Department of Computer Science

MSc in Computer Science

On percolation theory for agent-based financial modeling

Damien Deville

September 9, 2009

Supervisor: Dr. Matt Hall

Abstract

We apply percolation theory to financial modeling through the Cont-Bouchaud model. The project deals with the complete implementation in Java and testing of this model. We also derive major results and completely describe the statistics of the model.

A significative part treats the implementation of an algorithm able to efficiently find clusters in a lattice. We then present some results from percolation theory, and then discuss and implement the Cont-Bouchaud model. The results we find from percolation theory are in accordance with the literature. We find that the Cont-Bouchaud model is capable of generating results distributed as both a Gaussian and a power law, in function of the activity probability. The exponents found for the power law are also in accordance with the literature. A final part is dedicated to presenting and discussing the implementation of the model in Java.

A copy of this report and all the code written for the project can be found on my website:

http://www.ddeville.me

Contents

1 Introduction

During the past twenty years, there has been increasing interest from banks and financial firms in general for models able to represent asset price fluctuations [11]. These models were particularly needed for risk management purposes. Quantitative analysts originally from Mathematics or Physics university departments have been flooding the financial job market. This was mainly due to the fact that since the 1987 financial crisis the shape of the market has changed [1] [9] and particularly the distribution of returns that were until then assumed to be Gaussian [2] [23] nowadays clearly display fat tails [9]. The appearance of fat tails in the distribution of returns was the evidence that some risks inherent in asset prices were not taken into account if we assumed that the distribution of returns was Gaussian [1] [9]. The most famous model for option pricing, the Black-Scholes model created in 1973 [4], initially assumed that the asset price followed a log-normal distribution. It has widely been showed that this model is not consistent with real data [12]. Given the impossibility of representing financial returns with Gaussian models, many authors showed interest to the subject and many models have been presented to try to represent better asset price returns and particularly defining a new distribution for them.

Many approaches for modeling changes in asset prices traded on a market have been presented [18]. Some focus on the price itself assuming it follows a well-known stochastic process (such as the log-normal [4], but also some Lévy jump process [13] [21] or even mean-reverting processes [14]), others add a stochastic variance (such as stochastic volatility models [15] or GARCH models [5]).
Another approach is to focus on the agents composing the market and try to understand their behaviours, actions and interactions, in order to measure their effect on the system as a whole: this approach is called agent-based modeling [22]. In agent-based financial modeling, we are particularly interested in summing up individual demand and supply in order to get the global equilibrium directly proportional to the price change.
In agent-based modeling, a lot of models have been proposed (for a complete review of these

models see [22]). We decide to focus on the Cont-Bouchaud percolation model for its simplicity and its efficacy [22].

The Cont-Bouchaud model is based on percolation theory, we will first introduce this. Percolation theory is the study of the behavior and statistics of clusters on lattices. Suppose we have a large square lattice where each cell can be occupied with probability p and empty with probability $1 - p$. Examples of such occupied lattices are shown in Figure 1.

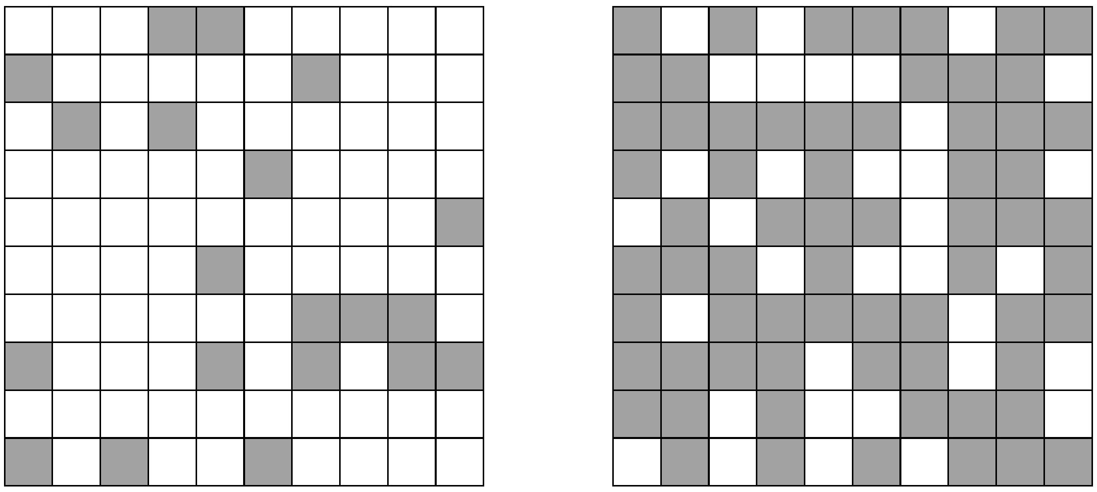


Figure 1: Examples of occupied 10x10 lattices with an occupation probability $p = 0.2$ (left) and $p = 0.7$ (right). Grey cells are defined as occupied.

Each group of neighbouring occupied cells forms a cluster. Neighbours are defined as cells having a common side but not those sharing only a corner as depicted in Figure 2 (thus a neighbour is a cell at the top, bottom, left or right of the current cell, but not on the diagonal). It is important to understand that the occupation of the cells is random and each occupied cell is occupied independently of the status of its neighbours. The number of clusters in the lattice, the size of each one and their distribution in the lattice are important topics in percolation theory.

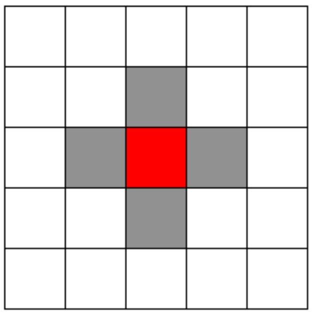


Figure 2: Definition of a neighbouring cell in the lattice. Assume the red cell is the current cell: its neighbours are the four grey cells on top, bottom, left and right.

In this project, we first discuss an algorithm for finding clusters in a lattice since it represents the keystone for discussing and implementing percolation theory. We then present and test some results from percolation theory. Finally, we introduce the Cont-Bouchaud model and discuss some results. The final section describes implementation.

2 Implementing the cluster-finding algorithm

As we will explain further below, running the model depends on correctly identifying all clusters on a lattice. This is a non-trivial operation and indeed, most of the efficiency of the whole model directly depends on the efficiency of the cluster-finding algorithm. In this setting, being able to compute the cluster sizes for each given lattice in a very small amount of time was a prerequisite for being able to study the model more in depth. Thus, a large amount of time has been devoted to designing a good cluster-finding algorithm. We also have to decide on a good data structure to store the clusters. We will need to get cluster sizes quite often while implementing the model, so a fast access to them will improve the efficiency of the whole model. Since checking if a lattice percolates means checking if a cluster spans the whole lattice from top to bottom or side to side, the quality of the cluster-finding algorithm could also be measured by its capability of detecting some unusual percolating clusters as the one depicted in Figure 3. It is an interesting subject since, while writing and modifying the code, it often seems that we will always find a new cluster shape that our algorithm cannot detect! However, the concordance between the value we found for p_c and the value usually found in the literature [24] is strong evidence that our algorithm is capable of finding all those clusters (see section 3).

The principle behind such an algorithm is to count the cluster number in the lattice and to store the size of each cluster. We thus have to span the whole lattice from top-left to bottom-right and assign each cell to a cluster.
For each occupied cell, we check if the cells at the top and left are occupied. We then have four possibilities:

1. If both cells are empty, create a new cluster for the current cell.
2. If only one cell is occupied, the current cell belongs to the cluster this occupied cell belongs to.
3. If both cells are occupied and belong to the same cluster, the current cell belongs to that same cluster.

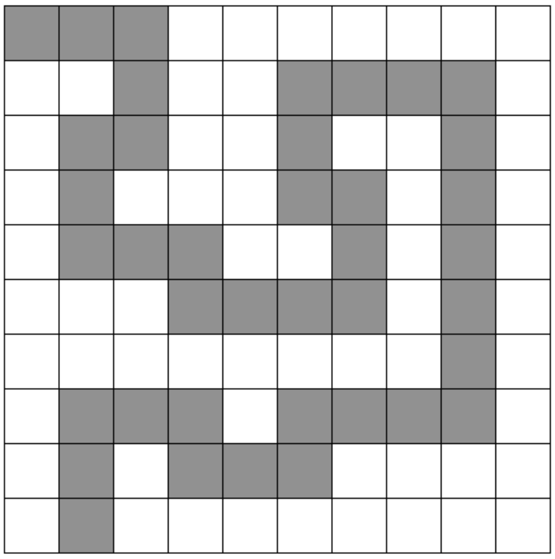


Figure 3: Example of a lattice pattern displaying an odd percolating spanning cluster that the cluster-finding has to be able to recognize

4. When both cells are occupied but each one belongs to a different cluster, the current cell will create a link between these two clusters. We have to find an easy way of linking these two clusters while setting the current cell to one of them.

The first idea we had was to implement a proper algorithm based on Java Collections. Each cluster was represented as an ArrayList containing cell objects. Each time we decided a cell belonged to a cluster, we had to add this cell to the corresponding cluster ArrayList. The problem was that each time we wanted to check if a cell belonged to a cluster, we had to check and span all ArrayLists in order to find the cell. Since for large lattice sizes, clusters can become quite large, this method was slow and definitely not scalable.
We then decided to look at the literature and found there were two famous algorithms that already tried to solve this problem: the Leath algorithm [20] and the Hoshen-Kopelman algorithm [16].
The Leath algorithm is based on recursions and while quite efficient, it is not fast enough for our purpose. Given its efficiency, we focus on the Hoshen-Kopelman algorithm.

2.1 The Hoshen-Kopelman cluster-finding algorithm

The Hoshen-Kopelman algorithm is based on the well-known union-finding algorithm. It works by assigning a label to each cluster. Then, if we have to link two clusters, we create a union between both labels and set the cell as the lowest of the two labels. When we span the lattice a second time, we find the unions and update the lattice. An example of the Hoshen-Kopelman algorithm performed on a 6x6 lattice is shown in Figure 4.

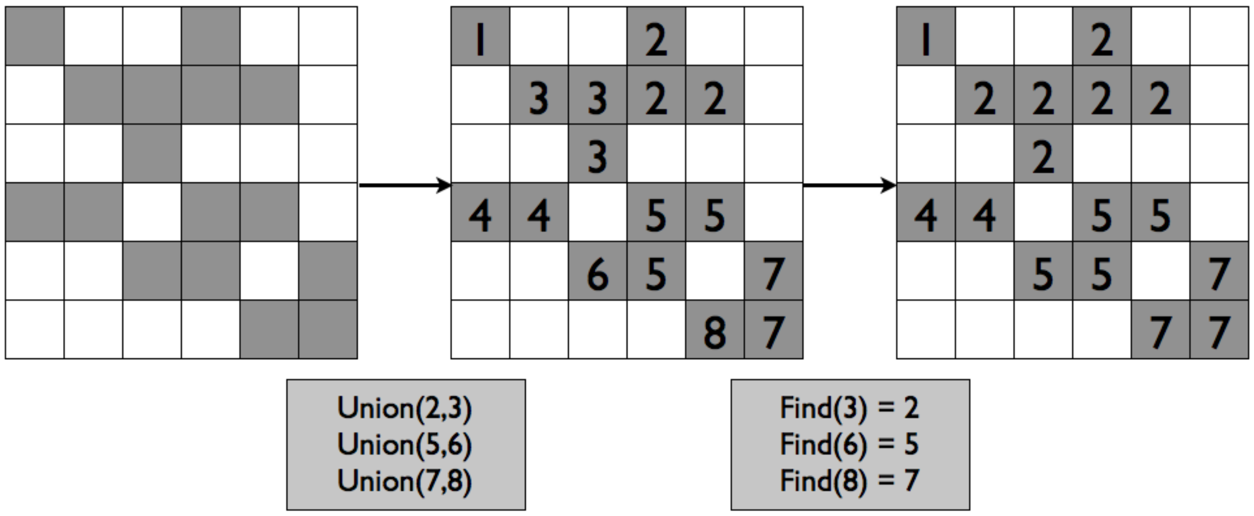


Figure 4: The two-steps Hoshen-Kopelman algorithm performed on a 6x6 lattice. The grey cells are occupied. The first step consists in spanning the lattice once and assigning cluster labels to each occupied cell. If a link between two clusters has to be made, we create a union relation between these two cluster labels. A second step consists in spanning the lattice a second time and finding and updating the cluster labels (to insure each cluster is represented by only one label).

The implementation of the Hoshen-Kopelman algorithm is as follows:
We first span the lattice once. Each time we find an occupied cell, we check the neighbors at the top and left of the current cell. We have then four possibilities:

1. Both cells are empty: we create a new cluster label and set it to the current cell
2. Only one cell is occupied: we set the cluster label of the occupied cell to the current cell
3. Both cells are occupied and have the same cluster label: we set this cluster label to the current cell
4. If both cells are occupied but have distinct cluster labels, we set the smallest to be the current cell cluster label and we add the union between both cluster labels as a new entry into a labels HashTable where the key is defined as the largest label of the two and the corresponding value is the smallest one. If the key already exists in the HashTable, we have to use the **find** function. We explain this further below.

In the labels HashTable, when a value is equal to its key, it means the cluster that the key's label represents is not linked to any other cluster. A label of the type $V(n) = n$ is called *good* label (while a label of the form $V(n) = m$ is called a *bad* label). We thus need a **find** function able to tell us the smallest *good* label each *bad* label is linked to.
The **find** function works as follows: given a *bad* cluster label, we go recursively through each union in the labels HashTable (of the type $V(n) = m$) until the key is equal to the value meaning this is the smallest *good* label the current label is linked to.
The second step of the algorithm consists of spanning the whole lattice a second time and applying this **find** function to the cluster label of each occupied cell we encounter. We can now be sure that each cluster in the lattice is represented by only one cluster label and this label is the smallest *good* label we can find.
Doing so, we manage to compute the algorithm and find clusters for a 1000x1000 lattice in around 800ms while a 500x500 lattice in 200ms.
Pseudocode for the algorithm is shown in Listing 1 and pseudocode for the **find** function is shown in Listing 2.

