Essays on Economic Sentiment Dynamics and Asymmetric Multifractal Models of Financial Volatility

Inaugural-Dissertation zur Erlangung des akademischen Grades eines Doktors der Wirtschafts- und Sozialwissenschaften der Wirtschafts- und Sozialwissenschaftlichen Fakultät der Christian-Albrechts-Universität zu Kiel

vorgelegt von
Stepan S. Sushko
M.Sc. aus Lemberg, UdSSR

Kiel, 2021

Erstgutachter:
Prof. Dr. Thomas Lux

Zweitgutachter:
Prof. Dr. Jan Kallsen

Tag der mündlichen Prüfung:
31. August 2021

Acknowledgements

Above all, I must express my appreciation for my scientific supervisor, Prof. Dr. Thomas Lux, for his encouragement, constructive criticism, assistance, guidance and valuable advice.

The writing of this thesis has been going with varying success until a turning point in the seventh year. I appreciate the Doctoral Programme Quantitative Economics for the help with financial support of my study during the first four years.

I would like also to express gratitude to the university staff and the programme for the help and support in a foreign country, and for creating such a welcoming environment during my study.

I greatly benefited from the valuable discussions in the very beginning of the second project, with Dr. Leövey, who gave me the first insight in the topic. Dr. Tae Seok and Dr. Gonghadze: thank you for your comments about the study process and your advice.

I am deeply indebted to my first supervisor Prof. Dr. Grigory Belyavsky for giving me my first insight into financial mathematics and option pricing.

I am forever grateful to my wife Ekaterina, for her support and endless patience during my long-term study. To my parents, who inculcated in me an understanding of the value of higher education: you always motivated me to finish it and not to give up.

Stepan S. Sushko
Anapa, Russia, July 2019

Abstract

In the first chapter of the dissertation an estimation of the continuous-time Markov chain (CTMC) model of experts' sentiment index is considered in the case of incomplete data. Particularly, three estimation approaches based on a discrete-time sample are presented: the EM algorithm and two versions of the maximum likelihood estimation method. The first approach for the estimation of the considered model is iterative and leads to massive recursive computations of matrices. The most crucial part of the second and third approaches is the numerical computation of the matrix exponential of the intensity matrix. In particular, the second approach is based on the eigendecomposition of the intensity matrix and the corresponding well-known property of matrix exponential for such decomposition. In order to increase the effectiveness of the method in the third approach the fact that the intensity matrix has a lower Hessenberg form is used. All three approaches are based on numerical optimization using the nonlinear conjugate optimizer. In order to test them, Monte Carlo simulations for few parametric sets and different number of agents are run. Further, all three estimation methods are approbated on real market data for the German economy.

The second chapter is dedicated to the development of the methods of calibration and estimation of the model belonging to the asset price class of models. Two variants of the generalization of the Markov Switching Multifractal (MSM) model, called the Asymmetric Markov-Switching Multifrequency, are considered. The modifications are aimed to reproduce such a phenomenon of asset returns as *leverage effect* (asymmetry). Other features of the model, namely the long memory stylized fact for different frequencies and degrees of persistence, the mean reversion of volatility, and the volatility clustering, are investigated and proven. The option pricing theory based on risk-neutral measure is developed for this model. The model parameters' calibration and estimation techniques are described and tested with simulated and real data including asset prices and option prices from the European financial market. In-sample and out-of-sample performance are tested.

Keywords: Continuous-time Markov Chain, agent-based models, EM-algorithm, matrix exponential, Maximum-likelihood estimation, Kolmogorov forward equation, transition probability estimation, Hessenberg matrix, calibration, option pricing, stylized facts, Local Risk-Neutral Valuation Relationship, martingale measure, Monte Carlo simulations, parallel computations on GPUs, equity risk premium.

