Swiss Federal Institute of Technology Zurich

Seminar for Statistics

Master Thesis Summer 2015

Philip Berntsen

Particle filter adapted to jump-diffusion model of bubbles and crashes with non-local crash-hazard rate estimation

Submission Date: July 30th 2015

Adviser:	Dr. Markus Kalisch
Supervisors:	Prof. Didier Sornette
	Prof. Yannick Malevergne

Abstract

Crashes in the financial sector probably represent the most striking events among all possible extreme phenomena. The impact of the crises have become more severe and their arrivals more frequent. The most recent financial crises shed fresh light on the importance of identifying and understanding financial bubbles and crashes.

The model developed by Malevergne and Sornette [2014] aims at describing the dynamics of the underlying occurrences and probability of crashes. A bubble in this work is synonymous with prices growing at a higher rate than what can be expected as normal growth over the same time period. A non-local estimation of the crash hazard rate takes into account unsustainable price growth, and increases as the spread, between a proxy for the fundamental value and the market price becomes greater. The historical evaluation of the jump risk is unique and expands the understanding of crash probability dynamics assumed embedded in financial log-return data.

The present work is mainly concerned with developing fast sequential Monte Carlo methods, using **C++**. The algorithms are developed for learning about unobserved shocks from discretely realized prices for the model introduced by Malevergne and Sornette [2014]. In particular, we show how the best performing filter - auxiliary particle filter - is derived for the model at hand. All codes are accessible in the appendix for reproducibility and research extensions.

In addition, we show how the filter can be used for calibration of the model at hand. The estimation of the parameters, however, is shown to be difficult.

Contents

1 Introduction 1

2 Jump-diffusion model of bubbles and crashes with non-local behavioral self-referencing 3

2.1 Jump-diffusion models . . . 3

2.2 Model of bubbles and crashes with non-local behavior . . . 4

2.3 Mispricing and inherent jump probability . . . 8

3 Particle filtering 10

3.1 Filtering . . . 10

3.1.1 Importance sampling . . . 13

3.1.2 Sequential importance sampling . . . 13

3.1.3 Sequential importance resampling . . . 15

3.1.4 Auxiliary particle filter . . . 15

3.2 Derivation of the filtering algorithm . . . 16

3.2.1 Jump times . . . 17

3.2.2 Jump sizes . . . 17

3.2.3 Filtering weights . . . 19

3.3 Convergence of particle filters . . . 20

4 Parameter estimation 21

4.1 Maximum likelihood . . . 21

4.1.1 Expectation-maximization algorithm . . . 22

4.2 State augmentation . . . 25

4.2.1 Fixed parameters . . . 26

4.2.2 Artificial dynamics . . . 26

4.2.3 Priors . . . 27

5 Simulations 28

5.1 Model simulation and stylized facts . . . 28

5.1.1 Model simulation . . . 28

5.1.2 Stylized facts . . . 29

5.2 Filtering of the states and convergence . . . 36

5.2.1 Filtering of the states . . . 36

5.2.2 Convergence of the particle filter . . . 36

5.3 Parameter estimation . . . 40

5.3.1 Parameter cross-sections . . . 40

5.3.2 State augmentation . . . 40

5.4 Conclusion of simulations . . . 41

6 Conclusion 46

Appendices **51**

A Pseudo code **52**
A.1 SIR particle filter algorithm 52
A.2 APF particle filter algorithm 52
A.3 Fixed state augmentation 53

B Code **55**
B.1 SIR C++ code 55
B.2 APF C++ code 61
B.3 Fixed state augmentation C++ code 69

List of Figures

5.1 Simulated model of bubbles and crashes with non-local behavior 31
5.2 Volatility and mispricing 32
5.3 Crash hazard rate density plot 33
5.4 Log returns and squared log returns with corresponding correlograms 34
5.5 Kurtosis and skewness of raw returns from the simulated series 35
5.6 SIR performance of estimating jumps and jump sizes 37
5.7 APF performance of estimating jumps and jump sizes 38
5.8 Convergence of SIR particle filter 39
5.9 Cross section of the log-likelihood space (α,β) 43
5.10 Cross section of the log-likelihood space a 44
5.11 State augmentation with fixed parameters 45

Chapter 1

Introduction

Throughout financial market history, there are numerous examples showing phases of extraordinary growth in prices, known as bubbles, followed by a crash. A bubble is created whenever the price of an asset rises above the fundamental value. The fundamental value is notoriously hard to estimate and, therefore, also the determination of whether extraordinary price growth is in fact a bubble. In the works of Malevergne and Sornette [2014] a bubble is quantified as growth in asset prices that exceed what can be expected over a certain period in time. This in turn renders an increase in the crash probability. The estimated growth rate is based on the available historic log-returns and an historic anchoring point by which the current price is compared gives a proxy for the fundamental value.

The model introduced by Malevergne and Sornette [2014] aims at explaining the dynamics of crashes. Estimation of the crash-hazard rate is based on the above commonly held conception. The modeling of real financial log return is said to depend on the volatility, crash probability and lastly jumps. The disentanglement of volatility and jump is important as they are compensated differently with respect to risk and expected return.

One of the main model strengths is the persistent crash-hazard rate. The historic estimation procedure allows for enduring positive crash probability. Comparable models, such as rational expectation models lack this property. Here the crash-hazard rate becomes zero as soon as the conditional expected return becomes zero. This comes from proportionally relating the conditional expected return with the crash-hazard rate, and is clearly wrong. Furthermore rational expectation models assume that despite explosive paths in the asset prices, the non-arbitrage condition remains true. Traders possessing different risk profiles, becoming more prominent in periods of market excitement, clearly contradicts this assumption. Malevergne and Sornette [2014] model allows for heterogeneous collection of traders. This grant market players to have different views on the timing of the crash and therefore diverse exit strategies. This is in line with typical behavior of crashes as they are formed over a series of successive drawdowns. Chapter 2 introduces the exciting model at hand.

The work in this thesis is concerned with adapting sequential Monte Carlo methods, in particular, particle filters to estimate the latent states (jump occurrences and jump sizes) from discretely realized asset prices using the Malevergne and Sornette [2014] model. The ultimate goal, however, is to use the filtering algorithms for parameter estimation. The underlying idea of filtering algorithms is to estimate the predictive distribution of the unobservable states given historic asset price information. They are especially powerful in the case of complex systems in which the predictive distribution, known as the filtering density, is intractable. From an initial sample, the filter is used to select, or filter, the most likely particles thereby obtaining a discrete approximation of the filtering density. In chapter 3 we thoroughly explain different filtering algorithms before developing a more advanced method, known as auxiliary particle filter (APF), for the present model.

In a next step, we propose and investigate algorithms for model calibration using the filter. Chapter 4 investigates two methods for parameter estimation. First the expectation-maximization (EM) algorithm is developed for maximizing the log likelihood. Next, a simpler alternative, known as state augmentation is pursued in efforts of jointly estimate the parameters and states.

In order to check model applicability as well as performance of the filter, chapter 5 is devoted to a simulation study. The conclusion drawn from this study were uplifting. Firstly, the model works very well in reproducing the stylized facts drawn from real log returns. Secondly, the algorithm developed for the model at hand works efficiently in identifying important movements in the simulated data. Sadly, the calibration did not yield the desired result. Even if the calibration remains unsolved, it is believed that further development, utilizing the algorithms in this master thesis, of more advanced algorithms for jointly learning about parameters and states will improve the results.

Chapter 2

Jump-diffusion model of bubbles and crashes with non-local behavioral self-referencing

In the following chapter the discrete time jump-diffusion of bubbles and crashes with non-local crash-hazard rate estimation, developed by Malevergne and Sornette [2014], is presented. The model belongs to a class known as jump-diffusion models, which will briefly be explained in section 2.1. Section 2.2 consider the dynamics of the jump-diffusion model at hand and focuses on the derivation of the different terms. The chapter closes with a discussion on the crucial benefits of the model and a short comparison with similar models.

2.1 Jump-diffusion models

Jump-diffusion models have since the introduction by Merton [1976], received a great deal of attention. They are used to capture discontinuous behavior in financial log-return data and shown to be very attractive in understanding the risk-return relationship resulting from the disentanglement of diffusion and jump risks. A true purist, believing in pure diffusion models without jumps, may argue that log-returns are inherently continuous and that the apparent discontinuity comes from the fact that observations are made in discrete time. Although true, they fail to realize that the task is not really to identify whether the price trajectory is discontinuous or not, but rather propose a model which reproduces the realistic properties of price behavior at the time scale of interest. In a pure diffusion model unpredictable market moves, corresponding to the normal perception of risk, is difficult to capture. This is why jump diffusion models are preferable. As a further motivational remark, Bates [1991] imposed a jump-diffusion model on the log returns of the S&P500 future options and found systematic behavior in the expected number of drawdowns before the 1987 stock market crash. Maybe then, these models are able to replicate the reality in such a way that crashes can be determined ex-ante.

More precisely, the general jump-diffusion model assumes that log-prices are a mixture between a Brownian motion and a jump size random variable controlled by a point process. It is therefore assumed that log-prices, $Y_t = \log S_t$, and the underlying state variables L_t

jointly solve:

$$dY_t = \mu^s(L_t)dt + \sigma^s(L_t)dW_t^s + d(\sum_{n=1}^{N_t^s} Z_n^s) \ , \tag{2.1a}$$

$$dL_t = \mu^l(L_t)dt + \sigma^l(L_t)dW_t^l + d(\sum_{n=1}^{N_t^l} Z_n^l) \ , \tag{2.1b}$$

where W_t^s and W_t^l are potentially correlated Brownian motions, N_t^s and N_t^l are point processes with predictable intensities $\lambda^s(L_t)$ and $\lambda^x(L_t)$, and Z_n^s and Z_n^x are jump size random variables with distributions $\Pi^s(L_t)$ and $\Pi^l(L_t)$ respectively. As an example, we may take S_t to be the S&P500 equity index, L_t as it's stochastic variance, Z_n^s the jumps in prices and Z_n^l the jumps in volatility. Omitting jumps in volatility and prices reduces the model to the well-known stochastic volatility model. Utilizing the full structure is, therefore, known as a stochastic volatility model with jumps. For feasible calibration one normally assumes a Poisson point process with independent increments and constant jump intensity.

A solution to the continuous time system of equations is presented in Johannes, Polson, and Stroud [2009] and, using a single discretization step, given by:

$$r_{t+1} = \mu_t^s(L_t) + \sigma_t^s\epsilon_{t+1}^s + Z_{t+1}^s J_{t+1}^s \ , \tag{2.2a}$$

$$L_{t+1} = L_t + \mu_t^l(L_t) + \sigma_t^l\epsilon_{t+1}^l + Z_{t+1}^l J_{t+1}^l \ , \tag{2.2b}$$

where $r_{t+1} = Y_{t+1} - Y_t$, ϵ_{t+1}^s and ϵ_{t+1}^l are iid standard Gaussian, and J_{t+1}^s and J_{t+1}^l are Bernoulli random variables with success probability $\lambda_s(L_t)$ and $\lambda_l(L_t)$ respectivelly.

The driving force behind the introduction of jump-diffusion models has been on developing an option pricing model. Such a model aims at capturing the features of option prices quoted in the market. They are also used to compute hedging strategies and quantify the risk associated with a given position. Compared with other models, jump-diffusion models are superior for pricing and hedging Cont and Tankov [2004].

2.2 Model of bubbles and crashes with non-local behavior

In this section, the model introduced by Malevergne and Sornette [2014] is considered in more detail. They look at a discrete-time economy in which two assets are traded; a risky- and a risk-free asset. The risk is defined on a filtered probability space $(\Omega, \mathcal{F}, \mathcal{P})$. The log-returns, $r_t = \log \frac{S_t}{S_{t-1}}$, where S_t is the price of the risky asset at time t, are assumed to follow the consecutive stochastic equation

$$r_t = \mu_t + \sigma_t \cdot \epsilon_t - \kappa \cdot J_t \cdot I_t \ , \tag{2.3}$$

where

- μ_t denotes the drift,
- σ_t the volatility at time t,
- $\epsilon_t \overset{iid}{\sim} \mathbb{N}(0,1)$ Gaussian,
- κ the average jump size assumed to be strictly positive,

- $J_t \sim \exp(1.0)$, exponentially distribution,
- $I_t \sim \text{Bernoulli}(\lambda_t)$, where λ_t is the inverse logit function of a constantly changing mispricing index and denotes the conditional success probability

$$\lambda_t = P[I_t = 1|\mathcal{F}_{t-1}] \ , \tag{2.4}$$

- and, lastly, it is assumed that ϵ_t ,J_t and I_t are independent conditional on $\mathcal{F}_{t-1}$.

The remaining part of this section will be devoted to analyzing each term in equation 2.3.

Firstly, the conditional expected return of the risky asset, determining the risk premium investors expect above the risk-free rate, is assumed to be constant and given by

$$\begin{aligned} \bar{r} &\overset{!}{=} \mathbb{E}[r_t|\mathcal{F}_{t-1}] \ , && (2.5a)\\ &= \mathbb{E}[\mu_t + \sigma_t \cdot \epsilon_t - \kappa \cdot J_t \cdot I_t|\mathcal{F}_{t-1}] \ , && (2.5b)\\ &= \mu_t - \kappa \cdot \mathbb{E}[J_t|\mathcal{F}_{t-1}] \cdot \mathbb{E}[I_t|\mathcal{F}_{t-1}] \ , && (2.5c)\\ &= \mu_t - \kappa \cdot \lambda_t \ . && (2.5d) \end{aligned}$$

Where $\bar{r}$ defines the risk-free rate in a risk-neutral framework and the unconditional expected return of the risky asset in a risk-averse world. Note that μ_t increases linearly as a function of the crash-hazard rate. This is perfectly in line with the risk-return relationship: investors require greater remuneration for investing in assets with increased uncertainty. κ may be viewed as a measure of investors sensitivity to an increase in the crash-hazard rate. The greater remuneration from higher risk, along with the crash not being a certain deterministic outcome of the bubble, means that it is rational for traders to remain in the market during a bubble phase.

Next σ_t denote what is commonly known as the volatility. Real volatility is continually changing in a partly predictable manner, due to the presence of clustering. The GARCH(1,1) process enables recovery of these stylized facts and is why it's sensible to model the conditional standard deviation accordingly. More formally, $\sigma_t \cdot \epsilon_t$ is said to follow a GARCH(1,1) process if

$$\sigma_t^2 = w + \alpha \cdot (r_{t-1} - \bar{r})^2 + \beta \cdot \sigma_{t-1}^2 \ . \tag{2.6}$$

The GARCH process, therefore, allows the squared volatility, σ_t^2, to depend on previously squared volatilities as well as previous squared values of the process. Hence, volatilities are large if either $|r_{t-1}|$ or σ_{t-1} are large. The GARCH(1,1) process is covariance-stationary white noise process if and only if $\alpha + \beta < 1$. The variance of the covariance-stationary process is then given by

$$\bar{\sigma}^2 = \frac{w}{1 - \alpha - \beta} \ . \tag{2.7}$$

For ease of modeling purposes, we will proceed with the following dynamics for the volatility

$$\sigma_t^2 = \bar{\sigma}^2 \cdot (1 - \alpha - \beta) + \alpha \cdot (r_{t-1} - \bar{r})^2 + \beta \cdot \sigma_{t-1}^2 \ . \tag{2.8}$$

The last term in 2.3 represents the crash. The realization of a jump is determined by the Bernoulli random variable, I_t, and it's success probability is time varying. The dynamics for the crash probability in the model incorporates a non-local estimation procedure and is both unique and revolutionary in terms of existing jump-diffusion models. Non-local implies that the estimation takes into account unsustainable price growth that has happened over longer historic time periods. Consider growth in an asset price, S_t, over a certain time period $[t-\tau, t]$. This growth is then compared with what can be considered as sustainable over the same period. The difference gives us the mispricing metric

$$\delta_{t,\tau} = \frac{S_t}{S_{t-\tau} \cdot e^{\tau \bar{r}}} \ , \tag{2.9}$$

where $e^{\tau \bar{r}}$ denotes the continuous compounding. This representation corresponds to an anchoring on price. The initial price is taken to be an approximation of the asset's fundamental value. Using the log-logistic function translates $\delta_{t-1,\tau}$ into the crash-hazard rate denoted by

$$\lambda_t = \frac{1}{1 + (\frac{\delta_{t-1,\tau}}{d})^{-1/s}} \ , \tag{2.10}$$

with scale parameter d and shape parameter s. The former represents the threshold above which the mispricing becomes too great. The latter represents the uncertainty or fuzziness around this level. Hence, extraordinary asset price growth translates to greater crash-hazard risk. This in turn renders increased expected return, compensating investors for the additional risk. Increased expected return again put greater pressure on elevated price growth and finally added on jump risk. This loop continues until the price is corrected. The correction is determined by the accumulation of jump sizes. Large accumulative jump sizes pushes the current price back towards its fundamental value and subsequently decreases the crash-hazard rate. This is synonymous with real market behavior where periods of extraordinary growth is associated with greater fluctuations and jumps in returns.

Malevergne and Sornette [2014] extends the non-local crash-hazard rate estimation by noticing that markets are populated by different types of investors. In particular, financial markets are populated by heterogeneous investors with different time horizons, ranging from intra-day traders to slow moving long-term investors. The time period over which the mispricing is estimated should, therefore, be a weighted average of the past mispricing's relating to the mixture of agents acting in the market. Noticing that the log-transform of the past mispricing is nothing but a moving average over the rolling window $[0, \tau]$ suggests the standard procedure of using an exponentially-weighted moving average

$$\log \delta_{t,a} e^{\bar{r}} = (1-a) \sum_{k=0}^{\infty} a^k \frac{S_{t-k}}{S_{t-1-k}} \ , \tag{2.11a}$$

$$= (1-a) \cdot \log \frac{S_t}{S_{t-1}} + a \cdot \log \left(\delta_{t-1,a} e^{\bar{r}}\right) \ . \tag{2.11b}$$

Malevergne and Sornette [2014] thereby summarizes market heterogeneity in the parameter a. a describes the collective market view on the severity of the current mispricing. For example, a market heavily populated by short-term investors weigh recent extraordinary growth higher than a market with a greater portion of long-term investors.

Equation 2.11b can equivalently be written as

$$\log \delta_{t,a} = (1-a) \cdot (\log \frac{S_t}{S_{t-1}} - \bar{r}) + a \log \delta_{t-1,a} \ . \tag{2.12}$$

For a typical investor memory of one year, counting 250 trading days in a year, $a = 1 - \frac{1}{250} = 0.996$. Next, by introducing the term

$$X_t := \frac{1}{s} \log \frac{\delta_{t-1,a}}{d} \ , \tag{2.13}$$

one can deduce that

$$\begin{aligned}
\lambda_t &= \frac{1}{1 + (\frac{\delta_{t-1,a}}{d})^{-1/s}} \ , & (2.14a)\\
&= \frac{1}{1 + (\frac{de^{sX_t}}{d})^{-1/s}} \ , & (2.14b)\\
&= \frac{1}{1 + e^{-X_t}} \ , & (2.14c)\\
&:= L(X_t) \ . & (2.14d)
\end{aligned}$$

Hence,

$$\begin{aligned}
X_t &= \frac{1}{s} \log \frac{\delta_{t-1,a}}{d} \ , & (2.15a)\\
&= \frac{1}{s} \left(\log \left(exp \left((1-a) \, (\log \frac{S_{t-1}}{S_{t-2}} - \bar{r}) + a \, \log(\delta_{t-2,a}) \right) \right) - \log d \right) \ , & (2.15b)\\
&= \frac{1}{s} \left((1-a)(\log \frac{S_{t-1}}{S_{t-2}} - \bar{r}) + a \, \log(\delta_{t-2,a}) - a \, \log d - (1-a) \, \log d \right) \ , & (2.15c)\\
&= \frac{1}{s} \left((1-a)(\log \frac{S_{t-1}}{S_{t-2}} - \bar{r}) + a \, \log \frac{\delta_{t-2,a}}{d} - (1-a) \, \log d \right) \ , & (2.15d)\\
&= -\frac{1-a}{s} \log d + a \, X_{t-1} + \frac{1-a}{s} (\log \frac{S_{t-1}}{S_{t-2}} - \bar{r}) \ , & (2.15e)\\
&= (1-a)\bar{X} + aX_{t-1} + \eta(r_{t-1} - \bar{r}) \ . & (2.15f)
\end{aligned}$$

X_t will be denoted the mispricing index. For ease of writing let

$$\bar{X} = -\frac{\log d}{s} \text{ and } \eta = \tfrac{1-a}{s} \ . \tag{2.16}$$

The mispricing index, therefore, depends on past mispricing indexes X_{t-1} of which its influence is determined by the parameter a, and the size of past return, r_{t-1}.

In conclusion, the complete system of equations is therefore given by

$$\begin{aligned}
r_t &= \bar{r} + \kappa \cdot L(X_t) + \sigma_t \cdot \epsilon_t - \kappa \cdot J_t \cdot I_t \ , & (2.17)\\
X_t &= (1-a) \cdot \bar{X} + a \cdot X_{t-1} + \eta \cdot (r_{t-1} - \bar{r}) \ , & (2.18)\\
\sigma_t^2 &= \bar{\sigma}^2 \cdot (1 - \alpha - \beta) + \alpha \cdot (r_{t-1} - \bar{r})^2 + \beta \cdot \sigma_{t-1}^2 \ , & (2.19)
\end{aligned}$$

where I_t denotes the jumps controlled by the Bernoulli random variable with success probability equal to $_t$ and J_t the unit exponential distribution controlling the corresponding jump sizes.

In closing of this section, summarizing the complete set of parameters

- $\bar{r}$ the unconditional expected return of the risky asset,
- κ the average jump size,
- a the smoothing factor of the exponential moving average of the mispricing index,
- $\bar{X}$ function of the threshold d and the fuzziness parameter s and represents the typical size of mispricings,
- η function of the smoothing parameter a and the fuzziness paramater s, and determines the variance around $\bar{X}$,
- $\bar{\sigma}$ denotes the volatility of covariance-stationary GARCH(1,1) process,
- α and β describes the squared volatility, σ_t^2 dependence on previous squared values of log-returns and squared volatilities respectively.

In order to give a deeper meaning of the parameters $\bar{X}$ and η, the asymptotic distribution of $\lambda_t = L(X_t)$ is presented in Malevergne and Sornette [2014] and given by the logistic-normal distribution

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma x(1-x)} \cdot exp(-\frac{(\log(\frac{x}{1-x}) - \mu)^2}{2\sigma^2}) \ , \tag{2.20}$$

where $\mu = \bar{X}$ and $\sigma^2 = \frac{\eta^2}{1-a^2}\text{var}(r_t)$.

2.3 Mispricing and inherent jump probability

The model builds upon the common conception that the underlying cause of a crash can be found in the preceding months or even years of financial log-return observations. The backward view, determining the downside risk associated with the growth, depends on the heterogeneous mixture of investors in the market. An accelerating ascent of prices forces the market to enter an unstable phase in which any small disturbance can trigger a crash. This representation of markets becoming increasingly more unstable is accounted for by the step up in crash-hazard rate. A Bernoulli random variable with success parameter equal to the crash-hazard rate replicates the sudden drop. The jump risk will get back to 'normal' levels if the accumulated size of the jumps is large enough. Otherwise, the crash-hazard rate remains significant. This estimation approach distinguishes itself from previous works as most existing jump-diffusion models derive the conditional expected return as being proportional to the underlying present crash hazard rate. We will in this section, therefore, shed light on these differences and highlight the improvements made by Malevergne and Sornette [2014].

Rational expectation models where originally introduced by Blanchard [1979]. They where intended to account for the possibility, often discussed in the empirical literature and by practitioners, that observed prices may deviate significantly and over extended time intervals from fundamental asset value. While allowing for deviations from fundamental value, rational expectation models of bubbles and crashes assume that all agents have rational expectations and share the same information. A rational bubble may arise when the actual market price depends positively on its own expected rate of change.

The instantaneous matching of the conditional expected return and crash-hazard rate in rational expectation models follows from the no-arbitrage condition and is both unrealistic and misleading because it assumes complete markets and no friction. Both conditions are highly unrealistic in periods of market hysteria in which prices increasingly deviates from its fundamental value. Humans are by nature not rational and exhibit different views on risk. This means a non-uniform distribution of traders acting in the market which Malevergne and Sornette [2014] accounts for.