The best approach for testing the Hoshen-Kopelman algorithm is to deterministically draw clusters in lattices, then run the algorithm and see how it behaves and if it gives the

```
For i = 1 To size(lattice)
   For j = 1 To size(lattice)
      get current cell(i,j) ;
      check top cell(i-1,j) ;
      check left cell(i,j-1) ;
      If(top and bottom cells are empty)
         Then create new cluster label and set to current cell ;
      Else If(just one cell is occupied)
         Then set the occupied cell cluster label to the current cell
      Else If(both cells have the same cluster label)
         Then set this cluster label to the current cell ;
      Else If(both cells are occupied but have distinct cluster label)
         Then
         {
            assign smallest cluster label to the current cell ;
            create a union relation $V(n)=m$ in the labels HashTable ;
         }
      End
   End
End
```

Listing 1: Hoshen-Kopelman algorithm for finding clusters in a site lattice (Pseudocode)

```
$labels$ ← labels HashTable ;
$labelNum$ ← initial bad label ;
While(value matching the key $labelNum$ in $labels$ $\neq$ $labelNum$)
   $labelNum$ ← value matching the $labelNum$ key in $labels$ ;
End
Return labelNum ;
```

Listing 2: The **find** method used in the Hoshen-Kopelman algorithm (Pseudocode)

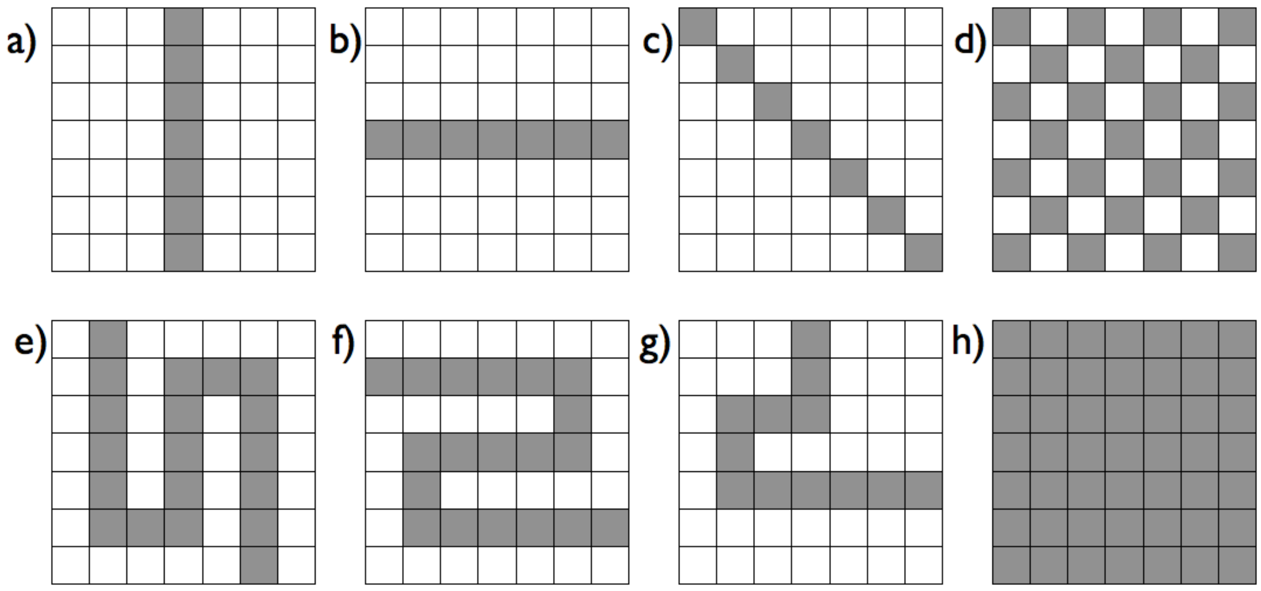


Figure 5: Lattices with different cluster shapes used to test the efficacy of the cluster-finding Hoshen-Kopelman algorithm

lattice	a	b	c	d	e	f	g	h
percolation?	yes	yes	no	no	yes	yes	no	yes
number of clusters	1	1	7	25	1	1	1	1

Table 1: Results of the test of the efficacy of the cluster-finding Hoshen-Kopelman algorithm on the lattices from Figure 5

results we expect. By repeatedly designing lattices with unusual cluster shapes (percolating or not) and testing how well the algorithm finds them and determines their size, we can perform an efficient testing procedure. Examples of tested lattices and the results of the applying algorithm on them are shown in Figure 5 and Table 1.

3 Results from percolation theory

After having discussed an efficient algorithm for finding clusters in a lattice, we can now present some results from percolation theory, implement and test them.

It is important to note that, in the following sections, in order to get the results, we often proceed computing the ensemble average. An ensemble consists of a large number of experiments of a system, considered all at once, each of which represents a possible state that the real system might be in. In our case, we can consider an ensemble as a large number of returns generated by the model for a given lattice. Finally, we take an average of a large number of ensembles. Even if the ensemble average is dependent on the ensembles chosen, if we consider a large number of ensembles, the value of the ensemble average should stabilize and tend to the real value.

3.1 The critical probability

Clearly, the expected size of a cluster of cells is a function of p. As p is increased, we would expect larger clusters and for some probability we expect to find clusters that span the entire lattice from one side to the other. This is referred to as the critical probability p_c. Assuming the size of the lattice is infinite, for a probability p below p_c, there cannot be any cluster that spans the whole lattice and for a probability p above p_c there is one cluster that spans the whole lattice. The exact value of p_c depends on lattice topology [25], as bonds and sites lattices. Here we concentrate on sites lattices.

We can find this critical probability by numerical simulation. We compute a number of experiments for each given probability on a lattice. For each experiment, we check how many lattices contain a spanning cluster. Some pseudocode illustrating this routine is listed in Listing 3.

As the size of the lattice increases, the interval of probabilities generating a spanning cluster narrows and it is then easier to find the critical probability. In Figure 6 we plot the

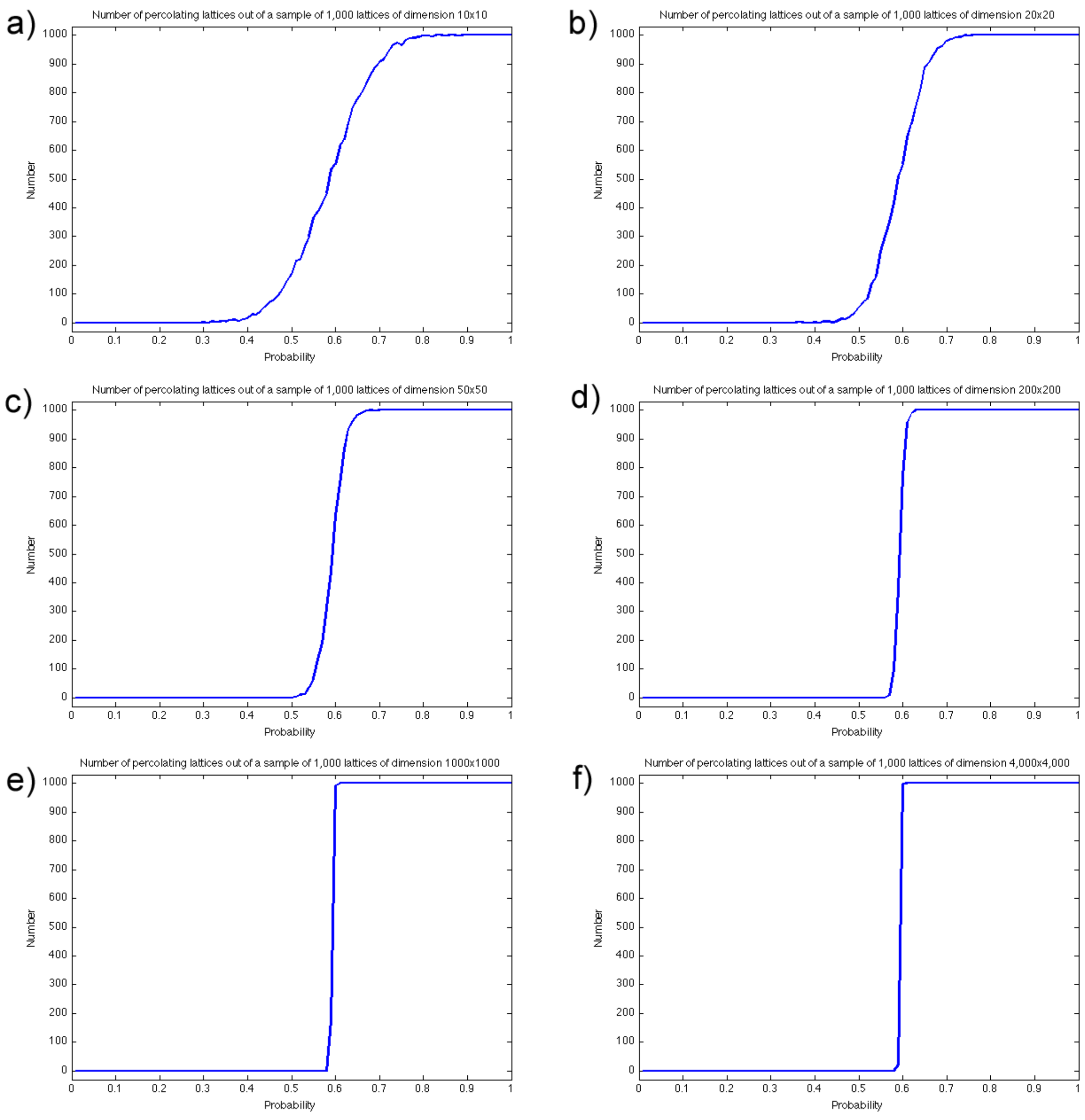


Figure 6: Looking for the critical probability. The x-axis is the probability p. The y-axis is the number of lattices percolating (containing a top-bottom spanning cluster) out of a sample of 1,000 lattices. The experiment is run on lattices of size (top-left to bottom-right) 10 (a), 20 (b), 50 (c), 200 (d), 1,000 (e) and 4,000 (f).

```
For $i$ = 1 To 100
    $prob$ ← $i$/100 ;
    $counter$ ← 0 ;
    For $j$ = 1 To number of experiments
        generate a new lattice with probability $prob$ ;
        check if the lattice percolates ;
        If(lattice percolates)
            increment $counter$ by one ;
    End
    Print $counter$ ;
End
```

Listing 3: Algorithm to find the critical probability

results of the numerical simulations we have performed. Taking a 0.01 step between each probability, we clearly see that the critical probability is between 0.59 and 0.60. Taking a smaller interval (0.001) between 0.59 and 0.60, we find that the critical probability is between 0.592 and 0.593. Going ahead, we find a critical probability of 0.5927 ± 10^{-4} which is in good agreement with the value found in the literature ($p_c = 0.5927464$) [25].

3.2 Number of clusters

The number of clusters in the lattice and the size of each one will be important in the definition of the Cont-Bouchaud model. We are therefore interested in finding the distribution and sizes of clusters and particularly the number of clusters present in the lattice for a given probability. We thus compute the number of clusters in lattices of various sizes for a range of probabilities between 0 and 1. Listing 4 describes an algorithm for generating cluster distribution data.

In Figure 7 we plot the number of clusters for each probability and for each lattice size. In order to compare the lattices of different sizes, we scale the number of clusters in each lattice dividing it by the square of the lattice size L. We can see that the result is approximately independent of the size of the lattice. The probability that generates more clusters is 0.27 ± 0.01 which is the value usually found in the literature [29].

```
For $i$ = 1 To 100
    $prob$ ← $i/100$ ;
    $counter$ ← 0 ;
    $num$ ← number of experiments ;
    For $j$ = 1 To $num$
        generate a new lattice with probability $prob$ ;
        get the number of clusters in this lattice ;
        $counter$ ← $counter$ + number of clusters ;
    End
    Print $counter/num$ ;
End
```

Listing 4: Algorithm to find the probability that insures the maximum number of clusters

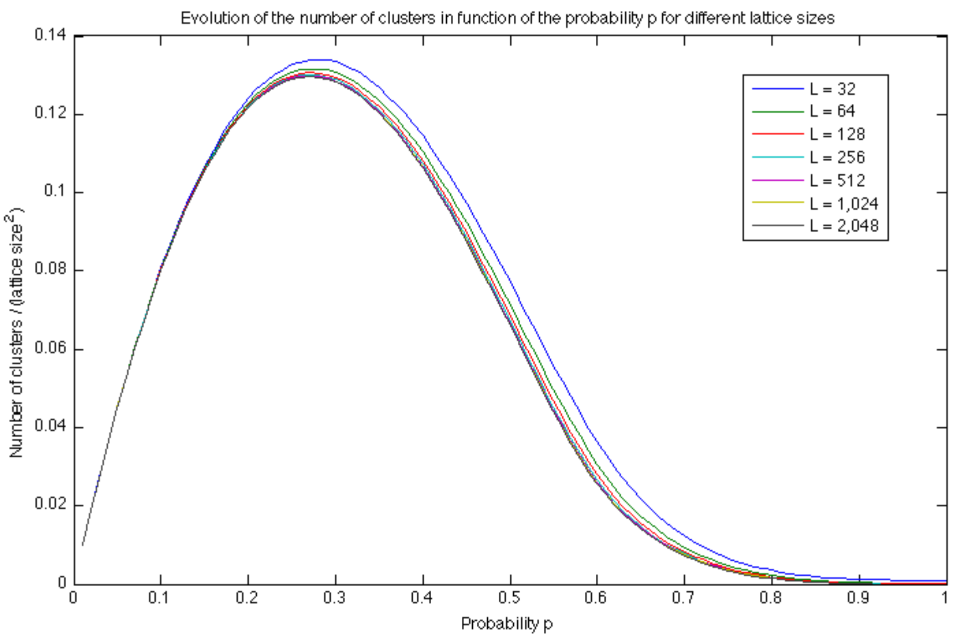


Figure 7: The number of clusters in the lattice for each probability. Experiments are performed for various sizes of the lattice. For each lattice size, we used a sample of 5,000 experiments (for sizes 32 and 64), 2,000 experiments (for size 128), 200 experiments (for size 256 and 512) and 50 experiments (for size 1,024 and 2,048).

3.3 The cluster size distribution

In order to determine the cluster size distribution, we simulate 10,000 experiments on a 501x501 latttice for probabilities ranging from 0.1 to 0.6. We then take an average of occurrences of each cluster size. The distribution of the cluster sizes follows a power law distribution [25]. A power law distribution describes a special relationship where the frequencies decrease very slowly as the sizes of the event increase.

In Figure 8 and Figure 9, we plot the cluster size distribution for various probabilities (0.1, 0.2, 0.3, 0.4, 0.5 and 0.6). Notice that when using a probability $p = 0.6 > p_c$, as assumed in [27] we do not take into account the percolating cluster (this special cluster that spans the lattice from one side to the other and that theoretically, assuming the size of the lattice is infinite, would have an infinite size) in the plotting of the distribution. Then, for each probability, we fit the distribution with the following power-law

$$f(x) = ax^k \tag{1}$$

where a and k are constant, x is the cluster size and k is called the *scaling exponent.*

All the plotting was performed using Matlab [1] and some basic code used for obtaining histograms is listed in Listing 5.

