Sensitivity Analysis of an Econophysical Model using Stochastic Collocation

Projektarbeit
Computational Engineering Science

submitted by Emma Pabich, emma.pabich@rwth-aachen.de,
Till Beemelmanns, till.beemelmanns@rwth-aachen.de,
Simon Cramer, simon.cramer@rwth-aachen.de

First Supervisor: Prof. Dr. Martin Frank
Department of Mathematics
RWTH Aachen

Second Supervisors: Torsten Trimborn
Department of Mathematics
RWTH Aachen

Philipp Otte
Department of Mathematics
RWTH Aachen
# Contents

1 Introduction 5

2 Model 7
   2.1 General set-up 7
   2.2 Related Models 7
   2.3 iAgent 9
      2.3.1 Three Sources of Information 9
      2.3.2 Opinion formation 10
      2.3.3 Trading decision 10
      2.3.4 Adaption of Trust Coefficients 11
      2.3.5 Cash and stock positions 12
      2.3.6 Neighbourhood Information 12
   2.4 iMarket 13
      2.4.1 Market Clearing 13
      2.4.2 Neighbourhood 13
   2.5 Standard Parameter Set 13

3 Implementation 15
   3.1 MATLAB 15
      3.1.1 Class structure 15
      3.1.2 Generation of random numbers 15
      3.1.3 Optimisation and Parallelisation 15
      3.1.4 Tests and remaining errors 16
   3.2 C++ 16
      3.2.1 Class structure 17
      3.2.2 Generation of random numbers 17
      3.2.3 Optimisation and Parallelisation 17
      3.2.4 Tests and remaining errors 18
   3.3 Comparison 18
   3.4 Initialisation 19

4 Early Simulations 25
   4.1 Reproduction 25
      4.1.1 The Excitable Regime 27
      4.1.2 Existence of Herding 28
      4.1.3 Number of Agents 29
   4.2 Claim Review 29
      4.2.1 Different Price Clearing Condition 30
      4.2.2 Different Neighbourhoods 30
      4.2.3 Different $E_i(s_j(t_k))$ 31
## 5 Stochastic collocation

5.1 Idea ......................................................... 32
5.2 Input Parameters ........................................ 33
5.3 Quantities of Interest .................................... 34
  5.3.1 Bubble and Crash Quantification .................. 35
5.4 Sparse Grids .............................................. 36
  5.4.1 Linear Interpolation on a Full Grid in one-dimension 36
  5.4.2 Linear Interpolation on a Full Grid with Hierarchical basis functions in one-dimension 37
  5.4.3 Linear Interpolation on a Full Grid with Hierarchical basis functions in arbitrary dimensions 38
  5.4.4 Sparse Grids ...................................... 39
  5.4.5 Clenshaw-Curtis Rule ............................. 40
  5.4.6 Grid Complexity comparison ...................... 43
  5.4.7 Sparse Grid Implementation ....................... 43
  5.4.8 Adaptive Sparse Grids ............................ 44
  5.4.9 Evaluation ......................................... 45
  5.4.10 Implementation test .............................. 45
5.5 Results .................................................... 46
  5.5.1 The Excitable Regime ............................... 46
  5.5.2 Influence of $\Omega$ ................................. 47

## 6 Conclusion and Outlook

Literatur 53
1 Introduction

In the history of financial markets, bubbles and crashes have always been of great importance and the consequences of a big financial crash were often devastating. In the last 30 years the connectivity between the international economic markets has advanced enormously [1] and the financial market has become detached from the real world economy [2]. Unfortunately bubble and crash patterns on the financial market can spill over to the real economy and affect these negatively due to the fact that the capital transfer to companies is disturbed [3]. Therefore, researchers and economics are interested in understanding the cause of bubbles and crashes which sometimes seem to occur out of no rational reason.

On the base of this hypothesis, Harras & Sornette developed an agent-based model which we examine in this paper.

One possible mechanism is that crashes emerge from a long run-up of the price above the rational price (bubble) and crash from this labile phase within a short time frame below the rational stock price. The stop of the run-up of the price and the emerging crash is triggered, in many cases by an unexpected negative influence on the market. Economic research has yet to agree on a precise definition of a bubble and its characteristics [4]. Bubbles seem to emerge from irrational decisions of traders and over-optimistic speculations. Therefore financial bubbles and crashes can be explained by several possible mechanism.

One of these mechanism is the social-informational interaction between the investors long term before the crash, summarised by Sornette (2003)[5]. He explains that the “underlying cause of a crash should be found in the preceding month and years”[4] where we can observe that investors tend to imitate the decision of traders which are part of their social network. This is called “herding behaviour” [6]. This market cooperation is amplified by “several positive feedback mechanisms” and leads to a “grow of the correlation between investors’ decisions” [4]. A progressive grow of the market price is caused which is eventually followed by an unstable market. Finally a crash occurs due to a small disturbance of the markets equilibrium. Some investors may obtain information which lets them decide to disinvest their money with the effect that other traders with strong trust in these will follow them blindly. The positive feedback beneath the investors cau-ses a panic and a mass divestment on the stock which then leads to a fast drop of the stock price. This model is interesting because it introduces psychology in terms of “trust” between agents.

In Econophysics, methods developed in physics are applied to solve problems in economics. Those problems often include uncertainty or stochastic processes and nonlinear dynamics (c.f. [7]). A driving force behind econophysics was the availability of financial data collected since the 1980s. While standard economic methods investigates homogeneous agents in equilibrium or close to equilibrium, more interesting observation can

5
be made with inhomogeneous agents and far of equilibrium condition. Physicist, mostly based in the field of statistical mechanics, started to match their models against real market data in contrast to economists who favour soluble models [7].

In classical economic theory, the agents are assumed to be rationally acting individuals, so called *Homo economicus* [8], who make their decisions only on the basis of their own benefit and rational rules. In Econophysics, agent-based models go beyond the prototype theories by using “representative” agents. It is common to use multiple agent types or agents with different character traits in one model. These models are often derived from statistical physics and are used to investigate the coherences between the micro behaviour of one individual and the macro behaviour of the whole system [9].

Econophysical models are often based on models from statistical mechanics. The model of Harras and Sornette has its roots in the Ising model developed in 1924 by Ernst Ising. Rather simplified the Ising model describes the two states of a particle in a ferromagnetic material. Each particle can have one of the two spins (-1,+1). Neighbours with different spins exert forces to each other, trying to homogenise their neighbourhood. In the two dimensional model the particles are arranged on a finite square lattice. One can think of a particle as an agent who is either selling (-1) or buying (+1) stock while the market and psychological effects mimic the forces particles exert on each other. Harras and Sornette extended the Ising model with a third passive state (0) in which the agent is neither selling nor buying stock [6][7].

These models are implemented as ideal test cases and make use of simplified assumptions compared to the mechanics of a real financial market and behaviour of humans. As soon as these models utilise random variables it is inevitable to verify the results with stochastic methods in order to reliably show dependencies between parameters and observations.

For this reason, we decided to conduct a sensitivity analysis on the model of Harras and Sornette to further the understanding of the influences that different parameters have on the model. The method of stochastic collocation offers great capabilities of analysis while escaping the course of dimensionality.

Harras and Sornette tried to explain in [4] why bubbles and crashes exist and how they emerge. They identify two behavioural regimes in their model depending on how much an agent is influenced by his neighbour. One “efficient” regime already reproducing stylised facts and an “excitable” regime where bubbles/crashes occur as agents start to imitate each others actions[4].

The paper is organised as follows: In section 2, we will introduce our model and relate it to others; section 3 will cover the implementation; section 4 we will present early results from the model. Section 5 gives a short introduction to stochastic collocation and present our results. Finally in section 6 we will draw our conclusion and give a short outlook.
2 Model

We re-implement a model from the paper “How to grow a bubble: A model of myopic adapting agents” by Georges Harras and Didier Sornette [4].

2.1 General set-up

The model considers a fixed amount of $N \in \mathbb{N}$ agents who are trading a single stock or asset. The market is organised by a market maker. At every time step each agent decides whether he wants to trade or remain passive based on his opinion.

The agents form their opinion with the use of three sources of information: private information, global information and neighbourhood information. The trust in the different sources is adapted over time based on their performance. Every agent has a fixed neighbourhood of $J \in \mathbb{N}$ neighbours, whose decision influences the neighbourhood information.

At the beginning of a time step a new stock price is computed based on the trading decisions and trading volumes from the last time step. Afterwards all agents are updated with the new stock price. This enables them to recalculate their liquid funds and their amount of stock. Furthermore every agent adapts his trusts in the different sources of information based on their performance. Now the agents can recompute their opinion, trading decision and trading volume. To avoid an identical flow of information in every time step, the update and adaptation process is done in a unique random order per time step. Finally the market maker fetches all trading volumes and decisions to determine the new stock price. Figure 1 summarises the process.

The simulation is conducted over a finite time interval $[T_0, T_1]; 0 \leq T_0 < T_1$. We discretise the time interval by choosing a fixed amount of time steps $D \in \mathbb{N}$ and define $\Delta t = \frac{T_1 - T_0}{D}; \Delta t > 0$. A time step $k$ is then given by $t_k = T_0 + k \cdot \Delta t$.

2.2 Related Models

Before presenting the model’s implementation it is useful to show how it is related to other models.

The conversion from the different pieces of information to the decision of one agent in the model of Harras and Sornette is based on the model of Zhou and Sornette [10]. It focuses on agent herding and not-rational decision making. Their ising model examines the imitating herding behaviour of individuals and the macro behaviour of the system in a more simple way. In contrast to the model considered in this paper, Zhou’s and Sornette’s [10] agents are not allowed to stay passive at a trading step and the three different sources of information are not weighted.

