Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Modelling of a primitive stock market with MATLAB

Steven Müllener & Thomas Walti

Zurich
May 2008

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Gruppenarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen-Hilsmittel verwenden habe, und alle Stellen, die wörtlich oder sinngemäss aus veröffentlichen Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Gruppenarbeit nicht, auch nicht auszugsweise, bereits für eine andere Leistung ausgefertigt wurde.

Steven Müllener Thomas Walti

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Steven Müllener Thomas Walti

Inhaltsverzeichnis

1 Individual contributions

1.1 Steven Müllener

- Implementation of the basic framework in MATLAB
- Module-based structure
- Price formation algorithm
- Concept of clustering and the associated algorithm
- Thorough analysis of parameters and their influence

1.2 Thomas Walti

- Mathematical concept of historical volatility and log price returns
- Statistical analysis of the model in MATLAB
- Concept of fat-tailed distributions
- Preparation of the plots for inclusion in the report

Needless to say, the points listed above just represent areas of specialisation and responsibility. Large parts of the work has been conducted by both authors often elucidating the material in collaboration.

2 Introduction and Motivations

A simple stock market model prevalently is based on agents trading one type of share. The market price is given by the price where the same number of shares are offered by the sellers as are ordered by the buyers. Therefore, every agent is either a seller or a buyer at a given point in time.

One of the more famous stock market models of a rather simple nature is the Genoa market microstructure circumscribed by M. Raberto et al. presented in (1). In cited paper some information about the model is given but the authors omitted to provide further details about the implementation as well as to further specify several parameters used in the model.

Consequently, in this report a simple stock market based on the Genoa market microstructure is described which was implemented and tested in MATLAB. The aim was to stay as close to the Genoa market model as possible and where there was no information about how a certain part of the algorithm was designed, several options were analysed and the one with the lowest disturbing impact on the simulation results has eventually been chosen.

An expected outcome of the simulation should yield typical features of a real-world stock market. In this report a few of these features will be named and later on in the simulation results section the named features will be looked out for.

3 Description of the Model

In this section the starting point for modelling a primitive stock market will be explained. What is more, the concepts of historical volatility and opinion propagation as two methods for expanding a primitive stock market towards a more realistic model of a real-world stock market behaviour are explained.

At first, a simple round- and agent-based market was implemented where every agent starts with a certain amount of money and a certain amount of shares, each of the same type and valued at a normalised price of 1. At the beginning of each round (hereafter referred to as "time-step"), each agent is randomly chosen to be either seller or buyer with equal probability.

Sellers then define a lower bound price influenced by the current share's market price. The limit-price is computed as follows: limit-price$_{\mathrm{sell}}$:= p/N(μ,σ) where p is the instantaneous market price and N(μ,σ) denotes a random draw from a gaussian distribution with expectation value μ=1.01. The variance is proportional to the historical volatility. This functional dependence establishes the link between nervous markets and agent uncertainty. Putting it in other words: when volatility is high, uncertainty towards the true price of the market grows and traders place orders with a broader distribution of limit-prices. Buy orders are created almost analogous to sell orders: limit-price$_{\mathrm{buy}}$:= p$\cdot N(\mu,\sigma)$. The amount of cash assigned to a buy order is a random fraction of the agent's available cash. Similarly, the amount of shares offered in a sell order is a random fraction of the agent's available shares. 'Random' here indicates a draw from a uniform distribution in both cases. By generating these limit-prices a spread of the average value of buy/sell orders arises naturally. It is worth noting that so far all the random draws happen independently of each other. However, they are subject to two constraints: the historical price volatility and the finiteness of the resources available to each trader. During the price formation process, the new price is set at the intersection of the demand and supply curves. The demand function f(p) is the total quantity of assets ordered to buy at a certain price p which is calculated by summing up all the assets demanded by buyers with a limit-price higher or equal than the price under test p. Let us label the agents who are buyers by u, the associated limit-price by b$_{\mathrm{u}}$ and the assets ordered to buy by a^{b}. The demand curve then becomes:

$$f(p) := \sum_{u|bu \geq p} a_u^b$$

Similarly, the supply curve is calculated by summing over the assets every agent is willing to sell at a certain price. Sellers are labeled by v, the associated limit-price by $\mathrm{s_v}$ and the assets ordered to sell are denoted by a^s. Hence we define the supply curve:

$$g(p) := \sum_{v|Sv \leq p} a_u^s$$

It is easy to see that the supply function is an decreasing step function of p and the demand function is an increasing step function of p. If p is less the the market price g(p) $\leq f(p)$. If conversely p is greater the the market price g(p) $\geq f(p)$. Therefore, to determine the market price one has to iterate the price at which the inequality flips from less than to greater than. There are two pathological cases. One in which the curves do not intersect at all and one in which the two step functions have a common horizontal segment. In the implementation this two rare cases have been respected in a way such that the stock price can go to zero or become very high in just one time step. Especially with just few agents and the opinion propagation model enabled, this possibility has to be accounted for. For maximum efficiency, an algorithm inspired by a binary search has been chosen in order to determine the current market price. In this algorithm, the possible price interval is cut in halves. Then the half is kept where at the lower boundary more stock are demanded whereas at the higher boundaries' price more shares are offered for sale. The other half is then discarded and the steps are repeated until the interval is small enough.