Also, the effect of proportionally relating the conditional expected return and the crash-hazard rate is catastrophic. It severely underestimates the risk that the investor faces as soon as the price evolution of the asset stabilizes. This is because the conditional expected return becomes zero and in turn also the crash probability. In the model by Malevergne and Sornette [2014] it remains high as a consequence of the existing mispricing, still accounting for the larger downside risk.

In another attempt, Bates [2015] considers a model in which the Poisson intensity, controlling the number of crashes, is calculated as a linear functional of the realized volatility. This relates to the works by Malevergne and Sornette [2014] but separates on two important aspects. Firstly, the sign of mispricing is ignored when only considering an absolute measure such as volatility. This may lead to odd results as the crash probability can in principle increase with prices fluctuating around a downward sloping trend. This may be true in some cases, but in general fails to hold for the majority of scenarios. Secondly, and rather importantly, Bates crash-hazard rate predictions are based on a local estimation procedure. This does not take into account unsustainable price growth over longer periods, say a year.

Chapter 3

Particle filtering

Particle filters are classified as recursive Monte Carlo methods. Like all Monte Carlo methods, particle filters can adapt to any model, a crucial advantage in non-linear models with fat-tailed and asymmetric error distributions. Furthermore, particle filters are easy to program and computationally very fast to run.

Before developing an accurate filtering algorithm for learning about the unobservable crash-hazard rate and jump times for the model at hand, the chapter gives an introduction to different particle filtering methods. The chapter continuous with explaining the convergence of particle filters, before deducing the APF particle filter for the model presented in Malevergne and Sornette [2014]. The **C++** codes can be found in the appendix.

3.1 Filtering

Filtering is a way of extracting a latent state variable, L_t, from noisy data, using a statistical model, specifically a dynamic system for which you have a Bayesian model. Filtering was originally used to track latent states in physical systems, such as tracking the location of an airplane using noisy radar signals. Lately it has become a useful tool in economics and finance due to widespread use of models incorporating latent variables. As seen in the previous chapter, the latent variables are intended to capture unobserved changes in the economic environment and follow their own dynamics. As an example it is a commonly held conception that assets volatility is time varying.

In full generality, we consider a statistical model which generates the observed data, r_t, where conditional distribution of r_t depends on the latent state variable, L_t. Hence, we have

$$\text{observation equation: } r_t = f(L_t, \epsilon_t^y) \ , \tag{3.1}$$

$$\text{state evolution: } L_t = g(L_{t-1}, \epsilon_t^x) \ , \tag{3.2}$$

where ϵ_t^y is the observation noise and ϵ_t^x the state shocks. The observation equation is often written as a conditional likelihood, $p(r_t \mid L_t)$, and the state evolution as $p(L_t \mid L_{t-1})$. The posterior distribution of L_t given the observed data, $p(L_t \mid r^t)$ is said to solve the particle filtering problem. The posterior, take into account observations measured up to time t and is given by

$$p(L_t \mid r^t) = p(L_t \mid (r_1, ..., r_t)) \ . \tag{3.3}$$

Computing $p(L_t \mid r^t)$ is a two step-procedure consisting in first predicting the states and then use Bayesian update taking into account the current observation. Firstly, the prediction step combines the filtering distribution at time t−1 with the state evolution:

$$p(L_t \mid r^{t-1}) = \int p(L_t \mid L_{t-1}) p(L_{t-1} \mid r^{t-1}) dL_{t-1} \ . \tag{3.4}$$

This provides a forecast of the state at time t. Next, given a new observation, r_t the instantaneous prediction of the states is updated by following Bayes theorem

$$p(L_t \mid r^t) = \frac{p(r_t \mid L_t) p(L_t \mid r^{t-1})}{p(r_t \mid r^{t-1})} \ , \tag{3.5a}$$

$$\propto p(r_t \mid L_t) p(L_t \mid r^{t-1}) \ . \tag{3.5b}$$

The terms, reading the equation from left to right, are denoted posterior and prior respectively. Note that the integral above may be very hard to solve, consequently one relies on Monte Carlo approximation. Direct Monte Carlo approximation of the integral is however only possible if one can sample from the filtering density, $p(L_{t-1}|r^{t-1})$. The filtering density is rarely known analytically and one relies on alternative methods. In particular one relies on sequential Monte Carlo, also known as particle filtering algorithms. For the moment assume that the filtering density is known and that sampling from this distribution is possible.

A particle filter is a discrete approximation, $p^N(L_{t-1} \mid r^{t-1})$, to $p(L_{t-1} \mid r^{t-1})$, generally written as $\{\pi_{t-1}^{(k)}, L_{t-1}^{(k)}\}_{k=1}^N$, where $\sum_{k=1}^N \pi_{t-1}^{(k)} = 1$. Hence an approximation to the target density, $p(L_{t-1} \mid r^{t-1})$, is given by

$$p(L_{t-1} \mid r^{t-1}) \approx \sum_{k=1}^{N} \pi_{t-1}^{(k)} \delta_{L_{t-1}^{(k)}} \ , \tag{3.6}$$

where $\delta_{L_{t-1}^{(k)}}$ is the Dirac function centered at $L_{t-1}^{(k)}$ and $\pi_{t-1}^{(k)} = p(L_{t-1}^{(k)} \mid r^{t-1})$. The latter term is also known as the importance weight at time t−1. Note that the particle approximation, $p^N(L_{t-1} \mid r^{t-1})$, can be transformed into an equally weighted random sample using any kind of sampling with replacement technique. Hence, sampling with replacement from the discrete distribution $\{\pi_{t-1}^{(k)}, L_{t-1}^{(k)}\}_{k=1}^N$ yields an equally weighted sample $\{1/N, L_{t-1}^{z(k)}\}_{k=1}^N$, where $z(k)$ denotes the resampling index from the vector of prior states. Normally, the resampling is performed using a multinomial resampling technique. The filtering algorithm in this dissertation, however, utilizes a much faster approach, based on Carpenter, Clifford, and Fearnhead [1999]. Having obtained a discrete approximation of $p(L_{t-1} \mid r^{t-1})$ enables us to deduce estimates of interest, such as $\mathbb{E}[f(L_{t-1}) \mid r^{t-1}]$, by applying Monte Carlo:

$$\mathbb{E}[f(L_{t-1}) \mid r^{t-1}] \approx \sum_{k=1}^{N} f(L_{t-1}^{(k)}) \pi_{t-1}^{(k)} \ . \tag{3.7}$$

Given a discretization of $p^N(L_{t-1}|r^{t-1})$, $p(L_t|r^t)$ can be expressed in the following way

$$p^N(L_t \mid r^t) \propto \int p(r_t \mid L_t)p(L_t \mid L_{t-1})p^N(L_{t-1} \mid r^{t-1})dL_{t-1} \ , \tag{3.8a}$$

$$\approx \sum_{k=1}^{N} p(r_t \mid L_t)p(L_t \mid L_{t-1}^{(k)})\pi_{t-1}^{(k)} \ . \tag{3.8b}$$

This, however, is nothing but a discrete mixture distribution. Sampling from a discrete mixture distribution is straightforward by first selecting the mixture index and then simulate from that mixture component. That is, we first select (k), then simulate $L_t^{(k)}$ according to $p(L_t|L_{t-1}^{(k)})$ receiving the corresponding probability weight $p(r_t|L_t^{(k)})$. The discrete probability weights of $p^N(L_t|r^t)$, are therefore given by

$$\pi_t^{(k)} = p(r_t|L_t^{(k)})\pi_{t-1}^{(k)} \ . \tag{3.9}$$

This procedure is known as exact particle filtering because we can sample directly from the filtering density $p(L_{t-1}|r^{t-1})$ and thereby obtain a particle approximation of $p(L_t|r^t)$. The algorithm can be summarized as follows

- Given initial particle set, approximating the filtering density at time $t-1$, $\{L_{t-1}^{(k)}, \pi_{t-1}^{(k)}\}$, for k=1,....,N simulate new states,

$$L_t^{(k)} \sim p(L_t \mid L_{t-1}^{(k)}) \ , \tag{3.10}$$

  and assign probability weight

$$\pi_t^{(k)} \propto \pi_{t-1}^{(k)} p(r_t|L_t^{(k)}) \ . \tag{3.11}$$

- For k=1,...,N draw

$$z(k) \sim Mult(N; \pi_t^{(1)}, ..., \pi_t^{(N)}) \ , \tag{3.12}$$

  where $\pi_t^{(i)}$ denotes the normalized probability weights. After setting $L_t^{(k)} = L_t^{z(k)}$, a Monte Carlo estimate of the state mean at time t is given by

$$\mathbb{E}[L_t|r^t] = \int L_t p^N(L_t|r^t)dL_t \ , \tag{3.13a}$$

$$\approx \sum_{k=1}^{N} L_t^{(k)}\pi_t^{(k)} \ , \tag{3.13b}$$

$$= \frac{1}{N}\sum_{k=1}^{N} L_t^{(k)} \ . \tag{3.13c}$$

As previously stated, if direct sampling is not possible, a technique known as importance sampling is applied. Before proceeding note that in order to derive the subsequent algorithms it is useful to consider the full posterior distribution at time t, $p(L_{0:t} \mid r^t)$.

3.1.1 Importance sampling

Importance sampling approximates the sample from the target distribution, $p(L_{0:t-1}|r^{t-1})$, which may be difficult or even impossible to sample from, by sampling from a proposal density $q(L_{0:t-1}|r^{t-1})$. Using the trick

$$p(L_{0:t-1} \mid r^{t-1}) = \frac{p(L_{0:t-1} \mid r^{t-1})}{q(L_{0:t-1} \mid r^{t-1})} q(L_{0:t-1} \mid r^{t-1}) \ , \tag{3.14}$$

where $\frac{p(L_{0:t-1}|r^{t-1})}{q(L_{0:t-1}|r^{t-1})}$ is denoted the importance weight. An approximation to the target density, $p(L_{0:t-1} \mid r^{t-1})$, is given by

$$p(L_{0:t-1} \mid r^{t-1}) \approx \sum_{k=1}^{N} \pi_{t-1}^{(k)} \delta_{L_{0:t-1}^{(k)}} \ , \tag{3.15}$$

where $\pi_{t-1}^{(k)} = \frac{p(L_{t-1}|r^{t-1})}{q(L_{0:t-1}|r^{t-1})}$. Note that the importance weights compensate for the fact that one is sampling $L_{0:t-1}$ from the wrong distribution, $q(L_{0:t-1}|r^{t-1})$.

A sample from the filtering density, $p^N(L_t|r^t)$, solves the filtering problem. Applying importance sampling directly to the filtering density at time t is possible, but would imply re-computation of the importance weights whenever new data, r_t, becomes available. Sequential importance sampling (SIS) is then much more efficient. SIS simply updates the filtering distribution at time t−1 by propagating $L_{t-1}^{(k)}$ through the state evolution $p(L_t|L_{t-1})$.

Before moving on, note that there are two sources of approximation errors in particle filter algorithms:

1. approximating $p(L_{t-1}|r^{t-1})$ by $p^N(L_{t-1}|r^{t-1})$ (discrete approximation) ,
2. and secondly approximating $p^N(L_{t-1}|r^{t-1})$ using a sampling method.

3.1.2 Sequential importance sampling

The most basic sequential Monte Carlo approximation scheme is the SIS. The idea is to utilize the discrete approximation of the filtering density at time t−1, $\{L_{0:t-1}^{(k)}, \pi_{t-1}^{(k)}\}_{k=1}^{N}$, then recursively update these particles to obtain an approximation of the posterior density at the next time step.

Applying importance sampling to the posterior distribution at time t gives

$$\pi_t^{(k)} = \frac{p(L_{0:t} \mid r^t)}{q(L_{0:t} \mid r^t)} \ . \tag{3.16}$$

Assuming that the importance density $q(L_{0:t} \mid r^t)$ admits as marginal distribution at time t−1, the importance function $q(L_{0:t-1} \mid r^{t-1})$, implies that

$$\begin{aligned}
q(L_{0:t} \mid r^t) &= q(L_{0:t-1} \mid r^{t-1})q(L_t \mid L_{0:t-1}, r^t) \ , && (3.17a)\\
&= q(L_{0:t-2} \mid r^{t-2})q(L_{t-1} \mid L_{0:t-2}, r^{t-1})q(L_t \mid L_{0:t-1}, r^t)q(L_t \mid L_{0:t-1}, r^t) \ , && (3.17b)\\
& .... \ , && (3.17c)\\
&= q(L_0)\prod_{k=1}^{t+1} q(L_k \mid L_{0:k-1}, r^k) \ . && (3.17d)
\end{aligned}$$

Furthermore, as the states evolve according to a Markov process and the observations are conditionally independent given the states:

$$\begin{aligned}
p(L_{0:t}) &= p(L_0)p(L_1 \mid L_0)p(L_2 \mid L_1, L_0)....p(L_t \mid L_t, ..., L_0) \ , && (3.18a)\\
&= p(L_0)\prod_{k=1}^{t} p(L_k \mid L_{k-1}) \ , && (3.18b)\\
p(r^t \mid L_{0:t}) &= \prod_{k=1}^{t} p(r_k \mid L_k) \ . && (3.18c)
\end{aligned}$$

Finally obtaining a recursive formula for the importance weights

$$\begin{aligned}
\pi_t &= \frac{p(L_{0:t} \mid r^t)}{q(L_{0:t} \mid r^t)} \ , && (3.19a)\\
&= \frac{p(r^t \mid L_{0:t})p(L_{0:t})}{q(L_0)\prod_{k=1}^{t} q(L_k \mid L_{0:k-1}, y^k)} \ , && (3.19b)\\
&= \frac{\prod_{k=1}^{t} p(r_k \mid L_k)p(L_0)\prod_{k=1}^{t} p(L_k \mid L_{k-1})}{q(L_0)\prod_{k=1}^{t} q(L_k \mid L_{0:k-1}, y^k)} \ , && (3.19c)\\
&= \frac{p(r_t \mid L_t)p(L_t \mid L_{t-1})p(L_0)\prod_{k=1}^{t-1} p(r_k \mid L_k)p(L_k \mid L_{k-1})}{q(L_t \mid L_{0:t-1}, r^t)q(L_0)\prod_{k=1}^{t-1} q(L_k \mid L_{0:k-1}, y^k)} \ , && (3.19d)\\
&= \pi_{t-1}\frac{p(r_t \mid L_t)p(L_t \mid L_{t-1})}{q(L_t \mid L_{0:t-1}, r^t)} \ , && (3.19e)\\
&= \pi_{t-1}\frac{p(r_t \mid L_t)p(L_t \mid L_{t-1})}{q(L_t \mid L_{t-1}, r_t)} \ , && (3.19f)
\end{aligned}$$

where in the last step it's assumed that $q(L_t|L_{0:t-1}, r^t) = q(L_t|L_{t-1}, r_t)$, so that the proposal at the next time step only depends on the most recent state and the most recent observation. This is reasonable for first order Markovian state evolutions.

The complete set of update equations, resulting from the SIS, simplify to:

$$\begin{aligned}
L_t^{(k)} &\sim q(L_t \mid L_{t-1}^{(k)}, r_t) \ , && (3.20a)\\
\pi_t^{(k)} &= \pi_{t-1}^{(k)}\frac{p(r_t \mid L_t^{(k)})p(L_t^{(k)} \mid L_{t-1}^{(k)})}{q(L_t^{(k)} \mid L_{t-1}^{(k)}, r_t)} \ , && (3.20b)
\end{aligned}$$

thereby obtaining a discrete approximation $\{L_t^{(k)}, \pi_t^{(k)}\}_{k=1}^N$ of the filtering density $p(L_t \mid r^t)$.

3.1.3 Sequential importance resampling

SIS suffers from a significant drawback, namely as t increases the distribution of the importance weights becomes highly skewed meaning that some particles will, eventually, have their weights set close to zero. This is denoted degeneracy. The sequential importance resampling (SIR) algorithm corrects this deficiency by resampling the updated particles, such that the resampled importance weights become $\pi_t^{(k)} = 1/N$.

SIR chooses the proposal density $q(L_t|L_{t-1}^{(k)}, r_t)$ to be the state transition distribution $p(L_t \mid L_{t-1})$. The update equations, therefore, become:

$$L_t^{(k)} \sim p(L_t \mid L_{t-1}^{(k)}) \ , \tag{3.21}$$

$$\pi_t^{(k)} \propto p(r_t \mid L_t^{(k)}) \ . \tag{3.22}$$

Hence, given current particle set $\{L_{t-1}^{(k)}, \pi_{t-1}^{(k)}\}_{k=1}^N$ from $p^N(L_{t-1} \mid r^{t-1})$ the algorithm consists of two steps:

1. For k=1,....,N simulate $L_t^{(k)} \sim p(L_t \mid L_{t-1}^{(k)})$
2. For k=1,....,N compute

$$\pi_t^{(k)} = p(r_t \mid L_t^{(k)}) / \sum_{k=1}^{N} p(r_t \mid L_t^{(k)}) \ , \tag{3.23}$$

draw

$$z(k) \sim \text{Mult}(N; \pi_t^{(1)}, ..., \pi_t^{(N)}) \ , \tag{3.24}$$

and set $L_t^{(k)} = L_t^{(z(k))}$. $\{L_t^{(k)}, \frac{1}{N}\}_{k=1}^N$ approximates the filtering density at time t.

3.1.4 Auxiliary particle filter

The SIR chooses the proposal density, $q(L_t \mid L_{t-1}^{(k)}, r_t)$, to be the state transition distribution, $p(L_t \mid L_{t-1})$. This samples new states, L_t, ignoring the new observation, r_t. In periods with large movements driven by outliers and rare events, this sampling procedure underestimates the state. The auxiliary particle filter (APF) corrects this deficiency and adapts to the structure of the given model. It consists of two steps: resampling old particles using $p(r_t \mid L_{t-1})$, then propagate states by simulating from $p(L_t \mid L_{t-1}, r_t)$. If it is not possible to evaluate $p(r_{t+1} \mid L_t)$ or sample directly from $p(L_t \mid L_{t-1}, r_t)$, one may utilize importance sampling. Approximating the exact distribution, $p(r_t \mid L_{t-1})$, by $\hat{p}(r_t \mid L_{t-1})$ and $p(L_t|L_{t-1}, r_t)$ by $\hat{p}(L_t \mid L_{t-1}, r_t)$, results in the following steps for the APF algorithm:

1. For k=1,...,N compute

$$w_t^{(k)} = \hat{p}(r_t \mid L_{t-1}^{(k)}) / \sum_{k=1}^{N} \hat{p}(r_t \mid L_{t-1}^{(k)}) \ , \tag{3.25}$$

draw

$$z(k) \sim Mult(N; w_t^{(1)}, ..., w_t^{(N)}) \ , \tag{3.26}$$

and set $L_{t-1}^{(k)} = L_{t-1}^{(z(k))}$

2. For k=1,....,N simulate
$$L_t^{(k)} \sim p(L_t \mid L_{t-1}^{(k)}, r_t) \ , \tag{3.27}$$
and compute
$$\begin{aligned} \pi_t^{(k)} &= \pi_{t-1}^{(k)} \frac{\text{target}}{\text{proposal}} \ , &(3.28a)\\ &= \pi_{t-1}^{(k)} \frac{p(r_t \mid L_{t-1}^{(k)}) p(L_t^{(k)} \mid L_{t-1}^{(k)})}{\hat{p}(r_t \mid L_{t-1}^{(k)}) \hat{p}(L_t^{(k)} \mid L_{t-1}^{(k)}, r_t)} \ , &(3.28b)\\ &= \pi_{t-1}^{(k)} \frac{p(r_t \mid L_t^{(k)}) p(L_t^{(k)} \mid L_{t-1}^{(k)})}{w_t^{z(k)} \hat{p}(L_t^{(k)} \mid L_{t-1}^{(k)}, r_t)} \ , &(3.28c)\end{aligned}$$
thereby obtaining the next step particle set, $\{L_t^{(k)}, \pi_t^{(k)}\}_{k=1}^N$ approximating $p^N(L_t \mid r^t)$.

As explained in Johannes and Polson [2009] there is no need to resample the importance weights calculated at the end of the algorithm. This, in fact, introduces additional Monte Carlo error.

3.2 Derivation of the filtering algorithm

In this section, the APF particle filtering algorithm will be derived for the model at hand. We thereby also implicitly determine the simpler SIR algorithm. The complete SIR algorithm can be found in the appendix. Recapping the dynamics of the jump-diffusion model introduced by Malevergne and Sornette [2014]:

$$\begin{aligned} r_t &= \bar{r} + \kappa \cdot L(X_t) + \sigma_t \cdot \epsilon_t - \kappa \cdot J_t \cdot I_t \ , &(3.29)\\ X_t &= (1-a) \cdot \bar{X} + a \cdot X_{t-1} + \eta \cdot (r_{t-1} - \bar{r}) \ , &(3.30)\\ \sigma_t^2 &= \bar{\sigma}^2 \cdot (1-\alpha-\beta) + \alpha \cdot (r_{t-1} - \bar{r})^2 + \beta \cdot \sigma_{t-1}^2 \ , &(3.31)\end{aligned}$$

where $\lambda_t = L(X_t) = \frac{1}{1+e^{-X_t}}$ and $J_t \sim \exp(1), I_t \sim \text{Bernoulli}(\lambda_t)$ and $\epsilon_t \sim \mathbb{N}(0,1)$ all independent. In what follows let $f_{emg}(x; \mu, \sigma, k)$ and $\phi(x; \mu, \sigma)$ denote the densities of the Exponentially Modified Gaussian and Gaussian respectively, and $L_t = (I_t, J_t)$

The first step in the APF algorithm requires the predictive likelihood given by $p(r_t \mid L_{t-1}) = \int p(r_t \mid L_t) p(L_t \mid L_{t-1}) dL_t$. Following the approach in Johannes et al. [2009] to approximate $p(r_t \mid L_{t-1})$ intermediate states, jump times and jump sizes need to be integrated out. Consider X_t and σ_t^2 to be known and given by their respective equations

$$\begin{aligned} X_t &= (1-a) \cdot \bar{X} + a \cdot X_{t-1} + \eta \cdot (r_{t-1} - \bar{r}) \ , &(3.32)\\ \sigma_t^2 &= \bar{\sigma}^2 \cdot (1-\alpha-\beta) + \alpha \cdot (r_{t-1} - \bar{r})^2 + \beta \cdot \sigma_{t-1}^2 \ . &(3.33)\end{aligned}$$

We have that

$$\hat{p}(r_t \mid (X_t, \sigma_t^2), J_t, I_t) = \phi(r_t; \bar{r} + \kappa \cdot L(\hat{X}_t) - \kappa \cdot J_t \cdot I_t, \sigma_t) \ . \tag{3.34}$$

Now, the jump times and sizes can be integrated out to obtain $\hat{p}(r_t \mid X_t, \sigma_t^2)$. First taking the expectation with respect to I_t yields

$$\begin{aligned}\hat{p}(r_t \mid (X_t, \sigma_t^2), J_t) = L(X_t) \cdot \phi(r_t; \bar{r} + \kappa \cdot L(X_t) - \kappa \cdot J_t, \sigma_t) + \\ (1 - L(X_t)) \cdot \phi(r_t; \bar{r} + \kappa \cdot L(X_t), \sigma_t) \ .\end{aligned} \tag{3.35}$$

Lastly, averaging the first Gaussian density with respect to J_t yields the Exponentially Modified Gaussian. Hence

$$\begin{aligned}\hat{p}(r_t \mid X_t, \sigma_t^2) = L(X_t) \cdot f_{emg}(r_t; \bar{r} + \kappa \cdot L(X_t), \sigma_t, \kappa) + \\ (1 - L(X_t)) \cdot \phi(r_t; \bar{r} + \kappa \cdot L(X_t), \sigma_t) \ .\end{aligned} \tag{3.36}$$

The previous section explained how APF adapts to the model structure by taking into account the current observation r_t when simulating new states. In the following two sections, it is showed how we obtain $p(L_t | L_{t-1}, r_t)$ for the jump times and sizes respectively.