Zhou and Sornette’s [10] model is inspired by the fundamental Ising-Model by Ernst
Ising. The Ising-Model is a physical model of ferromagnetism which interprets discrete integer variables \((-1, +1)\) as spins organised in a lattice and each spin is interacting with the neighbouring spins. Transferred to the financial model, the spins is interpreted as an agent selling \((-1)\) or buying \((+1)\) stocks.

A model described by Topol (1991) [11] also uses a combining of agents’ information sources and in addition to that, his agents have a learning behaviour likewise in [4]. The individuals’ learning strategy is based on the past price dynamics and furthermore the agents of Topol correct their highest price for paying and lowest price for buying, the so-called bid-ask prices, by gathering information from the other agents’ bid prices, ask prices and other rational information about the market. The agents of [4] have a similar information gathering. The agents observe the actings of their neighbours, include it into their information sources and are able to learn which information source is more trustful. But in contrast to Topol, the agents do not consider the bid-ask prices of other individuals.
Agents who adapt their behaviour with “experiences” from the past where implemented by Wyart and Bouchaud (2007) [12]. The agents use strategies based on the past correlation between quantitative information and the price. The strategies create a feedback loop which are able to destabilise the market which is demonstrated by overreaction and excess volatility. In addition to that, Wyart and Bouchaud give empirical evidence for the existence of these correlations in real markets. Harras and Sornette state that their systems with a similar agent’s strategy is also likely to build up a feedback loop among the agents which leads to a unstable market.

The information cascades which is a main topic in Harras’ and Sornette’s [4] model, was investigated by Bikhchandi et al. (1992) [13] and it was shown that an informational cascades occur when an individual imitates the action of individuals before him, despite possible contradictions with his own information/believes. The agents in this model have only limited information and use similar to the examined model a neighbourhood from which an agent draws information.

Another model which uses the concept of positive feedback among the agents was developed by Chari and Kehoe (2003) [14]. Their agents observe all other agents’ actions and try to derive the fundamental value of the stock in order to buy and sell at the right time with the motive to make profit with the difference in the price. The positive feedback in [4] is also explained by the fact that agents observe each other, but in [4] the agents only observe their direct neighbours and not all other agents.

Veldkamp designed a model to observe the creation of bubbles and crashes (2005)[15]. This model uses infrequent positive news regarding a successful unknown company resulting a slow rise in the asset price but also an increase in the sensitivity to the news. So that a single piece of bad news can trigger a crash. The main difference between the model of Veldkamp and the model by Harras and Sornette, is that Veldkamp uses only positive news to create a bubble, whereas Harras and Sornette use negative and positive information. Furthermore a crash is not inevitably induced by a single piece of bad information.

2.3 iAgent

2.3.1 Three Sources of Information

The agents form their opinion based on three sources of information. The first source is the agent’s social network. Every trader is connected to $J \in \mathbb{N}$ others from whom he gathers their opinions or trading decisions. The second source is the global information, which is the same for every agent. These information represent economic and financial news that may influence the future development of the market. The global information is conducted from a normal distribution.
and is uncorrelated with the personal information of each agent. The last source of information is unique for every agent. It represents knowledge which may not be available publicly or a subjective view on how the market will develop in the future. This personal information is uncorrelated with time and obtained from a normal distribution \( \mathcal{N}(0, 1) \).

### 2.3.2 Opinion formation

The opinion of agent \( i \in [1, N] \) at time \( t_k \) consists of the weighted sum of the three sources of information introduced above.

\[
\omega_i(t_k) = c_{1i} \sum_{j=1}^{J} k_{ij}(t_{k-1}) E_i[s_j(t_k)] + c_{2i} u(t_{k-1}) \eta(t_k) + c_{3i} \epsilon_i(t_k)
\]

In this formula \( \epsilon_i(t_k) \sim \mathcal{N}(0, 1) \) represents the personal information of agent \( i \), \( \eta(t_k) \sim \mathcal{N}(0, 1) \) describes the public information. \( J \) is the number of neighbours that agent \( i \) takes into consideration and \( E_i[s_j(t_k)] \) is the expected decision of these neighbours.

The underlying concept of equation (1) is based on a standard assumption in the social interaction literature ([16], [17]) and decision making theory [18] that an agent forms his opinion by combining different sources of information. To take into account the personality of each agent, and the heterogeneity of the agents we characterise each of them by a tripped of fixed traits \( (c_1, c_2, c_3) \). Thus we can describe how much weight an agent gives to each source of information. These character traits are chosen only once, randomly from three uniform distributions \( \bar{c}_1 \sim \mathcal{U}[0, C_1], \bar{c}_2 \sim \mathcal{U}[0, C_2], \bar{c}_3 \sim \mathcal{U}[0, C_3] \). These weights are fixed during one simulation.

The agents are able to modify the coefficients \( k_{ij} \in \mathbb{R} \) for the neighbourhood information and \( u \in \mathbb{R} \) for the global information in order to adapt to the recent market regime. The factor \( k_{ij} \) indicates the reliability of a certain neighbour. It increases if the neighbour proved to be helpful in the recent past and decreases if he predicted the market changes wrong. The factor \( u \) can be modified, to give more weight to global information if this seems reasonable.

### 2.3.3 Trading decision

Another important part each agent’s personality is their aversion to risks. This character trait is defined by the unique threshold \( \bar{\omega}_i \in [0, \bar{\Omega}] \) chosen from the uniform random variable \( \omega \sim \mathcal{U}[0, \bar{\Omega}] \). This threshold is fixed and drawn randomly from a uniform distribution in the interval \( [0, \bar{\Omega}] \). The trader decides to buy stock if \( \omega_i(t_k) > \bar{\omega}_i \) and sells a stock if \( \omega_i(t_k) < -\bar{\omega}_i \). If the agent’s opinion is between these two boundaries he remains passive.

Each time an agent decides to buy a stock, he uses a fixed percentage \( g \) of his cash. The amount of stocks he is able to buy with this percentage of stock is called trading volume.
$v_i \in \mathbb{R}^+$. Reversely he can sell the same fraction $g \in (0, 1)$ of his stocks. Thus our agent’s liquidity is constrained. The wealth of an agent is the sum of his cash and the value of his stocks at the current market price. This decision making process can be summarised as follows:

- if $\omega_i(t_k) > \bar{\omega}_i$ :
  
  $s_i = +1$ (buying)
  
  $v_i(t_k) = g \cdot \frac{\text{cash}_i(t_k)}{\text{price}(t_k-1)}$

- if $-\bar{\omega}_i \leq \omega_i(t_k) \leq \bar{\omega}_i$ :
  
  $s_i = 0$ (passive)
  
  $v_i(t_k) = 0$

- if $\omega_i(t_k) < -\bar{\omega}_i$ :
  
  $s_i = -1$ (selling)
  
  $v_i(t_k) = g \cdot \text{stock}_i(t_k)$

### 2.3.4 Adaption of Trust Coefficients

The agents are able to adapt their trust in the news $\eta(t_k)$ and in the advice $E_i[s_j(t_{k-1})]$ of their neighbours according to their performance in the past. Each agent adapts the time dependent coefficients $u(t_k)$ and $k_{ij}(t_k)$ according to the source’s prediction and the realised return. In this evaluation he gives recent data a higher priority using standard auto-regressive update:

\[
u(t_k) = \alpha u(t_{k-1}) + (1 - \alpha) \eta(t_{k-1}) \frac{r(t_k)}{\sigma_r(t_k)} \]  
\[
k_{ij}(t_k) = \alpha k_{ij}(t_{k-1}) + (1 - \alpha) E_i[s_j(t_{k-1})] \frac{r(t_k)}{\sigma_r(t_k)} \]  

with

\[
\sigma_r(t_k) = \sqrt{\alpha \sigma_r(t_{k-1})^2 + (1 - \alpha)(r(t_{k-1}) - <r(t_k)>)^2}
\]

\[
<r(t_k)> = \alpha <r(t_{k-1})> + (1 - \alpha)r(t_{k-1})
\]

Where $\alpha \in (0, 1)$ is the memory discount factor which controls the amount of time steps each agent looks back for his trust coefficient adaptation. This amount of time steps can be approximated by series expansion. Let $\bar{u}_k = \eta(t_{k-1}) \frac{r(t_k)}{\sigma_r(t_k)}$ then

\[
u(t_k) = \alpha u(t_{k-1}) + (1 - \alpha)\bar{u}_k
\]

$\iff$

\[
u(t_k) = \alpha^2 u(t_{k-2}) + (\alpha^2 - \alpha)\bar{u}_{k-1} + (1 - \alpha)\bar{u}_k
\]

$\iff$

\[
u(t_k) = \alpha^k u(t_0) + \ldots + \alpha^2 u(t_{k-2}) + (\alpha^2 - \alpha)\bar{u}_{k-1} + (1 - \alpha)\bar{u}_k
\]
Let $\epsilon > 0$ then
\[
\begin{align*}
\alpha^k & \leq \epsilon \\
\Leftrightarrow \quad k \cdot \log(\alpha) & \leq \log(\epsilon) \\
\Leftrightarrow \quad k & \geq \frac{\log(\epsilon)}{\log(\alpha)} \\
\Rightarrow \quad k & \approx \frac{1}{|\log(\alpha)|}
\end{align*}
\]

As shown agents remember approximately $\frac{1}{|\log(\alpha)|}$ time steps.