Power laws were fitted to the empirical data using the Ezyfit [2] Matlab toolbox.
We obtain good fits for all probabilities. The *R-squared* values are indeed close to 1 for all probabilities (see Figure 8 and Figure 9 for the precise values of the *R-squared*).
On the right-hand side of the figures, we plot the same distributions but on a log-log scale. We notice that if we take the logarithm on both sides of the power-law equation

$$\begin{aligned} \log(f(x)) &= \log\left(ax^k\right) && (2)\\ &= \log(a) + \log(x^k) && (3)\\ &= \log(a) + k\log(x). && (4) \end{aligned}$$

[1]http://www.mathworks.com
[2]http://www.fast.u-psud.fr/ezyfit

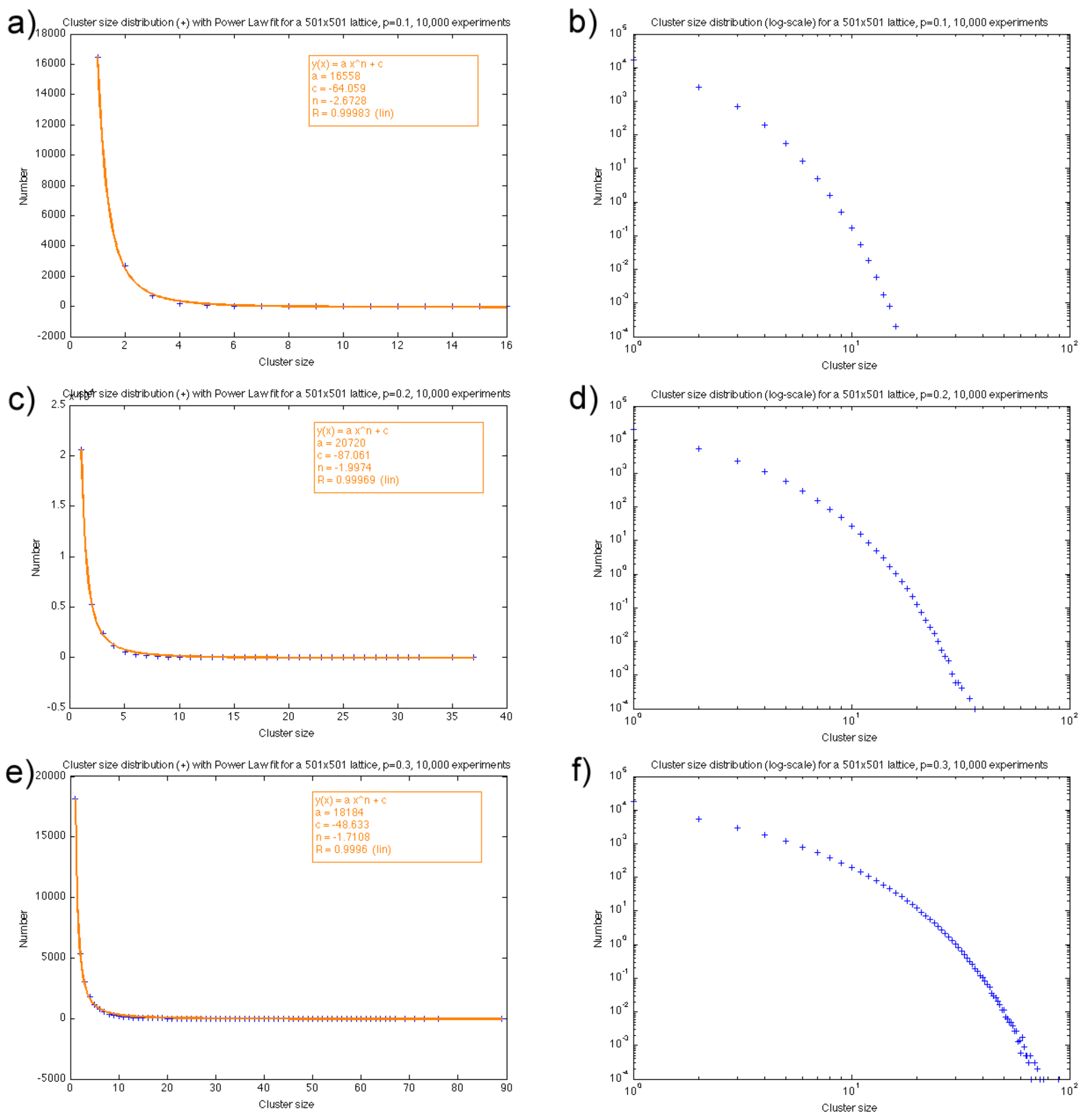


Figure 8: Cluster sizes distribution among a 501x501 lattice for various cell occupancy probabilities: 0.1 (a and b), 0.2 (c and d) and 0.3 (e and f). On the left-hand side, we plotted the distribution on a linear scale (blue “plus” signs) and we fitted a power law to it (orange line). On the right-hand side, we plotted the same empirical distribution on a log-log scale.

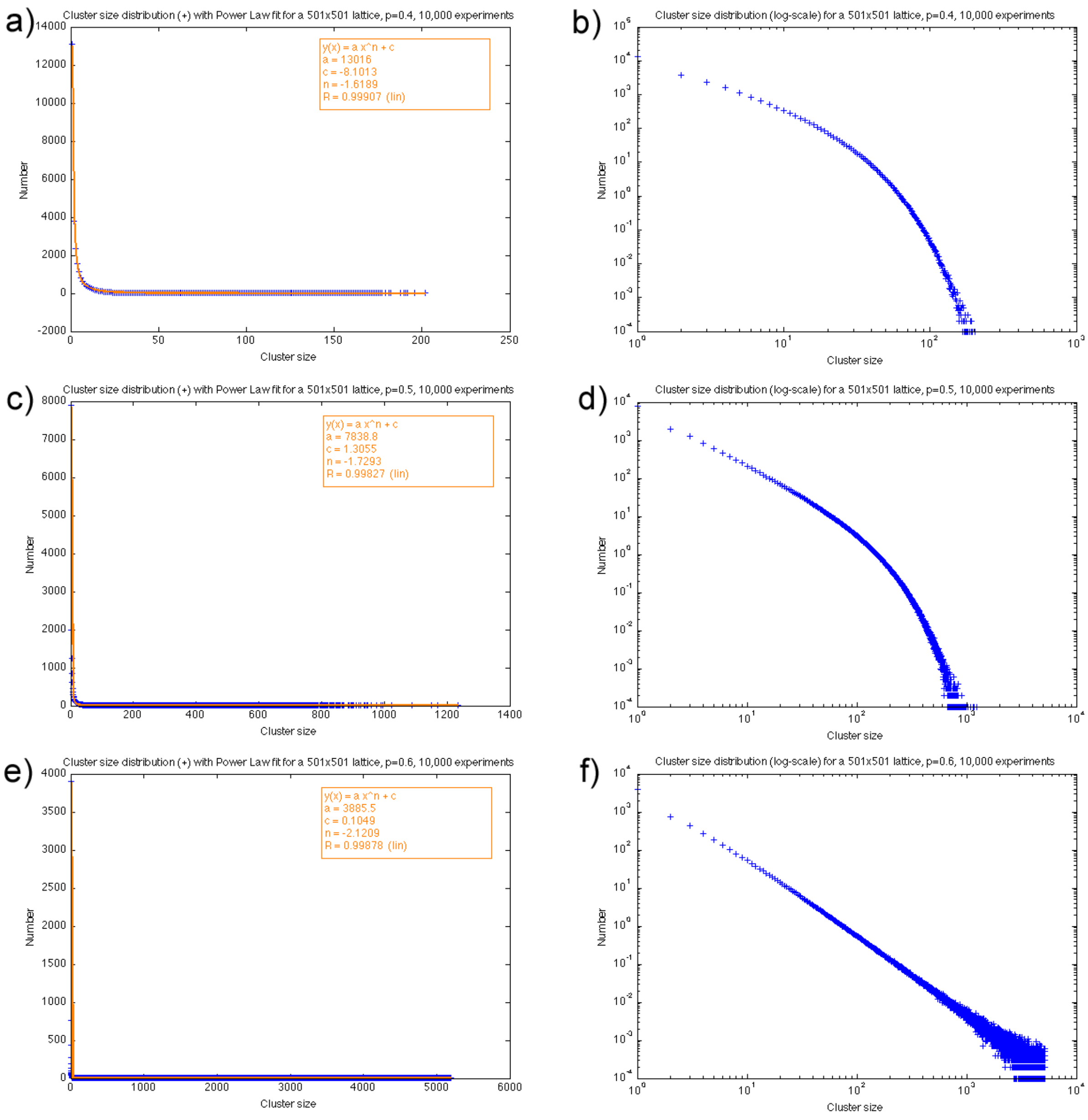


Figure 9: Cluster sizes distribution among a 501x501 lattice for various cell occupancy probabilities: 0.4 (a and b), 0.5 (c and d) and 0.6 (e and f). On the left-hand side, we plotted the distribution on a linear scale (blue "plus" signs) and we fitted a power law to it (orange line). On the right-hand side, we plotted the same empirical distribution on a log-log scale. In the case $p = 0.6$, since $p \geq p_c$ the critical probability, there is formation of an "infinite" spanning cluster. This was removed when plotting the clusters distribution.

```
S = 'output' ;
n = 100 ;
dim = 200 ;
xfinal = zeros(1,dim) ;
yfinal = zeros(1,dim) ;

for i=1:n
   num = int2str(i) ;
   x = [S num '.txt'] ;
   fid = fopen(x,'rt') ;
   a = fscanf(fid , '%f') ;
   a = abs(a) ;
   %a = sqrt(a) ;
   [y,x] = hist(a,dim) ;
   yfinal = yfinal + y ;
   xfinal = xfinal + x ;
end

yfinal = yfinal/n;
xfinal = xfinal/n ;
loglog(xfinal , yfinal , '+')
%plot(xfinal , yfinal , '+')
```

Listing 5: Matlab code for getting the histogram graphs

Then

$$\log(f(x)) = k\log(x) + \log(a) \tag{5}$$

the equation becomes a linear relationship where k is the slope. Thus the power law curve becomes a straight line on log-log plot. So, looking for the exponent of a power-law equation reduces to looking for the slope of an elementary linear equation.
A similar property is found for exponential functions where

$$f(x) = ae^{bx}. \tag{6}$$

Taking a logarithm on both sides leads to

$$\begin{aligned} \log(f(x)) &= \log\left(ae^{bx}\right) & (7)\\ &= \log(a) + bx & (8)\end{aligned}$$

Then

$$\log(f(x)) = bx + \log(a) \tag{9}$$

and the equation is again a linear relationship where b is the slope. Thus the exponential function curve becomes a straight line on log-log plot.

These properties of both the power law and the exponential function are useful since they facilitate the fitting, reducing it to a classic OLS.

For a probability $p < p_c$, the distribution of the cluster sizes looks pretty much like an exponential distribution while when $p \geq p_c$, it is clearly a power-law [22], [25].

4 The Cont-Bouchaud percolation model

Now we are capable of generating lattices with randomly occupied cells and compute the cluster number and sizes, we can apply this to financial modeling through the Cont-Bouchaud model.

4.1 Why percolation theory in financial modeling?

In agent-based financial modeling, we are looking for a way to represent each actor (such as a trader or any kind of investor) in the system on its own (at the *micro* scale) and a way to sum up the effects of each actor in order to have a scaled vision of the whole system (at the *macro* scale).
In this setting a cell represents a trader, a lattice represents the whole market and clusters represent groups of investors making joint decisions in the market. Percolation theory provides a good framework for agent-based financial modeling. The lattice allows different patterns for the market, different locations for each trader, and by the formation of clusters different centers of interest where traders interact, exchange information and end taking similar decisions in their investment.
Percolation theory is both a simple setting for a model but also a rich one, offering many possibilities for financial modeling.

4.2 The model

The main purpose of the Cont-Bouchaud model is to investigate the phenomenon of herding between traders. For example, traders working in the same bank may have similar opinions about the market due to communication between themselves. Hence, the model assumes that cells (traders) that are close enough to belong to the same cluster share the same opinion about the market and make the same moves. Then, neighbor cells form clusters that represent traders making joint decisions. This is how the Cont-Bouchaud model represents herding phenomena in the financial markets [10].
Each cluster can decide to buy with probability a, sell with probability a or sleep with

probability $1 - 2a$. Thus, a small a means a few trades at each time interval while a large a (close to its maximum value 0.5) means that a large fraction of the traders participates in the market. Traders in clusters behave identically, so the quantity purchased at each time step by each cluster is directly proportional to the cluster size.

If the cluster buys, it buys a quantity ϕ^{buy} proportional to the cluster size and if it sells, it sells a quantity ϕ^{sell} proportional to the cluster size. Then, at each time interval, the difference between supply and demand is given by the following formula

$$\Delta = \frac{1}{\lambda}\left(\sum_i \phi_i^{buy} - \sum_i \phi_i^{sell}\right) \tag{10}$$

where λ is a *scaling component* representing the excess demand needed to move the price by one unit.

The logarithm of the price is then supposed to change proportionally to Δ [6], [7], [10], [24], [30]

$$\log(P) = \Delta \tag{11}$$

In this way, the change in price is proportionally determined by the difference between supply and demand.

4.3 Generating time-series

Using the Cont-Bouchaud model in order to generate returns, we can then easily generate prices time-series. Figure 10 shows two time-series generated with the Cont-Bouchaud model for two different values of the activity probability a.

4.4 The distribution of returns generated by the Cont-Bouchaud model

We are particularly interested in the distribution of the returns generated by the Cont-Bouchaud model. We know [9] [10] that in the market, given the presence of bubbles and the risk of crashes (both characterized by sharp up or down price fluctuations) the returns are not Gaussian but follow a fat-tailed distribution (where the fat tails actually model these

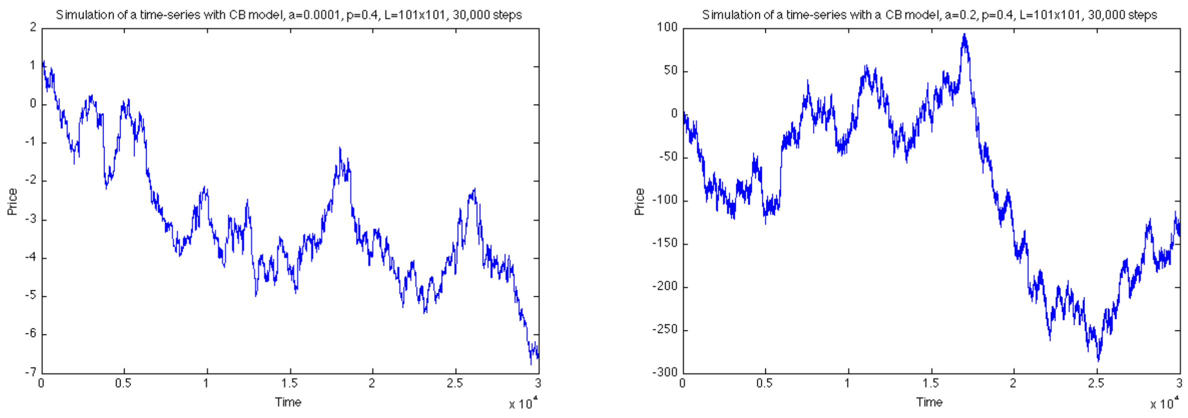


Figure 10: Two time-series generated with the Cont-Bouchaud model with the following parameters: lattice size=101x101, p=0.4, a=0.0001 (left) and a=0.2 (right) for 30,000 steps.

sharp fluctuations). Indeed, the fat tails in the distribution are the product of the additional risk in financial prices implied by the fear of crashes and bubbles.