3.2.1 Jump times

The second step in the APF algorithm simulates latent variables using the structure of the model. First sampling jump times from

$$p(I_t = k \mid (X_t, \sigma_t^2), r_t) \propto p(r_t \mid (X_t, \sigma_t^2), I_t = k) p(I_t) \ , \tag{3.37a}$$

$$= \begin{cases} \lambda_t \cdot f_{emg}(r_t; \bar{r} + \kappa \cdot \lambda_t, \sigma_t, \kappa) & \text{if } k = 1 \ , \\ (1 - \lambda_t) \cdot \phi(r_t; \bar{r} + \kappa \cdot \lambda_t, \sigma_t) & \text{if } k = 0 \ , \end{cases} \tag{3.37b}$$

where $\lambda_t = L(X_t)$. We introduce the following notation

$$p_1 \propto p(I_t = 1 \mid (X_t, \sigma_t^2), r_t) \ , \tag{3.38}$$

$$p_2 \propto p(I_t = 0 \mid (X_t, \sigma_t^2), r_t) \ . \tag{3.39}$$

3.2.2 Jump sizes

If a jump occurs, incorporating the new observation, r_t, effectively tilts the jump sizes towards values that could have generated the observation. Hence, one is interested in sampling from

$$p(J_t \mid I_t = 1, (X_t, \sigma_t^2), r_t) \ . \tag{3.40}$$

Again, considering (X_t, σ_t^2) as known, and noting that

$$r_t = \bar{r} + \kappa \cdot \lambda_t + \sigma_t \cdot \epsilon_t - \kappa \cdot J_t \cdot I_t \ , \tag{3.41a}$$

$$r_t - (\bar{r} + \kappa \cdot \lambda_t) = \tilde{\epsilon}_t - \tilde{J}_t \cdot I_t \ , \tag{3.41b}$$

$$\tilde{r}_t = \tilde{\epsilon}_t - \tilde{J}_t \cdot I_t \ , \tag{3.41c}$$

where $\tilde{\epsilon}_t \sim \mathbb{N}(0, \hat{\sigma}_t^2)$ and $\tilde{J}_t \sim exp(1/\kappa)$.

$$p(\tilde{J}_t = z \mid I_t = 1, X_t, \sigma_t^2, r_t) = \frac{p(\tilde{J}_{t+1} = z)p(\tilde{\epsilon}_t = \tilde{r}_t + z)}{p(\tilde{r}_t)} \ , \tag{3.42a}$$

$$\propto \begin{cases} \frac{1}{\sqrt{2\pi}\sigma_t} exp(-\frac{(z+\tilde{r}_t+\frac{1}{k}\sigma_t^2)^2}{2\sigma_t^2}) & \text{if } z \geq 0 \ , \\ 0 & \text{if } z < 0 \ . \end{cases} \tag{3.42b}$$

This is the truncated Gaussian distribution on the range $z \in (0, \infty)$ with mean and standard deviation given by:

$$\mu = -(\tilde{r}_t + \frac{1}{k}\sigma_t^2) \ , \tag{3.43}$$

$$\sigma = \sigma_t^2 \ , \tag{3.44}$$

and probability density function

$$f(x; \mu, \sigma) = \frac{\phi(\frac{x-\mu}{\sigma})}{1 - \Phi(-\frac{\mu}{\sigma})} \ . \tag{3.45}$$

Letting $P(\tilde{J}_{t+1} = z \mid I_t = 1, X_t, \sigma_t^2, r_{t+1})$ denote the cdf of the jump sizes, gives

$$P(\tilde{J}_{t+1} = z \mid I_t = 1, (X_t, \sigma_t^2), r_t) = \frac{\Phi(\frac{z-\mu}{\sigma}) - \Phi(-\frac{\mu}{\sigma})}{1 - \Phi(-\frac{\mu}{\sigma})} \ . \tag{3.46}$$

Simulating random variables z may be done using the relation; if $u \sim U[0,1]$ and $F = P(\tilde{J}_t = z \mid I_t = 1, X_t, \sigma_t^2, r_t)$ then $F^{-1}(U) \overset{d}{=} Z$. Hence a random variable z is given by

$$\Phi^{-1}(\Phi(-\frac{\mu}{\sigma}) + u(1 - \Phi(-\frac{\mu}{\sigma})))\sigma + \mu \ . \tag{3.47}$$

However, using the method above calls for a simultaneous evaluation of the normal cdf Φ and its inverse Φ^{-1}, requiring a great deal of precision in the approximations of these functions. In addition it may also be quite ineffective if μ is large. Lastly noting that if σ is close to zero and μ positive, such that $-\frac{\mu}{\sigma} << 0$, can cause problems since $\Phi(-\frac{\mu}{\sigma}) \to 0$ exponentially fast as $-\frac{\mu}{\sigma} \to -\infty$. This in turn implies that $(1 - \Phi(-\frac{\mu}{\sigma}))u + \Phi(-\frac{\mu}{\sigma}) \to u$. Enabling negative jump size realizations, since $\Phi^{-1}(u) \leq 0$ whenever $u \leq \frac{1}{2}$.

Another, readily available method is to simulate from a standard normal distribution, X. Then simulate jump sizes, using the transformation $Z = \mu + \sigma X$, until the generated number is larger than 0. This method is quite reasonable when $0 < \mu$, but of no use when μ is several standard deviations to the left of 0.

Both approaches presented above may not yield the desired results and efficiency. An accept-reject algorithm may be more profitable.

Accept-reject algorithm

First, the general accept-reject algorithm is based on the following result Devroye [1986];

let h and g be two densities such that $h(x) \leq Mg(x)$ for every x in the support of h and normalization constant M. The random variable x resulting from the algorithm:

1. generate $z \sim g(z)$, then
2. generate $u \sim U[0,1]$. If $u \leq h(z)/Mg(z)$, take $x = z$, otherwise repeat step 1,

is distributed according to h.

Following the approach in Robert [1995], choosing g to be the exponential distribution with density

$$g(z;\lambda) = \lambda e^{-\lambda z} \ . \tag{3.48}$$

The task is now to find the normalization constant M such that $h(x) \leq Mg(x)$. Where in our case, $h(x) = p(\tilde{J}_t = z \mid I_t = 1, X_t, \sigma_t^2, r_t)$. By first noting that

$$e^{\lambda z} e^{-\frac{(z-\mu)^2}{2\sigma^2}} = e^{-\frac{(z-(\mu+\lambda\sigma^2))^2}{2\sigma^2}} e^{\frac{(\mu+\lambda\sigma^2)^2-\mu^2}{2\sigma^2}} \ , \tag{3.49a}$$

$$\leq e^{\mu\lambda+\lambda^2\sigma^2/2} \ . \tag{3.49b}$$

Hence,

$$\frac{h(z)}{g(z)} \leq \frac{e^{\mu\lambda+\lambda^2\sigma^2/2}}{\lambda\sqrt{2\pi}\sigma(1-\Phi(-\frac{\mu}{\sigma}))} \ . \tag{3.50}$$

The normalization constant M is chosen to be:

$$M = \frac{e^{\mu\lambda+\lambda^2\sigma^2/2}}{\lambda\sqrt{2\pi}\sigma(1-\Phi(-\frac{\mu}{\sigma}))} \ , \tag{3.51}$$

and the ratio $h(z)/Mg(z)$ is then given by

$$\frac{h(z)}{Mg(z)} = e^{-\frac{(z-\mu)^2}{2\sigma^2}-(\mu\lambda+\frac{\lambda^2\sigma^2}{2}-\lambda z)} \ . \tag{3.52}$$

Where a sensible choice for λ may be such that the expectation of the exponential distribution is close to the average jump size.

3.2.3 Filtering weights

Since we are using approximations to $p(r_t \mid L_{t-1})$ and $p(L_t \mid L_{t-1}, r_t)$ an additional reweighting step is required at the end of the algorithm, as explained in Johannes et al. [2009]. These final weights define the filtering distribution at time t. These probabilities are defined as

$$\pi_t \propto \pi_{t-1} \cdot \frac{p(L_t|L_{t-1},X_t,\sigma_t^2)\cdot p(r_t|L_t,X_t,\sigma_t^2)}{w_t^{z()}\cdot p(L_t|L_{t-1},X_t,\sigma_t^2,r_t)} \ , \tag{3.53a}$$

$$= \pi_{t-1} \cdot \frac{p(I_t|L_{t-1},X_t,\sigma_t^2)\cdot p(\tilde{J}_t|L_{t-1},X_t,\sigma_t^2)\cdot \phi(r_t;(\bar{r}+\kappa L(X_t)-\tilde{J}_t\cdot I_t),\sigma_t)}{p(I_t|L_{t-1},r_t)\cdot p(\tilde{J}_t|L_{t-1},X_t,\sigma_t^2,r_t)\cdot w_t^{z()}} \ , \tag{3.53b}$$

$$= \pi_{t-1} \cdot \frac{p(I_t|X_t)\cdot p(\tilde{J}_t)\cdot \phi(r_t;(\bar{r}+\kappa\cdot L(X_t)-\tilde{J}_t\cdot I_t),\sigma_t)}{p(I_t|X_t,\sigma_t^2,r_t)\cdot p(\tilde{J}_t|I_t,X_t,\sigma_t^2,r_t)\cdot w_t^{z()}} \ , \tag{3.53c}$$

$$\propto \begin{cases} \pi_{t-1}\cdot \frac{L(X_t)^{I_t}\cdot(1-L(X_t))^{1-I_t}\cdot e^{-1/\kappa J_t}\cdot\phi(r_t;(\bar{r}+\kappa\cdot L(X_t)-\tilde{J}_t\cdot I_t),\sigma_t)}{exp(-\frac{(x+\tilde{r}_t+\frac{1}{k}\cdot\sigma_t^2)^2}{2\cdot\sigma_t^2})\cdot p_1^{I_t}\cdot p_2^{1-I_t}\cdot w_t^{z()}} & \text{if } I_t = 1 \ , \\ \pi_{t-1}\cdot \frac{L(X_t)^{I_t}\cdot(1-L(X_t))^{1-I_t}\cdot e^{-1/\kappa\cdot J_t}\cdot\phi(r_t;(\bar{r}+\kappa\cdot L(X_t)),\sigma_t)}{p_1^{I_t}\cdot p_2^{1-I_t}\cdot w_t^{z()}} & \text{if } I_t = 0 \ . \end{cases} \tag{3.53d}$$

Where $w_t^{z()}$ denotes the resampled first stage weights. Completely in line with the dependence structure of the model the derivations used that the states $L_t = (I_t, J_t)$ are conditionally independent of each other and of the current observed log-return r_t.

3.3 Convergence of particle filters

The above approach, which is based on Johannes et al. [2009], relies on particle approximations. Under some mild regularity conditions on the state transition and the likelihood, the SIR and APF algorithms are consistent. If the true model is given by $p(L_t|r_r)$, then the particle approximation $p^N(L_t|r_t)$, given by $\{L_t^{(i)}, \pi_t^{(i)}\}_{i=1}^N$, converges in a pointwise sense as N increases Crisan and Doucet [2002].

When data is simulated, the true value of the states and parameters are known. This makes it possible to check for convergence in the particle filtering algorithm. Following the same procedure as in Sylvain [2012], using the relation

$$p(r_t|r_{1:t-1}) = \int p(r_t|L_{t-1})p(L_{t-1}|r_{1:t-1})dL_{t-1} \ , \tag{3.54a}$$

$$= \sum_{i=1}^N \pi_{t-1}^{(i)} p(r_t|L_{t-1}) \ . \tag{3.54b}$$

Assuming that $p(I_t|L_{t-1})$ and $p(J_t|L_{t-1})$ are simulated correctly, the estimated cdf obtained, using the particle filtering algorithm, is given by

$$\hat{P}(r_t|L_{t-1}) = P(r_t|\hat{I}_t, \hat{J}_t) \ , \tag{3.55a}$$

$$= \Phi(r_t; \bar{r} + \kappa(X_t) - \kappa \cdot \hat{J}_t \cdot \hat{I}_t, \sigma_t) \ , \tag{3.55b}$$

where Φ is the cdf's of the Gaussian distribution with mean $\bar{r} + \kappa(X_t) - \kappa \cdot \hat{J}_t \cdot \hat{I}_t$ and standard deviation equal to σ_t.

Letting $P(r_t|L_{t-1})$ denote the true predictive distribution, and using that if $X \sim F$, then $F(X) \sim U[0,1]$. Hence, $\hat{P}(r_t|L_{t-1})$ is said to be a good approximation if it is approximately uniformly distributed between 0 and 1. If it has a U-shape, the true cdf is broader, i.e. has heavier tails, and the estimation is said to be under-dispersed. If, on the other hand, the shape is concave, the true predictive cdf has thinner tails. Lastly, any sign of a slope would indicate bias.

Chapter 4

Parameter estimation

Particle filtering has proven to be successful in many applications. A main problem arises when there is the presence of unknown static parameters, especially when the parameter space is large. Numerous papers have been written on the construction of estimation algorithms based on Markov Chain Monte Carlo, but have not proven very successful.

In the previous section, we displayed the filtering of the states, in which the parameters of the jump-diffusion model at hand where assumed to be known. Clearly, this is never true in reality and this chapter therefore presents two methods, using the particle filtering algorithm, to calibrate the model. In this master thesis, the focus is on

- maximum likelihood ,
- state augmentation .

First the maximum likelihood approach is presented in section 4.1. This may be used to display cross sections of the parameter space, but the computational cost of directly applying the maximum likelihood is too great. In order to find the maximum of the log-likelihood we, therefore, rely on standard optimization algorithms. In section 4.1.1, we present how the expectation-maximization algorithm adapts to the filter. Next the simplest calibration method, known as state augmentation, is reviewed in section 4.2.

4.1 Maximum likelihood

Maximum likelihood is the traditional way of calibrating a model. Assuming that the jump-diffusion process depends on an unknown static parameter vector θ and furthermore that the marginal likelihood $L(\theta \mid r_{1:t+1}) = p_\theta(r_{1:t+1})$ admits a sequential formulation

$$p_\theta(r_{1:t+1}) = p_\theta(r_1) \prod_{k=1}^{t} p_\theta(r_{k+1} \mid r_{1:k}) \ . \tag{4.1}$$

One can utilize the filter to estimate the log-likelihood by noting that

$$\begin{aligned}
p_\theta(r_{k+1} \mid r_{1:k}) &= \int p_\theta(r_{k+1}, L_{k+1} \mid r_{1:k}) dL_{k+1} \ , && \text{(4.2a)}\\
&= \int \int p_\theta(r_{k+1} \mid L_{k+1}) p_\theta(L_{k+1} \mid L_k) p_\theta(L_k \mid r_{0:k}) dL_k dL_{k+1} \ , && \text{(4.2b)}\\
&\approx \sum_{i=1}^{N} \pi_k^{(i,\theta)} \int p(r_{k+1}|L_{k+1}) p(L_{k+1}|L_k^{(i,\theta)}) dL_{k+1} \ , && \text{(4.2c)}\\
&= \sum_{i=1}^{N} \pi_{k+1}^{(i,\theta)} \ . && \text{(4.2d)}
\end{aligned}$$

Where one utilized the Monte Carlo approximation for the filtering density $p(L_t|r_{1:k})$. $\pi_k^{(i,\theta)}$ denotes the un-normalized weights at the k^{th} time step for a given θ. The particle approximation of the log-likelihood is therefore given by

$$\begin{aligned}
\log L(\theta \mid r_{1:t+1}) &= \log \prod_{k=1}^{t} p_\theta(r_{k+1} \mid r_{1:k}) \ , && \text{(4.3a)}\\
&= \sum_{k=1}^{t} \log p_\theta(r_{k+1} \mid r_{1:k}) \ , && \text{(4.3b)}\\
&\approx \sum_{k=1}^{t} \log \sum_{i=1}^{N} \pi_{k+1}^{(i,\theta)} \ . && \text{(4.3c)}
\end{aligned}$$

Maximizing the sum of unnormalized weights results in the maximum likelihood estimator:

$$\hat{\theta} = \arg\max_\theta [\sum_{k=1}^{t} \log \sum_{i=1}^{N} \pi_{k+1}^{(i,\theta)}] \ . \tag{4.4}$$

Displaying a cross section of the parameter space is done by computing the log-likelihood for a subspace of the full parameter space, where all other parameters are set to their true values. Noting, however, in this case, that despite having deduced a fast algorithm for solving the filtering problem, computing the likelihood on a grid over the full parameter space is not feasible. As an example, a grid corresponding to 10 points in each parameter direction means running the particle filter for 10^8 different θ values. Assuming that the time series over which we run the algorithm is 10000 with 5000 particles, the real computation time would amount to approximately $10^8 \times 180\text{sec} = 208333$ days of computation time on a Mac running on OS X Yosemite with 2.7 GHz Intel Core i7 processor.

4.1.1 Expectation-maximization algorithm

To summarize, we are interested in solving the following maximization problem

$$\hat{\theta} = \arg\max_\theta \log p_\theta(r_{1:T}) \ , \tag{4.5}$$

where $\log p_\theta(r_{1:T})$ denotes the complete likelihood of the model over the whole time series spanning from 1 to T. The main difficulty of solving this optimization problem arises from the need to perform a nonlinear filtering operation in order to calculate the likelihood. Suppose, however, that in addition to the measurements r one can also observe the states

$L_{1:T} = (L_1, ..., L_T)$. Based on these measurements one seeks the maximum log-likelihood estimate of θ via

$$\hat{\theta} = \arg\max_{\theta} \log p_\theta(L_{1:T}, r_{1:T}) \ . \tag{4.6}$$

Then maximizing $l_\theta(L_{1:T}, r_{1:T}) = \log p_\theta(L_{1:T}, r_{1:T})$ is done by using a gradient based search algorithm. The states are, however, not observed and employment of the EM algorithm is necessary. The E-step approximates the joint log-likelihood function, $l_\theta(L_{1:T}, r_{1:T})$, by taking the expected value over the unobserved states based upon some current guess of the parameters θ_k. That is

$$\mathbb{E}[l_\theta(L_{1:T}, r_{1:T})|r_{1:T}] = \int l_\theta(L_{1:T}, r_{1:T}) p_{\theta_k}(L_{1:T}|r_{1:T}) dL_{1:T} \ . \tag{4.7}$$

The M-step then maximizes this expectation with respect to the parameter θ. Hence the Expectation Maximization (EM) algorithm iteratively switches between the following two steps:

1. (E-step) Calculate the expected value of $l_\theta(L_{1:T}, r_{1:T})$ over the unobservable states L based on a current parameter estimate θ_k and the measurements r.

$$Q(\theta, \theta_k) := \mathbb{E}[l_\theta(L_{1:T}, r_{1:T})|r_{1:T}] = \int l_\theta(L_{1:T}, r_{1:T}) p(L_{1:T}|r_{1:T}) dL_{1:T} \ . \tag{4.8}$$

2. (M-step) Obtain a new estimate θ_{k+1} by maximizing $Q(\theta, \theta_k)$ over θ

$$\theta_{k+1} = \arg\max_{\theta} Q(\theta, \theta_k) \ . \tag{4.9}$$

A remarkable feature of the EM algorithm is that maximizing $Q(\theta, \theta_k)$ actually generates an increase in the log-likelihood $log(p_\theta(r_{1:T}))$ from above.

In order to apply Monte Carlo approximation to the integral above, note that from Bayes and the Markov property of the model that

$$\begin{aligned} p_\theta(L_{1:T}, r_{1:T}) &= p_\theta(L_{1:T}) p_\theta(r_{1:T}|L_{1:T}) \ , & (4.10a) \\ &= p_\theta(L_1) \prod_{t=1}^{T} p_\theta(L_{t+1}|L_t) p(r_t|L_t) \ . & (4.10b) \end{aligned}$$

Hence, the conditional expectation from above translates to

$$\begin{aligned} Q(\theta, \theta_k) = &\int \log p_\theta(L_1)\, p_{\theta_k}(L_{1:T}|r_{1:T}) dL_1 \\ &+ \sum_{t=1}^{T} \int \log p_\theta(L_{t+1}|L_t)\, p_{\theta_k}(L_{t+1}|r_{1:T}) dL_t \\ &+ \sum_{t=1}^{T} \int \log p_\theta(r_r|L_t)\, p_{\theta_k}(L_t|r_{1:T}) dL_t \ , \end{aligned} \tag{4.11}$$

now, utilizing the filtering weights, provides us with the desired approximation of the log-likelihood function

$$\begin{aligned}\hat{Q}(\theta,\theta_k) &= \sum_{i=1}^{N} \pi_1^{(i,\theta_k)} \log p(L_1^{(i,\theta_k)}) \\ &+ \sum_{t=2}^{T}\sum_{i=1}^{N} \pi_t^{(i,\theta_k)} \log p_\theta(L_t^{(i,\theta_k)}|L_{t-1}) \\ &+ \sum_{t=1}^{T}\sum_{i=1}^{N} \pi_t^{(i,\theta_k)} \log p_\theta(r_r|L_t^{(i,\theta_k)}) \ .\end{aligned} \tag{4.12}$$

Where $L_{1:t}^{(i,\theta_k)}$ denotes the state particle i under initial parameter guess θ_k at time t.

In the M-step, one uses a gradient optimization approach to maximize Q with respect to θ of which the gradient can be approximated using the expression from above. Noting that the initial state distributions are independent of θ gives us

$$\begin{aligned}\frac{\partial \hat{Q}}{\partial \theta} &= \sum_{t=2}^{T}\sum_{i=1}^{N} \pi_t^{(i,\theta_k)} \frac{\partial}{\partial \theta} \log p_\theta(L_t^{(i,\theta_k)}|L_{t-1}) \\ &+ \sum_{t=1}^{T}\sum_{i=1}^{N} \pi_t^{(i,\theta_k)} \frac{\partial}{\partial \theta} \log p_\theta(r_r|L_t^{(i,\theta_k)}) \ .\end{aligned} \tag{4.13}$$

Putting it in context of the model at hand, in which the the partial derivatives $\frac{\partial \log p_\theta(r_t|L_t^{(i,\theta_k)})}{\partial \theta}$ are given in Malevergne and Sornette [2014]. Note that the state transition densities are given by

$$p_\theta(L_t^{(i,\theta_k)}|L_{t-1}) = p_\theta(I_t^{(i,\theta_k)}) p_\theta(J_t^{(i,\theta_k)}) \ , \tag{4.14}$$

where

$$p_\theta(I_t^{(i,\theta_k)}) = (1-\lambda_t)^{1-I_t^{(i,\theta_k)}} \lambda_t^{I_t^{(i,\theta_k)}} \ , \tag{4.15}$$

$$p_\theta(J_t^{(i,\theta_k)}) = e^{-J_t^{(i,\theta_k)}} \ . \tag{4.16}$$

Hence, given that the process driving the conditional jump probability X_t only depends on the parameters $\bar{r}, \bar{X}, \eta$ and a (X_t depends on $\bar{r}$ through the initial condition), the partial derivatives of the jump time state transition density, are given by

$$\frac{\partial \log p_\theta(I_t)}{\partial \theta_i} = \left(\frac{I_t}{\lambda_t} - (1-I_t)\frac{1}{1-\lambda_t}\right)\frac{\partial \lambda_t}{\partial \theta_i} \ , \tag{4.17a}$$

$$= \left(\frac{I_t}{\lambda_t} - (1-I_t)\frac{1}{1-\lambda_t}\right)L'(X_t)\frac{\partial X_t}{\partial \theta_i} \ , \tag{4.17b}$$

where

$$\frac{\partial X_t}{\partial \bar{r}} = -\eta + a\frac{\partial X_{t-1}}{\partial \bar{r}} \ , \tag{4.18}$$

$$\frac{\partial X_t}{\partial \bar{X}} = (1-a) + a\frac{\partial X_{t-1}}{\partial \bar{X}} \ , \tag{4.19}$$

$$\frac{\partial X_t}{\partial \eta} = (r_{t-1} - \bar{r}) + a\frac{\partial X_{t-1}}{\partial \eta} \ , \tag{4.20}$$

$$\frac{\partial X_t}{\partial a} = -\bar{X} + X_{t-1} + a\frac{\partial X_{t-1}}{\partial a} \ , \tag{4.21}$$

with initial conditions

$$\frac{\partial X_0}{\partial \bar{r}} = 0 \ , \tag{4.22}$$

$$\frac{\partial X_0}{\partial \bar{X}} = 1 \ , \tag{4.23}$$

$$\frac{\partial X_0}{\partial \eta} = 0 \ , \tag{4.24}$$

$$\frac{\partial X_0}{\partial a} = 0 \ . \tag{4.25}$$

The transition density for the jump sizes is independent of the parameters. Hence, $\frac{\partial}{\partial\theta} \log p_\theta(L_t|L_{t-1})$ depends only on the partial derivative of the state transition density for the jumps.