The last section of this artificial market describes the modelling of opinion propagation. In real-life markets one often can study mass psychological phenomena such as herd behaviour periods of frenzied buying (bubbles) or selling (crashes). Many observers cite these episodes as clear examples of herding behaviour being irrational and driven by emotion. Individual investors join a crowd of others in a rush to get in or out of the market (2). In a further analysis Pak et al. subdivided traders into groups of what they called 'rational' and 'noise' traders (3). The latter group's behaviour is governed solely by studying the market dynamics. Not only for noise traders but also for rational traders opinion propagation is a crucial factor to consider in every model.

The Genoa market makes use of clustering as a method of implementing opinion propagation. At every time step each possible pair of traders is chosen to form a cluster with a preliminarily defined possibility. If both of the two traders already are members of a cluster, both of the clusters will merge into one single cluster. It goes without saying, depending on the predefined parameters for the possibilities, clusters can become rather large quickly. Then, at every time step a random cluster is chosen and activated with a preliminarily defined possibility as well. Once a cluster is activated, all of its members behave the same way for this time step meaning they all are sellers or buyers respectively.

Generally supporting the idea, the authors of this paper have found this method one major drawback of the Genoa market when it comes to modelling bubbles or crashes since after the single time-step when the cluster activation takes place, the cluster is destroyed and therefore does not have any further influence on the market price. On the other hand, as soon as historical volatility is considered, the activation of a big cluster is followed by a growing market uncertainty of the agents and it can be argued that this outcome is indeed a further reaching consequence of the herding behaviour.

4 Implementation

4.1 Modules

At first, the desired modules are being activated or disabled. Enabling a module leads to much longer calculation time during execution of the main loop. Depending on several parameters, duration of the calculation can be up to 120 times longer than without trading, historical volatility and opinion propagation enabled.

4.2 Parameter Initialisation

In this section global parameters are set. The more important ones are the following:

- N: Number of agents for the model
- timeSteps: number of trading phases (used in the main loop)
- clusterPairProbability: probability for an agent to pair with another agent in order to form clusters. Each agent can pair with any other agent during each timeStep
- clusterActivateProbability: at each time step a cluster is randomly chosen and activated with this probability
- globalBuyProb: probability that an agent is a buyer during a time step. normally set to 0.5 so agents are sellers and buyers equiprobably.
- sellMu: expectation value for the price factor of a seller
- sellSigmaK: standard deviation for the price factor of a seller
- buyMu: expectation value for the price factor for a buyer
- buySigmaK: standard deviation for the price factor of a buyer

4.3 Agent Initialisation

An agent has several individual parameters. The more important ones are the following:

- cash: amount of money the agent currently posesses
- assets: number of assets the agent currently holds
- buyProb: probability that the agent is a buyer during the current time-step
- isBuyer: indicates whether the agent is a seller during the current time-step
- isSeller: indicates whether the agent is a buyer during the current time-step
- buyCash: holds the amount of money a buyer is willing to pay for assets
- buyQuant: holds the quantity of stocks a buyer wants to buy
- buyUpperLimit: determines the highest price a buyer is willing to pay for assets
- sellQuant: holds the quantity of stocks a seller wants to sell
- sellLowerLimit: determines the lowest price a seller is willing to sell his assets
- cluster: holds the index of the cluster which the agent is a member of

4.4 Code Structure

```
function [status] = market()
% This function is simulating a stock market

%% Modules
TRADING = 1;
HISTORICAL_VOLATILITY = 1;
OPINION_PROPAGATION_CLUSTERS = 1;

%% Parameter Initialisation
...

%% Agent Initialisation
...

%% Main Loop
for t = 1:timeStep
    %% Opinion Propagation
    if OPINION_PROPAGATION_CLUSTERS
        opinion_prop()
    end

    for n = 1:N
        %% Agent is Buyer or Seller during this time-step?
        ...
        %% Historial Volatility
        if HISTORICAL_VOLATILITY
            hist_volat(n)
        end
        %% Agent buy or sell parameters
        ...
    end

    %% New Market Price
    ...

    %% Trading
    if TRADING
        trading()
    end
end

%% PLOT THE DATA
...

end
```

5 Simulation Results

5.1 Expected Results

In our model of the stock market the decision making process is based on drawing a random number which itself is gaussian distributed. Therefore one would expect the stock price to behave like Brownian motion. As long as clustering and volatility effects are neglected the simulation should yield such a result. In a real life financial market however, the stock price returns are not purely normally distributed. Empirically, one usually encounters fat-tailed distribution densities. The main reason for this phenomena is that decisions of traders are not made isolated or independently from other agents actions. Hence, a mechanism has to be found where traders can influence each other wether to sell or to buy shares. In correspondence to the Genoa model, as stated before, clusters are implemented in order to account for this phenomena.