### 2.3.5 Cash and stock positions

After the computation of the new market price we update each agent’s cash and stocks.

\[
\begin{align*}
cash_i(t_k) & = cash_i(t_{k-1}) - s_i(t_{k-1})v_i(t_{k-1})price(t_k) \\
stock_i(t_k) & = stock_i(t_{k-1}) + s_i(t_{k-1})v_i(t_{k-1})
\end{align*}
\]

### 2.3.6 Neighbourhood Information

The neighbourhood information $E_i[s_j(t_k)]$ is described by “the expected action of the neighbour $j$ estimated by agent $i$ at time $t$”\(^1\), but unfortunately Harras and Sornette do not provide an exact equation for this term in [4]. One would expect $E_i[s_j(t_k)]$ to be a real expected value similar to

\[
E_i[s_j(t_k)] = P(s_j = 1) \cdot 1 + P(s_j = 0) \cdot 0 + P(s_j = -1) \cdot (-1). \tag{7}
\]

Since there is no further information on $P(s_j)$ in [4] and other publications by Harras or Sornette, we used this space for interpretation to test two different approaches.

The simple version is to collect the trading decision directly from the neighbours. The more complex one is to collect the neighbours’ opinions and then estimate the trading decision of the neighbour by comparing the opinion to the agent’s own threshold.

\[
E_i[s_j(t_k)] = \begin{cases} 
+1, & \text{if } \omega_j(t_k) > \bar{\omega}_i \\
0, & \text{if } -\bar{\omega}_i \leq \omega_j(t_k) \leq \bar{\omega}_i \\
-1, & \text{if } \omega_j(t_k) < -\bar{\omega}_i
\end{cases} \tag{8}
\]

We found a second approach in Zhou et. al [10], where $E_i[s_j(t_k)]$ is simply replaced by the decision of the neighbour $j$ at time step $t_{k-1}$:

\[
E_i[s_j(t_k)] = s_j(t_{k-1}) \tag{9}
\]

For our model we use equation 9, because we found it in a paper by Sornette. A comparison of the two version can be found in section 4.2.3.

\(^1\)A sequential updating mechanism with random ordering is used. This causes agent $i$ to obtain a different mix of already updated and not yet updated decisions each time step.
2.4 iMarket

The implied market is greatly simplified from real financial markets. We assume it to be a closed, frictionless and interest-free system. A market is closed if the number of agent is fixed and there are no agents entering or leaving the market and further no cash can enter or leave the market. You can compare this to the inability of foreign investors to enter the market and the inability of agents to borrow money outside of the system. In addition the market is frictionless, which implies that there are no trading fees or transaction delays. Further there are no dividends from cash, therefore the market is interest-free.

The market maker has an unlimited amount of cash and stock, therefore he can execute all transactions immediately and is adapting the price to the excess demand before executing the trades [4]. The market can be classified as an ideal market as proposed by E. Fama in her theory of Efficient Market Hypothesis (1965) [6].

2.4.1 Market Clearing

The new price is computed based on the excess demand $r(t_k)$ after all agents have been queried for their decision and their trading volume:

$$r(t_k) = \frac{1}{\lambda N} \sum_{i=1}^{N} s_i(t_k)v_i(t_k)$$  \hspace{1cm} (10)

$$price(t_k) = \exp\{\log[price(t_{k-1})] + r(t_k)\}$$  \hspace{1cm} (11)

The excess demand $r(t_k)$ is a measurement of how much the demand for a product exceeds the supply of the product. The market depth $\lambda \in (0, 1]$ scales the influence of the excess demand on the new price evaluation. The market clearing equations can be describes as a market maker who collects all market orders from the agents, computes on this basis the nice price and executes all trades [4].

2.4.2 Neighbourhood

Sornett’s and Harras’ agents live on a virtual square lattice with $J := 4$ neighbours, with periodic boundary conditions (Figure 2). In addition to the grid, we implemented a linear neighbourhood with periodic boundary conditions (Figure 3), because we wanted to verify the assertion of Harras and Sornette in [19] that the results of the simulation are independent of the topology. In this neighbourhood each agent is connected to $J := 2$ neighbours on both sides. A comparison of the two version can be found in section 4.2.2.

2.5 Standard Parameter Set

We define a standard parameter set according to Harras and Sornette [4] in table 1.

While apparently Harras and Sornette simulated only $D = 3000$ time steps, we will conduct our simulations with $D = 10000$ time steps.
Figure 2: Neighbourhood grid

Figure 3: Neighbourhood list

| C1 | 1 |
| C2 | 1 |
| C3 | 1 |
| $\lambda$ | 0.25 |
| $\Omega$ | 2 |
| $\alpha$ | 0.95 |
| g | 0.02 |
| nAgents | 2500 |

Table 1: Standard Setup
3 Implementation

Having introduced the model we now want to give some insight into the implementation. We implemented the model twice, once using MATLAB by MathWorks\(^2\) and again using C++ with a third party library. In section 3.2 we explain why we chose two implementations and briefly compare them in section 3.3. Finally we give insight into the initialisation of variables as they might effect the simulation.

3.1 MATLAB

3.1.1 Class structure

We build our model based on an existing framework, which provides basic functionalities for the market, for the agents and for visualising the results. The ultimate goal of this framework is to simulate many agent types in one market. The classes of our model are depicted in diagram 4.

3.1.2 Generation of random numbers

As described in section 2 our model depends strongly on random numbers. As random number generator can differ greatly in the quality of their output, we will describe which random number generators were used.

Matlab uses by default a Mersenne Twister engine with seed 0 [20]. For comparing simulations with different parameters the same seed was used while random seeds for Monte Carlo simulations where used by calling \texttt{rng(’shuffle’)}. We are aware that software generated random numbers are not strictly random in a mathematical sense but they pass a variety of statistical tests. The main advantage of these pseudorandom numbers is that the sequence is repeatable for analysing and testing [21]. A short evaluation of Matlab’s random number generator can be found in Section 3.3.

For the personal information \(\epsilon_i(t_k)\) and the global information \(n(t_k)\) we use the \texttt{normrnd} method [22]. For the update of the agents in each time step we generate a new random permutation with the \texttt{randperm} function [23]. This ensures that the agents are never updated in the same order. The coefficients \(c_1, c_2, c_3\) and \(\omega_i\) are obtained by multiplying a random number returned from \texttt{rand} with the corresponding \(C_1, C_2, C_3\) and \(\Omega\) [24].

3.1.3 Optimisation and Parallelisation

The first step of our code optimisation was to enforce preallocation for all arrays and implement a counter variable, which can be interpreted as the current time step. The speed-up was enormous because Matlab internally copies an array to a new storage location when appending a new element with the \texttt{(end+1)} directive.

\(^2\)http://www.mathworks.com
The second step was about obviating unnecessary function calls by merging smaller functions and avoiding sub-function calls. This was necessary because MATLAB’s object orientation is based on Java and is significantly slower then in other languages like C/C++. We also noticed that it took a lot of time to reference member variables. Introducing temporary variables proofed to be efficient if a function uses a member variable of another class or itself more than once. Solely by code optimisation we were able to get a speedup of 5-8 depending on the machine used.

The goal of the third step was to parallelise the update of the agents. Unfortunately our model did not generate any data and MATLAB issued no errors. The problem occurred when retrieving the decisions of the neighbours. Because MATLAB’s Parallel Computation Toolbox lacks any kind of atomic or critical directives as known from OpenMP, we did not have further options to parallelise this step. We tried vectorising our code, which did not prove to be feasible due to if-statements and loops. Therefore we focused on running multiple simulations in parallel.

The runtime of one simulation increases linearly with the number of agents. Figure 5 was created with 10,000 time steps on the MathCCES system turbine averaging 30 simulations per data point. As setup we choose the lattice neighbourhood with four neighbours per agent.

### 3.1.4 Tests and remaining errors

In our MATLAB model we have been able to replicate the results of Georges Harras and Didier Sornette [4], which we will examine further in chapter 4. The only difference to their model is the price. While Harras and Sornette claim their average price to be the initial price, our average price does not depend on the initial price and is always around 2.3.

The average price of the stock is determined by the total amount of cash divided by the total amount of stocks in the system at $t_0$ before the first simulation step. Despite of in-depth testing no errors were found concerning the initialisation of the price in the market class or the amount of cash and stocks in the agent class.

### 3.2 C++

For a standard setup with 10000 time steps and 2500 agents our Matlab implementation needs about 3 hours of computation time. For a sensitivity analysis with stochastic collocation at least 26000 simulations are required. The total computation time of 3250 days greatly exceeds our contingent of computation time. Therefore we decided to re-implement our model in C++ to profit from the efficient object-oriented programming features and the extensive optimisation tools.
3.2.1 Class structure

The class structure differs from Matlab due to the lack of an existing framework. Figure 6 shows the C++ class setup. We developed a data class to exchange data between different program parts. It also enabled us to save our results to disk in a consistent way. The additional analysis class extracts our quantities of interest as defined in 5.3.

3.2.2 Generation of random numbers

We use the Intel Math Kernel Library 11.0\(^3\) (Intel MKL 11.0) to generate the random numbers for the simulation. This library is vectorised and threaded for high performance computing.

The Mersenne Twister Engine was chosen to preserve a comparability to the Matlab version. As the random numbers depend on the seed they are generated with, we have to implement a seeding strategy ensuring random seeds. On Unix operating systems, /dev/random is a special file in which environmental noise from device drivers and other sources is collected. With this noise, we can generate random seeds [25]. For our implementation, we have chosen the /dev/urandom which uses a random number generator in case not enough noise has been collected to generate seeds.

The personal and global information are obtained by the \textit{vdRngGaussian} with the method set to \texttt{VSL\_RNG\_METHOD\_GAUSSIAN\_ICDF}. The coefficients \(c_{1i}, c_{2i}, c_{3i}\) and \(\omega_i\) were generated with the \textit{vdRngUniform} function with the method \texttt{VSL\_RNG\_METHOD\_UNIFORM\_STD\_ACCURATE}.