Contents

List of Figures

List of Tables

Abbreviations

ABM	**A**gent-**B**ased **M**odel
AMSM	**A**symmetric **M**arkov-**S**witching **M**ultifrequency
ASA	**A**daptive **S**imulated **A**nnealing
ATM	**A**t-**T**he-**M**oney
BFGS	**B**royden-**F**letcher-**G**oldfarb-**S**hanno
CDF	**C**umulative **D**istribution **F**unction
CPU	**C**entral **P**rocessing **U**nit
CTMC	**C**ontinuous-**T**ime **M**arkov **C**hain
EM	**E**xpectation **M**aximization
ERP	**E**quity **R**isk **P**remium
GPU	**G**raphics **P**rocessing **U**nit
ITM	**I**n-**T**he-**M**oney
LM	**L**evenberg-**M**arquardt
LR	**L**ong-**R**un
LRNVR	**L**ocal **R**isk-**N**eutral **V**aluation **R**elationship
MAD	**M**edan **A**bsolute **D**eviation
MAE	**M**ean **A**bsolute **E**rror
MC	**M**onte **C**arlo
MLE	**M**aximum **L**ikelihood **E**stimation
MR	**M**id-**R**un
MSE	**M**ean **S**tandard **E**rror
OTM	**O**ut-of-**T**he-**M**oney
PRNG	**P**seudo**R**andom **N**umber **G**enerator
QRNG	**Q**uasi**R**andom **N**umber **G**enerator
RMSE	**R**oot-**M**ean **S**quare **E**rror
SR	**S**ort-**R**un
WRSS	**W**eighted **R**esidual **S**um of **S**quares

1

Introduction

Quantitative research in economics and finance has a history of more than a century. Many models have been developed and used in the industry and academia and they became classic and popular tools among professionals in the academia and industry. In general, all these models could be divided into two general classes: structural form models and reduced form models. Models of the first class are aimed to explain inner processes, while the latter ones – mimic statistical features of data. Computers' era gave and still gives new opportunities for both classes of models allowing their further complication. In particular, modern stage of development of economics, financial market and technologies allows more and more active digitalization and implementation of complicated quantitative modeling by the actors. These processes lead to increase of accuracy, precision and quality of analysis important for decision making in management and finance.

This dissertation considers both types of models – reduced and structural form models. The first topic of this dissertation is the modeling of experts' sentiments dynamics improving understanding of economical processes. In particular, a business climate index is considered as the main real world example. As the business climate index we use a generalized indicator that is an arithmetic mean of experts' assessments. In such form, business climate changes (as experts' sentiments) refer to the opinion dynamics that are well described by Weidlich's[1] agent-based model of opinion formation [97] (a much more general version of the model is presented in the book [98]). It can be implemented for modeling of migration, opinion formation processes (in particular, political processes), competition between firms, consumption of goods, or evolution of cities. In this work, a simplified version of the model is considered. In our case, the inputs of the model are regular surveys of relevant experts from throughout the society. Namely, key economists, business people, employees of analytical companies, and other specialists that are competent in the field of economics and finance give their opinions about a current economical situation. These experts' assessments are used to build an index of a current economical situation (a business climate index).

[1] Wolfgang Weidlich is a founder of the sociological subdiscipline - sociodynamics.

The second direction of the research presented in this dissertation was complete development of a new asset and option prices model: its theoretical basis, estimation and goodness-of-fit analysis. The topic of option pricing was also considered in author's master thesis [91] and two early articles [6, 90]. This new model, belonging to the class of reduced form models as well as the famous Black-Scholes model (1973) became a theoretical and practical basis for option pricing. Thus, it is aimed to describe observed statistical features of the data. Since the Black-Scholes model was invented, the search began for a model that better explains *stylized facts* (features) of real market financial data and shape of volatility surface. Among the most important stylized facts are heavy-tailed negatively skewed non-normal distribution of underlying asset returns, clustering, persistence and leverage effect in volatility. All these features, in particular the presence of the term structure in volatility, contradict with Black-Scholes model assumptions and led to the so-called Implied volatility "smirk", which is observed as out-of-the-money put option market prices and in-the-money call option market prices being higher than Black-Scholes prices. Moreover, there is a negative correlation between the asset returns and their volatility in case of equity market, which is called the *leverage effect*. The leverage effect is an important stylized fact documented for the first time by F. Black in his 1976 paper[2].