One interesting characteristic of the Cont-Bouchaud model is the possibility, by selecting different parameters, of generating returns distributed differently. For a very small activity probability a, at each time step, only one (or a few) cluster trades. It follows that the distribution of the returns scales as the well-known [25] cluster size distribution of percolation theory (see section 3) [27], that is to say an exponential distribution or a power law. However, when the activity probability a is increased, at each time step, many clusters trade simultaneously. The returns are then proportional to the sum of each cluster size (characterizing the size of the trade). The *Central Limit Theorem* [3] states that the sum of a sufficiently large number of independently generated random numbers will be approximately Normally distributed. From this, we can argue that for $a \to 0.5$ the distribution of returns will tend to a Gaussian.

We are now analyzing the distribution of returns in both regimes (when $0.1 \leq a \leq 0.5$ and when $a \leq 0.1$).

a	Mean	St. dev.	Skewness	Kurtosis	R^2
0.50	-0.5574	4,902.00	0.00065	2.6486	0.99920
0.40	-1.4762	4,360.20	0.00063	2.6884	0.99939
0.30	1.6341	3,814.70	-0.00120	2.7709	0.99982
0.25	0.0368	3,429.20	-0.00074	2.8239	0.99994
0.20	-1.0861	3,103.10	0.00063	2.9105	0.99994
0.16	0.8861	2,766.10	0.00075	3.0087	0.99995
0.10	-0.5435	2,154.20	-0.00045	3.3320	0.99810
0.05	0.4659	1,546.60	0.00120	4.2486	0.99385
0.01	-0.0505	707.95	0.00039	12.0436	0.99689

Table 2: Results from the Gaussian distribution fitting on the experimental returns generated with the Cont-Bouchaud model.

4.4.1 The distribution of returns for $0.1 \leq a \leq 0.5$

In Figure 11, we plot the distribution of returns generated with the Cont-Bouchaud model (summing up over all clusters generated by a lattice for p comprised between 0.01 and 0.59) for an activity probability a equal to (from top-right to bottom-left) 0.5, 0.4, 0.3, 0.25, 0.2, 0.1, 0.05 and 0.01.

We observe that for a value of the probability a comprised between 0.1 and 0.5 the distribution of the returns seems to look pretty much like a Gaussian distribution, however, when $a \leq 0.1$, we also clearly see that the distribution is no longer Gaussian. The results from the fit are summarized in Table 2. We can clearly see that while for a large value of a ($a \geq 0.1$) the regression curve is a good fit to the data (given the value of R^2 almost equal to 1) whereas the fit is a lot worse for values of $a \leq 0.1$ (the value of R^2 is lower).

In order to characterize better its distribution, we now consider the four first moment of the empirical data generated with the Cont-Bouchaud model.

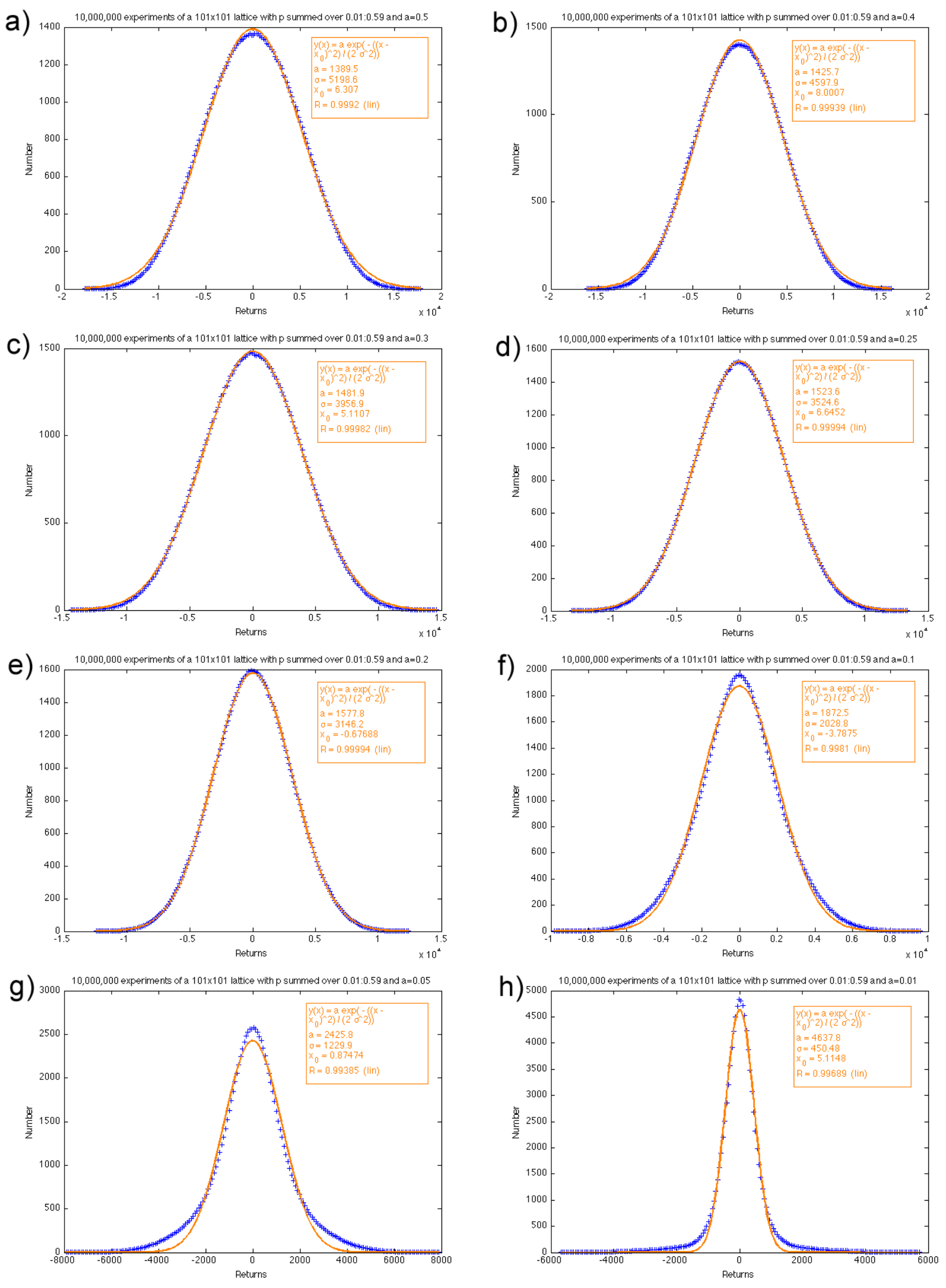


Figure 11: Distribution of returns generated with the Cont-Bouchaud model for various values of the probability a (from top-left to bottom-right: 0.5 (a), 0.4 (b), 0.3 (c), 0.25 (d), 0.2 (e), 0.1 (f), 0.05 (g), 0.01 (h)). The blue crosses are the experimental data while the orange curve is the Gaussian distribution fit

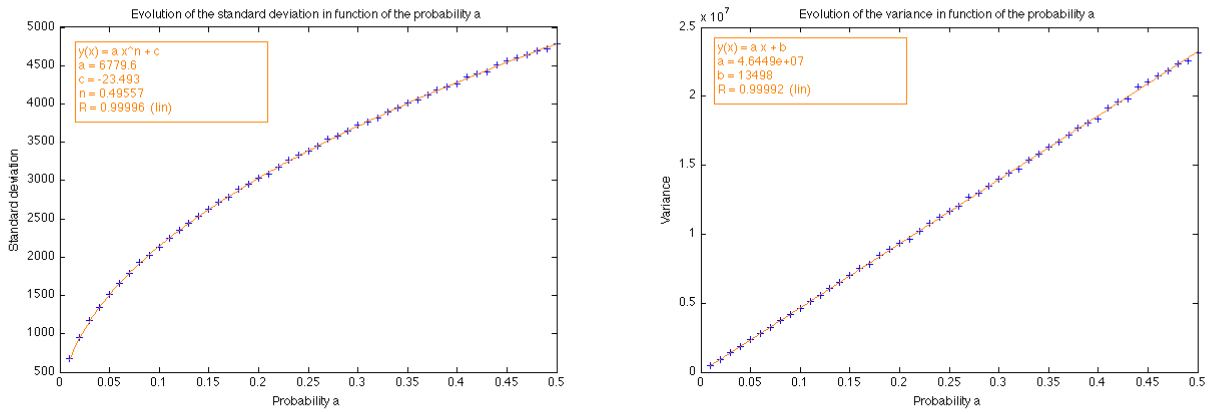


Figure 12: Evolution of the standard deviation (left) and variance (right) in function of the probability a.

Mean, μ

We clearly see that in all cases the mean of the returns is 0 ± 1.5, which is a characteristic of the Cont-Bouchaud model since positive and negative returns have exactly the same probability of occurring and in the same amount.

Standard deviation, σ

In Figure 11 and Table 2 we see that when the value of the probability a increases, the standard deviation also increases. This is due to the fact that a larger value of the activity a in the market leads to a greater number of actors participating in the market, synonym of greater supply and demand and thus larger variance of returns. We are particularly interested in knowing exactly how the standard deviation behaves when the the probability a increases. In Figure 12, we plot the value of the standard deviation and the variance. In order to get these values, we generate 1,000 steps of returns generated with the Cont-Bouchaud model using a lattice of size 101x101. We then compute the standard deviation out of these data. Performing this 50 times, we get 50 different values of the standard deviation out of which we compute the average to get the final value of the standard deviation for a given a. We perform this for all values of a in the range $0.01 - 0.50$ with an increment of 0.01 between

each value of a.
The shape of the curve representing the evolution of the standard deviation as a function of a looks like drawn from a square-root function and, indeed, when we try to fit it with a power law equation of the type $y = ba^n + c$, we find a value of the exponent n very close to 0.5 (exactly $n = 0.496 \pm 10^{-2}$), and the value of c being insignificantly small ($c = -23.5 \pm 10^{-1}$ while the range of the function is $[700, 5000]$). We can then assume that the standard deviation increases with the probability a as the function $y = b\sqrt{a}$.
Given that the variance is simply the square of the standard deviation, if we assume the standard deviation is driven by a square-root equation, the variance should be a linear equation. And indeed, looking at the right-hand side of Figure 12, we clearly see that the variance plot is a linear equation. When fitting the empirical data with a linear equation of the form $y = ba + c$, we get an almost perfect fit and again the value of c is insignificantly small. The linear equation is thus of the form $y = ba$. We can thus conclude that the variance of the returns generated with the Cont-Bouchaud model increases linearly with the activity probability a.

Skewness, s

Similarly, as referred in Table 2, the value of the skewness (a measure of the symmetry of the distribution) is almost equal to 0 for any value of the probability a ($s = 0 \pm 10^{-3}$ for any a). This is in accordance with a Gaussian distribution (the Gaussian distribution, being symmetric by definition, has a skewness of 0).

Kurtosis, k

Another important characteristic is the value of the kurtosis changing as a function of a. We know that a Gaussian distribution has a kurtosis of 3. A perfect fit between the empirical data and a Gaussian distribution should also display a kurtosis of 3. Looking at the fit between the empirical data and the Gaussian distribution in Figure 11, we see that for large values of a, the empirical data display smaller tails than the Gaussian distribution (and thus a lack of kurtosis) and for small values of a, the empirical data display fat tails (and thus

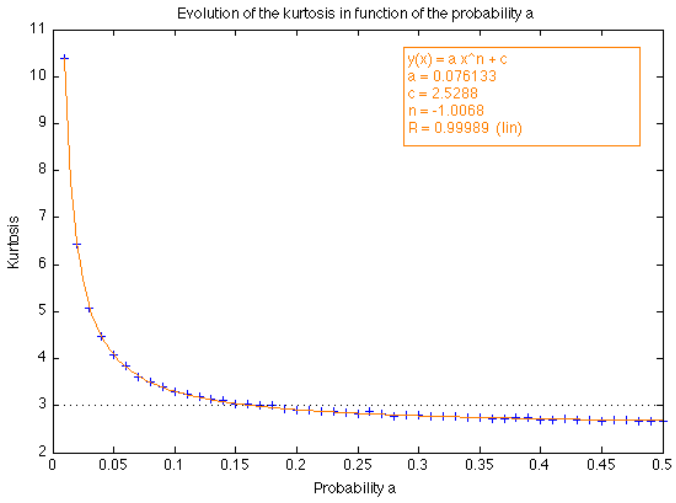


Figure 13: Evolution of the kurtosis in function of the probability a.

an excess of kurtosis). We can confirm this feature looking at the Table 2: for a value of $a < 0.10$, the empirical data display a kurtosis greater than 3 (which characterizes an excess of kurtosis); similarly, for a value of $a > 0.20$, the value of the kurtosis is less than 3 (which characterizes a lack of kurtosis).

We are interested in the evolution of the kurtosis as a function of the activity probability a and particularly in the value of a that gives a kurtosis of 3, i.e. describing a Gaussian distribution (assuming that the skewness is 0). In Figure 13, we plotted the evolution of the kurtosis in function of the activity probability a (with values taken in the range 0.01-0.50). The plot of such kurtosis evolution clearly has the shape of a power-law curve. This feature can be confirmed looking at Figure 14 which shows the same data on a log-log scale graph. We can then fit the empirical data to a power-law equation. We get a good fit and find a value of the exponent n close to -1 (exactly $n = -1.0068$). Assuming the power law equation has the following parameters $y = ba^n + c$, we can thus assume that the equation describing the kurtosis is function of the activity probability a is $y = b\frac{1}{a} + c$.