The random permutation to update the agents are generated with the \textit{random\_shuffle} from the C++ Standard Library, because the Intel MKL 11.0 is unfortunately not offering this feature. We use the method to shuffle an array containing the numbers from 0 to \(n\text{Agents} - 1\).

3.2.3 Optimisation and Parallelisation

Our first optimisation step was to compute the random numbers for all agents and all steps at once instead of creating a new stream of random numbers for each agent. This lead to a great reduction in computation time. The disadvantage of computing a lot of random numbers at once is the additional memory space.

The first approach to parallelisation was using OpenMP to run multiple simulations at the same time, allowing the IntelMKL only one thread per simulation. This however did not improve the performance considerably. Thus we have decided to run our simulations sequentially and instead allow the Intel MKL library to use multiple threads. On the RWTH compute cluster we observed a great speed up using up to six threads for the random generation. Using more threads does not improve our computation significantly.

\(^{3}\)Further information see https://software.intel.com/en-us/intel-mk
3.2.4 Tests and remaining errors

After reimplementing our MATLAB model in C++ we tested it carefully. Unfortunately it became obvious that our C++ model is not computing exactly the same results as the MATLAB model. The price history has a significantly greater variance than in MATLAB. This often makes it impossible to differentiate a bubble/crash from a normal turbulence in price. In our C++ simulation the price is slowly declining. With an initial value of 1 and 10000 steps the mean of price drops to 0.5.

As a consequence, we cannot detect bubbles or crashes after 3000 steps any more. If we choose a very large $C_1$ coefficient (e.g. $C_1 = 16$) one can see a transient effect in our price history. Furthermore the excess demand $r$ in our C++ implementation is always about three times higher than in our MATLAB implementation.

We have used different tests to find the source of our remaining errors. As a first step we have generated random numbers with MATLAB and imported them into our C++ model. This way we could compare the two implementations directly and make sure our errors are not resulting from using different random number generators. We have examined the various parameters to find an error in our C++ computations, which was not successful. Next we have used unit tests to make sure our code is computing correctly. The idea of unit testing is to isolate a single function of a program and test it with different input sets [26]. For testing we applied a bottom up approach starting with the smallest functions in the code. First we verified that all assignments in the constructors of the agent and the market class are correct, continuing with the agent’s `setup` function which provides configuration capabilities. The last test for the agent class was to assert the regularity of the `update` function, which succeeded. Proceeding similarly for the market class, the low level functions `updateU`, `updatePrice`, `save` and `createNeighbourhood` were tested extensively against a variety of data sets. The most demanding test was designed for the `run` function, which contains the entire control for the simulation steps. However having covered all implemented functions of the agent and the market class we did not find any faults or errors. All test were positive.

In section 3.3 we conduct a comparison of the two implementations.

3.3 Comparison

Compared to the MATLAB implementation, we achieved a speed-up greater than 2000 while preserving the linear run time behaviour with respect to agents and time steps. An average run on the RWTH Compute Cluster takes about 6 seconds.

To compare the two implementations we exported all random numbers from Matlab and imported them into our C++ implementation. As shown in figure 7, the price of the Matlab implementation swings around 2.5 while the price of the C++ implementation fluctuates around 1. You can also see how the blue price drops. We observed that the

\[ \text{http://www.ite.rwth-aachen.de/cms/IT\_Center/Dienste/fuer\_Externe/\~flxt/RWTH\_Compute\_Cluster/} \]
average price drop is depending on the variance of the price. The higher the variance the greater the drop of the average price.

Another difference is that the return in the C++ implementation is three times higher than the Matlab one (see figure 8). The Matlab price run also shows a random bubble/-crash event which is missing in the C++ price run. At this point we are unable to verify which code is still having errors. The Matlab implementation is reproducing the results of the paper concerning crash/bubble observations, while the C++ implementation is fulfilling the requirements concerning the price and all unit tests.

As the generation of random numbers is crucial to our simulations, we compared the random generator of Matlab with the IntelMKL. As shown in Figure 9 and Figure 10 the errors of the mean and the standard deviation are in the same order of magnitude. This plot was generated averaging 50 data samples with different seeds. This leads to the conclusion, that the different random generators should not have any major impact on our simulation results.

3.4 Initialisation

As our agents look back in time, it is obvious that our simulation results depend on the initialisation values. We have done no analysis on how significant this dependency is but for completeness we want to specify and give reason for our choice of initial values. A complete list of variables and their initialisation can be found in table 2.
Variable | Initialisation
---|---
$\sigma_r(t_0)$ | Initialised from uniform distribution $\mathcal{U}(0.000001, 0.1)$, because it was observed to lie in this interval.
$< r(t_0) >$ | Initialised from uniform distribution $\mathcal{U}(-0.01, 0.01)$, because it was observed to lie in this interval.
$u(t_0)$ | Initialised to $u(t_0) = 0$. This value is recomputed before it is used the first time.
$r(t_0)$ | Initialised to $r(t_0) = 0$. This value is recomputed before it is used the first time.
$price(t_0)$ | Initialised to $price(t_0) = 1$, as every agent starts out with one cash and one stock, the initial average value is one.
$cash_i(t_0)$ | Initialised to $cash_i(t_0) = 1$
$stock_i(t_0)$ | Initialised to $nStocks_i(t_0) = 1$
$k_{ij}(t_0)$ | Initialised from normal distribution $\mathcal{N}(0, 1)$, because this roughly imitates the distribution in a herding free state.
$s_i(t_0)$ | Initialised to $s_i(t_0) = 0$. This value is recomputed before it is used the first time.
$v_i(t_0)$ | Initialised to $v_i(t_0) = 0$. This value is recomputed before it is used the first time.
$\omega_i(t_0)$ | Initialised to $\omega_i(t_0) = 0$. This value is recomputed before it is used the first time.
$E_i[s_j(t_0)]$ | Initialised from a discrete uniform distribution $\mathcal{U}\{-1, 1\}$.

Tabel 2: Initialisation of Variables
Figure 4: Matlab class diagram
Figure 5: Runtime vs Agents
Figure 8: Return Comparison with same random numbers

Figure 9: Comparison of the Error in the Mean

Figure 10: Comparison the Error in Standard Deviation
4 Early Simulations

In this section, we present results obtained with our Matlab implementation. We are fully aware, that our code is not working faultlessly. But nevertheless we are able to reproduce most of the results of Harras and Sornette and test some of their claims.

Figure 11: $C1=1$, $C2=1$, $C3=1$, $\lambda=0.25$, $\Omega=2$, $\alpha=0.95$, $g=0.02$, nAgents = 2500, time steps = 10000

Figure 12: $C1=6.3$, $C2=1$, $C3=1$, $\lambda=0.25$, $\Omega=2$, $\alpha=0.95$, $g=0.02$, nAgents = 2500, time steps = 10000

4.1 Reproduction

We will reproduce some results of Harras and Sornette to prove that our implementations works as expected. In terms of an in depth analysis we will set our main focus on results
As claimed by Harras and Sornette the price trajectory is bounded, because there is no money creation. The price trajectory is a "mean-reverting random walk" around its initial value of one [4]. In our case the initial value is supposed to be one, but as you can see in figure 11 it is somewhere around 2.3 respectively 0.8 on a logarithmic scale. We can further confirm that the $\langle k_{ij} \rangle (t_k)$ is always around 0, and has peaks before a crash occurs. Figure 12 displays an example to illustrate this behaviour. Note that $\langle k_{ij} \rangle (t_k)$ is actually

$$\langle k_{ij} \rangle (t_k) = \frac{\sum_{i=1}^{nAgents} \sum_{j=1}^{J} k_{ij}(t_k)}{nAgents \times J}. \quad (12)$$

Harras and Sornette state that the trajectory of $u(t_i)$ is always around 0. However in our model $u(t_i)$ is around 1 or -1. It tends to change between 1 and -1 after a crash, but there are crashes after which the sign of $u(t_i)$ does not change. This can also be seen in figure 12. We have yet to find an explanation why our model behaves as it does, but we observe this behaviour of $u(t_k)$ in both implementations.

Econometrics analysed the returns of stock indices like the DAX\(^5\) or the S&P500\(^6\) and observed that the price returns are not Gaussian distributed. They deviate from this distribution and show so called “fat tails”, making the likelihood of extreme gains and losses on investments greater [6]. While the excess kurtosis of Gaussian distributed returns is zero, our model creates return with an excess kurtosis significantly deviating from zero. Figure 13 shows the excess kurtosis of the returns depending on $C_1$. For each datapoint we averaged the results of 20 simulations with different seeds. It is obvious

\(^5\)German stock index
\(^6\)Standard & Poor’s 500

Figure 13: Average Excess Kurtosis of 20 runs. Setup: $C_2=1$, $C_3=1$, $\lambda=0.25$, $\Omega=2$, $\alpha=0.95$, $g=0.02$, $nAgents = 2500$, time steps = 10000

of the sensitivity analysis in section 5.5.
that our model is not producing Gaussian fitted returns. The excess kurtosis is defined as

$$K := \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^4 \left( \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^2 - 3$$

and further information can be found in [6] and [27].

Figure 14: Average K Comparison. Setup: C2=1, C3=1, $\lambda=0.25$, $\Omega=2$, $\alpha=0.95$, $g=0.02$, nAgents = 2500, time steps = 10000

Figure 15: Price Comparison. Setup: C2=1, C3=1, $\lambda=0.25$, $\Omega=2$, $\alpha=0.95$, $g=0.02$, nAgents = 2500, time steps = 10000

4.1.1 The Excitable Regime

As stated by Harras and Sornette [4] the characteristic trait $C_1$, $C_2$ and $C_3$ are crucial parameters and $C_1$ is the most important one. As we can see in figure 15 we have only
small spikes in the price if we set $C_1 < 6.1$. If we set $C_1 > 6.1$ we see a qualitatively different price trajectory.

