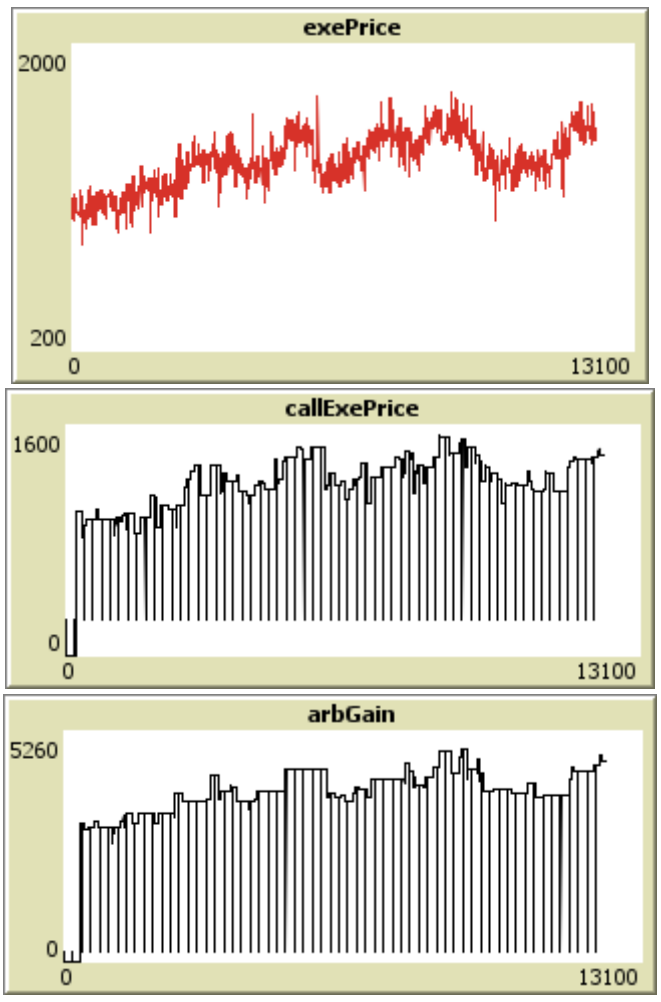
In our simplified model, the risk-free rate value enters only the Black-Scholes formula for the option pricing, meaning that the only variable of interest here is `arbGain`. According to the Black-Scholes-Merton model, a higher interest rate leads to a higher call price. This, according to the `gainP` formula shown above, translates into lower gains for the arbitrageur (result which is confirmed by the simulations, as emerges from the graphs).

Volatility

Always applying the “extreme values” technique, we choose here volatility values standing at the opposites of the total range.

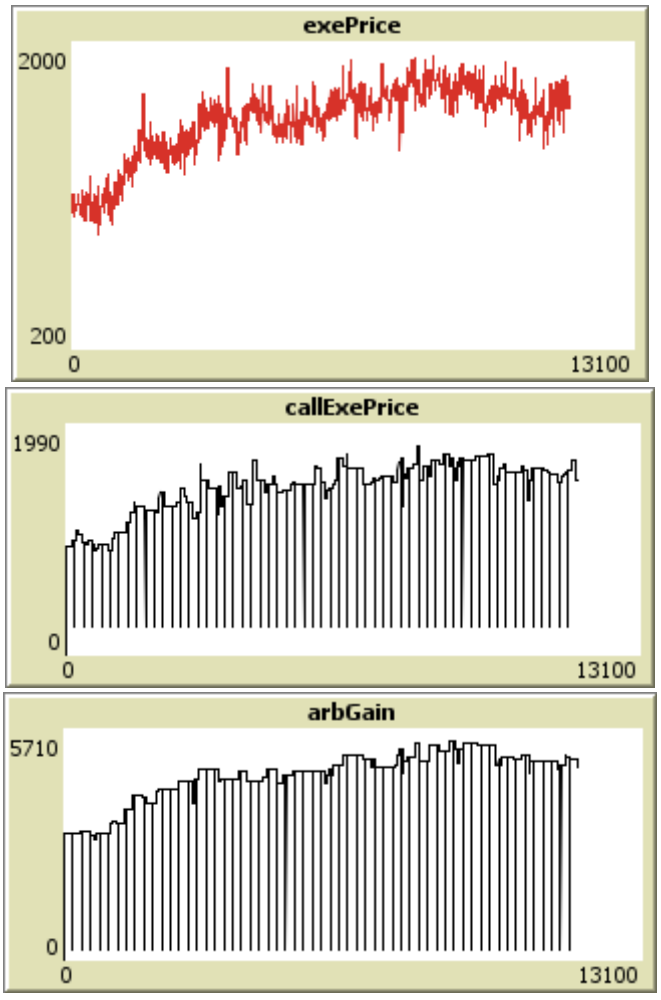
Parameters of the first simulation:

Number of agents trading stocks: 100
Number of agents trading call options: 100
Call option strike price: 100
Risk-free interest rate: 1%
Volatility: 0.01
Pass level: 0.3



Parameters of the second simulation:

Number of agents trading stocks: 100
Number of agents trading call options: 100
Call option strike price: 750
Risk-free interest rate: 10%
Volatility: 1.00
Pass level: 0.3



As for the risk-free rate case, the volatility of the stock price is used by the arbitrageur only, in order to compute the Black-Scholes option price. The financial theory tells us that an increased sigma results in a higher call option price, and our model is no exception. *Ceteris paribus*, this leads again to a lower arbitrage gain.

4. Conclusions

The Delta hedging strategy presented in our work grounds on solid theoretical bases, but we must be very careful about it. First of all, we based on the assumption of the absence of transaction costs and taxation, which is very helpful in simplifying the model and isolating some specific aspects, but is also extremely unrealistic. Second, we made further simplifying assumptions about several aspects.

While the formation of the stock price, based on the bid and ask theory, may seem a reasonable choice, it is still depurated from a series of aspects that are instead present in the real world: the influence of the dividends payments associated with the stocks, the possible existence of a mechanism of information learning (and sharing) for the agents, a much wider variety of assets (our work deals with one stock and one call option only), the preferences and the needs that the investors may have for various reasons (political, life-cycle, or even irrational reasons).

The same kind of assumptions is applied to the option market as well, with one more: we chose for the option agents a simple pricing method, based just on the difference between the stock market price and the strike price.

Although this wide set of simplifications, the empirical results still match the theoretical expectations. The choice of a Black-Scholes arbitrageur makes so that its gains maintain all the relations with the crucial variables that one would expect before running the simulations. The strike price, the volatility of the underlying price and the risk-free interest rate, when properly manipulated, still have the same kind of influence predicted by the formulas. And even the market call price, computed in the simple way we described, reacts well to changes in the strike price, going in the same direction suggested by the general theory.

In conclusion, our work catches a good number of dynamics proper of a complex environment such as the one represented by the intersection of a stock market, a derivatives market and an arbitrageur, even after a quite great amount of simplifying assumptions. On the contrary, we could even say that the presence of such assumptions is a very effective way to show the strength of certain theoretical results, which still valid in a more primitive environment like the one of our work.