We get a good fit (value of R^2 very close to 1). We can therefore assume this fitted power

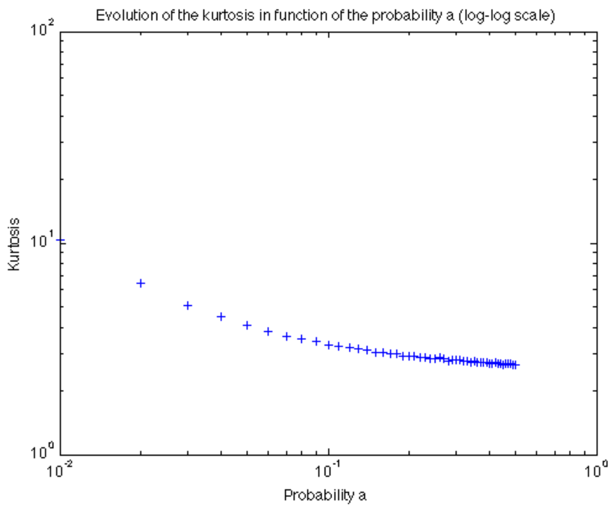


Figure 14: Evolution of the kurtosis in function of the probability a.

law equation describes the data well. We get the following values for the parameters

$$\begin{aligned} b &= 0.076 \pm 10^{-3} \\ n &= -1.007 \pm 10^{-3} \\ c &= 2.529 \pm 10^{-3} \end{aligned}$$

As we mentioned earlier, the empirical data display an excess of kurtosis for values of a below a value $a^* \in [0.10, 0.20]$ and a lack of kurtosis above a^*. Since we know the value of both the parameters of the equation and assuming that $y^* = 3$ being the value of the kurtosis for a Gaussian distribution, we can easily get the exact value of a^* insuring a kurtosis of 3 assuming that the power law equation can be rewritten as

$$y = ba^n + c \tag{12}$$

$$\frac{y-c}{b} = a^n \tag{13}$$

$$n \log a = \log\left(\frac{y-c}{b}\right) \tag{14}$$

$$\log a = \frac{\log\left(\frac{y-c}{b}\right)}{n} \tag{15}$$

$$a^* = \exp\left(\frac{\log\left(\frac{y^*-c}{b}\right)}{n}\right) \tag{16}$$

Substituting the fitted values into equation (16), we obtain

$$a^* = 0.1636 \pm 10^{-4}$$

We then assume that for a value of $a < 0.1636$, the returns generated with the Cont-Bouchaud model display an excess of kurtosis and then describe a fat tailed distribution whereas for a value $0.1636 < a \leq 0.50$, the returns display a lack of kurtosis. The returns generated with a value $a^* = 0.1636$ should give follow a distribution with, as the Gaussian distribution, a kurtosis of 3.

4.4.2 The distribution of returns for $a < 0.1$

Looking at Figure 11, we clearly see that for a small value of the probability a ($a \leq 0.1$), the distribution presents an excess of kurtosis (fat tails with much more values very close to the mean than in a Normal distribution) and can be recognized as an exponential distribution or a power law.

In Figure 15 we thus plotted on a log-log scale graph the distributions defined by the returns generated by the Cont-Bouchaud model with a value of the activity probability a equal to 0.0001, 0.0005, 0.001, 0.005, 0.01 and 0.05. Assuming the distribution is a power law, we find its exponent from the gradient of a straight line fit on a log-log plot. In Table 3, we show the slopes found for each activity probability a. We can note that, for all probabilities, we always find an exponent of $2.5 \pm 5 \times 10^{-2}$, precisely contained between 2.0 and 3.0, which seems to be the value usually found in the literature [22].

a	0.0001	0.0005	0.001	0.005	0.01	0.05
slope	-2.4680	-2.4952	-2.5290	-2.5484	-2.4001	-2.1960

Table 3: Slope of the power curves

Moreover, we know from the literature [26], that the returns for a small a follows an exponential distribution if $p < p_c$ whereas it follow a power law if $p = p_c$.

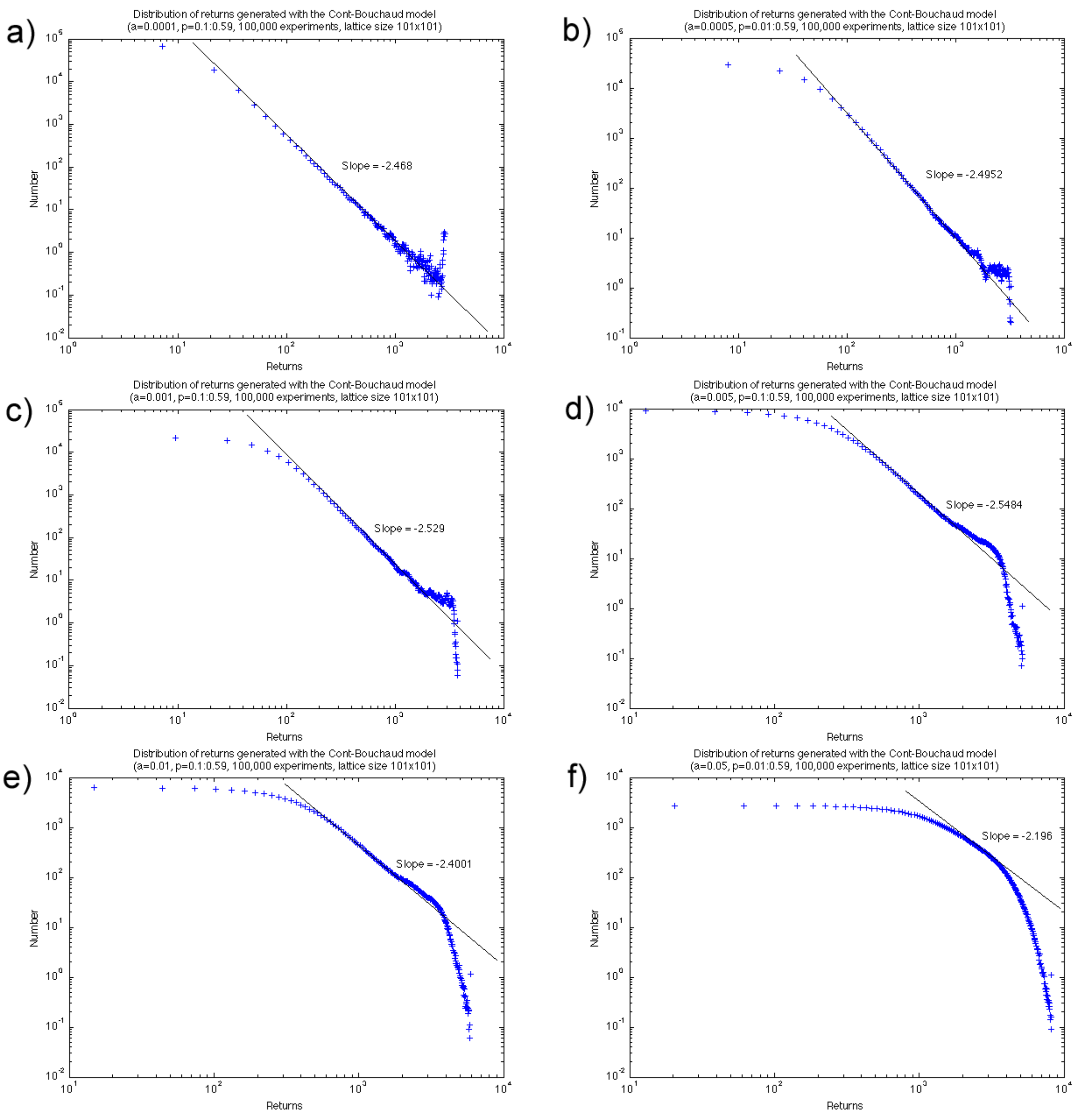
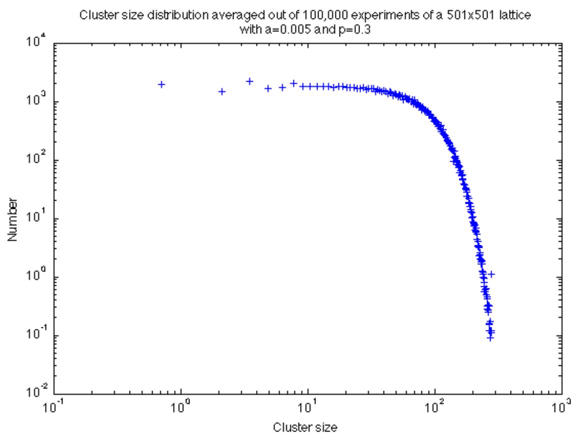


Figure 15: Distribution of returns generated with the Cont-Bouchaud model for different values of the activity probability a (from top-left to bottom-right: 0.0001, 0.0005, 0.001, 0.005, 0.01 and 0.05). The generated clusters have been summed up from all the range of occupancy probabilities p between 0.01 and 0.59. An average out of 100 experiments of 100,000 steps each on 101x101 lattices have been computed.



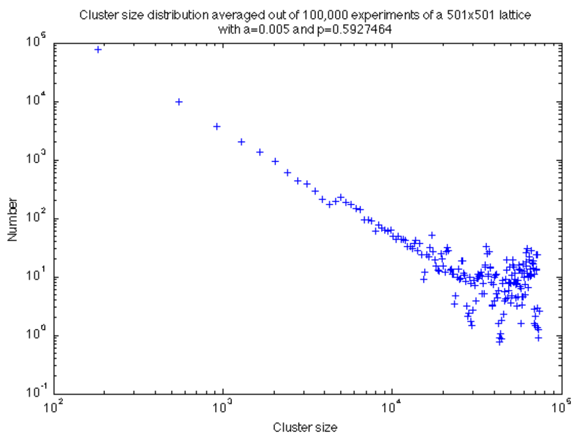


Figure 16: Comparison of the returns distribution when $p < p_c$ (left) vs. $p = p_c$ (right). The other parameters are the same in both cases (100,000 experiments, lattice size 501x501 and $a = 0.005$).

To verify this, in Figure 16, we plot two experimental distributions both generated from a Cont-Bouchaud model taken on a 501x501 lattice with an activity probability $a = 0.005$ but one (left) for a probability $p = 0.3 < p_c$ and the other one (right) $p_c = 0.5927464$ [25]. As discussed in [24], we clearly see the difference in the distribution: while for a value $p = p_c$ the distribution is a power law, for a value $p < p_c$ it is an exponential distribution.

4.5 Results

As we have seen in this section, the Cont-Bouchaud model is capable of generating time-series that look like actual asset prices time series. Looking closer at the statistics of the model, we note that, by changing the value of the parameters, it can generate returns from both a Gaussian distribution and a fat-tailed distribution. A distribution generate from the model with a large value of the activity probability (close to 0.5) will have a lack of kurtosis while a small a gives fat tails (excess of kurtosis). For a small a, we also find that the distribution is a power law. For a specific value $a^* = 0.1636$, the distribution has a kurtosis of 3, perfectly characterizing a Gaussian distribution.

4.6 Extensions

The Cont-Bouchaud model represents the basis of percolation theory applied to financial modeling. However, many extensions to this model have been discussed since then. Some extensions focus on changing the relationship between the price change and the difference between supply and demand. For example, in [7] Chakraborti considers the relative price change to be proportional to the "relative" difference of demand and supply. Furthermore, in [8], the authors assume different assumptions about the probabilities a and p. An interesting approach [26] is also to allow the activity probability a to vary in function of the price level (changing it proportionally to the last price change) to allow the activity reflecting the behaviour of traders.

In this project, time constraints meant we were not able to discuss them all, but we made their further hypothetical implementation as easy as possible. Since the basis of these models are the same as the Cont-Bouchaud model, we can reutilize al the code written for the purpose of this project and easily insert some extensions to it in the *Model* class (see next section for a presentation of the class design).

5 Implementation: the class design

This section describes the Java implementation of the models and analysis code.
For the purpose of this project we decided to implement the model in Java. The Java language offers a good object-oriented abstraction and makes it easy to represent entities by classes and objects. Also, computation is fast.

For representing percolation models, we first needed an abstract type defining the lattice. Then, we also needed an abstract type to represent each site of the lattice, as a cell.
Given that, we then had to design a **Cell** class and a **Lattice** class capable of generating **Cell** and **Lattice** objects respectively. Basically, a **Lattice** is made of L^2 **Cell**s (L being the size of an edge of the lattice). A **Cell** has some attributes being its coordinates (i, j) on the lattice, its status (empty or occupied) and its cluster label. It is important to notice that when it is first created the **Cell** is defined by default as empty and, since it does not belong to any cluster yet, its cluster value is set as -1 by default.
Similarly, a **Lattice** has some particular attributes which are its size N (defined as the length of the edge of the lattice) and the probability of each cell composing the lattice to be occupied.
Then, since the main purpose of lattices in the Cont-Bouchaud model is to generate clusters formed by groups of neighbor cells, we have to be able to retrieve cluster sizes from a populated lattice. We thus need a **Clusters** class that, given a populated **Lattice** made of **Cell**s, searches for clusters in the lattice and then generates a set of cluster sizes. The algorithm implemented to search for clusters is the *Hoshen-Kopelman* algorithm we discussed in the first section.
Given a **Clusters** object defining a set of cluster sizes, we can eventually define the Cont-Bouchaud model through the creation of a **ContBouchaud** class. The **ContBouchaud** class allows us to create **ContBouchaud** objects that compute a one-step price return based on the Cont-Bouchaud model definition, given a cluster set.
Then, it seems logical to create a **Model** class in which we can define and implement models

based on the Cont-Bouchaud basic model (from the classic Cont-Bouchaud model to further extensions of this model but nevertheless based on it). Through the **Model** class, we can define different time steps from a Cont-Bouchaud model for instance and generate returns for a given number of experiments.

Finally, in a **Functions** class, we integrate all the functions that we have been using through the project in order to perform some simulations or test the model.

The entry point to the code is through a **Project** class containing the *main* method where we can create **Model** objects and perform simulations on them.