The approach in wil [2008] uses a particle smoother where the smoothing weights in our case are given by:

$$q_{t-1}^{(i)} = \frac{p(L_t^{(i)}|L_{t-1}^{(i)})}{\sum_{j=1}^N p(L_t^{(i)}|L_{t-1}^{(j)})} \ , \tag{4.26a}$$

$$= \frac{p(I_t^{(i)})p(J_t^{(i)})}{\sum_{j=1}^N (p(I_t^{(i)})p(J_t^{(i)}))} \ , \tag{4.26b}$$

$$= \frac{1}{N} \ . \tag{4.26c}$$

The smoothed particle set, $\{\tilde{L}_t^{(i,\theta_k)}, 1/N\}$, is then used to approximate the partial derivatives of Q. We note, however, that since the weights identical

$$\frac{\partial \hat{Q}}{\partial \theta} = \sum_{t=2}^{T} \frac{\partial}{\partial \theta} \log p_\theta(\bar{\tilde{I}}_t^{(\theta_k)}) + \sum_{t=1}^{T} \frac{\partial}{\partial \theta} \log p_\theta(r_r|\bar{\tilde{I}}_t^{(\theta_k)}, \bar{\tilde{J}}_t^{(\theta_k)}) \ . \tag{4.27}$$

Where $\bar{\tilde{I}}_t = \frac{1}{N}\sum_{i=1}^N \tilde{I}_t^{(i)}$ and $\bar{\tilde{J}}_t = \frac{1}{N}\sum_{i=1}^N \tilde{J}_t^{(i)}$ represent the means of the smoothed particles.

4.2 State augmentation

In this section, we present a different calibration method directly using the filtering algorithm. Its simplicity comes at the cost of some drawbacks, but the advantage of being easy to understand and implement is considered to outweigh the cons. Actually, Sylvain [2012] also uses state augmentation to perform parameter estimation. Although on a different model, this approach proved to be the most successful. In state augmentation, one simply adds the parameters as new unobservable states. The new state vector then becomes:

$$L_t = (I_t, J_t, \theta_t) \ . \tag{4.28}$$

The time index in θ_t does not mean that the parameters are time varying, but rather that it is the filter estimation at time t. The initial selection of the parameters is done independently, except for (α, β). The latter are drawn from the uniform simplex due to the restriction $\alpha+\beta<1$. Particles are selected according to their usual state values and indirectly to their parameter value, dictating the direction in which they evolve. As explained in the Master thesis by Sylvain [2012], state augmentation increases the complexity of the filtering problem and it becomes more difficult to get good state estimates. In order to avoid sample impoverishment, a lot of particles needs to be drawn. Again, the speed of the algorithm developed in this thesis will become useful.

In order to utilize state augmentation, there are two obvious strategies one can employ with respect to the parameter evolution, namely

- fixed parameters and
- artificial dynamics .

4.2.1 Fixed parameters

This is the simplest form of state augmentation and the parameters are said to follow the dynamics

$$\theta_t = \theta_{t-1}$$

Implementing the parameters as states, the SIR pseudo algorithm becomes

1. Given initial particle set $\{L_{t-1}^{(i)}, \pi_{t-1}^{(i)}\}$, where $L_{t-1}^{(i)} = (I_{t-1}^{(i)}, J_{t-1}^{(i)}, \theta_{t-1}^{(i)})$ we simulate, for $i = 1, ..., N$, new states according to $p(L_t|L_{t-1})$ by first calculating

$$X_t^{(i)} = (1-a_{t-1}^{(i)})\bar{X}_{t-1}^{(i)} + a_{t-1}^{(i)}X_{t-1}^{(i)} + \eta_{t-1}^{(i)}(r_{t-1}-\bar{r}_{t-1}^{(i)}) \ , \tag{4.29}$$

$$(\sigma_t^2)^{(i)} = (\bar{\sigma}^2)_{t-1}^{(i)}(1-\alpha_{t-1}^{(i)}-\beta_{t-1}^{(i)}) + \alpha_{t-1}^{(i)}(r_{t-1}-\bar{r}_{t-1}^{(i)})^2 + \beta_{t-1}^{(i)}(\sigma_{t-1}^2)^{(i)} \ , \tag{4.30}$$

   then jump times and jump sizes are simulated according to

$$I_t^{(i)} \sim \text{Bernoulli}(L(X_t^{(i)})) \ , \tag{4.31}$$

$$J_t^{(i)} \sim \exp(1) \ , \tag{4.32}$$

   lastly, the parameters are evolved according to $\theta_t^{(i)} = \theta_{t-1}^{(i)}$

2. Estimate the filtering weights according to $p(r_t|L_t^{(i)})$ and resample.

Note that the particle selection process, determined by the filtering weights, is greatly influenced by the priors chosen for the parameters. The final estimation is restricted to the set of parameters initially drawn. Hence, one relies on simulating a significant amount of particles in order to explore the parameter space efficiently enough.

4.2.2 Artificial dynamics

Here the parameters are evolved according to some imposed artificial randomness with time decreasing variation. This diminishes sample impoverishment and enables a broader search of the parameter space. However, as pointed out in Sylvain [2012] the artificial dynamics can be dangerous because one particle could jump from one type of behavior to something completely different, even for small changes in parameter values. It would, therefore, be necessary to include some knowledge on how the parameters interact. Taking into account all interaction effects and dependencies become a very complex task. This dissertation will remain focused on the fixed state augmentation approach.

4.2.3 Priors

As previously stated, a crucial and important task of the state augmentation, especially involving fixed parameter dynamics, is selecting suitable prior distributions for θ. Initially, one might select the parameters independently of each other, but this approach falls short right away as $\alpha + \beta < 1$. Hence, one would need to sample at least these jointly.

The approach taken in this thesis is to sample almost all parameters independently from a uniform distribution with reasonable support. The support for the uniform distributions are chosen such that they contain the respective true value. This is necessary as the final estimate will be based on the initial sample.

In more detail, $\bar{r}$, denoting the expected daily growth rate of the risky asset is sampled between $-0.18/250$ and $0.32/250$. This is based on the historical annualized growth rates of the S&P500, taking into account the slight positive trend that; in the long run you can expect about 7% in inflation-adjusted return from investing in the stock market. This in turn would yield a long-term annual volatility of $\bar{\sigma} = 0.25$, and the volatility is therefore sampled between $0.1/\sqrt{250}$ and $0.5/\sqrt{250}$. The daily jump size, $K0$, is said to be strictly positive in this thesis and therefore sampled uniformly between 0.02 and 0.08. $\bar{X}$ between -6 and -5, η between 2 and 4 and lastly $a \sim U[0, 1]$.

In order to optimize the state augmentation procedure, it's clear that more thought can be put into selecting these priors. Nevertheless, it is hoped that generating a large amount of particles will diminish the dependency on the initial values and that the true values are profound enough to be noticed and selected by the algorithm. It is believed that the increased speed, allowing us to run the state augmentation algorithm on longer time series and to generate a lot of particles will improve the results and increase the precisions in the estimates compared with the works of Sylvain [2012]. Furthermore, note that a more sophisticated method of joint parameter and state learning is presented in Johannes and Polson. Attempts were made to implement this algorithm, but thus far not in a satisfactory way.

Chapter 5

Simulations

This chapter is devoted to a simulation study reviewing the model as well as the performance of the filtering algorithms. The first section analyzes the properties of the model. Next section 5.2 shows how the SIR and APF filter performs in estimating the filtering distribution and predicting the latent states. Section 5.3 use state augmentation to calibrate the model at hand. A short discussion on why it may be so difficult to calibrate the model by Malevergne and Sornette [2014] closes the chapter.

5.1 Model simulation and stylized facts

In the subsequent simulations a synthetic time series spanning 40 years will be used. Each time step represents one day and a year consists of 250 trading days. The parameters are chosen to be as follows $\bar{r} = 0.00028, \kappa = 0.04, \bar{X} = -5.0, \eta = 3.0, \bar{\sigma} = 0.25/\sqrt{250}, \alpha = 0.05, \beta = 0.94$ and $a = 0.996$. The parameters taking the values as described here takes advantage of what is commonly observed in real financial returns.

5.1.1 Model simulation

Figure 5.1 displays log prices (in black) as a function of time. Linear growth, in this linear-logarithmic plot, represents exponential growth with a constant rate of return. A convex shape represents rising returns and hence a super-exponential growth in prices. The dotted blue line represents the crash probability at time t and red bars the corresponding jump size. Note how the crash-hazard rate increases as a response to greater mispricing. A crash and its size will determine the subsequent decrease in the crash-hazard rate. The long-term memory of the mispricing index ensures non-negligible probability if the price gap remains large.

A second interesting observation is that the log prices fluctuate more before the peak around 1992, than after. The period before the peak is populated by more frequent jump occurrences as a response to greater mispricing. As the price increases, the risk of negative jumps become larger, in turn causing the expected return to escalate since investors require compensation for the greater uncertainty. The price trajectory will have a positive trend if the jumps do not correct enough and balance out the inherent mispricing. Figure 5.2 shows the corresponding evolution of the volatility and mispricing index. We note how the volatility exhibits large gaps. This illustrates how jumps account for extreme movement in discretely observed prices that are unexplainable by the diffusive component.

Figure 5.3 shows a Gaussian kernel density estimate of the mispricing in figure 5.2 along with the theoretical stationary distribution of λ_t colored in red.

5.1.2 Stylized facts

Financial returns exhibit empirical properties that are now so entrenched in econometric experience that they have been elevated to the status of facts. The stylized facts can be summarized as follows

1. return series are not iid, although showing little sign of serial correlation, their absolute or squared return do,
2. conditional expected returns are close to zero,
3. volatility varies over time and in clusters,
4. return series are leptokurtic and appear in clusters.

In the following each point is considered and estimated from the sample time series generated by the model. Whereby empirically showing how they coincide with stylized facts of real financial returns.

Serial correlation of a covariance-stationary process, denoted autocorrelations (ACF) $\rho(h)$, is estimated according to

$$\rho(h) = \rho(r_h, r_0) = \gamma(h)/\gamma(0) \ , \tag{5.1}$$

where γ denotes the autocovariance function

$$\begin{aligned} \gamma(h) &= \gamma(h,0) \ , && (5.2a) \\ &= \mathbb{E}[(r_t - \mu_t)(r_s - \mu_s)] \ . && (5.2b) \end{aligned}$$

In which μ_t is constant for a covariance stationary process and $h = |t - s|$. Hence the autocorrelations are estimated to be

$$\hat{\rho}(h) = \frac{\sum_{s=1}^{n-h}(r_{s+h} - \bar{r})(r_s - \bar{r})}{\sum_{t=1}^{n}(r_t - \bar{r})} \ . \tag{5.3}$$

The confidence bounds are estimated using the asymptotic result that for long iid time series $\hat{\rho}(h) \sim N(0, 1/n)$. Thus a 95% confidence interval for $\rho(h)$ is given by $\hat{\rho}(h) \pm 1.96/\sqrt{n}$.

A plot displaying the log-returns and squared log-returns with correspond autocorrelations can be found in figure 5.4. Figure 5.2 shows how the volatility varies over time and displays the appearance of clusters, properties following from the GARCH(1,1) process.

In order to test whether the simulated time series from the model is leptokurtic as real financial data are, we estimate the kurtosis. The kurtosis is a measure of the peakedness of a distribution. A kurtosis greater than 3 implies a leptokurtic distribution meaning sharper decline than the normal distribution for values concentrated around the mean and slower decay in the tails causing what is known as heavy tails. In addition to being leptokurtic, real log-returns are often also skewed. A negative skewness indicates that the mean of the data is less than the median. Both the kurtosis and skewness are estimated using the relation

$$\gamma_2 = \frac{\mu_4}{\mu_2} - 3 \ , \tag{5.4}$$

$$\gamma_1 = \frac{\mu_3}{\mu_2^{3/2}} \ , \tag{5.5}$$

where μ_2, μ_3 and μ_4 are the second, third and fourth central moments respectively. Below 5.5 is a histogram of the raw returns simulated from the model and a Gaussian kernel density estimate (blue) along with the best-fitting Gaussian density under the iid assumption. In the header the estimated moments from above are displayed.

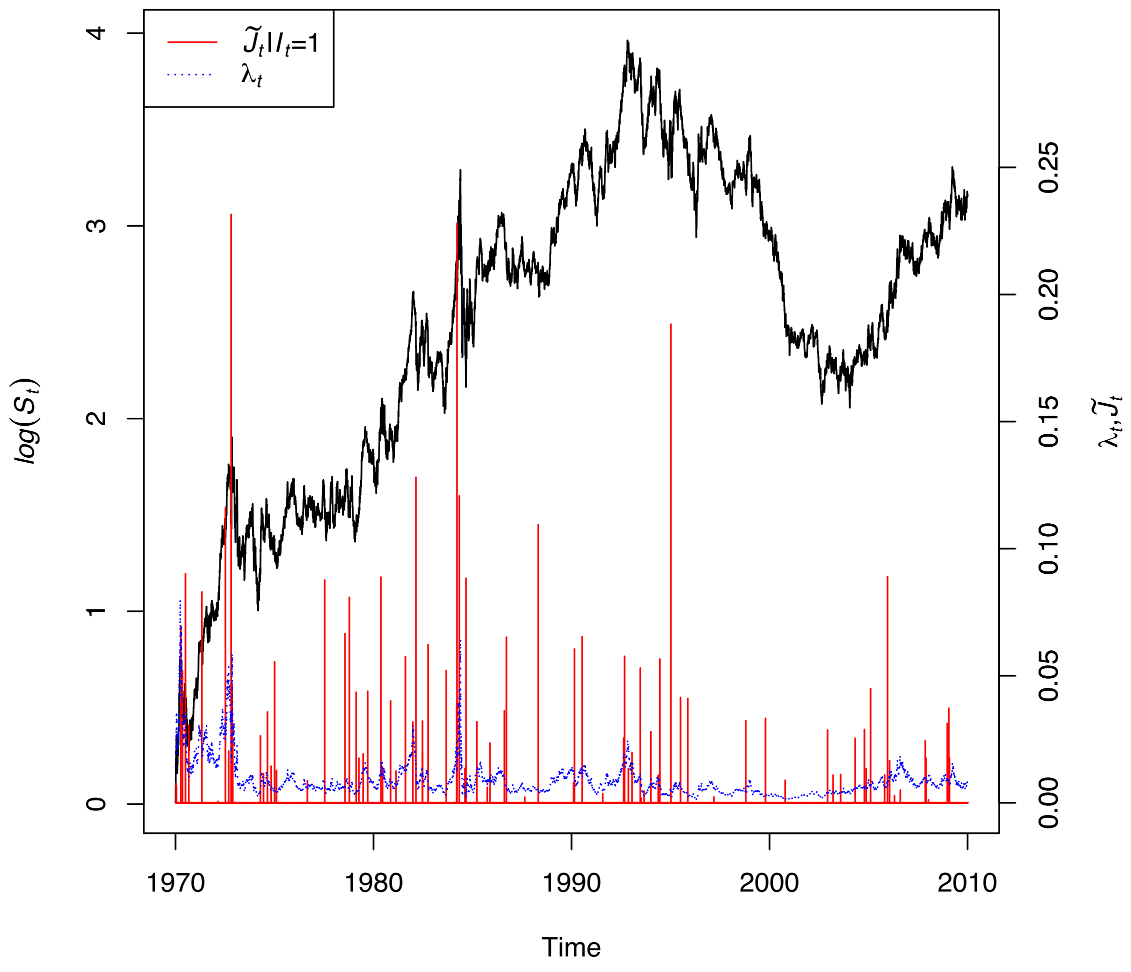

Figure 5.1: Simulated time series of the jump-diffusion model of bubbles and crashes with non-local behavior of length T=10 000,
black: log prices ref. left y-axis,
blue: Jump probability ref. right y-axis,
red: Jump sizes ref. right y-axis.

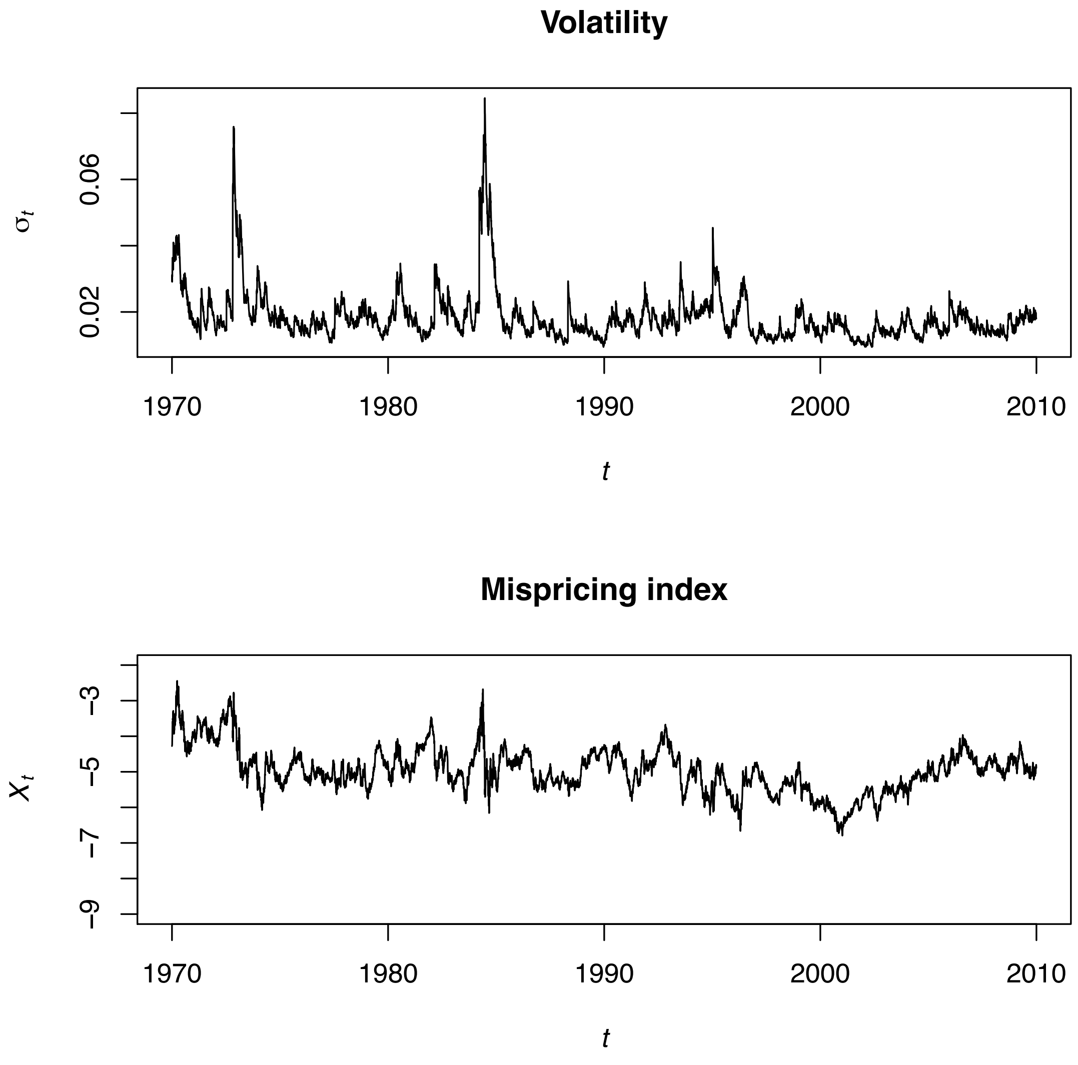

Figure 5.2: Trajectory of the volatility and mispricing index of the simulated jump-diffusion model.

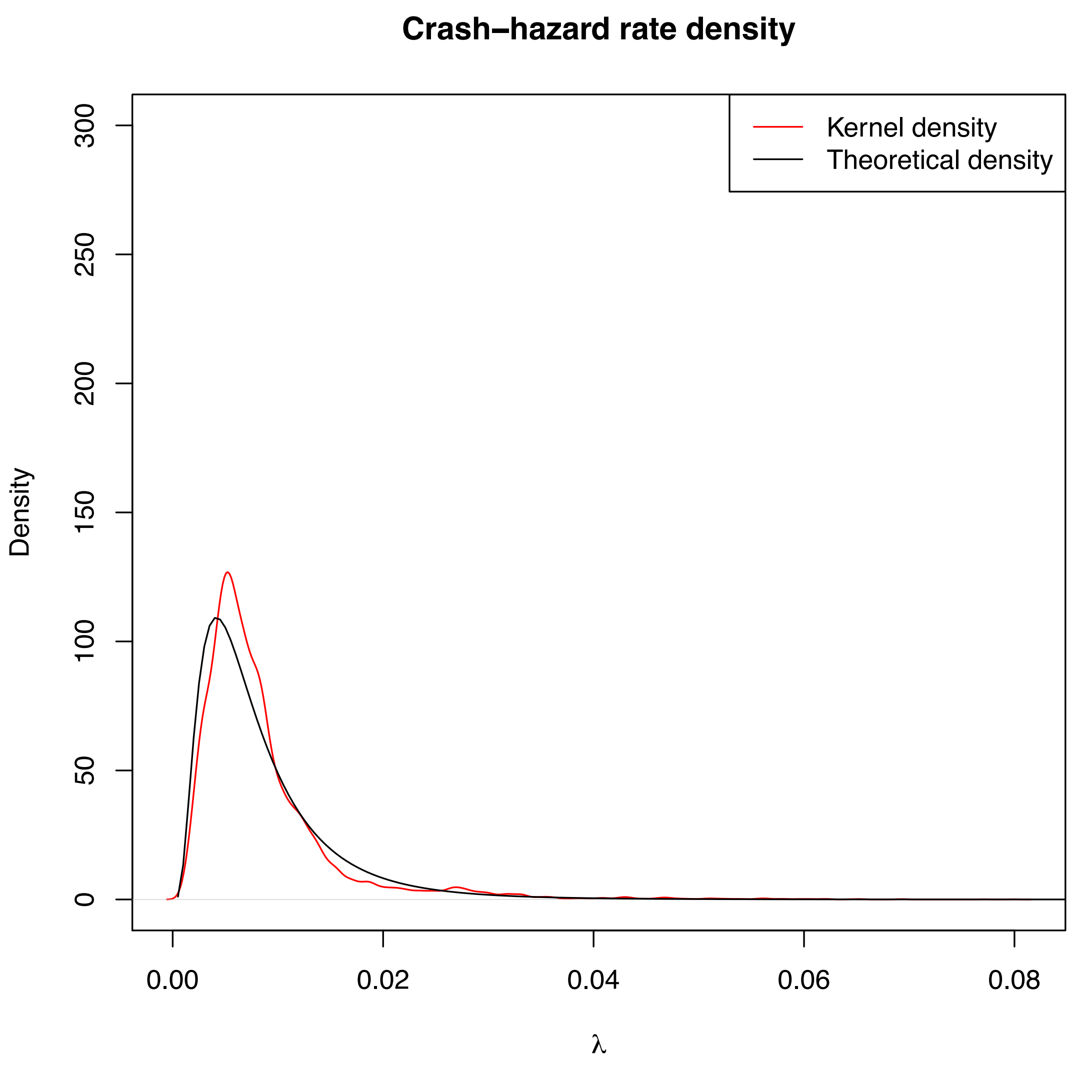

Figure 5.3: Gaussian kernel density estimate (red) of the simulated series from above and theoretical (black) stationary distribution of λ_t.