It is worth noting, in contrast to the first topic the second one is incomparably deeper investigated, many competitive models have been developed since the Black and Scholes work. The first attempts to describe some of the stylized facts, in particular excess kurtosis and fat-tails, were focused on an incorporation of jumps into models (see Merton 1976). The further focus of models development became incorporation of *stochastic volatility* (as an alternative, stochastic interest rates or random jumps) in a model that forms a class of two-factor[3] stochastic models. This allowed researchers to describe clustering and volatility smiles. Historically, Taylor (1982) originally incorporated the stochastic volatility in an asset price model, while Hull and White's (1988) stochastic volatility models firstly modeled the leverage effect stylized fact. Its importance for describing the volatility smirk was proven by many authors (Bates 2000 and others). In the early 1990s, Heston (1993) proposed the most famous model in the class of two factor stochastic volatility models: a diffusion model known as the Heston model incorporating the leverage effect and mean reversion property. Later, Bates (1996) complemented the Heston model with jumps. Carr and Madan (1999) suggested an efficient method of computation based on Fourier transformation in cases where the closed-form of characteristic function is known, in particular, in the case of the Heston model. Duffie, Pan and Singleton (2000) provided a closed-form transformation for pricing term-structure models, estimation of affine asset pricing models and pricing formulas for options that allowed the general definition of a broad class of *affine jump-diffusion stochastic volatility models* (see Appendix A.7) and, correspondingly, the rest continuous-time models were defined as a class of non-affine models. Toward the end of the 1990s, the two-factor (one stochastic volatility process) jump-diffusion models being used to explain volatility smiles/smirks were acknowledged as too restrictive to be effective (Bates 1997, Bakshi, Cao and Chen 1997), in particular due to the dependence of volatility level and a smile shape. Bates (2000) conducts an analysis of a large number

[2] It was then investigated many times by numerous authors, mostly by using linear regressions of returns and volatility, for instance, in Christie (1982), Duffee (1995), etc.

[3] Hereafter we consider sources of randomness as factors. So, we consider the Black-Scholes model as an one-factor model, while the Heston model is considered as a two-factor model.

of models comparing the two-factor stochastic volatility model with random jumps and a three-factor (two stochastic volatilities of different intensities) model in order to find out whether they are substitutes. He concludes that jumps are necessary even for three-factor models (two volatility factors).

It is clear that the general context of mathematical modeling for both dissertation topics is ambivalent. The main feature of the considered agent-based model of sentiments dynamics is that it takes into account the inner structure of a business climate change process as an opinion formation process. In other words, we deal with a structural form model that mimics the inner interaction of economic agents. In the same time, the option pricing project of the dissertation goes to the mainstream having an aim to create the reduced form model which is able to mimic all the main stylized facts of the data. Surprisingly, the structural form models have not been given much attention in economics and finance in contrast to such fields of science as physics. It seems to be natural, that the former fact led to creation of an interdisciplinary field of science known as *Econophysics*, but still very few researchers work with structural form models, in particular a related research based on Weidlich's sociodynamics was done by T. Lux [66]. This dissertation presents a framework for mathematical modeling of opinion dynamics, which is considered as a continuous-time Markov process of Poisson type with finite state space, in contrast to the differential equation formulation (master equation), as in the original formulation of Weidlich, T. Lux [66] uses the same simplistic case of the Weidlich agent-based model (ABM), but the author continues the line of Weidlich and constructs the Fokker-Plank equation on the base of the master equation. Further, the author uses finite difference methods in order to solve it and to use the solution for approximation of likelihood function, which allows the author to estimate the parameters of the model. As far as we know, no previous research has investigated the topic except the mentioned ones, at in the considered narrow context.