With our simulations we were able to confirm the thesis from Harras and Sornette that there exist two different regimes. Below a certain threshold for $C_1$, the regime is “efficient” and the price is always around its equilibrium value. If we set $C_1$ larger than this threshold we enter the “excitable regime”. In this excitable regime the agents tend to imitate each other, which leads to a positive or negative feedback loop and therefore to bubbles and crashes. In this regime the price trajectory can have a large deviation from the equilibrium price. In their paper Harras and Sornette state that the threshold for the different regimes is 3. In our case however the threshold is 6.1. This can be due to a difference in initialisation, as we have said earlier Harras and Sornette do not explain all their initial values in their paper. The difference in this threshold could also be resulting from our price. Our price in MATLAB is always around 2.3, and the price of Harras and Sornette is always around 1.

Figure 14 shows the corresponding $< k_{ij}(t_k) >$ to the price trajectories of figure 15. The peaks in $< k_{ij}(t_k) >$ correspond to the bubbles and crashes of the price. This confirms our thesis that herding behaviour amplifies instabilities in the price.

### 4.1.2 Existence of Herding

As introduced in section 1 herding is a plausible cause for bubbles and crashes. In our model the agent’s “trust” in its neighbours is indicated with the $k_{ij}(t_k)$. In figure 16 herding becomes visible, especially in the middle graph. The XY plane is representing the agents on the lattice, while the average $k_{ij}(t_k)$ per agent is plotted along the Z axis. Clusters of agents trust each other and therefore have high $k_{ij}(t_k)$. At time step $k=717$ early signs of herding behaviour get visible in small clusters of agents with a similar high average $k_{ij}(t_k)$. In the next 80 time steps these clusters grow until in time step $k=797$ the majority of agents trust their neighbours enormously. At this point the price has reached its highest velocity.
4.1.3 Number of Agents

Another variable in our setup is the number of agents. Apparently Harras and Sornette did all their simulations with 2500 agents. We were interested if a change in this variable affects our simulation results. Figure 17 shows that the standard deviation and the mean of the price remain almost unchanged, while the number of agents has an impact on the excess kurtosis. Note that the peak of the excess kurtosis is at 2500 agents. To fully understand the importance of this parameter further analysis is necessary.

4.2 Claim Review

Figure 18: Average Mean price of 20 runs. Setup: C2=1, C3=1, λ=0.25, Ω=2, α=0.95, g=0.02, nAgents = 2500, time steps = 10000

Figure 17: Averages of 20 runs. Setup: C1 = 4, C2=1, C3=1, λ=0.25, Ω=2, α=0.95, g=0.02, time steps = 10000
Figure 19: Average Standard Deviation of 20 runs. Setup: C2=1, C3=1, λ=0.25, Ω=2, α=0.95, g=0.02, nAgents = 2500, time steps = 10000

In the following we will review some claims Harras and Sornette made in [4], some of them were even presented without comprehensible proof.

4.2.1 Different Price Clearing Condition

Harras and Sornette claim in Section 3.7 of [4], that a different price clearing condition like updating the price after recomputing the cash and the stocks of the agents obtains the same qualitative results. To proof this claim we did some exemplary simulations. By visually comparing price trajectories we can confirm this behaviour. Further the excess kurtosis of the returns (See figure 13) and the standard deviation (See figure 19) are qualitatively identical until $C1 \approx 7$, while figure 18 shows that the mean price is substantially higher. In these figures the “Normal Setup” updates the agents with the price from the current time step as introduce in our model description, while the “Update Agents with $price(t_{k-1})$” setup uses the price of the previous time step to compute the agents cash and amount of stock.

4.2.2 Different Neighbourhoods

In section 2.3 of [4] Harras and Sornette claim, that the results are not sensitive to the topology the agents live on. Taking a different approach we examined if the results are unaffected by a total different topology like a line. We can show by example that also a linear neighbourhood has no significant impact. Assuming an extreme linear neighbourhood as in figure 3 with also four neighbours, figure 13, 19 and 18 show that the impact on the excess kurtosis of the return, standard deviation of the price and the mean price are marginal. We conclude that the model is robust to the topology, even though large $C1$ values remain untested.
4.2.3 Different $E_i[s_j(t_k)]$

As mentioned in the section 2.3.6 there are two ideas of how to compute the $E_i[s_j(t_k)]$. Figure 13,19 and 18 show once again that the impact on the excess kurtosis of the return, standard deviation of the price and the mean price are marginal. In the “normal setup” equation 9 is used, therefore the decision is taken directly from the neighbour. In the “Decision Estimated” setup equation 8 is used. Here we estimate the decision of the neighbour by evaluating his opinion against the agent’s personal threshold.
5 Stochastic collocation

5.1 Idea

Harras’ and Sornette’s [4] analysis of their own model is only done by inspecting one or two parameters at once. They investigated the impact of the parameter $C_1$, $C_2$ and $\alpha$ in particular. But we still do not know how the system behaves when we modify, in addition to $C_1$, $C_2$ and $\alpha$, other parameters like $C_3$, $N$, $\lambda$, $\Omega$, $g$, simultaneously. What happens when we increase the risk aversion of the agents (by decreasing $\Omega$)? Is the model still plausible? Do bubbles and crashes occur more often when we increase the amount of (dis)invested money (by increasing $g$)? ...

These questions about the impact of parameters can be summarised as a sensitivity analysis. We want to understand deeply which correlations between input and output variables exist and for which parameter intervals we obtain reasonable results. A sensitivity analysis may give additional insight about bubbles, crashes and limits of the model.

There exists a large number of possible approaches in order to perform a sensitivity analysis. Such as automatic differentiation where we apply the chain rule to any arithmetic operations and elementary functions of the source code in order to measure the impact of parameters with the help of the additional computed derivatives. To apply this method it is needed to overload operations in the implementation and the memory usage per simulation is enormous. Most common sensitivity analysis vary the input parameter for every sample within its interval boundaries after a specific rule and repeat this while observing the outputs. With a higher number of samples we can achieve a higher precision, but this also means a longer runtime. The One-factor-at-a-time (OFAT) analysis was done in [4], but only for $C_1$, $C_2$ with 40 different values and $\alpha$ (three different values) and as mentioned before they did not change more than two parameter simultaneously in one sample. In the scatter method the output is determined after randomly sampling the model over its input distribution [28]. Likewise in the full cartesian grid method an enormous amount of samples has to be simulated depending on the precision and the number of input parameters (Table 4). This results, due to a high sample runtime (Figure 5), in an phenomenal runtime for the whole analysis.

We can consider the implemented model as a black box (see figure 20) with input parameters and output variables, the so called quantities of interest (which we define in 5.3). In our considerations about outputs and inputs we identified seven input parameters as worth to be investigated (see chapter 5.2) and defined eleven quantities of interest as our output (see chapter 5.3). So we can consider the simulation as a function $f$ with

$$f : \Omega \subseteq \mathbb{R}^7 \rightarrow \mathbb{R}^{11}$$

where $\Omega$ is the space of our parameters. Table 4 on page 44 shows, that if we would use a cartesian full grid for our analysis we would need $3.43 \times 10^{10}$ samples which results in 3.000.000 years runtime using the MATLAB implementation (3h per run, see Figure
5) or 2.175 years with the C++ implementation (8s per run) with $10^4$ time steps, 40 simulations in parallel, $\times 10$ each sample with different seeds for Monte Carlo. Numerical analysis regarding functions with a higher dimension as $d = 4$ underlie the “Curse of dimensionality”\textsuperscript{7}, due to the fact that the function evaluations grows exponentially with the dimensionality [30].

The idea of our sensitivity analysis is to use sparse grids (see chapter 5.4) where the number of points grows slowly with the dimension “while preserving the asymptotic error decay of a full-grid interpolation with increasing grid resolution up to a logarithmic factor” [31]. For our 7-dimensional problem we have to evaluate 2,661 points (Table 4 on page 44, Clenshaw-Curtis Grid) on the sparse grids in contrast to the $3.43 \cdot 10^{10}$ samples on the full grid. In addition to that, the generation of the sampling points and the computation of the interpolant can be highly parallelised.

Since the model utilises random variables it is inevitable to repeat every simulation sample several times with different random seeds and average the outputs in order to show reliable dependencies between parameters and main results (Law of Large numbers, Monte Carlo). In the history of model based agents it has happened that model results were statistical artefacts and were disproved by a repeated evaluation of the model with a higher statistical quality. As a consequence, we have to verify that the results are not statistical artefacts which only appear under certain circumstance. Therefore we simulated over $10^5$ time steps, instead of computing only $3.000$ time steps as in [4] and made the agents analysis in 4.1.3.

### 5.2 Input Parameters

The here presented parameters are a selection of the entire input parameters and we define these now as our input parameters. We selected these parameters, because they seem to be the most fundamental and interesting ones. The parameter ranges where set due to logic consideration and experiences of previous runs.

- **$C1$:** Weight for the neighbourhood information with $C1 \in [0, 20]$. We want to observe the behaviour of the system with agents who do not regard their neighbourhood information $C1 = 0$ and also with agents with a extreme trust in the neighbourhood network.

\textsuperscript{7}A term that was coined by Bellmann [29]
• **C2**: Weight for the global information with $C2 \in [0, 8]$. The agents’ behaviour without and with a high weight of the global information should be observed.

• **C3**: Weight for the personal information with $C3 \in [0, 8]$. The agents’ behaviour without and with a high weight of the personal information should be observed.

• $\lambda$: market depth with $\lambda \in [0.1, 1]$. This factor controls the impact of the excess demand on the new price evaluation. This parameter range seems to be reasonable due to experiences of previous runs.