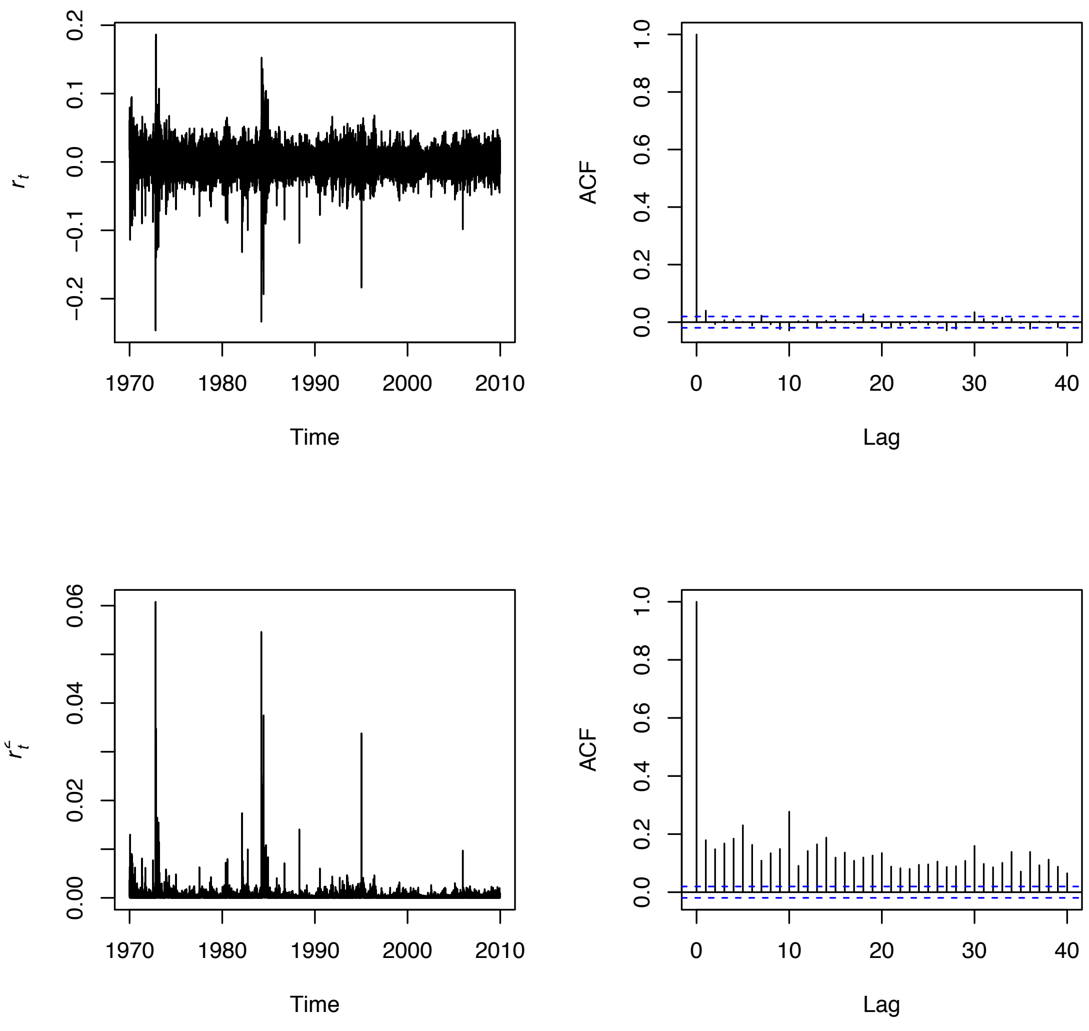

Figure 5.4: The upper left panel depicts the trajectory of log-returns and to the upper right its corresponding autocorrelations. The lower left figure shows the squared log returns whereas the lower right its corresponding autocorrelation.

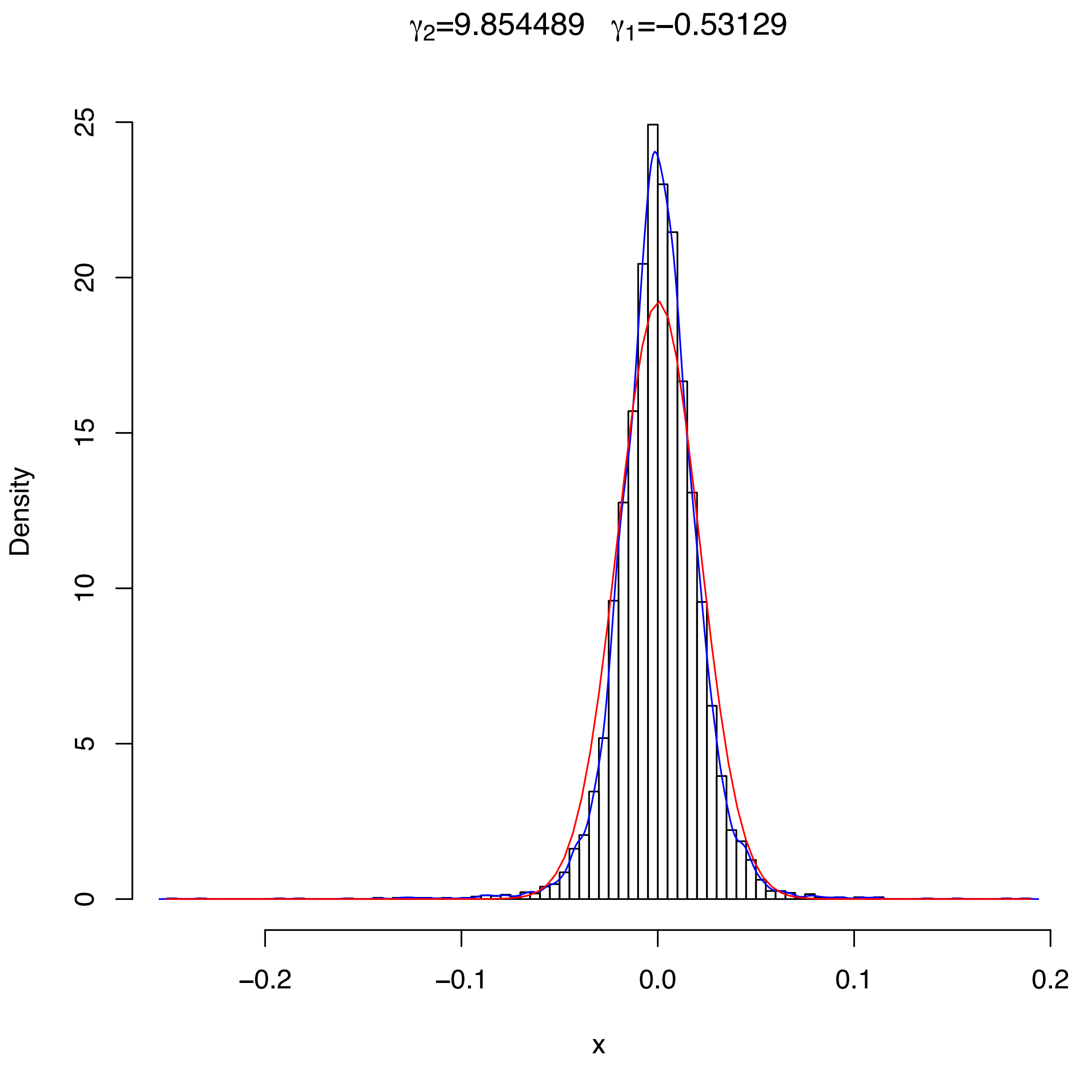

Figure 5.5: The plot shows a histogram of the log-returns and corresponding Gaussian kernel density estimate (blue) as well as best-fitting Gaussian distribution using MLE (red) under the assumption that the log-returns are iid. On the top, the kurtosis and skewness, respectively are calculated using the raw log-returns.

5.2 Filtering of the states and convergence

This section, conditional on the parameters being known, analyzes the performance of the filter. The APF algorithm developed in chapter 3 is run on the time series above. $N = 10000$ particles are generated. We choose $X = \bar{X}$ and $\sigma^2 = \bar{\sigma}^2$ and initialize the particles by selecting the jump times I_o from a Bernoulli distribution with initial success probability $L(X)$ and jump sizes from an exponential distribution with parameter 1/0.01.

5.2.1 Filtering of the states

Figure 5.7 shows the filtering of the underlying jump times and jump sizes using the APF and figure 5.6 the corresponding SIR filter. The true jump times are given by the black dots and the red crosses show the actual jump sizes with reference to the right y-axis. The black bars show, with reference to the left y-axis, how well the algorithm filters jump times. A bar stretching all the way up to 1 implies that the filter fully recovers the actual jump time.

The algorithm identifies large jumps very well, but misses many of the smaller ones. In particular jumps smaller than the true average jump size, 0.04, is not well accounted for. In this case, the price movements cannot be separated well enough from normal day-to-day movements.

Large jumps, being fully recovered and recognized, are very well accounted for in terms of their amplitudes. In particular, the APF filter performs remarkably well and the estimation error is small. One may improve the estimation power further by recognizing that the algorithm performs poorly in the case of small jumps that are of similar amplitude as normal returns. It also produces jumps, though negligible small, where in reality no jump has occurred. Increasing the jump-size threshold above which the algorithm starts assigning greater likelihood to jumps may not yield a significant improvement in estimation power. The algorithm already assigns a very low likelihood (small black bars) to the misclassifications. There are only very few cases, especially at the beginning of the time series, in which the filter assigns 'significant' probability to the misclassifications.

If data is sampled less frequent than daily, the algorithms ability to separate jumps from diffusion deteriorate and its judgment becomes even more clouded. This is because volatility aggregates and weekly variance is roughly 5 times larger than daily. Hence, sampling data at a greater frequency may yield an improvement referring to the missclassifications mentioned above.

Finally, note that there is only a marginal improvement of using the APF over the SIR for estimating the latent states in case of known parameters. As computation time is critical in parameter estimation, this master thesis proceeds with the SIR filter.

5.2.2 Convergence of the particle filter

In this section the method for model checking described in section 3.3 is applied. The predictive distribution function is obtained as a by-product of the filtering. Computing $\hat{P}(r_t|r_{1:t-1})$ for every observation enables us to analyze the convergence of the filtering algorithm through the histogram 5.8. The movie displays the evolvement of the histogram throughout time of the estimated predictive cdf, $\hat{P}(r_r|L_{t-1})$ resulting from the SIR filter. The result is quite convincing, with a distribution approaching the uniform distribution, implying that the estimated predictive cdf converges to the true predictive distribution $P(r_t|L_{t-1})$.

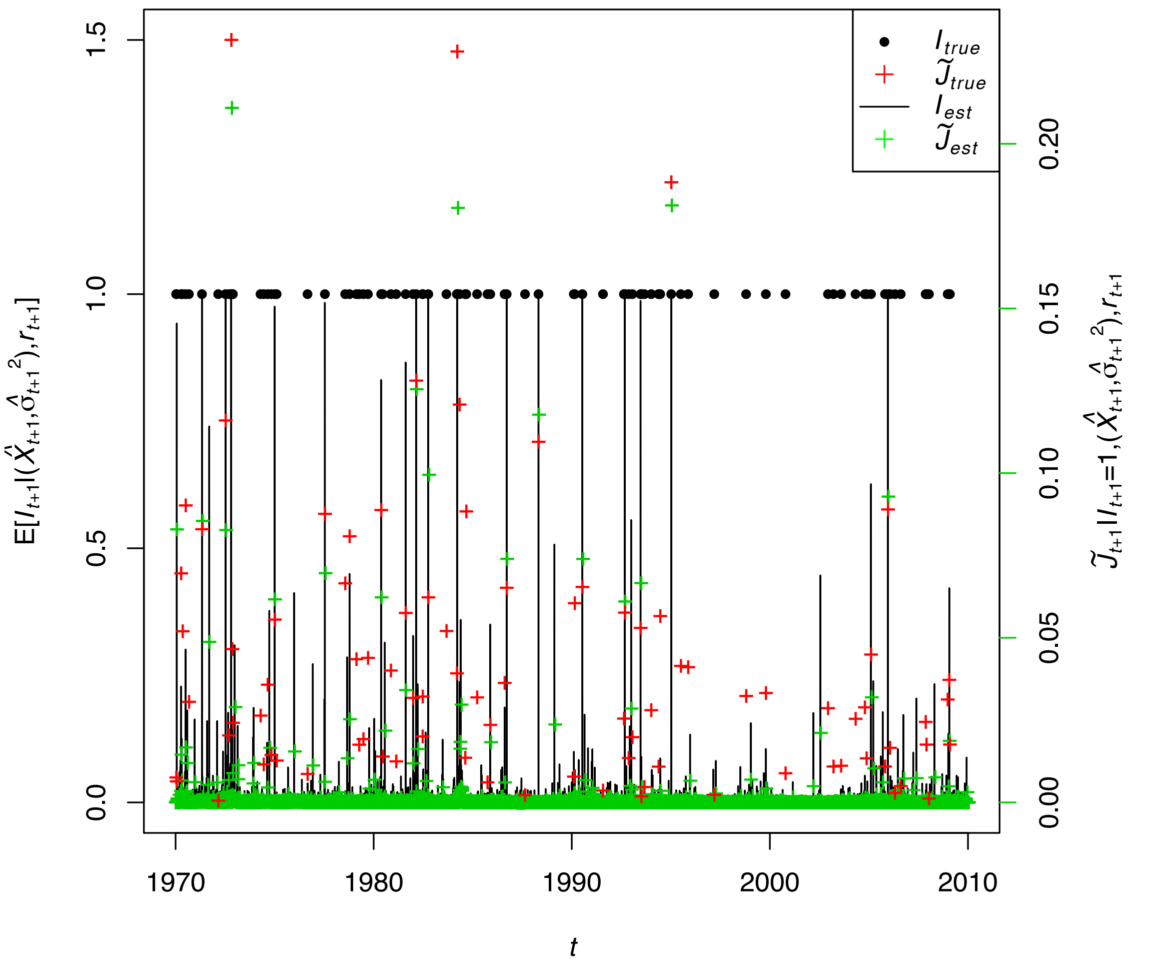

Figure 5.6: Filtering of the states using the SIR filter with N = 10 000 particles on a time series of length T = 10 000. The complete simulation took 49 sec (real time) on a Mac 2.7 GHz Intel Core i7 with OS X Yosemite and code written in C++. The true jump times are given by the black dots and the red crosses show the actual jump sizes with reference to the right y-axis. The black bars show, with reference to the left y-axis, displays how well the algorithm filters jump times. A bar stretching all the way up to 1 implies that the filter fully recovers the actual jump time.

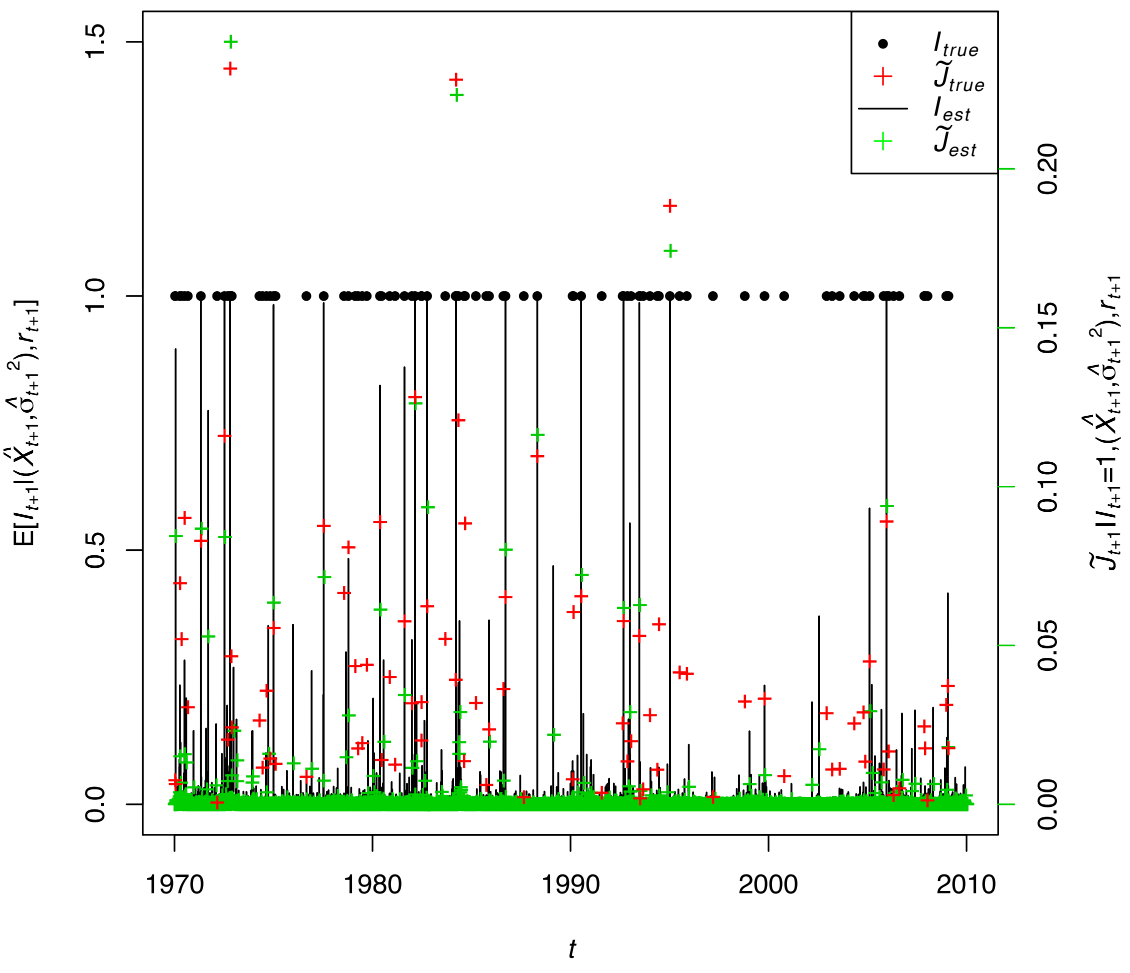

Figure 5.7: Filtering of the states using the APF filter with N = 10 000 particles on a time series of length T = 10 000. The complete simulation took 148 sec (real time) on a Mac 2.7 GHz Intel Core i7 with OS X Yosemite and code written in C++. The true jump times are given by the black dots and the red crosses show the actual jump sizes with reference to the right y-axis. The black bars show, with reference to the left y-axis, displays how well the algorithm filters jump times. A bar stretching all the way up to 1 implies that the filter fully recovers the actual jump time.

Figure 5.8: Histogram of $\hat{P}(r_t|r_{1:t-1})$ displaying the convergence of the estimated predictive cdf resulting from the filtering compared with the true predictive cdf. The histogram approaches, as t increases, the uniform distribtuion. The simulation study is conducted using the SIR filter with N = 10 000 particles on a time series of length T = 10 000. $\hat{P}(r_t|r_{1:t-1})$ is then computed every time step

5.3 Parameter estimation

In this section, the performance of the SIR filter to estimate the parameters is tested. In the subsequent simulations, we restrict ourselves to $N = 1000$ particles. We note that while 1000 particles may be enough to get good estimates for the states in the case of known parameters, it is actually very likely to be insufficient in the present case. If, however, the goal is to display the evolution of the estimates, restriction to a smaller number of particles is necessary. This is because otherwise the storage requirement and computation time becomes too overwhelming.

5.3.1 Parameter cross-sections

Here we display some cross sections of the parameter space. This is done to show how the filter can be used to directly apply the maximum likelihood. While keeping all parameters set at their true values, the log likelihood is computed over a grid of values for both the set (α, β) and a respectively. The result is displayed in figure 5.9 and figure 5.10. Notice that the shape of the log-likelihood as a function of (α, β) behaves quite well in the sense that it is possible to locate a maximum log-likelihood. The maximum log-likelihood of the simulation study is given by the pair $(\alpha, \beta) = (0.068965, 0.89655)$ which is not far from the true value $(\alpha, \beta) = (0.05, 0.94)$ of the simulated series. Note that the grid of $\theta's$ that were generated did not contain the true value and, therefore, might be the reason for the minor estimation bias.

The log-likelihood as a function of a does not behave as well, seen in figure 5.10. The area is very flat with a rough surface. The surface may be spikey due to jagged log-likelihood estimates produced by the filter. This suggests that the estimates of the states become more difficult with varying a.

In concluding this section, note that the cross sections do not take into account the parameter interdependencies. Hence, it does not really give a satisfactory view on how the parameter space actually looks like. Full application, directly maximizing the log-likelihood depends on computing the log-likelihood over a sufficiently dense parameter grid. This is too exhaustive computationally and gradient-based optimization algorithms are, therefore, necessary.

Sadly, the EM algorithm presented in section 4.1.1, was not implemented successfully within the timeframe of the thesis. It is, however, believed that it will not yield a satisfactory result. Malevergne and Sornette [2014] shows that direct maximization of the likelihood does not perform well. This comes from the fact that the log-likelihood as a function of some of the parameters are very flat. The gradient of the log-likelihood is, therefore, close to zero for a wide range of values outside the true ones. This in turn makes the gradient optimization cumbersome as some parameters can be changed with large magnitude without affecting the value of the log-likelihood too much.

5.3.2 State augmentation

Running the state augmentation with a satisfactory number of particles and feasible computation time meant restricting the above time series to 1000 observations, or equivalently

4 trading years. Further supporting the simplification we have that particle filters are dependent upon the initialization of the particles and in general on the seed used. The above simulation, therefore, needs to be repeated, with different seeds each run, to display the consistency in the estimates. Hence, there is a need for balancing the length of the time series, the number of particles and lastly the number of repetitions to get sound estimates using this approach. It's expected that if the algorithm works properly the estimates of the parameters will not fluctuate too much. In the subsequent simualtion the state augmentation algorithm was applied to the shortened time series of log-returns with length $T = 1000$ and using $N = 5000$ particles. This simulation procedure was then repeated $M = 200$.

Figure 5.11 shows the result of the simulation. The black dots represent a mean estimate produced from repeating state augmentation with static parameter evolution 200 times. The shaded gray area represents one estimated standard deviation away from the mean. Narrow bands will correspond to consistent estimates. The red lines in the plots represent the true parameter value.

The mean and standard deviation estimates are produced by taking the mean of the filter means produced at each run. That is, if $\hat{\theta}_t^{(j)}$ represents the filter mean at time t from the j^th^, i.e.

$$\hat{\theta}_t^{(j)} = \sum_{i=1}^{N} \pi_t^{(i)} \theta_t^{(i,j)} \ . \tag{5.6}$$

The black dotted line in figure 5.11 represents

$$\hat{\theta}_t = \frac{1}{M} \sum_{j=1}^{M} \hat{\theta}^{(j)} \ . \tag{5.7}$$

Notice that that the estimates are very poor for nearly all parameters. Except for maybe $\bar{\sigma}, \alpha$ and β the consistency in the estimates are bad and most standard deviation range estimates almost recover the whole support of the prior distributions chosen for the parameters. Also notice how, after around time step 200, only a small subset of the original particles survive due to sample depletion. This is visible because the estimation stops to vary after this point. The problem of sample depletion may be alleviated by applying the dynamic state augmentation approach. Focusing on the bright side, the estimates for α and β are rather good. The mean estimate displayes a clear convergence towards the true value. The spread in the estimates may, however, still be too wide in order to get accurate estimates, but scaling up the simulation would yield better results.

5.4 Conclusion of simulations

The EM tries to maximize the log-likelihood of the model by successively switching between estimating the states using the particle filter and a prior parameter guess and then maximize the log likelihood of the model with respect to the parameters. As the log-likelihood depends not only on the observations and the parameters but also on the unobservable states the EM algorithm may perform poorly when the estimates from the filter are bad. This is especially so whenever the probabilities are very jagged as a result of sample impoverishment. Evolutionary algorithms are methods intended to solve this problem by sampling particles in a smart way. The main idea behind is to combine the most probable particles in such a way that the fit becomes better as time increases by replacement of poorly fitting particles.

Even if the particle estimates would be good, Malevergne and Sornette [2014] shows that maximization of the log-likelihood does not perform well. The objective function is sloppy in the sense of Waterfall, Casey, Gutenkunst, Brown, Myers, Brouwer, Elser, and Sethna [2006] meaning that there are directions of the log-likelihood as a function of the eight parameters that are very flat. This can be seen as the gradient of the log-likelihood is consistently close to zero in a wide range outside the true parameter values. Hence, one can impose large changes in parameters almost maintaining the value of the log-likelihood. The EM algorithm, utilizing the particle filtering does therefore not perform well and one needs to use the filter in a different manner.

State augmentation did not yield any improvement in the efforts of model calibration. The 95% confidence bands were nearly as wide as the supports for the prior distributions of θ. This again shows how a wide range of parameters may be plausible for the model and calibration proved difficult using simple static state augmentation.