In general, most of existing research in the field of option pricing are reduced form models. Development of these models in the 2000s was focused on multi-factor (usually, two- or three-factor) models, often extending the Heston model by adding more volatility components or stochastic interest rates (Levin 2008; multi-dimensional and multi-factor affine diffusion, Byelkina and Levin 2010) or time-dependence of its parameters. For instance, the generalization of Christoffersen, Heston and Jacobs 2009 is aimed at incorporation of the stochastic correlation parameter. The presented work also considers an alternative to the ARCH-class models which were prominent in 1980s and 1990s – a generalization of the MSM model of Calvet and Fisher (2001) belonging to the class of *stochastic volatility* multi-factor models. The original MSM model is aimed to reproduce such strong phenomena of volatility as persistence (or *long memory*) and fractality. Moreover, it is able to model long memory for different frequencies and degrees of persistence [19], that fits real economic and business cycles well. In addition, it incorporates volatility clustering and mean reversion of volatility, in contrast to ARCH-type models that usually catch one or two of these features; besides, the MSM model generates non-Gaussian distribution of returns. The models of this class differentiate from FIGARCH by being able to have a different rate of long-memory property for different powers of returns. This class is relatively new and there is not much literature aimed at developing the topic. An investigation of the long-memory and other properties of MSM-class models can be found in [60, 84]. Liu and Lux [61] completed a generalization of the original MSM model of Calvet and Fisher. Its purpose is to create bi-variate long-memory time series with controlled covariance (recently, the mul-

tivariate approach was developed further in the series of papers [62, 63, 64]). This work points out modeling of a different stylized fact, namely *leverage effect* (asymmetry), which gives the name to this generalization – the Asymmetric Markov Switching Multifrequency (AMSM) model. There are considered to be two different ways to incorporate asymmetry in the original MSM model. The AMSM model does not abolish key features of the original MSM model, but supplements them, which is proven theoretically and reinforced by simulations and visualizations.

It can be concluded, that this dissertation well deserves careful analysis as a work closely related to business cycles in economics and finance[99]. Let us note, despite both projects are related to economical business cycles, modeling is very different at the same time. The agent-based model theoretically explains the nature of business cycles, while the asymmetric Markov switching model takes existence of this cycles as a statistical feature of financial data, this is another example of ambivalence of the dissertation projects.

The main objective of the first chapter of this dissertation dedicated to the agent-based model is development of an estimation technique for considered ABM model parameters alternative to the one proposed by T.Lux [66]. As it was mentioned above, the simple Weidlich's model is considered in this dissertation from another mathematical angle, namely as a continuous-time Markov chain (CTMC) with finite state space. The author assumed that the formulation of ABM as CTMC could allow constructing alternative likelihood functions[4] based on the transitions probabilities of the considered CTMC process. In order to attain this objective, the likelihood function developed and described by Billingsley [10] in 1961 could be used, with relaxation of certain conditions on asymptotic normality and consistency as done by Prakasa Rao [81] and Huber [50]. The limitation of their likelihood function is a necessity of continuous-time observation of the CTMC process. In this dissertation the likelihood function for the ABM model, which can be used for the case of discrete-time (incomplete) observations of a sentiment-based process, consequently, is presented. It is based on the transition probabilities which can be derived from Kolmogorov equations. Three methods of this likelihood function maximization with respect to the model parameters are developed and three new ABM model parameters estimation technique are thereby established. The subordinate research objectives are investigation of the estimation techniques statistical properties and performance, their strength and weaknesses, verification of the possibility of fitting the considered ABM model to the real world sentiment-based economic data series by the new techniques.

The second chapter of this dissertation dedicated to the option pricing based on two versions of the AMSM model has three important aims: establishing the mathematical features of the considered model versions, a mathematical basis for option pricing based on the AMSM model, an estimation/calibration of the AMSM model parameters (both versions), finally, the verification of proposed techniques using the real data from the financial market.

The ability to mimic stylized facts of returns, such as the non-normality of their distribution and influence of the AMSM model parameters on returns of AMSM process, were investigated and confirmed by simulations. The ability to mimic stylized facts of volatility by two versions of AMSM model, which are often united by the term clustering of volatility, was proven in the series of theorems and lemmas, namely: the mean reversion of volatility, the long memory of volatility and, finally, the leverage effect, crucial for this work. In ad-

[4] A likelihood function is necessary for maximum-likelihood estimation methods.

dition, the influence of model parameters at the long memory property measured by the Hurst exponent is investigated by simulations.