• $\Omega$: agents risk aversion with $\Omega \in [0, 5]$. We want to observe the systems’ behaviour when the agents can no stay passive $\Omega = 0$ (similar to [10]) and agents with a high acting threshold.

• $\alpha$: memory discount factor with $\alpha \in [0.4, 0.999]$. The time steps that an agents remembers corresponds to $\frac{1}{\ln(\alpha)}$ and we obtain the range of approximately $[1, 1000]$ time steps with the effect that we can investigate the impact of short-sighted and long-sighted agents.

• $g$: fraction of cash or stocks that an agents trades per action with a reasonable range of $g \in [0.001, 0.25]$.

### 5.3 Quantities of Interest

The most interesting outcome of each run is the price history, which shows us the computed price over the time steps. We could usually recognise visually and subjectively the quantitative and qualitative bubble-crash pattern. But for our stochastic analysis we needed an objective definition of a crash/bubble and its properties. In addition to the price we also observe the $\max \langle k_{ij} \rangle$ as in [4]. It gives us the averaged trust in the agents’ neighbourhood, which can be interpreted as a measurement of the herding intensity. On this basis, we declare for one sample the following quantities as our outputs. One sample means that we use $10 \times$ the same parameter set, but with different random seeds.

**Quantities for bubbles and crashes**

- **number of crashes**: quantitative measurement of crash occurrence
- **number of bubbles**: quantitative measurement of bubble occurrence
- **crash length**: qualitative measurement of crash duration
- **bubble length**: qualitative measurement of bubble duration
- **crash height**: qualitative measurement of crash impact
- **bubble height**: qualitative measurement of bubble impact

\[ \max \langle k_{ij} \rangle = \max_{t} \langle k_{ij} \rangle (t) = \max_{t} \frac{1}{N^{2}} \sum_{i=1}^{N} \sum_{j=1}^{J} k_{ij} (t) \]
- **crash area**: qualitative measurement of crash area
- **bubble area**: qualitative measurement of bubble area

### General Quantities
- **price**
- **standard deviation of price**
- **max** $\langle k_{ij} \rangle$: qualitative measurement of neighbourhood information importance

#### 5.3.1 Bubble and Crash Quantification

After examining different strategies for bubble/crash detection we have settled on a simple algorithm. We compute the simple moving average $m(t_k)$ (in figures 21 and 22 coloured in blue) which considers the last $n = 800$ steps of the price values.

$$m(t_k) = \frac{1}{n} \sum_{i=0}^{n-1} \text{price}(t_k-i)$$  \hspace{1cm} (14)

This way we take enough data to filter local peaks while still adjusting to long-term change in the price (purple coloured). We take the standard deviation of the price $\sigma_{\text{price}}$ and compute new aid lines $m(t_k) + \sigma_{\text{price}}$ (red coloured) and $m(t_k) - \sigma_{\text{price}}$ (yellow coloured). We then defined a fixed height $h$ that our price has to pass to be counted as a bubble. Using a fixed height ensures us that we do not detect any false bubbles in simulations with a low variance in price. We defined a crash symmetrically, i.e. the price has to pass $-h$ to be counted as a crash. With the help of the intersection points between the price and the aid lines we can compute the area, width and extreme values of each event.

![Price History with simple moving average and threshold](image)

**Figure 21: Bubble and crash quantification example**

| BUBBLE: 1 Position: 876 Maximum: 0.51478 Width: 209 Area: 75.6535 |
| BUBBLE: 2 Position: 2289 Maximum: 0.18577 Width: 133 Area: 17.5464 |
| BUBBLE: 3 Position: 2901 Maximum: 0.0038284 Width: 2 Area: 0.0045693 |
| CRASH: 1 Position: 631 Minimum: 0.12505 Width: 116 Area: 10.599 |
| CRASH: 2 Position: 1142 Minimum: 0.25255 Width: 122 Area: 22.62 |
| CRASH: 3 Position: 2501 Minimum: 0.42191 Width: 175 Area: 49.1798 |

| Listing 1: Quantification computation example result |
5.4 Sparse Grids

Sparse grids are a family of algorithms which help us overcome the “Curse of dimensionality”. They reduce the amount of function evaluations with the help of multidimensional quadrature construction and interpolation. In the following we provide a small derivation of general sparse grids from the full grid and we introduce a few notations we will make use of. All equations can be found in [32].

Let \( \vec{l} \) and \( \vec{k} \) be \( d \)-dimensional multi-indices with the component-wise relational operators,

\[
\vec{l} \leq \vec{k} \iff l_j \leq k_j, \quad 1 \leq j \leq d
\]

with the \( l_1 \)-norm and the maximum-norm,

\[
|\vec{l}|_1 := \sum_{j=1}^{d} l_j, \quad \text{and} \quad |\vec{l}|_{\infty} := \max_{1 \leq j \leq d} |l_j|.
\]

The \( L_p \)-norm is given by

\[
\|f\|_{L_p}^p := \int_{\Omega} |f(\vec{x})|^p \, dx
\]

and for vectors \( \vec{x} \) the 2-norm

\[
\|\vec{x}\|_2^2 := \sum_{j=1}^{d} |x_j|^2.
\]

5.4.1 Linear Interpolation on a Full Grid in one-dimension

We introduce now the interpolation by a linear combination of hat basis functions in the one dimensional case. We will reconstruct this method with hierarchal basis functions
in the next subchapters and eventually we will extend it to the \( d \)-dimensional case to derive Sparse Grids. For the sake of convenience, we consider an arbitrary function \( f : \Omega \to \mathbb{R} \) with \( \Omega := [0,1]^d \) and we assume that the function values are zero on the boundary \( f|_{\partial \Omega} = 0 \). The given parameter space is discretised by equidistant points \( x_i \) with a distance of \( h_n := 2^{-n} \) where \( n \) represents the refinement level. The standard hat function is given by

\[
\varphi(x) = \max(1 - |x|, 0)
\]

and is extended for the one-dimensional case with translation in the \( x \)-direction with the purpose to place a hat function on every single discretisation point

\[
\varphi_i(x) := \varphi(x - i)
\]

The interpolant \( u(x) \) can be computed by the weighted sum of hat basis functions with their corresponding coefficients \( \alpha_i \)

\[
f(\vec{x}) \approx u(\vec{x}) := \sum_i \alpha_i \varphi_i(\vec{x}).
\]

Figure 23 visualises an example of this method. On the left of the figure we can see an arbitrary function on \( \Omega \) and on the right the corresponding piecewise linear interpolant (red dotted) and the different coloured hat basis functions which are scaled by their weights.

![Figure 23: Arbitrary function on \( \Omega \) (left) and its interpolant (red dotted) by a weighted sum of hat basis functions (right), boundaries are considered to be zero](image)

### 5.4.2 Linear Interpolation on a Full Grid with Hierarchical basis functions in one-dimension

For the construction of sparse grids, it is needed to decompose the considered space into hierarchical subspaces. With the help of these subspaces, it is possible to reduce the number of evaluation points compared to a full grid and the accompanying “Curse of Dimensionality” in higher dimensional problems. For the construction of the one dimensional hierarchical decomposition we introduce now the hierarchical index set

\[
I_l := \{ i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, i \text{ odd} \}.
\]
The variable \( l \) describes the level on the hierarchy structure and furthermore we have to adapt the hat function with dilatation. From this, it follows that all basis functions on the same level have the same dilatation see figure 24.

\[
\varphi_{l,i} (x) := \varphi \left( 2^l x - i \right)
\]  

(23)

Thus we can define the set of hierarchical subspace \( W_l \) on the specific level \( l \) as,

\[
W_l := \text{span} \{ \varphi_{l,i} (x) : i \in I_l \}.
\]

(24)

The space of piecewise linear functions \( V_n \) on a full grid with mesh width \( h_n = 2^{-n} \) and with level \( n \) is the join of all its hierarchical subspaces, see figure 24:

\[
V_n = \bigoplus_{l \leq n} W_l.
\]

(25)

Then, the interpolant \( u (x) \in V_n \) is computed by

\[
u (x) = \sum_{l \leq n, i \in I_l} \alpha_{l,i} \varphi_{l,i} (x)
\]

(26)

with the hierarchical weights \( \alpha_{l,i} \) which are uniquely assigned to their basis functions \( \varphi_{l,i} (x) \). The interpolation still uses the all grid points from the full grid.