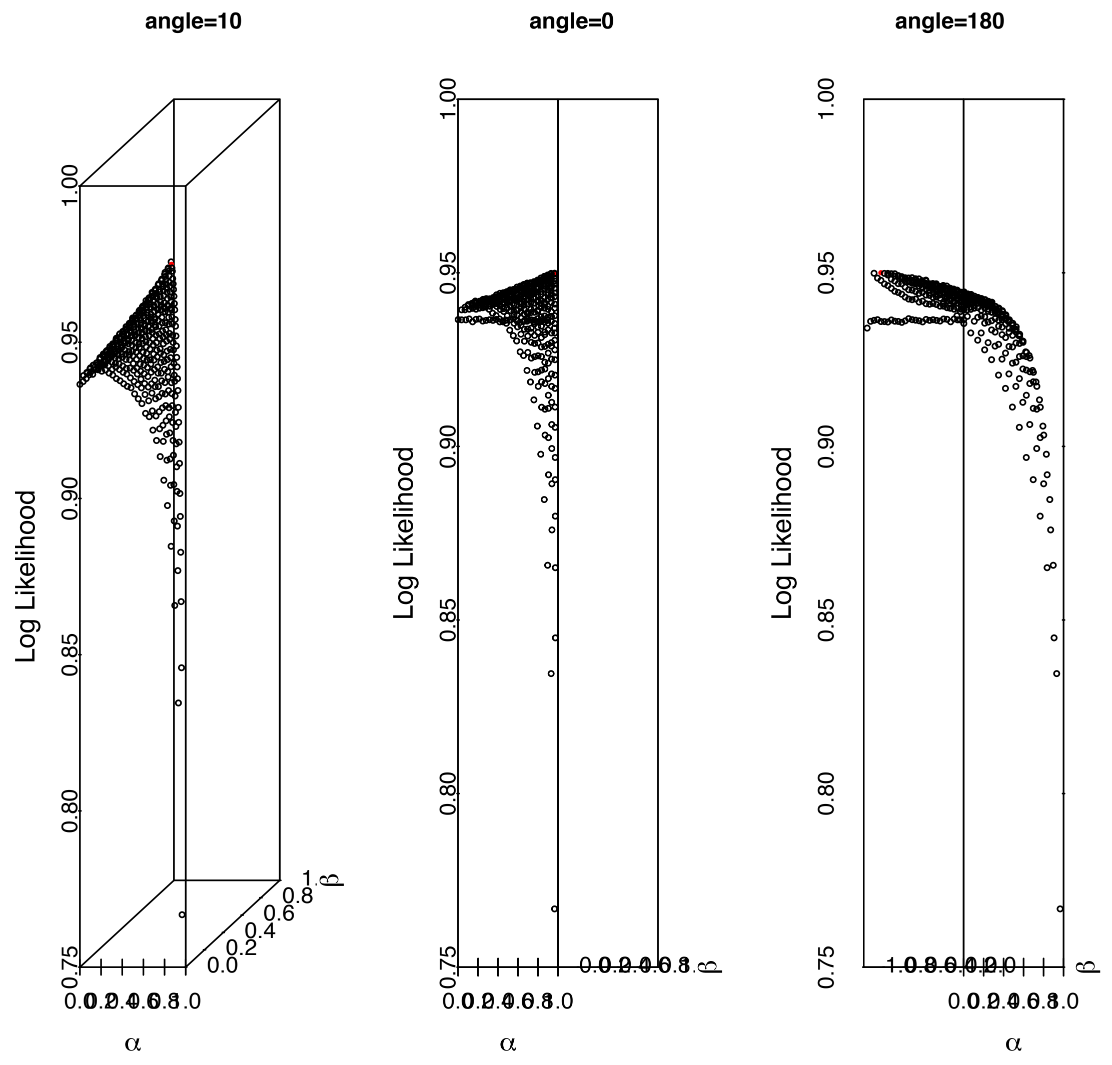

Figure 5.9: Log-likelihood as a function of (α, β) keeping all other parameters set to their true value. The three panels display a different viewpoint of the 3-dimensional scatterplot, in which angle denotes the angle between the α- and β-axis. Running the SIR particle filter with N = 1 000 particles on a time series of length T = 10 000. The simulation is repeated according to the number of grid points for θ and corresponded to run the filter 435 times. The black points correspond to the log-likelihood of the given parameter value. The red point corresponds to the maximum log-likelihood value over the given (α, β) grid and corresponds to $(0.068965, 0.89655)$, true value is given by $(0.05, 0.94)$. The complete simulation took 1 487 sec (real time) on a Mac 2.7 GHz Intel Core i7 with OS X Yosemite and code written in C++.

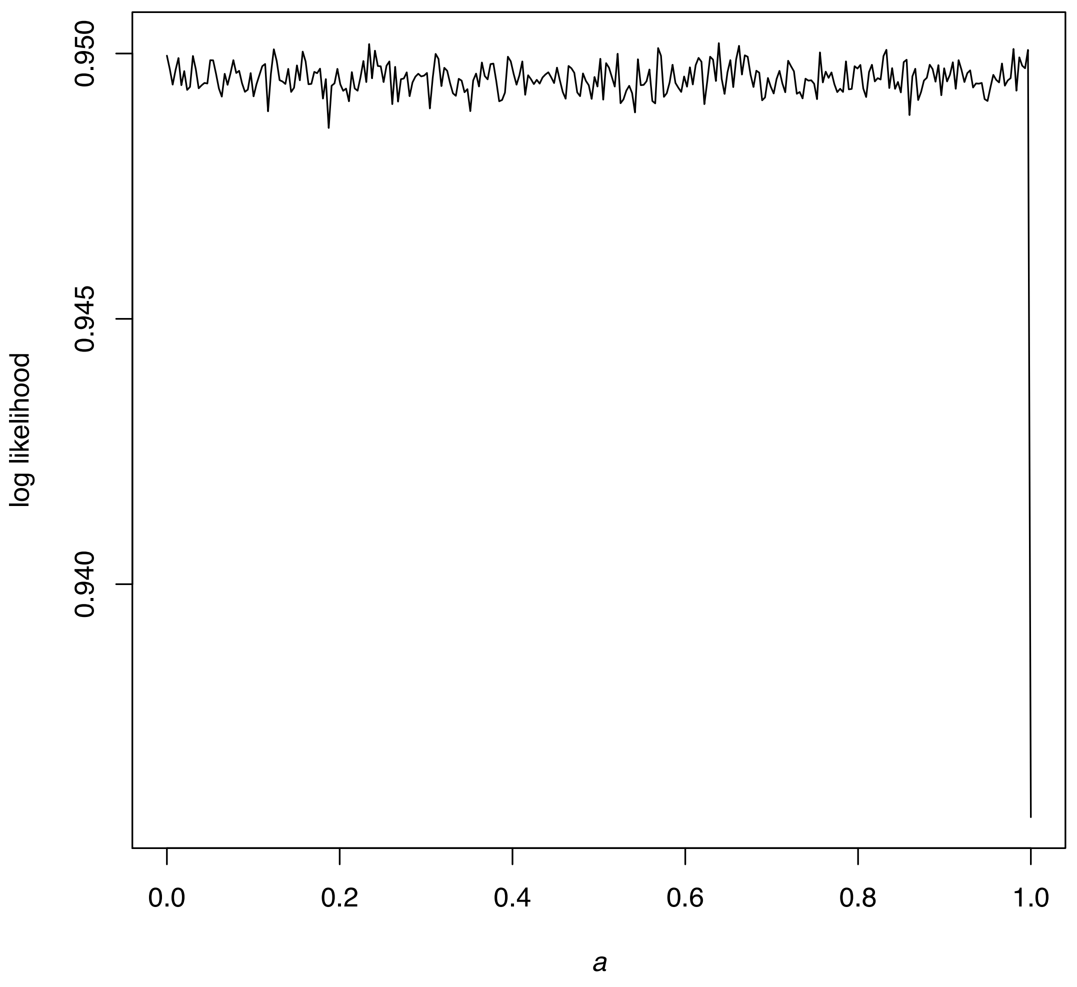

Figure 5.10: Plot showing the log-likelihood as a function of a, and keeping the remaining parameters set equal to their true values. The true value for a is given by 0.996. Running the SIR particle filter with N = 1 000 particles on a time series of length T = 10 000. The simulation is repeated according to the number of grid points generated for θ and corresponded to run the filter 300 times. The complete simulation took 1 016 sec (real time) on a Mac 2.7 GHz Intel Core i7 with OS X Yosemite and code written in C++.

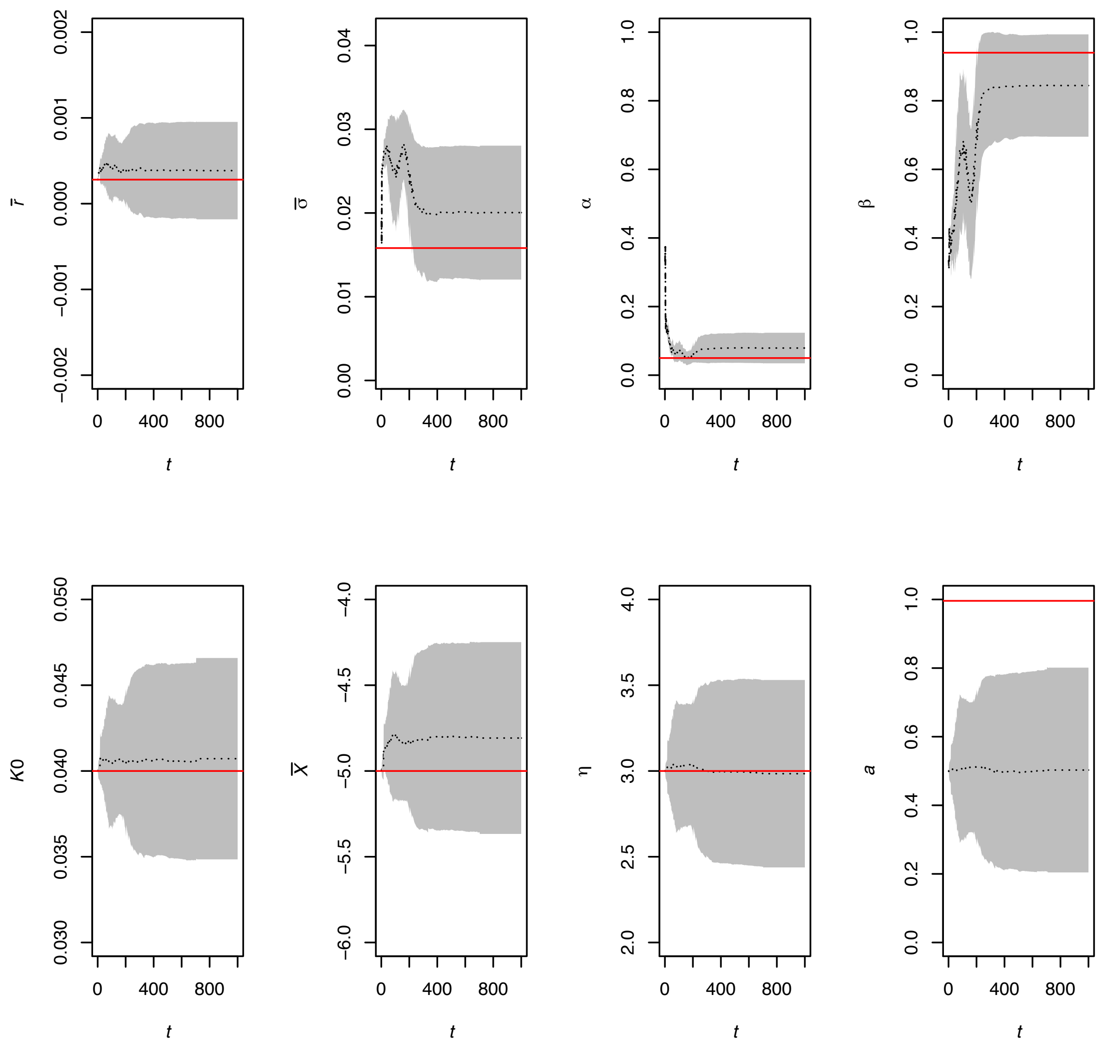

Figure 5.11: Black dotted line: the mean parameter estimate from the repeated simulations. Gray area: represents a sandard deviation from the mean. Red line: true parameter value. Running the SIR particle filter with N = 5 000 particles on a time series of T=1 000 observations. The simulation is then repeated M = 200. The complete simulation took 740 sec (real time) on a Mac 2.7 GHz Intel Core i7 with OS X Yosemite and code written in C++.

Chapter 6

Conclusion

The main work of this master thesis was concerned with adapting a fast sequential Monte Carlo algorithms, in particular a sequential importance resampling - and an auxiliary particle filter algorithm, to the model developed by Malevergne and Sornette [2014]. Quantitative properties and advantages of the model where highlighted and explained. Synthetic data was generated enabling model inference as well as checking the performance of the filters. The algorithms proved effective in identifying reasonably large sized jumps, which is important for practical applications. In particular, movements in log-returns exceeding a threshold corresponding to the true average crash size is effectively recognized by the algorithm. It was noted that the filter produced estimates of jumps even if in reality no jump had occurred. The estimated jumps were, however, in this case negligible and the likelihood assigned to the falsely generated jumps small.

A secondary goal, utilizing the particle filter, was to calibrate the model. It was explained how the EM-algorithm may be adapted and state augmentation was implemented and tested for the model at hand. Simulations in chapter 5 demonstrated that calibration is not straightforward. As also pointed out in Malevergne and Sornette [2014], the logllikelihood as a function of some of the parameters is very flat so that they can be changed in combination with large factors without altering significantly the log-likelihood value. Using fixed evolution for the parameters, state augmentation applied the filter directly to select the best fitting particles and thereby obtaining an estimate for the parameters. This attempt was shown to be unsuccessful as the estimate range of nearly all parameters almost fully recovered the support of the prior distributions for θ. The algorithm, however, proved somewhat successfully in estimating α and β.

It is well known that parameter estimation of these complex systems of equations is hard. Referring to the master theses Bertolace [2009], Sylvain [2012], Lagerqvist [2014], although on a different model yielded a similar conclusion. Lagerqvist [2014] concluded that the parameter estimation works poorly due to model sloppiness, a term introduced by Brown, Hill, Calero, Myers, Lee, Sethna, and Cerione [2004]. A similar study may be performed for the model at hand.

In future work we encourage development of an algorithm for sequentially learning about the state variables and parameters utilizing and building upon the algorithms developed in this dissertation. It would be very interesting to see if this yields any improvements in calibration over the EM and state augmentation pursued in this thesis.

Another line of research is to design a smarter way to evolve the parameters and select better prior distributions for θ. This would utilize a more effective exploration of the parameter space, referring to state augmentation in chapter 4. This would also yield a better starting point for employment of a random search algorithm such as simulated annealing, genetic algorithms etc., in which one can also build upon the present filter.

A more critical question is related to the actual fit and applicability of this model. How do we test whether the crash-hazard rate follows the dynamics proposed in the works of Malevergne and Sornette [2014]. Since the true crash-hazard rate cannot be observed, it is not evident how to do so. A possible way to check the validity could be to systematically test predictions generated by the model and compare with reality. There are numerous bubbles characterized by super-exponential growth followed by crashes, but also evidence of 'sustainable' bubbles. That is, prices that experience super-exponential growth which are not followed by a crash but rather stabilization in price on a higher level. Hence, we know in advance that some of these predictions will fall short. Extending the model to include a distinction between these two types of bubbles would indeed yield a significant improvement.

Bibliography

Parameter estimation for discrete-time nonlinear systems using EM, 2008. ISBN 9783902661005. doi:10.3182/20080706-5-KR-1001.00675.

T.G Andersen, T. Bollerslev, F.X. Diebold, and P. Labys. Modeling and forecasting realized volatility. *Econometrica*, 71(2):579–625, 2003. doi:10.1111/1468-0262.00418.

O.E. Barndorff-Nielsen. Econometric analysis of realized volatility and its use in estimating stochastic volatility models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 64(2):253–280, 2002. doi:10.1111/1467-9868.00336.

O.E. Barndorff-Nielsen and N. Shephard. Estimating quadratic variation using realized variance. *Journal of Applied Econometrics*, 17(5):457–477, 2002. doi:10.1002/jae.691.

D.S. Bates. The crash of '87: Was it expected? the evidence from options markets. *The Journal of Finance*, 46(3):1009–1044, 1991. doi:10.1111/j.1540-6261.1991.tb03775.x.

S.D. Bates. How crashes develop: intradaily volatility and crash evolution. `http://www.biz.uiowa.edu/faculty/dbates/` or `http://tippie.uiowa.edu/people/profile/profile.aspx?id=194916`, May 2015.

A. Bertolace. Study of a nonlinear model of the price of an asset: Kalman filter calibration to data. Master's thesis, ETH, 2009. URL `http://www.er.ethz.ch/media/publications/phd-and-master-theses.html`.

O.J Blanchard. Speculative bubbles, crashes and rational expectations. *Economics Letters*, 3(4):387–389, 1979. doi:10.1016/0165-1765(79)90017-X.

B. Bollen and B. Inder. Estimating daily volatility in financial markets utilizing intraday data. *Journal of empirical finance*, 9(5):551–562, 2002. doi:10.1016/S0927-5398(02)00010-5.

K. Brown, C. Hill, G. Calero, C. Myers, K. Lee, J. Sethna, and R. Cerione. The statistical mechanism of complex signaling networks: Nerve growth factor signaling. *Physical Biology*, 1:184–195, 2004.

J. Carpenter, P. Clifford, and P. Fearnhead. Improved particle filter for nonlinear problems. *IEE Proceedings - Radar, Sonar and Navigation*, 146(1):2, 1999. doi:10.1049/ip-rsn:19990255.

R Cont and P. Tankov. *Financial modelling with jump processes*, volume 2. Chapman Hall/CRC, 2004.

D. Crisan and A. Doucet. A survey of convergence results on particle filtering methods for practitioners. *Signal Processing, IEEE Transactions on*, 50(3):736–746, 2002. doi:10.1109/78.984773.

S. Das. *Particle filtering on large dimensional state spaces and applications in computer vision*. PhD thesis, Iowa State University, 2010.

L. Devroye. *Sample-based non-uniform random variate generation*. 1986. ISBN 0-911801-11-1. doi:10.1145/318242.318443.

V. Filimonov and D. Sornette. Apparent criticality and calibration issues in the hawkes self-excited point process model: application to high-frequency financial data. *Quantitative Finance*, 15(8):1293–1314, 2015. doi:10.1080/14697688.2015.1032544.

Michael S. Johannes and Nick G. Polson. Particle filtering and parameter learning (march 2007). Available at SSRN: `http://ssrn.com/abstract=983646` or `http://dx.doi.org/10.2139/ssrn.983646`.

Michael S. Johannes and Nick G. Polson. *Particle filtering*. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-540-71297-8.

Michael S. Johannes, Nick G. Polson, and Seung M. Yae. Sequential inference for nonlinear models using slice variables (november 19, 2009). Available at SSRN: `http://ssrn.com/abstract=1509782` or `http://dx.doi.org/10.2139/ssrn.1509782`.

Michael S Johannes, Nick G Polson, and J.R. Stroud. Optimal filtering of jump diffusions: Extracting latent states from asset prices. *Review of Financial Studies*, 22(7):2759–2799, 2009. doi:10.1093/rfs/hhn110.

A. Johansen and D. Sornette. Shocks, crashes and bubbles in financial markets. *Brussels Economic Review*, 53(2):201–253, 2010. URL `http://EconPapers.repec.org/RePEc:bxr:bxrceb:2013/80942`.

A. Johansen, D. Sornette, and O. Ledoit. Predicting financial crashes using discrete scale invariance. *Journal of Risk*, 1(4):5–32, 1999.

A. Johansen, O. Ledoit, and D. Sornette. Crashes as critical points. *International Journal of Theoretical and Applied Finance*, 3(2):219–255, 2000. ISSN 0219-0249. doi:10.1142/S0219024900000115.

T. Kaizoji, M. Leiss, A. Saichev, and D. Sornette. Super-exponential endogenous bubbles in an equilibrium model of fundamentalist and chartist traders. *Journal of Economic Behavior & Organization*, 112:289–310, 2015. doi:10.1016/j.jebo.2015.02.001.

C.P. Kindleberger and R.Z. Aliber. *Manias, panics and crashes: a history of financial crises*. Palgrave Macmillan, 2011. ISBN 9781403936516. doi:10.1057/9780230628045.

A. Lagerqvist. Parameter estimation of a non-equilibrium asset pricing model and performance analysis of the calibration in terms of sloppiness. Master's thesis, ETH, 2014. URL `http://www.er.ethz.ch/media/publications/phd-and-master-theses.html`.

C. Lee, A.C. Lee, and S. Wang. *Encyclopedia of Finance*, chapter Jump diffusion model, pages 525–534. Springer, 2013. ISBN 978-1-4614-5359-8. doi:10.1007/978-1-4614-5360-4.

Y. Malevergne and D. Sornette. Jump-diffusion model of bubbles and crashes with non-local behavioral self-referencing. 2014.

A.J. McNeil, R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques, and Tools: Concepts, Techniques, and Tools*. Princeton Series in Finance. Princeton University Press, 2005. ISBN 9780691122557.

R.C. Merton. Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, 3(1-2):125–144, 1976. doi:10.1016/0304-405X(76)90022-2.

G. Poyiadjis, S.S. Singh, and A. Doucet. Gradient-free maximum likelihood parameter estimation with particle filters. In *American Control Conference*, page p.6 pp., 2006. ISBN 1-4244-0209-3. doi:10.1109/ACC.2006.1657187.

C.P. Robert. Simulation of truncated normal variables. *Statistics and Computing*, 5(2): 121–125, 1995. doi:10.1007/BF00143942.

T.B. Schön, A. Wills, and B. Ninness. System identification of nonlinear state-space models. *Automatica*, 47(1):39–49, 2011. doi:10.1016/j.automatica.2010.10.013.

D. Sornette. *Why stock markets crash: Critical events in complex financial systems.* Princeton University Press, Princeton NJ, 2009. doi:10.1515/9781400829552.

D. Sornette and Y. Malevergne. From rational bubbles to crashes. *Physica A: Statistical Mechanics and its Applications*, 299(1):40–59, 2001. doi:10.1016/S0378-4371(01)00281-3.

R. Sylvain. Sequential monte carlo methods for a dynamical model of stock prices. Master's thesis, ETH, 2012. URL `http://www.er.ethz.ch/media/publications/phd-and-master-theses.html`.

J.J. Waterfall, F.P. Casey, R.N. Gutenkunst, K.S. Brown, C.R. Myers, P.W. Brouwer, V. Elser, and J.P. Sethna. Sloppy-model universality class and the vandermonde matrix. *Phys. Rev. Lett.*, 97:150601, Oct 2006. doi:10.1103/PhysRevLett.97.150601.

M.J. Werner, K. Ide, and D. D. Sornette. Earthquake forecasting based on data assimilation: sequential monte carlo methods for renewal point processes. *Nonlin. Processes Geophys.*, 18(1):49–70, 2011. doi:10.5194/npg-18-49-2011.

X. Yang and K. Xing. *Stochastic Optimization - Seeing the Optimal for the Uncertain*, chapter Joint State and Parameter Estimation in Particle Filtering and Stochastic Optimization. InTech, 2011. ISBN 978-953-307-829-8. doi:10.5772/14658.

Y. Yukalov, D. Sornette, and E.P. Yukalova. Nonlinear dynamical model of regime switching between conventions and business cycles. *Journal of Economic Behavior Organization*, 70(1-2):206–230, 2009. doi:10.1016/j.jebo.2008.12.004.

Appendices

Appendix A

Pseudo code

A.1 SIR particle filter algorithm

Using the notation in Johannes et al. [2009] the SIR particle filter is implemented using the following generic algorithm:

1. given initial particles $\{(I_{t-1}^{(i)}, J_{t-1}^{(i)}), \pi_{t-1}^{(i)}\}_{i=1}^{N}$ we simulate new states $(I_t^{(i)}, J_t^{(i)})$ by first computing

$$X_t = (1-a)\bar{X} + aX_{t-1} + \eta(r_{t-1} - \bar{r}) \ , \tag{A.1}$$

$$\sigma_t^2 = \bar{\sigma}^2(1-\alpha-\beta) + \alpha(r_{t-1} - \bar{r})^2 + \beta\sigma_{t-1}^2 \ , \tag{A.2}$$

then for $i = 1, ..., N$

$$\hat{I}_t^{(i)} \sim \text{Bernoulli}(L(X_t)) \ , \tag{A.3}$$

$$\hat{J}_t^{(i)} \sim \exp(1) \text{ if } \hat{I}_t^{(i)} = 1 \ . \tag{A.4}$$

Then evaluate the filtering weights according to

$$w_t^{(i)} \propto p(r_t \mid \hat{L}_t^{(i)}) \tag{A.5a}$$

$$= \phi(r_t|\mu = (\bar{r} + \kappa L(X_t) - \kappa J_t^{(i)} I_t^{(i)}, \sigma = \sigma_t) \tag{A.5b}$$

Resample the particles by drawing indices

$$z(i) \sim \text{Mult}(N; w_t^{(1)}, ..., w_t^{(N)}) \ , \tag{A.6}$$

and set $(I_t^{(i)}, J_t^{(i)}) = (I_t^{z(i)}, J_t^{z(i)})$.