The financial time series have been simulated in three different modes:

- basic mode with limit-prices that are determined by a fixed gaussian function
- basic mode including limit-prices that exhibit a functional dependence on past price volatility. The time window during which recent events are determinative was defined to be 50 time-steps wide.
- the previous mode additionally including the formation of opinion networks (clusters)

All modes were simulated for 1000 time-steps and 200 Agents. As mentioned above the basic mode should lead to gaussian distributed log returns. The including of volatility and clustering effects should give rise to deviations from such a behaviour. One expects to see key stylised facts of financial time series (i.e. fat tails and volatility clustering). In this section plots will be presented for each of the described modes.

5.2 Simulation Results

An interesting result of the simulation is that even in absence of clustering, the artificial market exhibits some key stylised facts of financial time series (i.e. fat tails and volatility) using just simple trading rules in a environment characterised by the finiteness of agent resources, order limit-prices as well as the creation and matching of demand and supply curves. However the plotted results show that there is only a small deviation from the expected normal distribution.

If we include the ability of the agents to consider the past development we find that for values above a certain σ the market dynamics react quiet sensitively. The higher the recent volatility the higher is the agents uncertainty. Therefore price returns increase

dramatically. What is more, we can observe that large changes tend to be succeeded by large changes whereas small changes tend to be succeeded by small changes i.e. volatility clustering.
The last two plots show the results of the third simulation mode in which clusters are formed. These clusters can be interpreted as a model for opinion propagation among the agents. Hence decisions are not made independently. The consequence is a departure from the gaussian probability density. The figures also show another remarkable fact. The price tends to be more stable if the probability of clustering is set to a high value.

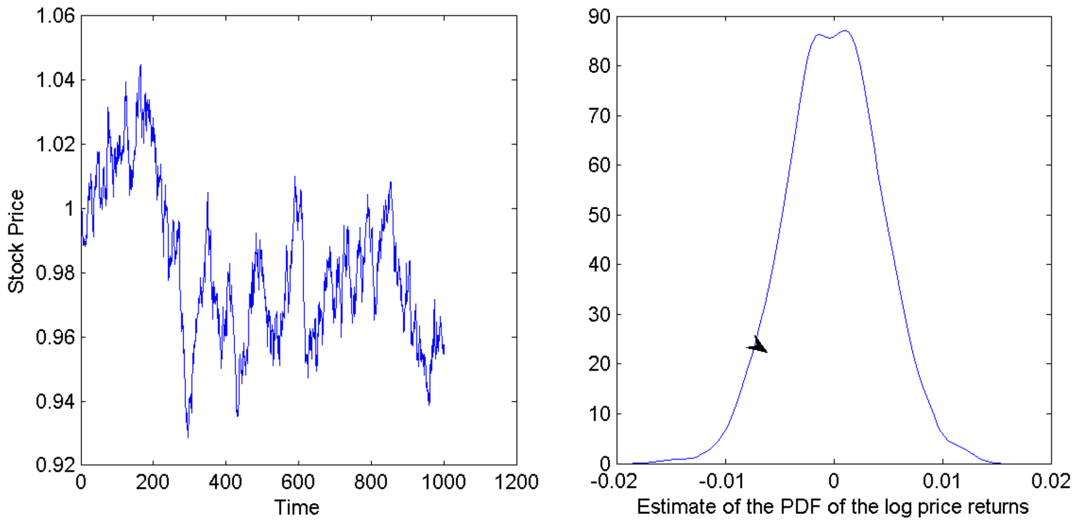

Fig.1 Results for the basic simulation mode i.e in absence of clustering or volatility effects. The picture shows almost no deviation from the hypothesis of normal distribution