For the AMSM model it was intended to use calibration of the model parameters with option prices from the real financial market. This led to the mandatory stage of developing mathematical basis for option pricing based on the AMSM model as well as for any other model. As the mathematic basis for option pricing, the modification of the risk-neutral (LRNVR) approach of Duan [28], developed for the GARCH model, was used. This assumption led to another series of theorems and lemmas necessary for the option pricing with AMSM model. In particular, the Duan's approach assumes construction of the stochastic discount factor based on utility function, which allowed to define Radon-Nikodym derivative. This derivative allows to transform a physical measure into a risk-neutral one. Then, the construction and distribution of returns and volatility of AMSM process under this risk-neutral measure was defined and, lastly, the fact, that this constructed risk-neutral measure is a martingale measure, was proven.

The final stage of the project dedicated to the AMSM model is aimed to explore calibration and estimation techniques for its parameters. In order to extract the parameters of the original MSM model, Calvet and Fisher constructed the likelihood function, that allows the use of the MLE method. Note, Liu and Lux [62, 63, 64] used an alternative estimation technique, namely, the GMM method of estimation. The first method of fitting the AMSM model parameters in this work is calibration, based on minimization of weighted sum of squared residuals (RSS) between real market option prices[5] and theoretical ones obtained using the AMSM model. In addition, the equity risk premium, that is part of the model, was calibrated jointly with the model parameters using option prices. The second approach is based not only on option prices data alone, but it also uses historical asset returns with corresponding option prices during a joint estimation procedure or historical asset returns alone. This goal was achieved by construction of the likelihood function for each of three types of dataset, and then by the maximum likelihood method applied firstly to the simulated data then to the real data from the financial market.

Let us describe the projects briefly sketched above with more details. The main limitation for direct implementation of the ABM model is absence of the closed-form of the likelihood function necessary for the maximum likelihood estimation method in the case of discrete observations, because the transition probabilities of the underlying CTMC process are a subject to numerical solution in this case. One of possibilities to estimate CTMC in the case of incomplete (discrete) data is to use the Expectation-Maximization (EM) algorithm [27] based on the iterative maximization of the likelihood function using conditional expectations of complete data given the observed data instead of missing data. Many authors apply this method to CTMC and among them are such frequently cited authors as Asmussen [5], M. Bladt, and M. Sørensen [12], with the broad review done in the work of Mezner [70]. These authors consider the generator matrix of CTMC process as a matrix with constant elements, whilst the corresponding matrix is tridiagonal with the elements of diagonal and subdiagonals being functions of three parameters in the ABM model case. Nevertheless, the similar construction to the one from the paper by Bladt and Sorensen (2005) was used. Particularly, the idea of eigendecomposition for computing of the expectation step of the EM algorithm used in the presented dissertation is based on the paper by

[5] Broad cross-sectional option prices (set of Call option contracts with different strike prices and maturities) are used as a data set.

P. Metzner, I. Horenko, and C. Schutte [71]. There are two difficulties in the application of the EM algorithm in the considered case. First, it is necessary to obtain a mathematical expectation of the likelihood function as a result of the expectation step of the EM algorithm, which is quite a complicated procedure. Second, it is necessary to maximize this expectation on the maximization step, which can be done only numerically. This happens because the system of first order conditions does not have an exact solution in our case, but it has the same structure as for the case of ordinary MLE based on complete data. One of the ways to solve this maximization problem is to use the Generalized EM algorithm incorporating features of either the Newton-Raphson type method or another numerical optimization method (see [69]) on the maximization step of the EM algorithm. Namely, the modification of the nonlinear conjugate gradient numerical method [14, 42] was used. Nevertheless, the ABM with limited number of agents can be used with EM algorithm approach, namely up to around 30 agents.

The idea of the second ABM model's estimation approach presented in the dissertation is based on direct implementation of the eigendecomposition of the infinitesimal generator matrix of CTMC process in order to calculate the transition probability matrix of this process necessary for constructing of the discrete-time likelihood function. Further, this likelihood function is maximized using Levenberg-Marquart optimization method. The second approach is two orders of magnitude faster than the previous one, based on EM algorithm, as it was revealed during simulations, still this approach has the same limitation on the model defining constant – the agents number, due to instability of eigendecomposition procedure for large sparse matrices.