A.2 APF particle filter algorithm

Following the APF algorithm approach in Johannes et al. [2009] the pseudo code becomes:

1. given initial particles $\{(I_{t-1}^{(i)}, J_{t-1}^{(i)}), \pi_{t-1}^{(i)}\}_{i=1}^{N}$ we compute $(\hat{I}_t^{(i)}, \hat{J}_t^{(i)})$ by first computing

$$X_t = (1-a)\bar{X} + aX_{t-1} + \eta(r_{t-1} - \bar{r}) \ , \tag{A.7}$$
$$\sigma_t^2 = \bar{\sigma}^2(1-\alpha-\beta) + \alpha(r_{t-1} - \bar{r})^2 + \beta\sigma_{t-1}^2 \ , \tag{A.8}$$

then for $i = 1, ..., N$

$$\hat{I}_t^{(i)} \sim \text{Bernoulli}(L(X_t)) \ , \tag{A.9}$$
$$\hat{J}_t^{(i)} \sim \exp(1) \text{ if } \hat{I}_t^{(i)} = 1 \ . \tag{A.10}$$

Then evaluate the first stage weights, using $\{\hat{I}_t^{(i)}, \hat{J}_t^{(i)}\}$, according to

$$w_t^{(i)} \propto p(r_t \mid \hat{L}_t^{(i)}) \ , \tag{A.11a}$$
$$= \phi(r_t|\mu = (\bar{r} + \kappa L(X_t) - \kappa J_t^{(i)} I_t^{(i)}, \sigma = \sigma_t) \ . \tag{A.11b}$$

Resample the intial particles by drawing indices

$$z(i) \sim \text{Mult}(N; w_t^{(1)}, ..., w_t^{(N)}) \ , \tag{A.12}$$

and set $(I_{t-1}^{(i)}, J_{t-1}^{(i)}) = (I_{t-1}^{z(i)}, J_{t-1}^{z(i)})$ and $(\hat{I}_t^{(i)}, \hat{J}_t^{(i)}) = (\hat{I}_t^{z(i)}, \hat{J}_t^{z(i)})$.

2. Generate new states, i.e number of jumps I_t from $p(I_t \mid X_t, \sigma_t^2, r_t)$, using the probabilities $\{p_k^{(i)}\}_{k=0}^{1}$ defined by

$$p_0^{(i)} = \lambda_t \cdot f_{emg}(r_t; \bar{r} + \kappa\lambda_t, \sigma_t, \kappa) \ , \tag{A.13}$$
$$p_1^{(i)} = (1-\lambda_t) \cdot \phi(r_t; \bar{r} + \kappa\lambda_t, \sigma_t) \ . \tag{A.14}$$

3. Generate the total jump sizes $\tilde{J}_t$ from $p(\tilde{J}_t \mid I_t = 1, X_t, \sigma_t^2, r_t)$.
4. Finally compute the second stage weights.

$$\pi_t^{(i)} \ , \tag{A.15}$$

where $\lambda_t = 1/(1 + exp(X_t))$.

A.3 Fixed state augmentation

Fixed state augmentation follows the generic algorith as follows:

1. At t=0, For i=1,....N simulate $\theta_0^{(i)} = ((\bar{r}^{(i)}, k^{(i)}, \bar{X}^{(i)}, \eta^{(i)}, w^{(i)}, \alpha^{(i)}, \beta^{(i)}, a^{(i)}))$ according to some prior distribution and similarly for the jump times and sizes
2. Given initial particles $\{(I_{t-1}^{(i)}, J_{t-1}^{(i)}), \theta_{t-1}^{(i)}\}_{i=1}^{N}$ compute $(X_t^{(i)}, (\sigma_t^2)^{(i)})$ for each parameter particle $\theta_{t-1}^{(i)}$ and set $\theta_t^{(i)} = \theta_{t-1}^{(i)}$ and simulate new jump times and sizes according to

$$\hat{I}_t^{(i)} \sim \text{Bernoulli}(L(X_t^{(i)})) \ , \tag{A.16}$$
$$\hat{J}_t^{(i)} \sim \exp(1) \text{ if } \hat{I}_t^{(i)} = 1 \ . \tag{A.17}$$

3. Estimate the filtering weights at time t according to

$$\pi_t^{(i)} = p(r_t \mid L_t^{(i)}) \ , \tag{A.18a}$$

$$= \phi(r_t|\mu = (\bar{r}_t^{(i)} + \kappa_t^{(i)} L(X_t^{(i)}) - \kappa_t^{(i)} J_t^{(i)} I_t^{(i)}, \sigma = \sigma_t^{(i)}) \ . \tag{A.18b}$$

4. Resample the particles by drawing indices

$$z(i) \sim \text{Mult}(N; \pi_t^{(1)}, ..., \pi_t^{(N)}) \ , \tag{A.19}$$

and set $(I_t^{(i)}, J_t^{(i)}) = (I_t^{z(i)}, J_t^{z(i)})$ and $\theta_t^{(i)} = \theta_t^{z(i)}$

Appendix B

Code

B.1 SIR C++ code

Listing B.1: SIR C++ code

```
/*          Input           */
// Time series of log returns
// Number of particles

/*          Output          */
//      I      J            */
//      pi          loglik  */

#include<iostream>          // Include in/out function
#include<fstream>
#include<string>
#include<vector>
#include<random>
#include<math.h>
#include<cstdio>
#include<ctime>
#include<time.h>
#include<algorithm>         // for low_bound and upper_bound
#include<iterator>          // std::begin, std::end
#include<iomanip>           // for setw
using namespace std;        // The namespace

#ifndef M_PI
    #define M_PI 3.14159265358979323846264L
#endif

// Normal cumulative density function:
long double pnorm(long double q, long double mu=0.0L, long double sd=1.0L) {
  const long double SQRTHL = 7.071067811865475244008e-1L;
  long double x, y, z;

  x = (q-mu)/sd * SQRTHL;
  z = abs(x);

  if( z < SQRTHL )
    y = 0.5L + 0.5L * erf(x);
  else
    {
      y = 0.5L * erfc(z);

      if( x > 0.0L )
        y = 1.0L - y;
    }
  return y;
}

// sgn function
long double sign(long double x) {
  if (x >= 0.0L) {
    return 1.0L;
  }
  else {
    return -1.0L;
  }
}

// Normal density:
long double dnorm(long double x, long double mu=0.0L, long double sd2=1.0L) {
  // sd2 denotes the variance
  return exp(-.5L*(x-mu)*(x-mu)/sd2-.91893853320467274178L)/sqrt(sd2);
}

// Exponential modified Gaussian density:
long double demg(long double x, long double mu, long double sd, long double k) {
  return 1.0L/abs(k)*exp((x-mu)/k + sd*sd/(2.0L*k*k)) *
    (1.0L - pnorm((sign(k)*((x-mu)/sd + sd/k))));
}

long double L(long double x) {
  return (1.0L/(1.0L+exp(-1.0L*x)));
}

int main() {

  random_device rd;
  uint32_t seed=time(NULL);

  // Number of Monte Carlo simulations:
  unsigned int N = 10000;

  // Exact values:
  long double mu; long double sigma0; long double alpha; long double beta;
  long double K0; long double Xbar; long double eta;long double a;
  mu = 0.00028L; sigma0 = sqrt((0.25L*0.25L)/250.0L); alpha = 0.05L;
  beta = 0.94L; K0 = 0.04L; Xbar = -5.0L; eta = 3.0L; a = 0.996L;

  // Reading data from simulated the Jump Diffusion model:
  //------------------------------------------------------------------------------//
  // Length of time series:
  unsigned int Ts = 0;

  vector<long double> r;
  vector<long double> X_real;
  vector<long double> V_real;
  vector<int> chrash_activity;
  vector<long double> chrash_size;

  FILE * myFile;
```

```
myFile = fopen("sim10000.csv", "r");

if (myFile!=NULL) {
  cout << "Successfully␣opened␣the␣file␣\n";

  long double aux_r, aux_X_real, aux_V_real, aux_chrash_size;
  int aux_chrash_activity;

  while (fscanf(myFile, "%Lf,%Lf,%i,%Lf,%Lf\n",
                &aux_X_real, &aux_V_real, &aux_chrash_activity,
                &aux_chrash_size, &aux_r) == 5) {
    X_real.push_back (aux_X_real);
    V_real.push_back (aux_V_real);
    chrash_activity.push_back (aux_chrash_activity);
    chrash_size.push_back (aux_chrash_size);
    r.push_back (aux_r);
    Ts++;
  }

  fclose (myFile);
  cout << "Data␣read␣\n";
}
else {
  cout << "Unable␣to␣open␣the␣file␣\n";
  return 0;
}

cout << "Ts=" << Ts << endl;
cout << "Random␣Seed=" << rd() << endl;
cout << "Random␣Seed=" << seed << endl;

unsigned int T = Ts*N;

vector<unsigned int> I(T);
vector<long double> J(T);
vector<long double> pi(T);

// For checking convergence in particle filters:
vector<double> FF(Ts);

// Distributions needed for the computations / algorithm:
default_random_engine generator;
generator.seed( seed ); // Seeded differently each time

exponential_distribution<long double> exponential(1.0L);
normal_distribution<long double> normal(0.0L,1.0L);
uniform_real_distribution<long double> unif(0.0L,1.0L);

cout << "End␣initialisation" << endl;

//---------------------------------------------------------------------------//
// Vector containing the first stage weights at time t:
vector<long double> W(N);

// Vector containing the normalizing first stage weights at time t
vector<long double> nW(N);
// Vector containing the cumulative sum of the normalized first stage weights
// at time t+1
vector<long double> Q((N+1));
```

```
// Vector containing the position of the re-sampled states at time t+1
vector<unsigned int> B(N);

// Vectors containg the resampled states at time t
vector<long double> is(N);
vector<long double> js(N);

vector<long double> TT((N+1));

// Timing the SIR algorithm;
clock_t start1;
double duration1;
duration1 = 0;

start1 = clock();
//---------------------------------------------------------------------------//

// Initialisation , sampling initial states :
bernoulli_distribution bernoulli_init(0.0006);
exponential_distribution<long double> exponential_init(20.0L);

for (unsigned int n=0; n<N; n++) {
  I[n] = bernoulli_init(generator);
  J[n] = I[n]*exponential_init(generator);
  pi[n] = 1.0L/N;
}

cout << "Initial␣sampling␣done" << endl;

// The log likelihood sum for the given theta:
long double lls = 0;

cout << "Starting␣APF␣particle␣filter" << endl;

long double X = Xbar;
long double V = sigma0 * sigma0 * (1 - alpha - beta);

for (unsigned int t=0; t<(T-N); t+=N)
  {
    X = (1.0L - a) * Xbar + a*X +
      eta * (r[t/N] - mu);
    V = sigma0 * sigma0 * (1.0L - alpha - beta) + alpha *
      pow((r[t/N] - mu), 2.0L) + beta * V;
    bernoulli_distribution bernoulli1(L(X));

    // Simulating new state variables
    for (unsigned int n=0; n<N; n++) {
      I[n+t+N] = bernoulli1(generator);
      if (I[n+t+N] == 1) {
        J[n+t+N] = exponential(generator);
      }
    }

    // computing the filtering weights at time t:
    for (unsigned int n=0; n<N; n++) {
      W[n] = dnorm(r[(t+N)/N],
                   (mu + K0 * L(X) - K0 * J[n+t+N] * I[n+t+N]) , V);
    }

    // First stage resampling from the discrete distribution
    // {L_{t+1}, W_{t+1}} using algorithm in multinomialsampling.pdf

    // Normalizing the weights:
    long double s1;
    s1 = 0.0L;
    for (int n=0; n<N; n++) {
      s1 += W[n];
    }

    // Resampling step
    for (unsigned int n=0; n<N; n++) {
      nW[n] = W[n]/s1;
      // Calculating the cumulative sum of the normalized weights:
      // corresponding to Q:
      if (n == 0) {
        Q[n] = 0.0L;
        TT[n] = exponential(generator);
      }
      if (n > 0) {
        Q[n] = Q[n-1] + nW[n-1];
        TT[n] = TT[n-1] + exponential(generator);
      }
    }
    Q[N] = Q[(N-1)] + nW[(N-1)];
    TT[N] = TT[(N-1)] + exponential(generator);

    unsigned int i=0; unsigned int j=1;
    while (i < N) {
      if ( TT[i] < (Q[j] * TT[N])) {
        B[i] = (j-1);
        is[i] = I[(j-1) + t + N];
        js[i] = J[(j-1) + t + N];
        i++;
      }
      else {
        j++;
      }
    }
    // Updating the states
    for (unsigned int n=0; n<N; n++) {
      I[n+t+N] = is[n];
      J[n+t+N] = js[n];
    }

    // Computing the log likelihood at each time step
    long double llst = 0.0L;

    for (unsigned int n=0; n<N; n++) {
      // Using the unnormalised probability weights at time t
      llst += W[n];
    }
    long double ss;
    for (unsigned int n=0; n<N; n++) {
      pi[n+t+N] = W[n];
      ss = pi[n+t+N]/llst;
      pi[n+t+N] = ss;
    }

    // Updating the complete log-likelihood of the full time series:
    lls += log(llst);
```

```
    //cout << t << "  " << lls << "\n";
    if (isnan(lls) == 1) {
      cout << "NaN␣log␣likelihood␣value␣!" << "\n";
      break;
    }

    // Checking convergence of particle filters:
    double SC;
    SC = 0;
    for (unsigned int j=0; j<N; j++) {
      SC += pi[j+t] * pnorm(r[(t+N)/N],
                            mu + K0 * L(X) - K0 * J[j] * I[j],
                            sqrtl(V));
    }
    FF[t/N] = SC;

} // end of time loop

duration1 += (clock() - start1) / (double) CLOCKS_PER_SEC;
cout << duration1 << "seconds"  << endl;

//---------------------------------------------------------------------------------//
// Estimating the filtered means:
vector<long double> I_mean(Ts);
vector<long double> J_mean(Ts);

for (unsigned int t=0; t<T; t+=N) {
  long double i_mean = 0.0L;
  long double j_mean = 0.0L;
  long double j_weight = 0.0L;
  for (unsigned int n=0; n<N; n++) {
    // Calculating the conditional moment of
    // E[I]
    i_mean += 1.0L / N * I[t+n];
    if (I[t+n] == 1) {
      j_weight += 1.0L;
    }
  }

  for (unsigned int n=0; n<N; n++) {
    if (I[n+t] ==  1) {
      // Calculating the conditional mean of J
      j_mean += 1.0L / j_weight * J[t+n];
    }
    else {
      j_mean += 0.0L;
    }
  }

  I_mean[t/N] = i_mean;
  J_mean[t/N] = j_mean;
}
// Saving the data:
ofstream myfile_sim;
myfile_sim.open ("SIR10000.csv");
for (unsigned int i=0; i<Ts; i++) {
  myfile_sim << I_mean[i] << "," << J_mean[i]
             << "\n";
}
myfile_sim.close();
```

```
  //--------------------------------------------------------------------------//
  // Saving the convergence data:
  ofstream myfile_sim2;
  myfile_sim2.open ("convergenceSIR.csv");

  for (unsigned int i=0; i<Ts; i++) {
    myfile_sim2 << FF[i] << "\n";
  }
  myfile_sim2.close();
  //--------------------------------------------------------------------------//
  return 0;
}
```

B.2 APF C++ code

Listing B.2: APF C++ code

```
/*          Input           */
// Time series of log returns
// Number of particles

/*          Output          */
//      I      J            */
//      pi          loglik  */

#include<iostream>          // Include in/out function
#include<fstream>
#include<string>
#include<vector>
#include<random>
#include<math.h>
#include<cstdio>
#include<ctime>
#include<time.h>
#include<algorithm>         // for low_bound and upper_bound
#include<iterator>          // std::begin, std::end
#include<iomanip>           // for setw
using namespace std;        // The namespace

#ifndef M_PI
    #define M_PI 3.14159265358979323846264L
#endif

// Normal cumulative density function:
long double pnorm(long double q, long double mu=0.0L, long double sd=1.0L) {
  const long double SQRTHL = 7.071067811865475244008e-1L;
  long double x, y, z;

  x = (q-mu)/sd * SQRTHL;
  z = abs(x);

  if( z < SQRTHL )
    y = 0.5L + 0.5L * erf(x);
  else
    {
      y = 0.5L * erfc(z);

      if( x > 0.0L )
        y = 1.0L - y;
    }
  return y;
}

// sgn function
long double sign(long double x) {
  if (x >= 0.0L) {
    return 1.0L;
  }
  else {
    return -1.0L;
  }
}

// Normal density:
long double dnorm(long double x, long double mu=0.0L, long double sd2=1.0L) {
  // sd2 denotes the variance
  return exp(-.5L*(x-mu)*(x-mu)/sd2-.91893853320467274178L)/sqrt(sd2);
}

// Exponential modified Gaussian density:
long double demg(long double x, long double mu, long double sd, long double k) {
  return 1.0L/abs(k)*exp((x-mu)/k + sd*sd/(2.0L*k*k)) *
    (1.0L - pnorm((sign(k)*((x-mu)/sd + sd/k))));
}

long double L(long double x) {
  return (1.0L/(1.0L+exp(-1.0L*x)));
}

int main() {

  random_device rd;
  uint32_t seed=time(NULL);

  // Number of Monte Carlo simulations:
  unsigned int N = 10000;

  // Exact values:
  long double mu; long double sigma0; long double alpha; long double beta;
  long double K0; long double Xbar; long double eta;long double a;
  mu = 0.00028L; sigma0 = sqrt((0.25L*0.25L)/250.0L); alpha = 0.05L;
  beta = 0.94L; K0 = 0.04L; Xbar = -5.0L; eta = 3.0L; a = 0.996L;

  // Reading data from simulated the Jump Diffusion model:
  //-------------------------------------------------------------------------------//

  // Length of time series:
  unsigned int Ts = 0;

  vector<long double> r;
  vector<long double> X_real;
  vector<long double> V_real;
  vector<int> chrash_activity;
  vector<long double> chrash_size;

  FILE * myFile;
  myFile = fopen("sim10000.csv", "r");

  if (myFile!=NULL) {
    cout << "Successfully␣opened␣the␣file␣\n";
    long double aux_r, aux_X_real, aux_V_real, aux_chrash_size;
    int aux_chrash_activity;

    while (fscanf(myFile, "%Lf,%Lf,%i,%Lf,%Lf\n",
                  &aux_X_real, &aux_V_real, &aux_chrash_activity,
                  &aux_chrash_size, &aux_r) == 5) {
      X_real.push_back (aux_X_real);
      V_real.push_back (aux_V_real);
      chrash_activity.push_back (aux_chrash_activity);
      chrash_size.push_back (aux_chrash_size);
      r.push_back (aux_r);
      Ts++;
    }

    fclose (myFile);
    cout << "Data␣read␣\n";
  }
  else {
    cout << "Unable␣to␣open␣the␣file␣\n";
    return 0;
  }

  cout << "Ts=" << Ts << endl;
  cout << "Random␣Seed=" << rd() << endl;
  cout << "Random␣Seed=" << seed << endl;

  unsigned int T = Ts*N;

  vector<unsigned int> I(T);
  vector<long double> J(T);
  vector<long double> pi(T);

  // For checking convergence in particle filters:
  vector<double> FF(Ts);

  default_random_engine generator;
  generator.seed( seed ); // Seeded differently each time the code is run

  // Distributions needed for the computations / algorithm:
  exponential_distribution<long double> exponential(1.0L);
  normal_distribution<long double> normal(0.0L,1.0L);
  uniform_real_distribution<long double> unif(0.0L,1.0L);

  cout << "End␣initialisation" << endl;

  //---------------------------------------------------------------------------//
  // Vector containing the first stage weights at time t:
  vector<long double> W(N);

  // Vector containing the normalizing first stage weights at time t
  vector<long double> nW(N);
  // Vector containing the cumulative sum of the normalized first stage weights
  // at time t+1
  vector<long double> Q((N+1));

  // Vector containing the position of the re-sampled states at time t+1
  vector<unsigned int> B(N);

  // Vectors containg the resampled states at time t
```

```
vector<long double> I_hat(N);
vector<long double> i_hat(N);
vector<long double> J_hat(N);
vector<long double> j_hat(N);
vector<long double> is(N);
vector<long double> js(N);

// Vector containing the density of jump sizes at time t
vector<long double> DE1(N);
vector<long double> DE2(N);

// Vectors containg the unormalized / normalized jump probabilities,
// respectivelly, at time t:
vector<long double> p1(N);
vector<long double> P1(N);
// probability of no jump:
vector<long double> P2(N);

vector<long double> TT((N+1));

// Timing the Particle Filter APF algorithm;
clock_t start1;
double duration1;
duration1 = 0;

start1 = clock();
//---------------------------------------------------------------------------//

// Initialisation, sampling initial states :
bernoulli_distribution bernoulli_init(0.0006);
exponential_distribution<long double> exponential_init(20.0L);

for (unsigned int n=0; n<N; n++) {
  I[n] = bernoulli_init(generator);
  J[n] = I[n]*exponential_init(generator);
  pi[n] = 1.0L/N;
}

cout << "Initial␣sampling␣done" << endl;

// The log likelihood sum for the given theta:
long double lls = 0;

cout << "Starting␣APF␣particle␣filter" << endl;

// Initializing the mispricing and volatility
long double X = Xbar;
long double V = sigma0 * sigma0;

for (unsigned int t=0; t<(T-N); t+=N)
  {
    X = (1.0L - a) * Xbar + a*X +
      eta * (r[((t+N)/N - 1)] - mu);
    V =  sigma0 * sigma0 * (1.0L - alpha - beta) + alpha *
      pow((r[((t+N)/N - 1)] - mu), 2.0L) + beta * V;
     bernoulli_distribution bernoulli1(L(X));

     // Evaluating the first stage weights:
     for (unsigned int n=0; n<N; n++) {
       I_hat[n] = bernoulli1(generator);
       if (I_hat[n] == 1) {
         J_hat[n] = exponential(generator);
       }
     }

     // calculating the mean of the log-returns including new observation at
     // time t:
     for (unsigned int n=0; n<N; n++) {
       W[n] = dnorm(r[(t+N)/N],
                    (mu + K0 * L(X) - K0 * J_hat[n] * I_hat[n]),
                    V);
     }

     // First stage resampling from the discrete distribution
     // {L_{t+1}, W_{t+1}} using algorithm in multinomialsampling.pdf

     // Normalizing the weights:
     long double s1;
     s1 = 0.0L;
     for (int n=0; n<N; n++) {
       s1 += W[n];
     }

     // First stage resampling
     for (unsigned int n=0; n<N; n++) {
       nW[n] = W[n]/s1;
       // Calculating the cumulative sum of the normalized weights:
       // corresponding to Q:
       if (n == 0) {
         Q[n] = 0.0L;
         TT[n] = exponential(generator);
       }
       if (n > 0) {
         Q[n] = Q[n-1] + nW[n-1];
         TT[n] = TT[n-1] + exponential(generator);
       }
     }
     Q[N] = Q[(N-1)] + nW[(N-1)];
    TT[N] = TT[(N-1)] + exponential(generator);

     unsigned int i=0; unsigned int j=1;
     while (i < N) {
       if ( TT[i] < (Q[j] * TT[N])) {
         B[i] = (j-1);
         i_hat[i] = I_hat[(j-1)];
         j_hat[i] = J_hat[(j-1)];
         is[i] = I[(j-1)+t];
         js[i] = J[(j-1)+t];
         i++;
       }
       else {
         j++;
       }
     }
     // Updating the states
     for (unsigned int n=0; n<N; n++) {
       I_hat[n] = i_hat[n];
       J_hat[n] = j_hat[n];
       I[n+t] = is[n];
       J[n+t] = js[n];
     }

     // Calculating the jump probability at time t+1:
     for (unsigned int n=0; n<N; n++) {
       P1[n] = L(X) *
         demg(r[(t+N)/N],(mu + K0 * L(X)),sqrtl(V), K0);
       P2[n] = (1.0L - L(X)) *
         dnorm(r[(t+N)/N],(mu + K0 * L(X)), V );
       // Normalized jump probability for n=1,...,N
       // Rounding off using floor
       p1[n] = P1[n] / (P1[n] + P2[n]);
     }

     // Simulating new jumps:
     for (unsigned int n=0; n<N; n++) {
       // Probability of jump for n=1,...,N
       bernoulli_distribution bernoulli(p1[n]);
       I[n+t+N] = bernoulli(generator);
     }

     // Simulate jump sizes in case a jump has occured:
     for (unsigned int n=0; n<N; n++) {
       if (I[n+t+N] == 1) {
         long double a_tilde = -1.0L * (
                                       r[(t+N)/N] - (mu + K0 * L(X) ) +
                                       1.0L / K0 * V
                                       );

         // accept-reject algorithm
         exponential_distribution<long double> expj(1.0L/K0);
         long double zz = expj(generator);
         long double ratio = expl(-
                                (zz - a_tilde)*(zz - a_tilde) /
                                (2.0L * V) -
                                (
                                 a_tilde * 1.0L / K0 + 1.0L / K0 *
                                 1.0L / K0 * V / 2.0L -
                                 1.0L / K0 * zz
                                 )
                                );

         long double u = unif(generator);
         while (u > ratio) {
           zz = expj(generator);
           ratio = exp(-
                       (zz - a_tilde)*(zz - a_tilde) /
                       (2.0L * V) -
                       (
                        a_tilde * 1.0L / K0 + 1.0L / K0 *
                        1.0L / K0 * V / 2.0L -
                        1.0L / K0 * zz
                        )
                       );
           u = unif(generator);
         }

         J[n+t+N] = zz;
       }
       else {
         J[n+t+N] = 0.0L;
       }
       if (J[n+t+N] < 0.0L) {
         break;
       }

     }
     // Calculating the jump size density at time t:
     for (unsigned int n=0; n<N; n++) {
       if (I[n+t+N] == 1) {

         long double a_tilde = - (
                                r[(t+N)/N] -
                                (mu + K0 * L(X)) +
                                1.0L / K0 * V
                                );
         DE1[n] = 1.0L / K0 * expl(- (1.0L / K0) * J[n+t+N]);
         DE2[n] = 1.0L / (1.0L - pnorm(-1.0L * a_tilde / sqrtl(V))) *
           dnorm(J[n+t+N], a_tilde, V);
       }
       else {
         // Density of the jump sizes not defined if no jump has occured..
         // hence it will not affect the discrete probability weights and
         // is therefore set to 1
         DE1[n] = 1.0L;
         DE2[n] = 1.0L;
       }
     }

     // Estimating the unweighted discrete probability weights of the
     // filtering distribution at time t,
     for (unsigned int n=0; n<N; n++) {
       pi[n+t+N] = pi[n+t] *
         ((
           dnorm(r[(t+N)/N],
                 ((mu + K0 * L(X)) -
                  J[n+t+N] * I[n+t+N]
                  ),
                 V
                 ) *
           pow ( L(X), I[n+t+N] ) *
           pow ( (1.0L - L(X)) , (1.0L - I[n+t+N]) ) *
           DE1[n]
           ) / (
                 pow ( p1[n] , I[n+t+N] ) *
                 pow ( (1.0L - p1[n]) , (1.0L - I[n+t+N]) ) *
                 DE2[n] *
                 W[B[n]]
                 )
          );
     }

     // Computing the log likelihood at each time step
     long double llst = 0.0L;

     for (unsigned int n=0; n<N; n++) {
       llst += pi[n+t+N];
     }
     long double ss;
     for (unsigned int n=0; n<N; n++) {
       ss = pi[n+t+N]/llst;
       pi[n+t+N] = ss;
     }

     // Updating the complete log-likelihood of the full time series:
     lls += log(llst);
     //cout << t << "  " << lls << "\n";
     if (isnan(lls) == 1) {
       cout << "NaN␣log␣likelihood␣value␣!" << "\n";
       break;
     }

     // Checking convergence of particle filters:
     double SC;
     SC = 0;
     for (unsigned int j=0; j<N; j++) {
       SC += pi[j+t] * pnorm(r[(t+N)/N],
                               mu + K0 * L(X) - K0 * J_hat[j] * I_hat[j],
                               sqrtl(V));
     }
     FF[t/N] = SC;
   } // end of time loop
   cout << "End␣of␣APF␣particle␣filter" << endl;

   // Applying a particle smoother:

   // Estimating the computation time:
   duration1 += (clock() - start1) / (double) CLOCKS_PER_SEC;
   cout << duration1 << "seconds" << endl;

   //------------------------------------------------------------------//

   // Estimating the means:
   vector<long double> I_mean(Ts);
   vector<long double> J_mean(Ts);

   for (unsigned int t=0; t<T; t+=N) {
     long double i_mean = 0.0L;
     long double j_mean = 0.0L;
     long double j_weight = 0.0L;
     for (unsigned int n=0; n<N; n++) {
       // Calculating the conditional moment of
       // E[I]
       i_mean += pi[n+t] * I[t+n];
       if (I[t+n] == 1) {
         j_weight += pi[n+t];
       }
     }

     for (unsigned int n=0; n<N; n++) {
       if (I[n+t] ==  1) {
         // Calculating the conditional mean of J
         j_mean += pi[n+t] / j_weight * J[t+n];
       }
       else {
         j_mean += 0.0L;
       }
     }
     I_mean[t/N] = i_mean;
     J_mean[t/N] = j_mean;
   }
   // Saving the data:
   ofstream myfile_sim;
   myfile_sim.open ("APF10000.csv");
   for (unsigned int i=0; i<Ts; i++) {
     myfile_sim << I_mean[i] << "," << J_mean[i]
                << "\n";
   }
   myfile_sim.close();
   //---------------------------------------------------------------------------//
   // Saving the convergence data:
   ofstream myfile_sim2;
   myfile_sim2.open ("convergenceAPF.csv");

   for (unsigned int i=0; i<Ts; i++) {
     myfile_sim2 << FF[i] << "\n";
   }
   myfile_sim2.close();
   //---------------------------------------------------------------------------//
   return 0;
}
```