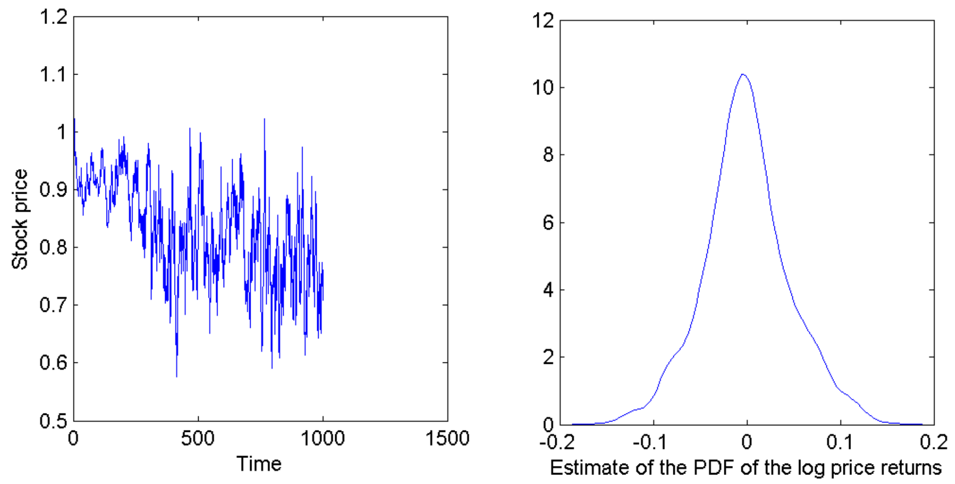

Fig 2. Shows results for a simulation setting the volatility parameter σ to 6.5 and respecting past volatility in a time window of 50 time-steps. Clustering is still inactive. Market dynamics change very sensitive with respect to varying the parameter σ.

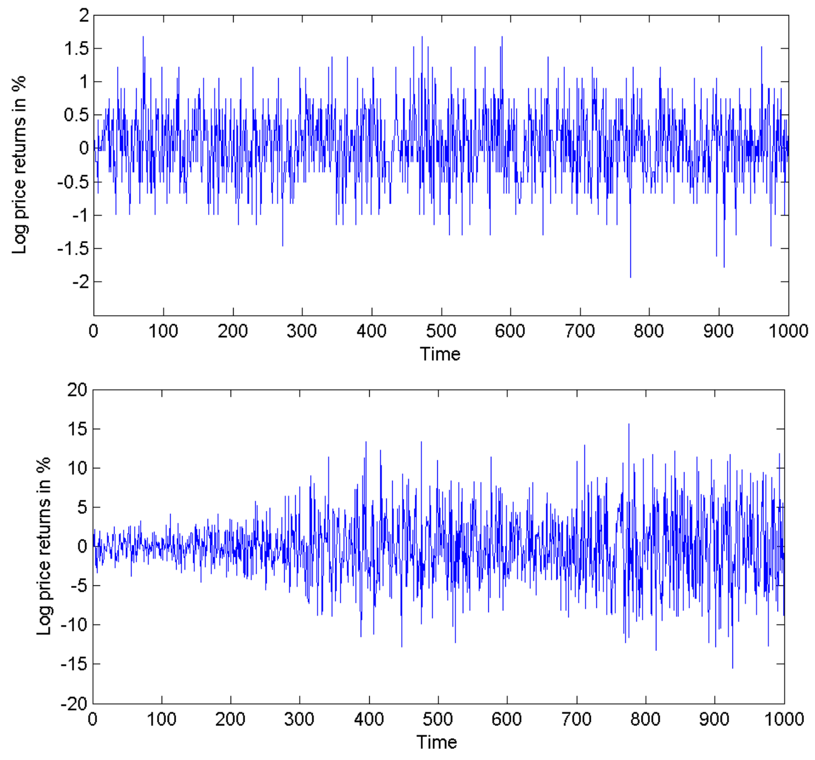

Fig 3. Upper plot: without volatility effects. Lower plot: volatility parameter σ is set to 6.5 and a time window of 50 time-steps is considered.

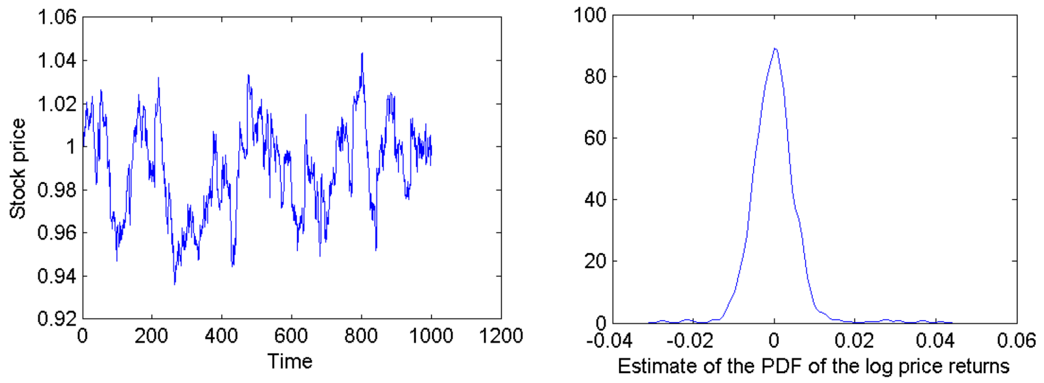

Fig 4. Shows results for a simulation setting the volatility parameter σ to 3.5 and respecting past volatility in a time window of 50 time-steps. Clusters are formed with probability 0.002 and activated with probability 0.4