For the third model parameters estimation approach another realization of the idea of direct computation of the transition probability matrix is suggested, the one, which permits us to overcome the instability of the second approach for a large number of agents. Namely, another method of the matrix's exponential computation is used, the one, that is based on the fact that the infinitesimal matrix of the considered CTMC process underlying the ABM model has a lower Hessenberg form (see the paper of Moler and Loan[72]). This makes it possible to compute the transition probability matrix in a more robust and efficient manner. As a result, the likelihood function is constructed and maximized by a nonlinear congruential optimizer. This approach led to greater stability of estimation of the ABM models with larger number of agents (350 agents settings were successfully tested) and further up to 10 times decrease of computational intensity. In order to test the performance of the methods, Monte Carlo simulations are done. This enables us to verify the quality of the estimates.

The main obstacle in the second dissertation project with modification of Duan's approach, created for the GARCH model, is that in the case of the MSM model, we have two sources of randomness against one in the GARCH model. The second source of randomness leads to changing of the measure and the sigma-algebra constructions. In order to overcome this difficulty, the assumption that the second source of randomness of the MSM model is known at the moment of time $t-1$ rather than t, like in the original MSM model, was done in order to prove theorems analogical to the ones in Duan's paper. Another obstacle, but a computational one, is the non-affine[6] construction of Asymmetric MSM models, which does not allow the use of semi-closed form expressions and leads to the Monte Carlo methods. AMSM models have a complicated construction of choice oper-

[6] See Appendix A.7.

ators, which makes paths generation relatively slow. Any option price computed using the AMSM model requires thousands of Monte Carlo simulations. Calibration and estimation of the model parameters procedures require computation of not just a single option price, but a cross-section of option prices. Further, the calibration procedure is an iterative search algorithm of model parameters values; this means, that hundreds and thousands of option basket price evaluations are necessary. As a result, a calibration of the AMSM model may take hours. In order to overcome these difficulties, it was necessary to use a sophisticated search algorithm and tricky programming solutions based on C++ programming language enhanced by parallelization technology for graphic cards computations (OpenCL). Another issue is convergence of the search algorithm in conditions of noisy data, because option prices obtained by the Monte Carlo method are imprecise (probabilistic) values slightly varying depending on a seed. This fact complicates the minimization task significantly (objective function becomes noisy), reduces precision/accuracy of calibrated parameters and increases computation time.

A data collecting strategy was and still is complicated task in many researches, the one presented is not an exception, despite the fact that internet made this task easier for many fields of science, including quantitative economics and finance. In particular, the internet brought to a new level the idea of open data. There are plenty of institutions providing an access to scientific open data, one of the most famous institutions is The Organization for Economic Co-operation and Development (OECD)[7]. The OECD provides economical and financial data collected in 36 countries. In particular, the OECD established the business confidence index and consumer confidence index which are examples of sentiment-based indices investigated in this research. The ZEW – Leibniz Centre for European Economic Research in Mannheim[8] is a German organization sharing open data values[9] and providing the sentiment-based economic data. The ZEW financial market monthly survey was established in December 1991. About 350 experts (agents) from financial and industrial sectors participate in this survey. The ZEW Financial Market Survey is used as a basis for the ZEW Indicator of Economic Sentiment. This research uses the ZEW Indicator of Economic Sentiment index data[10] as a real world data example for the ABM in the first dissertation project. The raw dataset is not always ready for direct use, it has to be prepared to fit the ABM model's format. Thus, the ZEW index is the difference between the percentage shares of optimistic and the pessimistic experts (agents) concerning the German economy in the next six months. Therefore, the ZEW index values belong to the interval from -100 to 100 and has to be transformed according to the assumptions about the data structure of the considered ABM model, this procedure is described further in Section 2.8.

The data from financial markets, necessary for the second project, is relatively more assessable and published online by both private and public institutions. For example, Deutsche Börse Exchange website provides raw and visualized data for many assets, such as equities, bonds, ETF/ETP, funds, commodities. It has its own search engine, which can be