B.3 Fixed state augmentation C++ code

Listing B.3: Fixed state augmentation C++ code

```
// State Augmentation with fixed parameters::

/*          Input            */
// Time series of log returns
// Number of particles

/*          Output           */
//      I      J    \theta     */

#include<iostream>          // Include in/out function
#include<fstream>
#include<string>
#include<vector>
#include<random>
#include<math.h>
#include<cstdio>
#include<ctime>
#include<time.h>
#include<algorithm>         // for low_bound and upper_bound
#include<iterator>          // std::begin, std::end
#include<iomanip>           // for setw
using namespace std;        // The namespace

#ifndef M_PI
    #define M_PI 3.14159265358979323846264L
#endif

// sgn function
long double sign(long double x) {
  if (x >= 0.0L) {
    return 1.0L;
  }
  else {
    return -1.0L;
  }
```

```
}

// Normal density:
long double dnorm(long double x, long double mu=0.0L, long double sd2=1.0L) {
  // sd2 denotes the variance
  return exp(-.5L*(x-mu)*(x-mu)/sd2-.918938533204672741780L)/sqrt(sd2);
}

// Normal cumulative density function:
long double pnorm(long double q, long double mu=0.0L, long double sd=1.0L) {
  const long double SQRTHL = 7.071067811865475244008e-1L;
  long double x, y, z;

  x = (q-mu)/sd * SQRTHL;
  z = abs(x);

  if( z < SQRTHL )
    y = 0.5L + 0.5L * erf(x);
  else
    {
      y = 0.5L * erfc(z);

      if( x > 0.0L )
        y = 1.0L - y;
    }
  return y;
}

// Exponential modified Gaussian density:
long double demg(long double x, long double mu, long double sd, long double k) {
  return 1.0L/abs(k)*exp((x-mu)/k + sd*sd/(2.0L*k*k)) *
    (1.0L - pnorm((sign(k)*((x-mu)/sd + sd/k))));
}

long double L(long double x) {
  return (1.0L/(1.0L+exp(-1.0L*x)));
}

int main() {

  random_device rd;
  uint32_t seed=time(NULL);
  // Number of Monte Carlo simulations:
  unsigned int N = 5000;
  // Number of state augmentation runs
  unsigned int M=200;

  // Reading data from simulated the Jump Diffusion model:
  //-------------------------------------------------------------------------------//

  // Length of time series:
  unsigned int Ts = 0;

  vector<long double> r;
  vector<long double> X_real;
  vector<long double> V_real;
  vector<int> chrash_activity;
  vector<long double> chrash_size;

  FILE * myFile;
```

```
myFile = fopen("sim10000.csv", "r");

if (myFile!=NULL) {
  cout << "Successfully␣opened␣the␣file␣\n";

  long double aux_r, aux_X_real, aux_V_real, aux_chrash_size;
  int aux_chrash_activity;

  while (fscanf(myFile, "%Lf,%Lf,%i,%Lf,%Lf\n",
                &aux_X_real, &aux_V_real, &aux_chrash_activity,
                &aux_chrash_size, &aux_r) == 5 & Ts < 1001) {
    X_real.push_back (aux_X_real);
    V_real.push_back (aux_V_real);
    chrash_activity.push_back (aux_chrash_activity);
    chrash_size.push_back (aux_chrash_size);
    r.push_back (aux_r);
    Ts++;
  }
  fclose (myFile);
  cout << "Data␣read␣\n";
}
else {
  cout << "Unable␣to␣open␣the␣file␣\n";
  return 0;
}

cout << "Ts=" << Ts << endl;
cout << "Random␣Seed=" << rd() << endl;
cout << "Random␣Seed=" << seed << endl;

unsigned int T = Ts*N;

// State vectors:
vector<unsigned int> I(T);
vector<long double> J(T);
//vector<long double> pi(T); due to SIR resampling step
// at the end all weights are identical and equal to 1

vector<long double> mu(T);
vector<long double> sigma0(T);
vector<long double> alpha(T);
vector<long double> beta(T);
vector<long double> K0(T);
vector<long double> Xbar(T);
vector<long double> eta(T);
vector<long double> a(T);

// Distributions needed for the computations / algorithm:
default_random_engine generator;
generator.seed( seed ); // Seeded differently each time

exponential_distribution<long double> exponential(1.0L);
normal_distribution<long double> normal(1.0L,1.0L);
uniform_real_distribution<long double> unif(0.0L,1.0L);

cout << "End␣initialisation" << endl;

//------------------------------------------------------------------------------------------//
 // Vector containing the first stage weights at time t:
vector<long double> W(N);
```

```
// Vector containing the normalizing first stage weights at time t
vector<long double> nW(N);
// Vector containing the cumulative sum of the normalized first stage weights
// at time t+1
vector<long double> Q((N+1));

// Vector containing the position of the re-sampled states at time t+1
vector<unsigned int> B(N);

// Vectors containg the resampled states at time t
vector<long double> is(N);
vector<long double> js(N);

vector<long double> Xhat(N);
vector<long double> Vhat(N);

vector<long double> mus(N);
vector<long double> sigma0s(N);
vector<long double> alphas(N);
vector<long double> betas(N);
vector<long double> K0s(N);
vector<long double> Xbars(N);
vector<long double> etas(N);
vector<long double> as(N);

vector<long double> TT((N+1));

// Timing the Particle Filter APF algorithm;
clock_t start1;
double duration1;
duration1 = 0;

start1 = clock();
//------------------------------------------------------------------------//

// Initialisation, sampling initial states :
bernoulli_distribution bernoulli_init(0.0006);
exponential_distribution<long double> exponential_init(1.0L/0.04L);

// Exact values::
// mu = 0.00028L; sigma0 = sqrt((0.25L*0.25L)/250.0L); alpha = 0.05L;
// beta = 0.94L; K0 = 0.04L; Xbar = -5.0L; eta = 3.0L; a = 0.996L;

uniform_real_distribution<long double> unif_mu(-0.18L/250.0L,0.32L/250.0L);
uniform_real_distribution<long double> unif_sigma0(0.10L/sqrtl(250.0L),
                                                   0.5L/sqrtl(250.0L));
uniform_real_distribution<long double> unif_K0(0.03L,0.05L);
uniform_real_distribution<long double> unif_Xbar(-6.0L,-4.0L);
uniform_real_distribution<long double> unif_eta(2.0L,4.0L);
uniform_real_distribution<long double> unif_a(0.0L,1.0L);

// Mean estimates obtained running the state augmentation M times:
// Vectors holding the different mean estimates:
vector<long double> mu_mean((Ts-1));
vector<long double> sigma0_mean((Ts-1));
vector<long double> alpha_mean((Ts-1));
vector<long double> beta_mean((Ts-1));
vector<long double> K0_mean((Ts-1));
vector<long double> Xbar_mean((Ts-1));
```

```
vector<long double> eta_mean((Ts-1));
vector<long double> a_mean((Ts-1));

// Vectors holding the different standard deviations:
vector<long double> sdmu((Ts-1));
vector<long double> sdsigma0((Ts-1));
vector<long double> sdalpha((Ts-1));
vector<long double> sdbeta((Ts-1));
vector<long double> sdK0((Ts-1));
vector<long double> sdXbar((Ts-1));
vector<long double> sdeta((Ts-1));
vector<long double> sda((Ts-1));

// Vectors holding the different theta_i^{(j)}
vector<long double> muj((Ts-1)*M);
vector<long double> sigma0j((Ts-1)*M);
vector<long double> alphaj((Ts-1)*M);
vector<long double> betaj((Ts-1)*M);
vector<long double> K0j((Ts-1)*M);
vector<long double> Xbarj((Ts-1)*M);
vector<long double> etaj((Ts-1)*M);
vector<long double> aj((Ts-1)*M);

for (unsigned m=0; m<M; m++)
  {
    uint32_t seed=time( NULL );
    generator.seed( seed ); // Seeded differently each time


    for (unsigned int n=0; n<N; n++) {
      // Initial sampling of the states:
      I[n] = bernoulli_init(generator);
      J[n] = I[n]*exponential_init(generator);

      mu[n] = unif_mu(generator);
      sigma0[n] = unif_sigma0(generator);
      long double E1 = exponential(generator);
      long double E2 = exponential(generator);
      long double E3 = exponential(generator);
      long double se = E1 + E2 + E3;
      alpha[n] = E1 / se;
      beta[n] = E2 / se;
      K0[n] = unif_K0(generator);
      Xbar[n] = unif_Xbar(generator);
      eta[n] = unif_eta(generator);
      a[n] = unif_a(generator);

      // Initialization of the mispricing and volatility
      // for each parameter value:
      Xhat[n] = Xbar[n];
      Vhat[n] = sigma0[n] * sigma0[n];

    }

    // The log likelihood sum for the given theta:
    //long double lls = 0;

    // Starting Particle filter:
    for (unsigned int t=0; t<(T-N); t+=N)
      {
```

```
    // Evaluating the first stage weights:
    for (unsigned int n=0; n<N; n++) {
      Xhat[n] = (1.0L - a[n+t]) * Xbar[n+t] + a[n+t] * Xhat[n] +
        eta[n+t] * (r[t/N] - mu[n+t]);
      Vhat[n] = sigma0[n+t] * sigma0[n+t] *
        (1.0L - alpha[n+t] - beta[n+t]) + alpha[n+t] *
        pow ((r[t/N] - mu[t+n]), 2.0) + beta[n+t] * Vhat[n];

      bernoulli_distribution bernoulli1(L(Xhat[n]));
      I[n+t+N] = bernoulli1(generator);
      if (I[n+t+N] == 1) {
        J[n+t+N] = exponential(generator);
      }

      mu[n+t+N] = mu[n+t];
      sigma0[n+t+N] = sigma0[n+t];
      alpha[n+t+N] = alpha[n+t];
      beta[n+t+N] = beta[n+t];
      K0[n+t+N] = K0[n+t];
      Xbar[n+t+N] = Xbar[n+t];
      eta[n+t+N] = eta[n+t];
      a[n+t+N] = a[n+t];
    }

    // calculating the mean of the log-returns including new
    // observation at time t:
    for (unsigned int n=0; n<N; n++) {
      W[n] = dnorm(r[(t+N)/N],
                   (mu[n+t+N] + K0[n+t+N] * L(Xhat[n]) -
                    K0[n+t+N] * J[n+t+N] * I[n+t+N]) ,
                   Vhat[n]);
    }

    // First stage resampling from the discrete distribution
    // {L_{t+1}, W_{t+1}} using algorithm in multinomialsampling.pdf

    // Normalizing the weights:
    long double s1;
    s1 = 0.0L;
    for (int n=0; n<N; n++) {
      s1 += W[n];
    }
    // First stage resampling

    for (unsigned int n=0; n<N; n++) {
      nW[n] = W[n]/s1;
      // Calculating the cumulative sum of the normalized weights:
      // corresponding to Q:
      if (n == 0) {
        Q[n] = 0.0L;
        TT[n] = exponential(generator);
      }
      if (n > 0) {
        Q[n] = Q[n-1] + nW[n-1];
        TT[n] = TT[n-1] + exponential(generator);
      }
    }
    Q[N] = Q[(N-1)] + nW[(N-1)];
    TT[N] = TT[(N-1)] + exponential(generator);
```

```
          unsigned int i=0; unsigned int j=1;
          while (i < N) {
            if ( TT[i] < (Q[j] * TT[N])) {
              B[i] = (j-1);
              // Resampling of the states:
              is[i] = I[(j-1)+t+N];
              js[i] = J[(j-1)+t+N];

              mus[i] = mu[(j-1)+t+N];
              sigma0s[i] = sigma0[(j-1)+t+N];
              alphas[i] = alpha[(j-1)+t+N];
              betas[i] = beta[(j-1)+t+N];
              K0s[i] = K0[(j-1)+t+N];
              Xbars[i] = Xbar[(j-1)+t+N];
              etas[i] = eta[(j-1)+t+N];
              as[i] = a[(j-1)+t+N];

              i++;
            }
            else {
              j++;
            }
          }

          for (unsigned int n=0; n<N; n++) {
            // Updating the states:
            I[n+t+N] = is[n];
            J[n+t+N] = js[n];

            mu[n+t+N] = mus[n];
            sigma0[n+t+N] = sigma0s[n];
            alpha[n+t+N] = alphas[n];
            beta[n+t+N] = betas[n];
            K0[n+t+N] = K0s[n];
            Xbar[n+t+N] = Xbars[n];
            eta[n+t+N] = etas[n];
            a[n+t+N] = as[n];
          }
        } // End of SIR filter
      //------------------------------------------------------------------------//
      // Saving the data:
      for (unsigned int t=0; t<T; t+=N) {
        long double mu_m = 0.0L;
        long double sigma0_m = 0.0L;
        long double alpha_m = 0.0L;
        long double beta_m = 0.0L;
        long double K0_m = 0.0L;
        long double Xbar_m = 0.0L;
        long double eta_m = 0.0L;
        long double a_m = 0.0L;

        for (unsigned int n=0; n<N; n++) {
          mu_m += 1.0/N * mu[t+n];
          sigma0_m += 1.0/N * sigma0[t+n];
          alpha_m += 1.0/N * alpha[t+n];
          beta_m += 1.0/N * beta[t+n];
          K0_m += 1.0/N * K0[t+n];
          Xbar_m += 1.0/N * Xbar[t+n];
          eta_m += 1.0/N * eta[t+n];
          a_m += 1.0/N * a[t+n];
        }
        // storing the different \theta_i^{(j)} for j=1,...,M
        muj[t/N + (Ts-1)*m] = mu_m;
        sigma0j[t/N + (Ts-1)*m] = sigma0_m;
        alphaj[t/N + (Ts-1)*m] = alpha_m;
        betaj[t/N + (Ts-1)*m] = beta_m;
        K0j[t/N + (Ts-1)*m] = K0_m;
        Xbarj[t/N + (Ts-1)*m] = Xbar_m;
        etaj[t/N + (Ts-1)*m] = eta_m;
        aj[t/N + (Ts-1)*m] = a_m;

        mu_mean[t/N] += mu_m;
        sigma0_mean[t/N] += sigma0_m;
        alpha_mean[t/N] += alpha_m;
        beta_mean[t/N] += beta_m;
        K0_mean[t/N] += K0_m;
        Xbar_mean[t/N] += Xbar_m;
        eta_mean[t/N] += eta_m;
        a_mean[t/N] += a_m;
      }
      cout << m << "\n";
    }
  for (unsigned int t=0; t<(Ts-1); t++) {
    mu_mean[t] /= M;
    sigma0_mean[t] /= M;
    alpha_mean[t] /= M;
    beta_mean[t] /= M;
    K0_mean[t] /= M;
    Xbar_mean[t] /= M;
    eta_mean[t] /= M;
    a_mean[t] /= M;

    for (unsigned int j=0; j<M; j++) {
      sdmu[t] += 1.0L / M * ( muj[t+(Ts-1)*j] - mu_mean[t] ) *
        ( muj[t+(Ts-1)*j] - mu_mean[t] );
      sdsigma0[t] += 1.0L / M * ( sigma0j[t+(Ts-1)*j] - sigma0_mean[t] ) *
        ( sigma0j[t+(Ts-1)*j] - sigma0_mean[t] ) ;
      sdalpha[t] += 1.0L / M * ( alphaj[t + (Ts-1)*j] - alpha_mean[t] ) *
        ( alphaj[t + (Ts-1)*j] - alpha_mean[t] );
      sdbeta[t] += 1.0L / M * ( betaj[t + (Ts-1)*j] - beta_mean[t] ) *
        ( betaj[t + (Ts-1)*j] - beta_mean[t] );
      sdK0[t] += 1.0L / M * ( K0j[t + (Ts-1) * j] - K0_mean[t] ) *
        ( K0j[t + (Ts-1) * j] - K0_mean[t] );
      sdXbar[t] += 1.0L / M * ( Xbarj[t + (Ts-1) * j] - Xbar_mean[t] ) *
        ( Xbarj[t + (Ts-1) * j] - Xbar_mean[t] );
      sdeta[t] += 1.0L / M * ( etaj[t + (Ts-1) * j] - eta_mean[t] ) *
        ( etaj[t + (Ts-1) * j] - eta_mean[t] );
      sda[t] += 1.0L / M * ( aj[t + (Ts-1) * j] - a_mean[t] ) *
        ( aj[t + (Ts-1) * j] - a_mean[t] );
    }
    sdmu[t] = sqrtl(sdmu[t]);
    sdsigma0[t] = sqrtl(sdsigma0[t]);
    sdalpha[t] = sqrtl(sdalpha[t]);
    sdbeta[t] = sqrtl(sdbeta[t]);
    sdK0[t] = sqrtl(sdK0[t]);
    sdXbar[t] = sqrtl(sdXbar[t]);
    sdeta[t] = sqrtl(sdeta[t]);
    sda[t] = sqrtl(sda[t]);
  }
  duration1 += (clock() - start1) / (double) CLOCKS_PER_SEC;
  cout << duration1 << "seconds"  << endl;

  // Saving the data:

  ofstream myfile_sim;
  myfile_sim.open ("SIR_SAm1000.csv");
  for (unsigned int i=0; i<(Ts-1); i++) {
    myfile_sim << mu_mean[i] << "," << sdmu[i] << ","
               << sigma0_mean[i] << "," << sdsigma0[i] << ","
               << alpha_mean[i] << "," << sdalpha[i] << ","
               << beta_mean[i] << "," << sdbeta[i] << ","
               << K0_mean[i] << "," << sdK0[i] << ","
               << Xbar_mean[i] << "," << sdXbar[i] << ","
               << eta_mean[i] << "," << sdeta[i] << ","
               << a_mean[i] << "," << sda[i]
               << "\n";
  }
  myfile_sim.close();

  return 0;
}
```

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PARTICLE FILTER ADAPTED TO JUMP-DIFFUSION MODEL OF BUBBLES AND CRASHES WITH NON-LOCAL CRASH-HAZARD RATE ESTIMATION

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
BERNTSEN	PHILIP

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
30.07.2015, ZURICH	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.