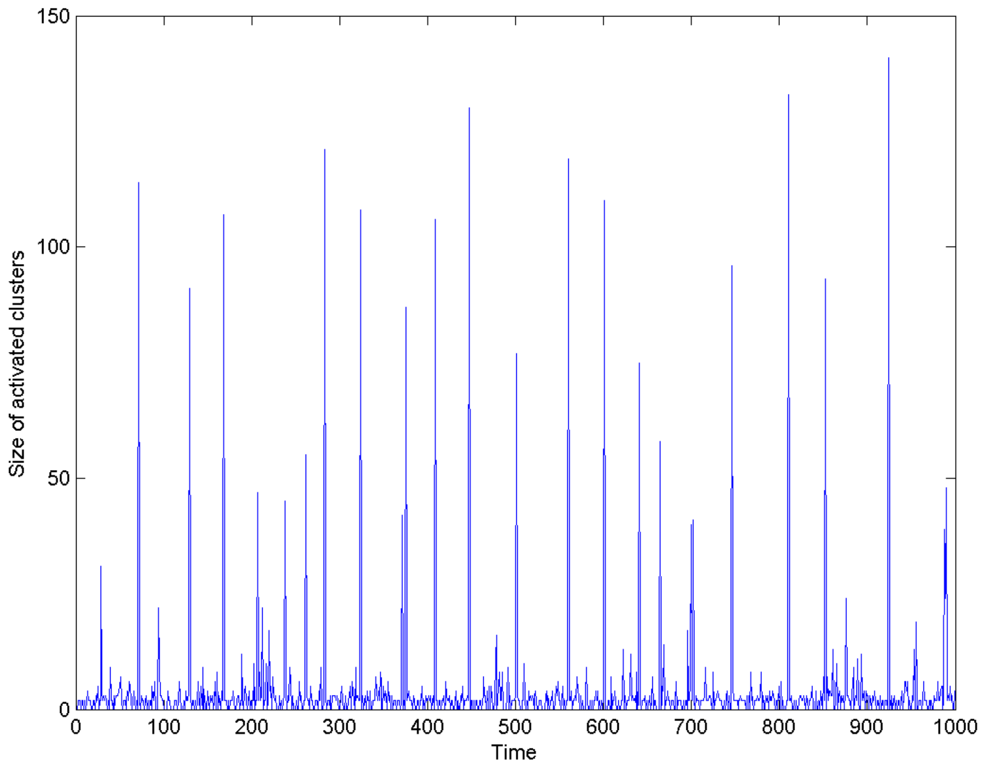

Fig 5. Activation time and size of the activated clusters. Compare with stock price development in Fig 4.

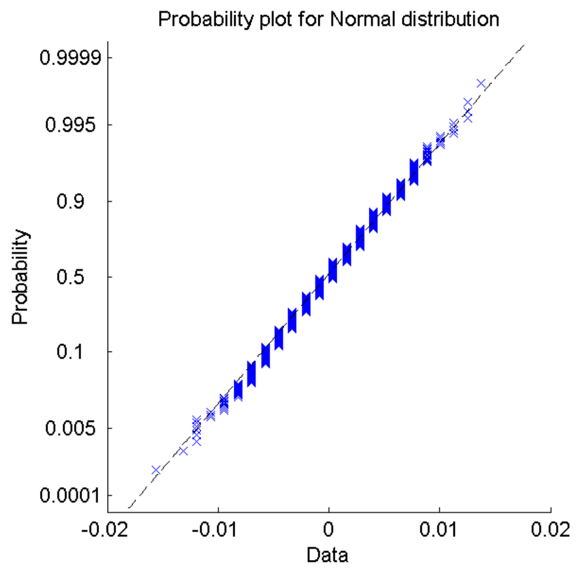

Fig 6. Simulated time series of the log return vs. gaussian normal distribution. Neither volatility nor clustering is activated.

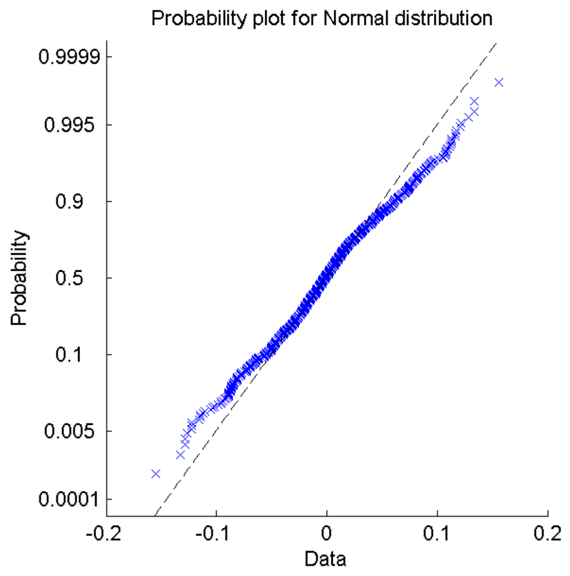

Fig 7. Simulated time series of the log return vs. gaussian normal distribution. Volatility is activated but clustering disabled.