Figure 24: Grid points \( x_{l,i} \) and the one-dimensional basis functions \( \varphi_{l,j} \) up to a level of \( n = 3 \) on the hierarchical basis (left) and the nodal point basis (right) [32]

### 5.4.3 Linear Interpolation on a Full Grid with Hierarchical basis functions in arbitrary dimensions

In this section we will transform all equations from the one-dimensional interpolation (section 5.4.2) to the arbitrary \( d \)-dimensional case. We will make use of the \( d \)-dimensional
multi-indices $\vec{l}$ and $\vec{i}$ and construct the piecewise $d$-dimensional basis function $\varphi_{\vec{j},\vec{i}}(\vec{x})$ by the product of the one-dimensional basis functions

$$\varphi_{\vec{j},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \varphi_{l_{j},i_{j}}(x_{j}).$$

(27)

We transfer the index set $I_{l}$ to the arbitrary-dimensional case $I_{\vec{l}}$,

$$I_{\vec{l}} := \{ \vec{i} : 1 \leq i_{j} \leq 2^{l_{j}} - 1, i_{j} \text{ odd}, i \leq j \leq d \}$$

(28)

and the level based subspace $W_{l}$ to $W_{\vec{l}}$,

$$W_{\vec{l}} := \text{span} \{ \varphi_{\vec{l},\vec{i}} : \vec{i} \in I_{\vec{l}} \}$$

(29)

and the space of piecewise $d$-linear functions $V_{n}$ is given by,

$$V_{n} = \bigoplus_{|\vec{l}|_{\infty} \leq n} W_{\vec{l}}$$

(30)

which results into a full grid with $(2^{n} - 1)^{d}$ grid points and the interpolant $u(\vec{x}) \in V_{n}$ can be computed by

$$u(\vec{x}) = \sum_{|\vec{l}|_{\infty} \leq n, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}).$$

(31)

For an error estimation with a sufficient smooth $f$ we obtain (c.f. [32]),

$$\| f(\vec{x}) - u(\vec{x}) \|_{L^{2}} \in \mathcal{O}(h_{n}^{2})$$

(32)

but with a computation complexity of,

$$\mathcal{O}(h_{n}^{-d}) = \mathcal{O}(2^{nd}).$$

(33)

5.4.4 Sparse Grids

The idea of Sparse Grids is to reduce the set of subspaces of a full grid to a new set of subspaces with a good cost-performance ratio. The subspaces of the Sparse Grid should have a high contribution to the computation of the interpolation leaving out those subspaces with a small contribution. Since we can estimate bounds for the contribution of the different subspaces, it is possible to make an a-priori selection of subspaces which have a high contribution to the interpolant but have only a small computational cost compared to other subspaces. The selection can be considered as an optimisation problem which is a continuous knapsack problem, because every single subspace has its own computation complexity (mass) and its contribution to the accuracy of the interpolant (use value). The problem is continuous, because of the fact that we can choose an arbitrary refinement level $n \in \mathbb{N}$ for the construction. In particular, we obtain the sparse
grid space with a reduced amount of subspaces by selecting only those subspaces whose
$l_1$-norm $|\vec{l}|_1$ is lower or equal than $n + d - 1$,

$$V^{(1)}_n := \bigoplus_{|\vec{l}|_1 \leq n + d - 1} W_{\vec{l}}$$

(34)

$$u (\vec{x}) = \sum_{|\vec{l}|_1 \leq n + d - 1, \vec{x} \in I_{\vec{l}}} \alpha_{\vec{l}, \vec{x}} \phi_{\vec{l}, \vec{x}} (\vec{x})$$

(35)

Equation (35) describes eventually the computation of the sparse grid interpolant with
$u (\vec{x}) \in V^{(1)}_n$. In figure 25 we can see a comparison of the full grid (left) and the sparse
grid (right) both with the same refinement level $n = 5$.

![Full Grid, n=5](image1)

![Sparse Grid, n=5](image2)

Figure 25: Full Grid (left), Sparse Grid with boundary consideration (right), refinement
level $n = 5$, figure generated with [33]

Figure 26 gives us a visualisation of the selection in the two-dimensional case. Every big
square on the left represents a subspace from the full grid $V_3$, but only the black painted
squares are a-priori selected and build the sparse grid with its associated sample points.
The grey ones are those subspaces with a hight computational cost but with a relative
low contribution to the interpolant and thus they are not considered in the sparse grid.
Finally, we archive on the top right of the figure the Sparse Grid with $n = 3$ which is
a join of all a-priori subspaces (in black). The full grid on the bottom right in grey is
archived if we would also join the grey subspaces.

Compared to the full grid with the same refinement level $n$, the asymptotic accuracy
decreases slightly from $O (h_n^2)$ to $O \left( h_n^2 (\log h_n^{-1})^{d-1} \right)$ [35] with respect to the $L_2$-norm,
but the computation cost decreases enormously from $O \left( 2^{nd} \right)$ to $O \left( 2^n n^{d-1} \right)$ which finally
leads to the circumvention of the “Curse of dimensionality”.

5.4.5 Clenshaw-Curtis Rule

In our sparse grid we use the Clenshaw-Curtis quadrature rule, since the CC-grid has
a good compromise between computational cost and accuracy. It is well suited for both
lower and higher dimensional problems [36]. Referred to our simulation model, we chose
the CC-Grid type with boundary in order to observe the behaviour of the model when we cancel out e.g. the neighbourhood information \( C1 = 0 \). The quadrature rule uses \( n + 1 \) nodes (one-dimensional case) which are the extreme points of \( T_n(x) = \cos(n \arccos x) \) in \( \Omega := [-1, 1] \) and the zeros of the Chebyshev polynomial of the second kind. The discretisation points are given by (equations taken from [37])

\[
x_j = \cos \frac{j\pi}{n}, \quad j = 1, \ldots, n - 1
\]

\[
x_0 = 1, \quad x_n = -1
\]

and the weights of each node can be computed by

\[
\alpha_{jc} = \frac{c_j}{n} \left(1 - \sum_{k=1}^{\lfloor n/2 \rfloor} \frac{b_k}{4k^2 - 1} \cos \left(2k\frac{j\pi}{n}\right)\right), \quad j = 0, \ldots, n
\]

with

\[
b_k = \begin{cases} 
1 & \text{if } k = n/2 \\
2 & \text{if } k < n/2 
\end{cases}
\]

and

\[
c_j = \begin{cases} 
1 & \text{if } j = 0, n \\
2 & \text{otherwise.}
\end{cases}
\]
Figures 28 and 27 give us an impression how a CC-Grid would look like in the two-dimensional case and in the three-dimensional case on $\Omega$. We can recognise with the increasing refinement level $n$ that the construction of the Sparse Grid has a nested structure.

Figure 27: two-dimensional CC-Grids with standard basis function and with differing refinement level $n$ on $\Omega := [0, 1]^2$, figure generated with [33]

Figure 28: three-dimensional CC-Grids with standard basis function and with differing refinement level $n$ on $\Omega := [0, 1]^3$, figure generated with [33]

Table 3 provides specific data of the points and weights for an one-dimensional example. With the points $x_j$ and weights $\alpha_j$ on the domain $\Omega := [-1, 1]$, whereas $\tilde{x}_j$ and $\tilde{\alpha}_j$ are transformed to $\tilde{\Omega} := [0, 8]$. Note that the sum of the weights is $\sum_{j=0}^{n} \alpha_j = 2$ and $\sum_{j=0}^{n} \tilde{\alpha}_j = 8$, corresponding to the integral of $f(x) = 1$ over $\Omega$, respectively $\tilde{\Omega}$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>$x_j$</th>
<th>$\alpha_j$</th>
<th>$\tilde{x}_j$</th>
<th>$\tilde{\alpha}_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>0.05</td>
<td>8.0</td>
<td>0.266667</td>
</tr>
<tr>
<td>1</td>
<td>0.707107</td>
<td>0.56667</td>
<td>6.8248</td>
<td>2.13333</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>0.76667</td>
<td>4.0</td>
<td>3.2</td>
</tr>
<tr>
<td>3</td>
<td>-0.707107</td>
<td>0.56667</td>
<td>1.17157</td>
<td>2.13333</td>
</tr>
<tr>
<td>4</td>
<td>-1.0</td>
<td>0.05</td>
<td>0.0</td>
<td>0.266667</td>
</tr>
</tbody>
</table>

Tabel 3: one-dimensional Clenshaw-Curtis example, $n=4$
5.4.6 Grid Complexity comparison

For the estimation of the overall runtime of the stochastic collocation it is necessary to know the number of sample points of the grid types and furthermore we want to emphasise here why we chose the Clenshaw-Curtis grid type. When we use \( n \) as the refinement level and \( d \) as the number of dimensions as before, we can estimate the number of grid points that we have to evaluate by the following equations [38].

The number of samples in a full grid without boundaries is given by,

\[
|G_n^d| = (2^n - 1)^d
\]

and for a full grid including the boundaries,

\[
|G_n^d| = (2^n + 1)^d
\]

and thus an overall complexity of \( O(2^{nd}) \) function evaluations.

The complexity of a regular inner sparse grids without boundary points is given by

\[
|G_n^d| = 2^n \left( \frac{n^{d-1}}{(d-1)!} + O(n^{d-2}) \right)
\]

where the \( O(\cdot) \) notation covers terms of lower order in the refinement \( n \) and hides \( d \)-dimensional coefficients. Whereas the sparse grids with boundaries is bounded by

\[
|G_n^d| \leq 3^d|G_n^d| = 3^d \left( 2^n \frac{n^{d-1}}{(d-1)!} + O(n^{d-2}) \right)
\]

and we get an overall degrees of freedom of \( O(2^n n^{d-1}) \).

In table 4 we can recognise the different growth rates of the evaluation points for the different grid types. The growth rate of the nodes of the CC-Grid type is enormously lower than the amount of samples in the standard sparse grid, although they have the same asymptotic complexity. If we would consider higher dimensional problems and functions with high evaluation costs it is absolutely essential to consider the absolute amount of sample points and is not sufficient to consider only the asymptotic behaviour of the different growth rates.

5.4.7 Sparse Grid Implementation

For our sparse grid implementation we used the Tasmanian Sparse Grid Generator[39] which is a toolkit, implemented in C++. The generator computes the sample nodes on basis of the number of dimensions \( d \), the parameter space \( \Omega \), refinement level \( n \) and the one-dimensional quadrature rule. In addition to the discretisation points, a grid file is being written containing the basis information about the grid in order to regenerate the samples and their weights for a later interpolation. Furthermore, we can store the results of the function evaluation into the grid file connecting each sample input to its
output. When all outputs are completely stored into the grid file, the file is ready for the interpolation. On the HPC Cluster, we encountered the problem that batch-jobs crash or were aborted due to exceeding memory limit or other unknown reason. To avoid losing all previous simulation results we distributed all sample points into input files each containing 200 samples. A batch-job is responsible for a single input file, so that if a batch job crashes only a small number of computations has to be redone. Furthermore, the batch job is able to distribute its work up to seven CPUs and we archived an overall runtime of $37 \, h$ for 26,610 sample points. For the management of inputs and outputs we use the data class (see chapter 3.2).