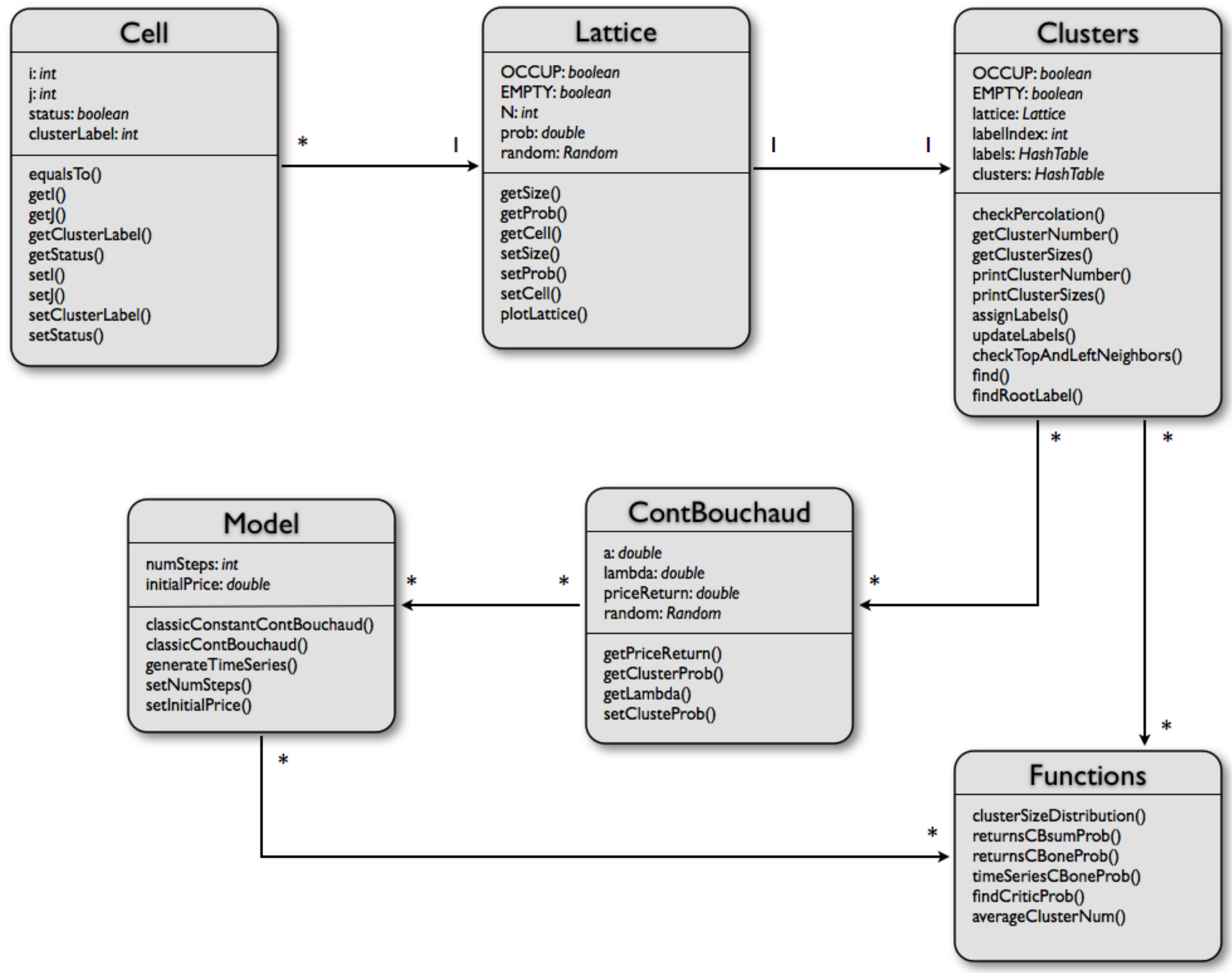


Figure 17: Class diagram

6 Conclusion

In this project we presented and implemented the Cont-Bouchaud model for financial modeling based on percolation theory.

The model is capable of generating time-series that look like actual asset prices time series. The distribution of returns generated with the model can be both Gaussian (assuming a large activity) or be a power law (assuming a small activity). The results we find from both pure percolation theory and the Cont-Bouchaud model are in accordance with those found in the literature insuring the correctness of our program.

Overall, the program allows a fast computation and offers a simple basis for developing various extensions to the model on. Some extensions, for example, would include models capable of interpreting the behaviour of traders allowing the activity to change in function of the price level.

The Cont-Bouchaud model, unless most of agent-based financial models, gives a simple representation of the market and only requires a few parameters. However, focusing on the agents instead of the price itself (such as a geometric Brownian motion for representing price fluctuations) implies having to model the whole market in order to generate a change in price. This technique, even if closer to reality and less abstract than pure price-based model, is slower.

Appendix

A Java code

In this appendix, we list all the Java code written in order to implement the Cont-Bouchaud model for the purpose of this project. The code can be downloaded from my website:

http://www.ddeville.me

A.1 Cell.java

```java
/**
 * Defines Cell objects that are formed by a pair of numbers representing
 * the coordinates of a cell in the lattice
 * @author Damien Deville
 *
 */
public class Cell
{
   private int i ;             // The i-coordinate of the cell
   private int j ;             // The j-coordinate of the cell
   private boolean status ;    // If the cell is defined as occupied (true) or empty (false)
   private int clusterLabel ; // The label of the cluster that the cell belongs to



   /**
    * Constructor definition
    * @param i: The i-coordinate of the cell
    * @param j: The j-coordinate of the cell
    * @param status: the status of the cell (true = occupied, false = empty)
    */
   public Cell(int i, int j, boolean status)
   {
      setI(i) ;
      setJ(j) ;
      setStatus(status) ;
      setClusterLabel(-1) ;    // Initially, the cell belongs to no cluster
   }

   /**
    * Constructor definition: if the user does not want to specify the status,
     * we define the cell as empty
     * @param i: The i-coordinate of the cell
     * @param j: The j-coordinate of the cell
     */
    public Cell(int i, int j)
    {
        setI(i) ;
        setJ(j) ;
        setStatus(false) ;
        setClusterLabel(-1) ;
    }

    /**
     * Returns true if two coordinates are equal, otherwise returns false
     * @param cell: the Cell object we want to compare it to
     * @return true if the two objects are equal, otherwise false
     */
    public boolean equalsTo(Cell cell)
    {
        if(i == cell.getI() && j == cell.getJ())
            return true ;
        else
            return false ;
    }

    /**
     * Get the i-coordinate of the cell
     * @return the i-coordinate of the cell
     */
    public int getI()
    {
        return i ;
    }

    /**
     * Get the j-coordinate of the cell
     * @return the j-coordinate of the cell
     */
    public int getJ()
    {
        return j ;
    }

    /**
     * Get the status of the cell (occupied or empty)
     * @return the status of the cell (occupied = true or empty = false)
     */
    public boolean getStatus()
    {
       return status ;
    }

    /**
     * Get the label of the cluster that the cell belongs to
     * @return the label of the cluster that the cell belongs to
     */
    public int getClusterLabel()
    {
       return clusterLabel ;
    }

    /**
     * Set the i-coordinate of the cell
     * @param i: the i-coordinate of the cell
     */
    private void setI(int i)
    {
       this.i = i ;
    }

    /**
     * Set the j-coordinate of the cell
     * @param j: the j-coordinate of the cell
     */
    private void setJ(int j)
    {
       this.j = j ;
    }

    /**
     * Set the status of the cell (occupied or empty)
     * @param status: the status of the cell (true = occupied, false = empty)
     */
    public void setStatus(boolean status)
    {
       this.status = status ;
```

```java
    }

    /**
     * Set the label of the cluster that the cell belongs to
     * @param clusterLabel: the label of the cluster that the cell belongs to
     */
    public void setClusterLabel(int clusterLabel)
    {
        this.clusterLabel = clusterLabel ;
    }
}
```

Listing 6: Cell.java

A.2 Lattice.java

```java
import java.util.Random;

/**
 * Defines lattice objects
 * @author Damien Deville
 *
 */
public class Lattice
{
    private final boolean OCCUP = true ;   // defines an occupied cell
    private final boolean EMPTY = false ;  // defines a non-occupied cell
    private int N ;                        // size of the lattice
    private double prob ;                  // probability of an occupied node
    private Cell[][] lattice ;             // array that represents the lattice
    private Random random ;                // random seed generator


    /**
     * Constructor definition, creates an empty lattice of size N
     * All the sites are defined like empty
     * @param N : the lattice size
     */
    public Lattice(int N)
    {
        setSize(N) ;   // we set the lattice size
        lattice = new Cell[N][N] ; // we create the actual lattice
        for(int i = 0 ; i < N ; i++)  // we populate it given the probability
        {
            for(int j = 0 ; j < N ; j++)
            {
                lattice[i][j] = new Cell(i,j,EMPTY) ;
            }
        }
    }

    /**
     * Constructor definition, creates a lattice of size N populated given the probability p.
     * If the randomly generated number is greater than p, we set the given cell of the
     * lattice to be occupied, otherwise we set it as empty
     * @param N: the lattice size
     * @param prob: the probability of each node being occupied
     */
    public Lattice(int N, double prob)
    {
        random = new Random() ; // we define a random object
        setSize(N) ;    // we set the lattice size
        setProb(prob) ;    // we set the probability
        lattice = new Cell[N][N] ; // we create the actual lattice
        for(int i = 0 ; i < N ; i++)  // we populate it given the probability
        {
            for(int j = 0 ; j < N ; j++)
            {
                if(random.nextDouble() <= prob)  // we set the cell to be occupied
                    lattice[i][j] = new Cell(i,j,OCCUP) ;
                else                          // we set the cell to be empty
                    lattice[i][j] = new Cell(i,j,EMPTY) ;
            }
        }
    }

    /**
     * Constructor definition, creates a lattice of size N, populated given the probability p
     * for a particular random seed. If the randomly generated number is greater than p, we
     * set the given cell of the lattice to be occupied, otherwise we set it as empty
     * @param N: the lattice size
     * @param prob: the probability of each node being occupied
     * @param seed: a particular seed for the random generator
     */
    public Lattice(int N, double prob, long seed)
    {
        random = new Random(seed) ;    // we define a random object given the particular seed
        setSize(N) ;    // we set the lattice size
        setProb(prob) ;    // we set the probability
        lattice = new Cell[N][N] ; // we create the actual lattice
        for(int i = 0 ; i < N ; i++)  // we populate it given the probability
        {
            for(int j = 0 ; j < N ; j++)
            {
                if(random.nextDouble() <= prob)  // we set the cell to be occupied
                    lattice[i][j] = new Cell(i,j,OCCUP) ;
                else                        // we set the cell to be empty
                    lattice[i][j] = new Cell(i,j,EMPTY) ;
            }
        }
    }



    /**
     * Get the lattice size
     * @return the lattice size
     */
    public int getSize()
    {
        return N ;
    }

    /**
     * Get the probability
     * @return the probability
     */
    public double getProb()
    {
        return prob ;
    }

    /**
     * Get the cell at the i,j coordinates in the lattice
     * @param i: the i-coordinate in the lattice
     * @param j: the j-coordinate in the lattice
     * @return the cell corresponding to these coordinates
     */
    public Cell getCell(int i, int j)
    {
        return lattice[i][j] ;
    }

    /**
     * Set the lattice size
     * @param N: the lattice size
     */
    private void setSize(int N)
    {
        this.N = N ;
    }

    /**
     * Set the probability
     * @param p: the probability
     */
    private void setProb(double prob)
    {
        this.prob = prob ;
    }

    /**
     * Set a cell to the defined position in the lattice
     * @param cell: the cell we want to set
     */
    public void setCell(Cell cell)
    {
        int i = cell.getI() ;
        int j = cell.getJ() ;
        lattice[i][j] = cell ;
    }

    /**
     * Loop through the lattice and plot it. Plot a "*" if the cell is occupied and
     * an empty space if it's empty
     */
    public void plotLattice()
    {
        for(int i = 0 ; i < N ; i++)
        {
            for(int j = 0 ; j < N ; j++)
            {
                if(lattice[i][j].getStatus() == OCCUP)
                    System.out.print("*") ;     // if the cell is occupied, we draw a *
                else
                    System.out.print(" ") ;     // if the cell is empty, we draw an empty space
            }
            System.out.println("") ;
        }
    }
}
```

Listing 7: Lattice.java

A.3 Clusters.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Enumeration ;
import java.util.Hashtable ;

/**
 * Defines clusters objects representing the clusters in
 * a populated lattice
 * @author Damien Deville
 *
 */
public class Clusters
{
    private final boolean OCCUP = true ;    // defines an occupied cell
    private final boolean EMPTY = false ;   // defines an empty cell
    private Lattice lattice ;               // defines a lattice
    // contains the various labels for the clusters
    private Hashtable<Integer,Integer> labels ;
    // contains the number of cells for each cluster
    private Hashtable<Integer,Integer> clusters ;
    private int labelIndex ;                // value of the first label

    /**
     * Constructor definition
     * @param lattice: the lattice we are analysing clusters on
     */
    public Clusters(Lattice lat)
```

```java
    {
        lattice = lat ;
        labelIndex = 1 ;  // we set the value of the first label to be 1
        // we create a new hashtable for the cluster labels
        labels = new Hashtable<Integer,Integer>() ;
        // we create a new hashtable for the cluster numbers
        clusters = new Hashtable<Integer,Integer>() ;
        // we first assign raw cluster labels to the occupied cells
        assignLabels() ;
        // we then link the clusters using the root label for each linked cluster
        updateLabels() ;
    }


    /**
     * Get the cluster number in the lattice
     * @return the cluster number in the lattice
     */
    public int getClusterNumber()
    {
        return clusters.size() ;   // the cluster number
    }


    /**
     * Get an array populated with the cluster sizes in the lattice
     * @return an array of cluster sizes
     */
    public Integer[] getClusterSizes()
    {
        Collection<Integer> c = clusters.values() ;
        Integer[] clusterSizesArray = (Integer[])c.toArray(new Integer[c.size()]) ;
        return clusterSizesArray ;
    }


    /**
     * Print the cluster number
     */
    public void printClusterNumber()
    {
        System.out.println(clusters.size()) ;
    }


    /**
     * Print the size of each cluster in the lattice
     */
    public void printClusterSizes()
    {
        Enumeration<Integer> k = clusters.keys() ;
        while (k.hasMoreElements())
        {
            int key = (int)k.nextElement() ;
            System.out.println((int)clusters.get(key));
        }
    }


    /**
     * Span the lattice once and assign cluster labels to each occupied cell.
     * However, some cluster can still be defined by more than one label.
     */
    private void assignLabels()
    {
        for(int i = 0 ; i < lattice.getSize() ; i++) // we span the lattice
        {
            for(int j = 0 ; j < lattice.getSize() ; j++)
            {
                Cell cell = lattice.getCell(i,j) ;  // we get the current cell in the lattice
                if(cell.getStatus() == OCCUP)     // we check if the cell is occupied
                    checkTopAndLeftNeighbors(cell) ; // we check its top and left neighbors
            }                                   // to assign the right label to the cell
        }
    }


    /**
     * Span the lattice once and update the labels for the clusters that are still
     * defined by more than one label.
     * For each label, it checks if it is the root one and if not, it looks for the root one.
     */
    private void updateLabels()
    {
        for(int i = 0 ; i < lattice.getSize() ; i++) // we span the lattice
        {
         for(int j = 0 ; j < lattice.getSize() ; j++)
         {
            Cell cell = lattice.getCell(i,j) ;  // we get the current cell in the lattice
            if(cell.getStatus() == OCCUP)     // we check if the cell is occupied
            {
               // we find the root label corresponding to the cell
               int label = findRootLabel(cell) ;
               cell.setClusterLabel(label) ;     // we set this new label to the cell
               // if the clusters hashtable does not already contain this label
               if(!clusters.containsKey(label))
                  clusters.put(label, 1) ;       // we add it to it
               else  // if it already contains it, we increment its size by 1
                  clusters.put(label, clusters.get(label) + 1) ;
            }
         }
      }
   }