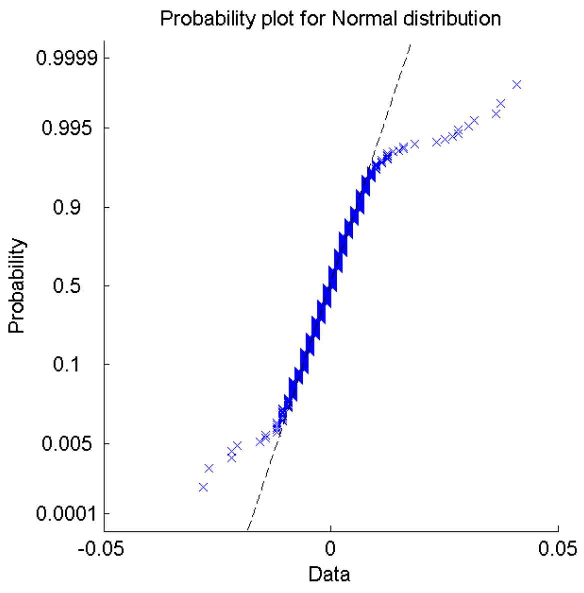

Fig 8. Simulated time series of the log return vs. gaussian normal distribution. Both volatility and clustering are activated.

6 Summary and Conclusions

The artificial market is embedded in a trading environment characterised by the finiteness of agent resources, order limit-prices, and the creation and matching of supply and demand curves. The model exhibits some substantial properties of financial time series such as fat tails and volatility clusters. Simulating the model without allowing the agents to form opinion clusters already yields a rather surprising behaviour that deviates from a gaussian distribution.

Including the functional dependence between recent volatility effects and the limit price determining process has a measurable impact on market dynamics. Volatility clusters are observed and the estimate of the probability density function of logarithmic price returns tends to diverge from the normal distribution. The activation of clustering gives also rise to fat tails. This reflects the fact that decisions of agents are not made independently. In addition, the formation of networks has a somewhat stabilising effect on the price time series.

This model is a computational tool where many experiments can be carried out and that incorporates the key features of the Genoa market model. However, there are some deficits in the model that can be addressed in future research. The creation and annihilation of both agents and money can be introduced. Realistically, the market should be populated with heterogenous groups of agents. Future projects may also include the notion of short sells which is considered as contributing factor to undesirable market volatility.