### Listing 2: Input file example

10000;2500;1;1;0.55;2.5;0.9;0.125;4;4;4;Run1;0;0;0;0;0;0;0;0;0;0;0;
10000;2500;1;1;0.55;2.5;0.9;0.125;4;4;0;Run2;0;0;0;0;0;0;0;0;0;0;0;
10000;2500;1;1;0.55;2.5;0.9;0.125;4;4;8;Run3;0;0;0;0;0;0;0;0;0;0;0;
...

5.4.8 Adaptive Sparse Grids

Assuming that the considered function shows significantly differing characteristics, the strategy of adaptive refining sparse grids is a possible faster and more precise way to perform the analysis [32]. Adaptive sparse grids try to refine the grid in regions of high curvature and spend less time in regions with smooth variation. The result is an interpolant with a smaller error and a grid with less points. The adaptation is computed iteratively level by level after the evaluation of all points from one level is done. It is possible that regions with local characteristics are not “discovered” at the first levels and
thus not more concerned in the next levels. Since we generate the grid at the beginning of our analysis and write it into several files for safer a batch job execution on the HPC Cluster (see chapter 5.4.7), an adaptive sparse grid could be only implemented with a high programming effort including iterative rewriting of the grid files and the usage of MPI.

5.4.9 Evaluation

The Tasmanian Sparse Grid Generator provides an interpolation method which returns the value of the interpolant $\tilde{y} \in \mathbb{R}^{11}$ at the provided point $\tilde{x} \in \mathbb{R}^{7}$ as defined by equation (35). The idea is to visualise correlations of input parameter and output data with the help of 2D, 3D plots or even 4D plots where the fourth-dimension is a colour in the surface plot. We select up to two parameters and up to two output variables which we want to investigate. Then we use the standard parameter set (see chapter 2.5) and we iterate over the selected parameter and its upper and lower bounds while observing the selected output variables. The results are written into plot files and are plotted with gnuplot. Table 5 gives us an overview about the different evaluation methods. After analysing the first results we would consider to change some parameters of the standard parameter set as well and repeat the evaluation methods in order to compare results with more than two differing input parameters.

<table>
<thead>
<tr>
<th>Evaluation method</th>
<th>#parameter</th>
<th>#output</th>
<th>Plot setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Evaluation</td>
<td>1</td>
<td>1</td>
<td>line plot, x-axis: parameter, y-axis: output variable</td>
</tr>
<tr>
<td>3D Evaluation</td>
<td>2</td>
<td>1</td>
<td>surface, x-axis: first parameter, y-axis: second parameter, z-axis: output variable</td>
</tr>
<tr>
<td>4D Evaluation</td>
<td>2</td>
<td>2</td>
<td>surface with colour, x-axis: first parameter, y-axis: second parameter, z-axis: first output, surface colour: second output</td>
</tr>
</tbody>
</table>

Table 5: Evaluation methods

5.4.10 Implementation test

To verify the implementation of the sparse grid generator, the writing and reading of data files with the data class and the evaluation routine, we used known functions and computed their interpolant and the according errors. In figure 30 we used the rosenbrock function [40]

$$f(x,y) = (1 - x^2) + 100(y - x^2)^2$$

(45)
on $\Omega = [-2,2] \times [-1,3]$ with CC-Grid and refinement level $n = 7$ obtaining 49 nodes. The interpolant was evaluated with 2600 grid points in order to plot the surface. The
error is negligible and only based on computation inaccuracy, because the polynomial $f$ with order 4 is interpolated exactly. Figure 30 shows on the left the original function, in the middle the interpolant and on the right the interpolation error. The plots are created with the 3D evaluation routine as described in 5.4.9.

The next function is non-polynomial and has differing characteristics (see figure 31) and is known to be ill-suited for interpolation [41].

\[ f(x, y) = \frac{1}{|0.5 - x^4 - y^4| + 1} \]  

(46)

On $\Omega = [0, 1] \times [0, 1]$ with $n=50$ and with 17 input files containing 3201 evaluation points, we get a maximal error at the discretisation points of $\max_{i=1,\ldots,2600} |f(\vec{x}_i) - u(\vec{x}_i)| = 0.6$.

Figure 31 shows again the original function, the interpolant and the absolute error with the 3D evaluation routine and with the 2D evaluation routine in figure 32.

The small error of the interpolants of both test functions verified that the sparse grid, data handling, interpolation on the sparse grid and the visualisation methods of our implementation seem to work well. But we can also see that the error of the interpolant of the function described by equation 46 is more inaccurate than the smooth polynomial function. In addition to that, the non-polynomial function was evaluated by a sparse grid with a higher refinement level ($n = 50$) in compared to the polynomial function ($n = 7$). As a consequence, we can assume that if our simulation function has strong differing characteristics, the final interpolation might be inaccurate.

5.5 Results

We are fully aware of the fact that the C++ implementation does not show the same behaviour as the MATLAB implementation (see chapter 3.2.4) and unfortunately the model showed some unreasonable behaviour when some extreme parameter values were used to perform the simulation. Nevertheless, we performed the sensitivity analysis with the C++ implementation and the results presented here should be treated with caution.

For our simulations, we set the number of agents to $N = 2500$, applied the sparse grid generator to the input parameters with a refinement level $n = 5$ obtaining 2661 evaluation points and executed each sample 10× with different random seeds while averaging the results as described in chapter 5.4.7. In the following we want to present some few noteworthy correlations between the input parameters and the output parameters.

5.5.1 The Excitable Regime

Harras and Sornette state in [4] that their “efficient regime” appears for a $C1 > 3$ and that at this point that the tendency to imitate others is strong enough that large price deviation from the equilibrium price appear. We can observe some similar but shifted behaviour in our results. In figure 34 we can see that registered amount of crashes is increasing with an increasing $C1$ and the maximum of crashes can be found at $C1 = 10$ with average of 30.5 crashes per run. Unfortunately, we are not able to explain the
following drop. It may be caused by the interpolation and maybe a higher refinement level is needed for reliable evaluation. Another explanation could be that the model does not represent any reasonable behaviour for this height $C_1$ values. In [4] the claim was made, that with an increasing $C_2$ the $C_1$ must be also be increased to obtain an excitable regime. Figure 33 shows us the inter parameter correlation between $C_1$, $C_2$ and the dependent measured average bubble length per run. We can see that the bubble length increases with an increasing $C_1$, especially for higher $C_2$ values. The maximum is eventually reached with $C_1 = 20$ and $C_2 = 8$. For a $C_2$ between 2 and 0, we obtain the lowest average bubble length.

### 5.5.2 Influence of $\Omega$

Figure 35 shows the behaviour of the average bubble count per run with a differing $\Omega$ on the x-axis. We can recognise that maximal bubble count is archived when the agents acting threshold is decreased to $\Omega = 1$ (standard parameter $\Omega = 2$). This seems to make sense due to the fact that the agents with a lower acting threshold are more responsive for neighbourhood information and herding behaviour. The minimum of bubbles per run can be found at the increased $\Omega = 3$ which reduces the agents trading occurrence and lowers the risk of herding behaviour.
Figure 29: Sparse Grid Implementation
Figure 30: Rosenbrock test function, 3D evaluation method

Figure 31: Test function (equation 46), 3D evaluation method

Figure 32: Test function (equation 46), 2D evaluation method
Figure 33: Average Bubble Length dependent on differing C1 and C2

Figure 34: Count of Crashed depended on C1

Figure 35: Count of Crashed depended on Omega
6 Conclusion and Outlook

In this paper we analysed the model by Harras and Sornette and tried to prove or refute their results. We were able to reconstruct phenomena such as herding and the excitable regime. Further, we dis-proved the behaviour of \( u(t) \) and showed with our sensitivity analysis that the model is not as robust as Harras and Sornette thought, even though a critical judgement has to be applied to our results.

In our early simulations (c.f. section 4) we show that the model produces non-Gaussian fitted returns, so-called fat tails. We are able to present clear signs of herding behaviour among the agents and additionally to establish the existence of an excitable regime with frequent bubbles and crashes. Furthermore we confirmed that the model is insensitive to the topology of the agents, whereas the influence of the number of agents \( N \) needs more research.

We could not perform the sensitivity analysis as planned because of the differing behaviour of the implementations. Due to the unexpected transient behaviour of the price in the C++ implementation, we refrain from presenting more inter parameter correlations. Although these problems exit we were able to depict some dependencies between input and output parameters. The stochastic collocation was a well suited tool to analyse the model because it allowed us to treat the model as a black box function mapping parameters to quantities of interest. Utilising sparse grids facilitated us to bypass the Curse of Dimensionality.

It was possible to achieve an enormous speed up switching from Matlab to C++, while maintaining the linear dependency of the runtime on agents and time steps. The sole code optimisation of the Matlab code gave us also an acceptable speedup.

This model offers many possible extensions. Even though the model was never intended to do so, one could import real market data and try to recognise bubbles by monitoring \( \langle k_{ij}(t_k) \rangle \). A difficulty would be to model the global news stream. A possibility would be to use a sentiment analysis of popular economic newspapers like the New York Times and the BBC.

Before extending the model, we will find the possible error in the code and re-examine the results of the stochastic collocation.

Another interesting analysis would be an in-depth evaluation of the changes in the results when the number of agents and/or the number of time steps is increased. As concluded from section 4.1.3 the number of agents is an important parameter. Further one can decrease the \( \Delta t \) and conduct an asymptotic analysis. Potentially one can extract the underlying differential equations for additional analysis.
Acknowledgments:

We would like to thank our advisors Torsten Trimborn and Philipp Otte for the help, proofreading and advice during the development of this project.
Literatur