   /**
    * For each occupied cell, check the cells on top and left of the current cell.
    * There are 4 cases:
    * 1: both cells are empty, we then have to create a new cluster label and apply it to
         the current cell
    * 2: only one cell is empty, we then have to apply the label of the cluster the
         non-empty cell belongs to to the current cell
    * 3: both cells are occupied and have the same cluster label: we then apply this cluster
         label to the current cell
    * 4: both cells are occupied but have different cluster labels: we then apply the
         smallest label to the current cell and define a union between the labels
    * @param cell: the current cell we are checking
    */
   private void checkTopAndLeftNeighbors(Cell cell)
   {
      Cell cell_top ;       // cell at the top of the current cell
      Cell cell_left ;  // cell at the left of the current cell

      // we make sure that cell_top is actually not outside of the lattice
      if(cell.getI() > 0)      // if it is actually INSIDE the lattice
         cell_top = lattice.getCell(cell.getI()-1, cell.getJ()) ; // we get the actual cell
      else                 // if it is actually OUTSIDE the lattice
         // we define it as an empty cell with coordinates (-1,-1)
         cell_top = new Cell(-1, -1, EMPTY) ;
```

```java

        // we make sure that cell_left is actually not outside of the lattice
        if(cell.getJ() > 0)      // if it is actually INSIDE the lattice
           cell_left = lattice.getCell(cell.getI(), cell.getJ()-1) ; // we get the actual cell
        else               // if it is actually OUTSIDE the lattice
           // we define is as an empty cell with coordinates (-1,-1)
           cell_left = new Cell(-1, -1, EMPTY) ;

        // we get the cluster label to which the cell at the top belongs
        int topClusterLabel = cell_top.getClusterLabel() ;
        // we get the cluster label to which the cell at the left belongs
        int leftClusterLabel = cell_left.getClusterLabel() ;

        // 1st case:
        // if both cell_top and cell_left are empty, we have to create a new cluster label
        if(cell_top.getStatus() == EMPTY && cell_left.getStatus() == EMPTY)
        {
           cell.setClusterLabel(labelIndex) ;      // we set the cluster label to be labelIndex
           // we add this cluster label to the labels hashtable. Note that the key
           // and the value is the same since this label is not linked to another one yet
           labels.put(labelIndex, labelIndex) ;
           labelIndex++ ;     // we increment the labelIndex by 1
        }

        // 2nd case:
        // if only one of both cell is empty, we have to set the cell cluster label same as
        // the non-empty cell's one
        else if(cell_top.getStatus() == EMPTY)     // if cell_top is empty
           // we find the root label and set it as cell_left's label
           cell.setClusterLabel(find(leftClusterLabel)) ;
        else if(cell_left.getStatus() == EMPTY)       // if cell_left is empty
           // we find the root label and set it as cell_top's label
           cell.setClusterLabel(find(topClusterLabel)) ;

        // 3rd case:
        // if both cell cluster labels are equal, we set the cell cluster label same as
        // this cluster label
        else if(topClusterLabel == leftClusterLabel)
           // we find the root label and set it as cell_top and cell_left's label
           cell.setClusterLabel(find(topClusterLabel)) ;

        // 4th case:
        // if cell_top and cell_left belong to different clusters, we set the current cell's
        // label as the smallest label
        // we also have to update the labels relation in the labels hashtable
        // in the label hashtable, if one cluster is defined by only one label, the key entry
        // (label) will be the same as its value
        // if one cluster is defined by more than one label, say 2, the key entry will be the
        // biggest label and the value the smallest
        else
        {
            // we look for the smallest label between both
            int smallLabel=(topClusterLabel<leftClusterLabel)?topClusterLabel:leftClusterLabel ;
            // we look for the biggest label between both
            int bigLabel=(topClusterLabel<leftClusterLabel)?leftClusterLabel:topClusterLabel ;
            // we set the smallest label for the current cell
            cell.setClusterLabel(smallLabel) ;

            // if the smallest label is actually smaller than the value corresponding
            // to the biggestLabel key
            if(smallLabel < labels.get(bigLabel))
                // we update the key and label to materialize the link
                labels.put(find(labels.get(bigLabel)), find(smallLabel)) ;
            // if the smallest label is actually bigger than the value corresponding
            // to the biggestLabel key
            else
                // we update the key and label to materialize the link
                labels.put(find(smallLabel), find(labels.get(bigLabel))) ;
        }
    }


    /**
     * Find the smallest "good" label given a "bad" label
     * @param label: the bad label we want to find the good label for
     * @return the good label
     */
    private int find(int label)
    {
        int initLabel = label ;     // we store the original bad label
        // while the label is not good, we loop through the links
        while(labels.get(initLabel) != initLabel)
            initLabel = labels.get(initLabel) ;
        // the value of initLabel is now the "good" label

        // we loop through the labels from the bad one another time
        while(labels.get(label) != label)
        {
            int temp = labels.get(label) ;   // we get the current label at each loop step
            // we set the value of each current label to the "good" label
            labels.put(label, initLabel) ;
            label = temp ;
        }
        return initLabel ;      // we return the good label
    }


    /**
     * Find the root label for each cluster label from the labels hashtable.
     * The root label is defined to be the smallest label among all labels constituing a
     * common cluster
     * @param cell: the cell we are checking the cluster label for
     * @return the root cluster label
     */
    private int findRootLabel(Cell cell)
    {
        int labelNum = cell.getClusterLabel() ;      // we get the cluster label of the cell
        while(labels.get(labelNum) != labelNum)      // we loop through the labels hashtable
        {                                   // until we reach the root label
            labelNum = labels.get(labelNum) ;
        }
        return labelNum ;                   // we return the root label for this cell
    }


    /**
     * Check if percolation (a top-bottom spanning cluster) occurs in the lattice
     * @return true if there is percolation, false if not
     */
    public boolean checkPercolation()
    {
        // we check if there is the same label on the top and bottom rows of the lattice
        // arraylist that contains all top-row cells
        ArrayList<Integer> top = new ArrayList<Integer>() ;
        // arraylist that contains all bottom-row cells
        ArrayList<Integer> bottom = new ArrayList<Integer>() ;
        // arraylist that contains all left-column cells
        ArrayList<Integer> left = new ArrayList<Integer>() ;
        // arraylist that contains all right-column cells
```

```java
        ArrayList<Integer> right = new ArrayList<Integer>() ;
        // we loop through the columns of the lattice
        for(int i = 0 ; i < lattice.getSize() ; i++)
        {
            // we get each cell from the top row
            Cell topCell = lattice.getCell(0,i) ;
            // we get each cell from the bottom row
            Cell bottomCell = lattice.getCell(lattice.getSize()-1,i) ;
            if(topCell.getStatus() == OCCUP)           // if the top cell is occupied
                top.add(topCell.getClusterLabel()) ;       // we add it to the top arraylist
            if(bottomCell.getStatus() == OCCUP)           // if the bottom cell is occupied
                bottom.add(bottomCell.getClusterLabel()) ;    // we add it to the bottom arraylist
        }
        // we remove from the top arraylist all elements that are not in bottom
        top.retainAll(bottom) ;
        if(top.size() > 0)          // we check if there are still elements in the arraylist
            return true ;
        // if there is not a spanning cluster from top to bottom,
        // we check if there is one from left side to right side
        // we loop through the rows of the lattice
        for(int j = 0 ; j < lattice.getSize() ; j++)
        {
            // we get each cell from the left column
            Cell leftCell = lattice.getCell(j,0) ;
            // we get each cell from the right column
            Cell rightCell = lattice.getCell(j,lattice.getSize()-1) ;
            if(leftCell.getStatus() == OCCUP)              // if the left cell is occupied
                left.add(leftCell.getClusterLabel()) ;     // we add it to the left arraylist
            if(rightCell.getStatus() == OCCUP)             // if the right cell is occupied
                right.add(rightCell.getClusterLabel()) ;   // we add it to the right arraylist
        }
        // we remove from the left arraylist all elements that are not in right
        left.retainAll(right) ;
        if(left.size() > 0)         // we check if there are still elements in the arraylist
            return true ;
        return false ;     // if there is no spanning cluster, we eventually return false
    }
}
```

Listing 8: Clusters.java

A.4 ContBouchaud.java

```java
import java.util.Random;

/**
 * Create a ContBouchaud objects representing as the return
 * for one time step computed from the Cont-Boucaud model
 * @author Damien Deville
 *
 */
public class ContBouchaud
{
   private Random random ;
   private double a ;            // the probability that a cluster be active (buys or sells)
   private double lambda ;       // the scaling factor
   private double priceReturn ;  // the price change given by the model

   /**
    * Constructor definition.
    * Compute the price return with the Cont-Bouchaud model. First populate a lattice , find
    * the cluster sizes and then compute the price return.
    * @param latticeSize: the size of the lattice
    * @param prob: the probability that a site in the lattice be occupied
    * @param a: the probability that a cluster be active (buys or sells)
    * @param lambda: the scaling factor in the Cont-Bouchaud model
    */
   public ContBouchaud(int latticeSize , double prob, double a, double lambda)
   {
      setProb(a) ;                    // we set the active cluster probability
      setLambda(lambda) ;                // we set the scaling factor

      double sumBuy = 0 ;      // we assume the sum of buying clusters is initially zero
      double sumSell = 0 ; // we assume the sum of selling clusters is initially zero
      random = new Random() ; // we define a new random object

      // we create a new lattice given the parameters
      Lattice lat = new Lattice(latticeSize , prob) ;
      // we check the clusters in this lattice
      Clusters clusters = new Clusters(lat) ;
      // we get the size of each cluster
      Integer[] clusterSizes = clusters.getClusterSizes() ;

      for(int i = 0 ; i < clusterSizes.length ; i++)  // we loop through the clusters
```

```
        {
            // we get a new uniformly generated random number
            double rnd = random.nextDouble() ;
            if(rnd < a) // if the random number is less than a (probability a)
                sumSell += clusterSizes[i] ;  // we assume the cluster is selling
            else if(rnd > 1-a)   // if the random number is greater than 1-a (probability a)
                sumBuy += clusterSizes[i] ;       // we assume the cluster is buying
            // else the cluster sleeps
        }
        // the price return is then given by the scaled difference
        priceReturn = (sumBuy-sumSell)/lambda ;
    }

    /**
     * Constructor definition.
     * Compute the price return using the Cont-Bouchaud model assuming we already have an
     * array containing the cluster sizes. This constructor is useful when we want to
     * compute various time steps from the same lattice pattern (for computation efficiency).
     * @param latticeSize: the size of the lattice
     * @param prob: the probability that a site in the lattice be occupied
     * @param a: the probability that a cluster be active (buys or sells)
     * @param lambda: the scaling factor in the Cont-Bouchaud model
     */
    public ContBouchaud(double a, double lambda, Integer[] clusterSizes)
    {
        setProb(a) ;                    // we set the active cluster probability
        setLambda(lambda) ;                 // we set the scaling factor

        double sumBuy = 0 ;      // we assume the sum of buying clusters is initially zero
        double sumSell = 0 ; // we assume the sum of selling clusters is initially zero
        random = new Random() ; // we define a new random object

        for(int i = 0 ; i < clusterSizes.length ; i++)  // we loop through the clusters
        {
            // we get a new uniformly generated random number
            double rnd = random.nextDouble() ;
            if(rnd < a) // if the random number is less than a (probability a)
                sumSell += clusterSizes[i] ;  // we assume the cluster is selling
            else if(rnd > 1-a)   // if the random number is greater than 1-a (probability a)
                sumBuy += clusterSizes[i] ;       // we assume the cluster is buying
            // else the cluster sleeps
        }
        // the price return is then given by the scaled difference
        priceReturn = (sumBuy-sumSell)/lambda ;
    }


    /**
     * Get the price return computed with the Cont-Bouchaud model
     * @return the price return
     */
    public double getPriceReturn()
    {
        return priceReturn ;
    }

    /**
     * Set the probability that a cluster be active (buys or sells)
     * @param a: the probability that a site is active (buys or sells)
     */
    private void setProb(double a)
    {
        this.a = a ;
    }

    /**
     * Set the scaling factor defined in the Cont-Bouchaud model
     * @param lambda: the scaling factor
     */
    private void setLambda(double lambda)
    {
        this.lambda = lambda ;
    }

    /**
     * Get the probability that a cluster be active (buys or sells)
     * @return the probability that a cluster is active (buys or sells)
     */
    public double getClusterProb()
    {
        return a ;
    }

    /**
     * Get the scaling factor defined in the Cont-Bouchaud model
     * @return the scaling factor
     */
    public double getLambda()
    {
        return lambda ;
    }
}
```

Listing 9: ContBouchaud.java

A.5 Model.java

```
/**
 * Implement particular models from the Cont−Bouchaud setting
 * @author Damien Deville
 *
 */
public class Model
{
    private int numSteps ;          // the number of time steps
    private double initialPrice ; // the initial asset price

    /**
     * Constructor definitio
     * @param numSteps: the number of time steps in the model
     * @param initialPrice: the initial asset price
     */
    public Model(int numSteps, double initialPrice)
    {
        setNumSteps(numSteps) ;         // we set the number of steps
        setInitialPrice(initialPrice) ;  // we set the initial asset price
    }

    /**
     * Set the number of time steps
     * @param numSteps: the number of time steps
     */
    public void setNumSteps(int numSteps)
    {
        this.numSteps = numSteps ;
    }

    /**
```

```
     * Set the initial asset price
     * @param initialPrice: the initial asset price
     */
    public void setInitialPrice(double initialPrice)
    {
       this.initialPrice = initialPrice ;
    }

    /**
     * Generate returns from the Cont-Bouchaud model assuming the
     * lattice does not change at each time step (more efficient)
     * and that we already have this lattice (inputed in the method)
     * @param a: the activity probability
     * @param lambda: the price scaling factor
     * @param clusterSizes: array containing cluster sizes
     * @return array containing returns from the model
     */
    public double[] classicConstantContBouchaud(double a,double lambda,Integer[] clusterSizes)
    {
       double[] returns = new double[numSteps] ; // we define an array to contain the returns
       ContBouchaud priceReturn ; // ContBouchaud object that models one return from the model
       for(int i = 0 ; i < numSteps ; i++) // we loop as many times as the number of steps
       {
          // get a price return from the Cont-Bouchaud model
          priceReturn = new ContBouchaud(a, lambda, clusterSizes) ;
          returns[i] = priceReturn.getPriceReturn() ;
       }
       return returns ;
    }

    /**
     * Generate returns from the Cont-Bouchaud model assuming the
     * lattice does not change at each time step (more efficient)
     * @param latticeSize: the size of the lattice
     * @param prob: the occupancy probability
     * @param a: the activity probability
     * @param lambda: the price scaling factor
     * @return array containing returns from the model
     */
    public double[] classicConstantContBouchaud(int latticeSize, double prob, double a, ...
                                                ... double lambda)
    {
       Lattice lat = new Lattice(latticeSize, prob) ;  // define a new lattice
```