7 Matlab Source-code

7.1 market.m

```
function [] = market()
% This function simulates a primitive stock market

clear all
close('all','hidden')

%% Modules
TRADING = 1;
HISTORICAL_VOLATILITY = 1;
OPINION_PROPAGATION_CLUSTERS = 1;

%% Parameter Initialisation
N = 200;
timeSteps = 100;
PercentageDone = 0;

% set global market parameters
stockPrice = ones(timeSteps,1);
activatedClusterSize = zeros(timeSteps,1);
clusters = zeros(N/2,1); % maximum number of possible clusters: N/2
clusterPairProbability = 0.0001;
clusterActivateProbability = 0.2;

% set global agent parameters
globalBuyProb = .5;
sellMu = 1.01;
sellSigmaK = 3.5;
buyMu = 1.01;
buySigmaK = 3.5;

% plot options
SUBPLOT_NUMBER_Y = 2;
SUBPLOT_NUMBER_X = 2;
figure(1)
hold on

%% Agent Initialisation
agent(N) = struct( ...
                'cash',                 10000, ...
                'assets',               10000, ...
                'volatSigma',            1, ...
                'volatTimeInt',          2, ...
                'buyProb',               0.5, ...
                'isBuyer',               0, ...
                'isSeller',            0, ...
                'buyCash',             0, ...
                'buyQuant',            0, ...
                'buyUpperLimit',       0, ...
                'sellQuant',           0, ...
                'sellLowerLimit',      inf, ...
                'cluster',             0 ...
                );

for n = 1:N
   agent(n).cash = 1000;
   agent(n).assets = 1000;
   agent(n).volatSigma = 0.02;
   agent(n).volatTimeInt = 50;
   agent(n).buyProb = globalBuyProb;
   agent(n).isBuyer = 0;
   agent(n).isSeller = 0;
   agent(n).buyCash = 0;
   agent(n).buyQuant = 0;
   agent(n).buyUpperLimit = 0;
   agent(n).sellQuant = 0;
   agent(n).sellLowerLimit = inf;
   agent(n).cluster = 0; % if member of cluster: [clusternumber] | else: [0]
end

global agent;
global N;
global t;
global globalBuyProb;
global clusterPairProbability;
global clusterActivateProbability;
global activatedClusterSize;
global clusters;
global stockPrice;
global buySigmaK;
global sellSigmaK;

%% Main Loop
for t = 1:timeSteps

    %% Opinion Propagation
    if OPINION_PROPAGATION_CLUSTERS
        opinion_prop()
    end

    for n = 1:N

        %% Agent is Buyer or Seller during this time-step?
        if rand(1)<agent(n).buyProb
            agent(n).isSeller = 0;
```

```
                agent(n).isBuyer = 1;
                % buyCash update
                agent(n).buyCash = rand(1)*agent(n).cash;
            else
                agent(n).isSeller = 1;
                agent(n).isBuyer = 0;
                % sellQuant update
                agent(n).sellQuant = round(rand(1)*agent(n).assets); %av
            end

            %% Historial Volatility
            if HISTORICAL_VOLATILITY
                hist_volat(n)
            end

            %% Agent buy or sell parameters
            if (agent(n).isBuyer == 1)
                agent(n).buyUpperLimit = stockPrice(t)*...
                                          (normrnd(buyMu,agent(n).volatSigma));
                agent(n).buyQuant = round(agent(n).buyCash/agent(n).buyUpperLimit);
            else
                agent(n).sellLowerLimit = stockPrice(t)/...
                                           abs(normrnd(sellMu,agent(n).volatSigma));
            end

        end

        %% New Market Price
        stockPrice(t+1) = price_formation(stockPrice(t));

        %% Trading
        if TRADING
            trading()
        end

        if PercentageDone ≠ round(100*(t/timeSteps))
            PercentageDone = round(100*(t/timeSteps))
        end

    end

    %% PLOT THE DATA
    %  Stock price
    subplot(SUBPLOT_NUMBER_Y,SUBPLOT_NUMBER_X,1)
    plot(stockPrice)
    %  Log price return of the Stock price
    subplot(SUBPLOT_NUMBER_Y,SUBPLOT_NUMBER_X,2)
    stockPriceReturnLog = log(stockPrice(2:end))-log(stockPrice(1:end-1));
    plot(100*stockPriceReturnLog)
    %  Probplot of the log price return
```

```
subplot(SUBPLOT_NUMBER_Y,SUBPLOT_NUMBER_X,3)
probplot(stockPriceReturnLog)
%  Estimate of the PDF
subplot(SUBPLOT_NUMBER_Y,SUBPLOT_NUMBER_X,4)
ksdensity(stockPriceReturnLog)

figure(2)
plot(activatedClusterSize)


end
```

7.2 trading.m

```
function [] = trading()
% This function is simulating a stock market
% INPUT:
%
% OUTPUT:
%

global agent
global N
global t
global stockPrice

% lower bound of stocks to trade
[f,g] = supply_demand(stockPrice(t+1));
tradeVolume = min(f,g);
tradedSell = 0;
tradedBuy = 0;
for n = 1:N
    if agent(n).isSeller == 1
        if stockPrice(t+1)≥agent(n).sellLowerLimit && tradedSell<tradeVolume
            % This seller sells
            numAssetsToSell = min(agent(n).sellQuant, tradeVolume-tradedSell);
            agent(n).assets = agent(n).assets-numAssetsToSell;
            tradedSell = tradedSell + numAssetsToSell;
            agent(n).cash = agent(n).cash + (numAssetsToSell * stockPrice(t+1));
        end
    else if stockPrice(t+1)≤agent(n).buyUpperLimit && tradedBuy<tradeVolume
        % This buyer buys
        numAssetsToBuy = min(agent(n).buyQuant, tradeVolume-tradedBuy);
        agent(n).assets = agent(n).assets+numAssetsToBuy;
        tradedBuy = tradedBuy + numAssetsToBuy;
        totalBuyPrice = numAssetsToBuy * stockPrice(t+1);
        agent(n).cash = agent(n).cash - totalBuyPrice;
        end
    end
end
```

7.3 hist_volat.m

```
function [] = hist_volat(n)
% This function is simulating a stock market
% INPUT:
%
% OUTPUT:
%

global agent
global t
global stockPrice
global buySigmaK
global sellSigmaK

%calculate volatSigma
if t==1
    agent(n).volatSigma = 0;
else
    if t < agent(n).volatTimeInt
        stockPricesRecent = stockPrice(1:t);
        logPriceReturn = log( stockPricesRecent(2:end)./...
                              stockPricesRecent(1:end-1) );
    else
        stockPricesRecent = stockPrice(t-agent(n).volatTimeInt+1:t);
        logPriceReturn = log( stockPricesRecent(2:end)./...
                              stockPricesRecent(1:end-1) );
    end
    if (agent(n).isBuyer == 1)
        agent(n).volatSigma = buySigmaK * std(logPriceReturn);
    else
        agent(n).volatSigma = sellSigmaK * std(logPriceReturn);
    end
end
```

7.4 opinion_prop.m

```
function [] = opinion_prop()
% This function is simulating a stock market
% INPUT:
%
% OUTPUT:
%
global agent
global N
global t
global globalBuyProb
global clusterPairProbability
global clusterActivateProbability
global activatedClusterSize
global clusters
% Pairing
% Reset last buyProb at first (changed for activated clusters)
agent(N).buyProb = globalBuyProb;
%keyboard
for i = 1:N-1
    % Reset all buyProb to globalBuyProb (changed for activated clusters)
    agent(i).buyProb = globalBuyProb;
    for j = i+1:N
        % form pair with Pa
        if rand(1)<clusterPairProbability && ...
            ((agent(i).cluster≠agent(j).cluster)||agent(i).cluster==0)
            % form pair:
            % if i agent is member of cluster:
                % set j agent (and all his cluster members)
                % to i cluster (and free j cluster)
            if agent(i).cluster≠0
                if agent(j).cluster≠0
                    clusters(agent(j).cluster)=0;
                    clusterToChange = agent(j).cluster;
                    for k = 1:N
                        if agent(k).cluster==clusterToChange
                            agent(k).cluster=agent(i).cluster;
                        end
                    end
                end
                agent(j).cluster=agent(i).cluster;
            % elseif j agent is member of cluster (and i is not):
                % set i agent to j cluster
            elseif agent(j).cluster≠0
                agent(i).cluster = agent(j).cluster;
            % else
            % form new cluster and reserve slot in cluster-arr
            else
                k = 1;
                while clusters(k)≠0; k=k+1; end
                %if k ≠ 1; keyboard; end
                clusters(k)=1;
                agent(i).cluster = k;
                agent(j).cluster = k;
            end
        end
    end
end

% Cluster activation
if rand(1)<clusterActivateProbability
    % find nonzero clusters
    activeClusters = find(clusters);
    % random draw of a cluster (first element after randperm)
    randElemNum = randperm(length(activeClusters));
    %activeClusters = randperm(activeClusters);
    % activate the cluster
    if ¬isempty(activeClusters)
        if rand(1)<0.5
            buyProb = 0;
        else
            buyProb = 1;
        end
        tempSum = 0;
        for k = 1:N
            if agent(k).cluster == activeClusters(randElemNum(1));
                agent(k).buyProb = buyProb;
                agent(k).cluster = 0; % leave cluster
                tempSum = tempSum + 1;
            end
        end
        activatedClusterSize(t) = tempSum;
        if activatedClusterSize > 0.95*N
            %keyboard % Too big a cluster was activated
        end
        clusters(activeClusters(randElemNum(1)))=0; % free cluster
    end
end

end
```

7.5 price_formation.m

```
function [p] = price_formation(marketPriceLast)

global agent
global N


%% Price Formation
    pLower = 0;
    pUpper = 10*marketPriceLast;

    % check wether f(pLower)>g(pLower) AND f(pUpper)<g(pUpper) at beginning
    % exit with old price if not
    [f1,g1] = supply_demand(pLower);
    [f2,g2] = supply_demand(pUpper);
    if f1≤g1
        p = 0;
    elseif f2≥g2
        p = 10 * marketPriceLast;
    else
        pTest = (pUpper+pLower)/2;
        [f,g] = supply_demand(pTest);
        while((pUpper-pLower)>marketPriceLast/1000 && f≠g)
            pTest = (pUpper+pLower)/2;
            [f,g] = supply_demand(pTest);
            if f>g
                pLower=pTest;
            else
                pUpper=pTest;
            end
        end
        p = pTest;
    end
end
```

7.6 supply_demand.m

```
function [f,g] = supply_demand(p)

global agent
global N

%% Supply/Demand values for given price p
    f = 0;
    g = 0;
    for n = 1:N

        %% Buy orders
        if (agent(n).buyUpperLimit≥p)&&(agent(n).isBuyer)
            f = f + agent(n).buyQuant;

        %% Sell orders
        elseif (agent(n).sellLowerLimit≤p)&&(agent(n).isSeller)
            g = g + agent(n).sellQuant;

        end

    end

end
```

8 References

Literatur

[1] M. Raberto et al. *Agent based simulation of a financial market* Physica A 299 (2001) 319-327

[2] M. K. Brunnermeier *Asset Pricing under Asymmetric Information: Bubbles, Crashes, Technical Analysis, and Herding* Oxford University Press (2001)

[3] P. Bak, M. Paczuski and M. Shubik *Price Variations in a Stock Market with Many Agents* Cowles Foundation Discussion Papers 1132, Cowles Foundation, Yale University. (1996)

[4] Ruey S. Tsay *Analysis of Financial Time Series* Wileysons.(2005)

[5] M. Raberto, Silvano Cincotti. *Modelling and simulation of a double auction artificial market* Physica A 355 (2005) 34-35

[6] M.Raberto et al. *Volatility in the Italian Stock Market: an Empirical Study*, Quantitative Finance Papers (1999)