```
        Clusters clusters = new Clusters(lat) ;    // we check the clusters in this lattice
        Integer[] clusterSizes = clusters.getClusterSizes() ; // we get the size of clusters

        double[] returns = new double[numSteps] ; // we define an array to contain the returns
        ContBouchaud priceReturn ; // ContBouchaud object that models one return from the model
        for(int i = 0 ; i < numSteps ; i++) // we loop as many times as the number of steps
        {
            // get a price return from the Cont-Bouchaud model
            priceReturn = new ContBouchaud(a, lambda, clusterSizes) ;
            returns[i] = priceReturn.getPriceReturn() ;
        }
        return returns ;
    }

    /**
     * Generate returns from the Cont-Bouchaud model assuming the
     * lattice changes at each time step
     * @param latticeSize
     * @param prob
     * @param a
     * @param lambda
     * @return
     */
    public double[] classicContBouchaud(int latticeSize, double prob, double a, double lambda)
    {
        double[] returns = new double[numSteps] ; // we define an array to contain the returns
        ContBouchaud priceReturn ; // ContBouchaud object that models one return from the model
        for(int i = 0 ; i < numSteps ; i++) // we loop as many times as the number of steps
        {
            // get a price return from the Cont-Bouchaud model
            priceReturn = new ContBouchaud(latticeSize, prob, a, lambda) ;
            returns[i] = priceReturn.getPriceReturn() ;
        }
        return returns ;
    }

    /**
     * Generate the price time series from an initial price based on the returns
     * @param initialPrice: the initial price
     * @return the price time series
     */
    public double[] generateTimeSeries(double[] priceReturns)
    {
```

```java
        double[] timeSeries = new double[numSteps] ; // we define an array to contain prices
        timeSeries[0] = initialPrice ;   // the first value is the initial price
        for(int i = 0 ; i < numSteps-1 ; i++)  // we loop as many times as the number of steps
            timeSeries[i+1] = timeSeries[i] + priceReturns[i] ;   // generate prices
        return timeSeries ;
    }
}
```

Listing 10: Model.java

A.6 Functions.java

```java
import java.util.Enumeration;
import java.util.Hashtable;

/**
 * Defines functions used in order to get some results
 * from the model
 * @author Damien Deville
 *
 */
public class Functions
{

    /**
     * Constructor definition
     */
    public Functions()
    {

    }

    /**
     * Get the cluster size distribution for a given probability p
     * print out frequency of each cluster size in the lattice
     * @param p: the probability p
     * @param latticeSize: the lattice size
     * @param numExp: number of experiments
     */
    public void clusterSizeDistribution(double p, int latticeSize , int numExp)
    {
        // HashTable that will contain the cluster size as a key and
        // the frequency as the corresponding value
        Hashtable<Integer,Double> nums = new Hashtable<Integer,Double>() ;
        // we loop as many times as number of experiments
        for(int n = 0 ; n < numExp ; n++)
        {
            Lattice lat = new Lattice(latticeSize,p) ;    // create new lattice
            Clusters clusters = new Clusters(lat) ;    // create new clusters
            Integer[] sizes = clusters.getClusterSizes() ;  // get cluster sizes
            // loop as many times as the sizes of cluster sizes array
            for(int i = 0 ; i < sizes.length ; i++)
            {
                // if the cluster size already exists as a key in the hashtable
                if(nums.containsKey(sizes[i]))
                    // we increment the value (frequency) by 1
                    nums.put(sizes[i], nums.get(sizes[i])+1) ;
                else  // if it does not exist
                    nums.put(sizes[i], 1.0) ;  // we create a new hashtable entry
            }
        }
        // we then print out the averaged frequency for each cluster size
        Enumeration<Integer> k = nums.keys() ;
        while (k.hasMoreElements())
        {
            int key = (int)k.nextElement() ;
            double value = (double)nums.get(key) ;
            System.out.println(key + " " + value/(double)numExp);
        }
    }


    /**
     * Compute returns from the Cont-Bouchaud model given a probability a and
     * summing over all clusters sizes for all probabilities p (step 0.01)
     * between 0.01 and 0.59. Write down the returns in a text file
     * @param a: the probability a
     * @param latticeSize: the lattice size
     * @param numSteps: number of time steps
     * @param numExp: number of experiments
     */
    public void returnsCBsumProb(double a, int latticeSize, int numSteps, int numExp)
    {
        String s = "output" ;
        String ext = ".txt" ;
        for(int j = 1 ; j <= numExp ; j++)  // loop as many times as number of experiments
        {
            String output = s + j + ext ; // create txt file name
            System.out.println(output) ;
            Integer[] originalClusterSizes = new Integer[0] ;
            for(int n = 1 ; n <= 59 ; n++)   // loop through all probabilities p
            {
                double proba = (double)n/100 ;    // define the probability p
                Lattice lat = new Lattice(latticeSize , proba) ; // create a new lattice
                Clusters clusters = new Clusters(lat) ;    // new clusters
                Integer[] clusterSizes = clusters.getClusterSizes() ; // get all cluster sizes
                // create a new array containing previous and new cluster sizes
                Integer[] newClusterSizes = new Integer[originalClusterSizes.length + ...
                                                        ... clusterSizes.length] ;
                // we add to it all previous clusters sizes
                for(int i = 0 ; i < originalClusterSizes.length ; i++)
                    newClusterSizes[i] = originalClusterSizes[i] ;
                // we add to it all new clusters sizes
                for(int i = 0 ; i < clusterSizes.length ; i++)
                    newClusterSizes[i+originalClusterSizes.length] = clusterSizes[i] ;
                originalClusterSizes = newClusterSizes ;
            }
            // given the new list of cluster sizes, we get the returns from the model
            Model model = new Model(numSteps, 1) ;
            double[] priceReturns=model.classicConstantContBouchaud(a,1,originalClusterSizes) ;
            FileOutput out = new FileOutput(output) ; // create txt file
            // for each time step, we write the result as a new line in the txt file
            for(int i = 0 ; i < priceReturns.length ; i++)
            {
                out.writeDouble(priceReturns[i]) ;
                 out.writeNewline() ;
            }
            out.close() ;  // close the text file
        }
    }


    /**
     * Compute returns from the Cont-Bouchaud model given a probability a and
     * a probability p. Write down the returns in a text file
     * @param p: the probability p
     * @param a: the probability a
     * @param latticeSize: the lattice size
     * @param numSteps: the number of time steps
     * @param numExp: the number of experiments we average out
     */
    public void returnsCBoneProb(double p,double a,int latticeSize ,int numSteps,int numExp)
    {
        String s = "output" ;
        String ext = ".txt" ;
        for(int n = 1 ; n <= numExp ; n++)  // loop as many times as number of experiments
        {
            String output = s + n + ext ; // create txt file name
            System.out.println(output) ;
            FileOutput out = new FileOutput(output) ; // create txt file
            Model model = new Model(numSteps, 1) ; // create a new model object
            // we get the returns from the Cont-Bouchaud model
            double[] priceReturns = model.classicConstantContBouchaud(latticeSize , p, a, 1) ;
            // for each time step, we write the result as a new line in the txt file
            for(int i = 0 ; i < numSteps ; i++)
            {
                out.writeDouble(priceReturns[i]) ;
                 out.writeNewline() ;
            }
            out.close() ;  // close the text file
        }
    }

    /**
     * Compute prices from the Cont-Bouchaud model given a probability a and
     * a probability p. Write down the returns in a text file
     * @param p: the probability p
     * @param a: the probability a
     * @param latticeSize: lattice size
     * @param numSteps: number of time steps
     * @param numExp: number of experiments
     * @param initialPrice: the initial asset price
     * @param lambda: the price scaling factor
     */
    public void timeSeriesCBoneProb(double p, double a, int latticeSize , int numSteps, ...
                                      ... int numExp, double initialPrice , double lambda)
    {
        String s = "output" ;
        String ext = ".txt" ;
        for(int n = 1 ; n <= numExp ; n++)  // loop as many times as number of experiments
        {
            String output = s + n + ext ; // create txt file name
            System.out.println(output) ;
            FileOutput out = new FileOutput(output) ; // create txt file
            Model model = new Model(numSteps, initialPrice) ;  // create a new model object
            // we get the returns from the Cont-Bouchaud model
            double[] priceReturns = model.classicConstantContBouchaud(latticeSize,p,a,lambda) ;
            // we generate a time-series from the returns
            double[] timeSeries = model.generateTimeSeries(priceReturns) ;
            // for each time step, we write the result as a new line in the txt file
            for(int i = 0 ; i < numSteps ; i++)
            {
                out.writeDouble(timeSeries[i]) ;
                 out.writeNewline() ;
            }
            out.close() ;  // close the text file
        }
    }


    /**
     * Find the critical probability checking if percolation occurs in each lattice
     * for each p. Average the results out of a number of trials
     * @param latticeSize: the lattice size
     * @param trials: number of trials
     */
    public void findCriticProb(int latticeSize, int trials)
    {
        Lattice lat ; // define new lattice
        double p ;      // probability p
        int probMesh = 100 ; // the probability mesh size
        //long seed = 12318991921L ;
        int[] results = new int[probMesh] ; // array containing results
        int count ;
        for(int i = 1 ; i <= probMesh ; i++)
        {
            p = (double)i/probMesh ;   // defines the probability p
            count = 0 ;
            // loop as many times as trials number
            for(int j = 0 ; j < trials ; j++)
            {
                System.out.println("Trial number: " + j) ;
                lat = new Lattice(latticeSize, p) ; // create a new lattice
                Clusters clusters = new Clusters(lat) ;   // get the clusters
                if(clusters.checkPercolation())  // check if percolation occurs
                    count++ ;   // if percolation occurs, increase count by 1
            }
            results[i-1] = count ;
            System.out.println(p + " " + results[i-1]) ;
        }
    }


    /**
     * Print the average cluster number for each probability p
     * @param latticeSize: the lattice size
     * @param numExp: number of experiments to average out
     */
    public void averageClusterNum(int latticeSize, int numExp)
    {
        for(double i = 1 ; i <= 100 ; i++)
        {
            double prob = i / 100 ; // probability p
            int sum = 0 ;
            for(int j = 0 ; j < numExp ; j++)
            {
                //long seed = 123189914592L ;
                Lattice lat = new Lattice(latticeSize, prob) ;
                Clusters clusters = new Clusters(lat) ;
                sum += clusters.getClusterNumber() ;
            }
            System.out.println(prob + " " + (double)sum/numExp) ;
        }
    }
    }
```

Listing 11: Functions.java

References

[1] Andersen, T., Benzoni, L., and Lund, J. An empirical investigation of continuous-time equity return models. *Journal of Finance*, 57 (2002), 1239.1284.

[2] Bachelier, L. Théorie de la spéculation. *Annales de l'Ecole Normale Supérieure*, 17 (1900), 21–86.

[3] Billingsley, P. *Probability and Measure*, 3rd edition ed. John Wiley sons, 1995.

[4] Black, F., and Scholes, M. The pricing of options and corporate liabilities. *Journal of Political Economics*, 81 (1973), 637–659.

[5] Bollerslev, T. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31 (1986), 307–327.

[6] Castiglione, F., and Stauffer, D. Multi-scaling in the cont–bouchaud microscopic stock market model. *Physica A: Statistical Mechanics and its Applications* (Jan 2001).

[7] Chakraborti, A. Market application of the percolation model: Relative price distribution. *International Journal of Modern Physics C 13*, 1 (2002), 25–30.

[8] Chang, I., Stauffer, D., and Pandey, R. B. Asymmetries, correlations and fat tails in percolation market model. *Arxiv preprint cond-mat/0108345* (2001).

[9] Cont, R. Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance* (Jan 2001).

[10] Cont, R., and Bouchaud, J.-P. Herd behavior and aggregate fluctuations in financial markets. *Macroeconomic dynamics 4*, 02 (2000), 170–196.

[11] Derman, E. *My Life as a Quant.* John Wiley & Sons - New York, 2004.

[12] Ederington, L., and Guan, W. Why are those options smiling?

[13] Geman, H. Pure jump levy processes for asset price modelling. *Journal of Banking and Finance* (2002).

[14] G.E.Uhlenbeck, and L.S.Ornstein. On the theory of brownian motion. *Phys.Rev.*, 36 (1930), 823841.

[15] Heston, S. L. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *The Review of Financial Studies*, Volume 6, number 2 (1993), 327–343.

[16] Hoshen, J., and Kopelman, R. Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm. *Physical Review B 14*, 8 (1976), 3438–3445.

[17] Hoshen, J., Kopelman, R., and Monberg, E. Percolation and cluster distribution. ii. layers, variable-range interactions, and exciton cluster model. *Journal of Statistical Physics 19*, 3 (1978), 219–242.

[18] Hull, J. C. *Options, Futures and Other Derivatives*, 7th edition ed. Prentice Hall, 2008.

[19] Klymko, P., Hoshen, J., and Kopelman, R. Percolation and cluster distribution. iii. algorithms for the site-bond problem. *Journal of Statistical Physics 21*, 5 (1979), 583–600.

[20] Leath, P. L. Cluster size and boundary distribution near percolation threshold. *Phys. Rev. B 14*, 11 (Dec 1976), 5046–5055.

[21] Madan, D., and Seneta, E. The variance gamma (v.g.) model for share market returns. *Journal of Business*, 63 (1990), 511–524.

[22] Samanidou, E., Zschischang, E., Stauffer, D., and Lux, T. Agent-based models of financial markets. *Reports on Progress in Physics* (Jan 2007).

[23] Samuelson, P. Rational theory of warrant pricing. *Indutrial Management Review*, 6 (1965), 13–31.

[24] Stauffer, D. Percolation models of financial market dynamics. *Advances in Complex Systems 4*, 1 (2001), 19.

[25] Stauffer, D., and Aharony, A. *Introduction to percolation theory*, 2nd edition ed. Taylor Francis, Ltd, 1991.

[26] Stauffer, D., and Jan, N. Sharp peaks in the percolation model for stock markets. *Physica A: Statistical Mechanics and its Applications* (Jan 2000).

[27] Stauffer, D., and Sornette, D. Self-organized percolation model for stock market fluctuations. *Arxiv preprint cond-mat* (Jan 1999).

[28] Tanaka, H. A percolation model of stock price fluctuations. *1264* (2002), 203–218.

[29] Tsang, I., and Tsang, I. Critical probabilities for diversity and number of clusters in randomly occupied square .... *Journal of Physics A-Mathematical and General* (Jan 1997).

[30] Wang, J., Yang, C.-X., Zhou, P.-L., Jin, Y.-D., Zhou, T., and Wang, B. Evolutionary percolation model of stock market with variable agent number. *Physica A: Statistical Mechanics and its Applications 354* (2005), 505–517.