[7] www.oecd.org/

[8] www.zew.de/en/das-zew/ueber-das-zew/

[9] https://www.zew.de/en/das-zew/ueber-das-zew/open-access/

[10] www.zew.de/en/publikationen/zew-gutachten-und-forschungsberichte/forschungsberichte/konjunktur/zew-finanzmarktreport/

used to find assets used in this research: EURO STOXX 50 index[11], DAX30 index[12], Siemens share[13] and SAP share[14]. The website of Eurex derivatives exchange operated by Deutsche Börse AG also publishes plenty of trade data offline[15], online data for equity options[16], which was used to obtain cross-section option prices data for the second project. A more complicated task is extracting a risk-free interest rate data necessary for option pricing. The problem here is availability of interest rates only for certain maturities, for example one week, one month and etc. Besides, there are two main sources of borrowing: the money market and the capital market. The capital market provides borrowing and interest rates for longer maturities with quarterly fixed interest rates, while the money market provides borrowing for shorter maturities. At the same time a derivative maturity is measured in days, this leads to necessity of interpolation of various interest rates for different maturities from both money market and capital market. As a risk-free interest rate in the European money market EURIBOR[17] interest rates are usually assumed. The EURIBOR quotes are accessible online on the website of the Bundesbank[18], for example the EURIBOR one-week daily quotations[19] were used among other in this research. The European Central Bank publishes so-called yield-curves, representing the term structure of interest rates on the capital market. In particular, the ECB website provides the raw data for yield curves online, the "Current year - AAA" dataset[20] was used in the interest rate interpolation procedure together with the EURIBOR quotation mentioned earlier.

The artificial data is another popular option for research purposes, taking into account difficulties with obtaining real data and necessity to have data with certain known properties in many research circumstances. The estimation methods developed in both projects of this dissertation were tested in a playground with a huge amount of simulated data. The artificial data was constructed by the author with the properties allowed to investigate: the sensitivity of the models with respect to their parameters, the statistical properties of the obtained models' parameters estimates, to tune the estimation procedures, to choose optimization methods that fit better for the objective (likelihood) functions used. Note, through the use of artificially simulated data a few optimization methods have been tested and selected, namely the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [18], the Levenberg-Marquart algorithm [58] and adaptive simulated annealing (ASA) [53], and a few additional ones were discarded, namely the ordinary simulated annealing, the threshold accepting, great deluge and evolutionary algorithms. In the case of AMSM model the sample paths of the process are simulated in parallel, which led to solving certain specific issues, such as quasirandom number generation of uniform and normal random variables

[11] `www.boerse-frankfurt.de/indices/euro-stoxx-50`

[12] `www.boerse-frankfurt.de/indices/dax`

[13] `www.boerse-frankfurt.de/equity/siemens-ag`

[14] `www.boerse-frankfurt.de/equity/sap-se`

[15] `www.eurexchange.com/exchange-en/market-data/file-services`

[16] `www.eurexchange.com/exchange-en/products/equ/opt`

[17] The Euro Interbank Offered Rate (EURIBOR) is the averaged interest rates in the interbank market published by European Money Markets Institute.

[18] `www.bundesbank.de/dynamic/action/en/statistics/time-series-databases/time-series-databases/743796/743796`

[19] These time series quotes are labeled as BBK01.ST0307 by Bundesbank.

[20] `https://www.ecb.europa.eu/stats/financial_markets_and_interest_rates/euro_area_yield_curves/html/index.en.html`

in parallel. To make simulation experiments as fast as possible a few measures were taken: the models were programmed with efficient C++ programming language, paths simulation was realized using GPU (Graphic Processor Units) rather than Computing Processor Units (CPU) and, finally, the bulk of the experiments were conducted on the recently established, powerful and flexible service of cloud computing Amazon AWS cloud (in the case of AMSM model from the second dissertation project).

It would be impossible or at least time-consuming to store and analyze all the simulated and, later, real data manually, in order to manage it R[21] [80] was used. R is an open source software environment for statistical computing and visualization. It allowed to systematize the research workflow, to enrich it with statistical analysis, complex plots and statistical tables with error metrics, which were automatically generated and stored for further use in the thesis text. In addition, a few open source R libraries allowed to relatively easily estimate Hurst exponent and several alternative (benchmark) option pricing models. So, the workflow of the research was highly integrated and based on three main information technologies (IT): C++ language for fast computations, R statistical language for data analysis/visualization and LaTeXas a document preparation system. This workflow is also based on the principles of reproduced research that was implemented using another technology coming from IT sphere – the Git[22], which is a version-control system allowing tracking changes in text files, usually with a code. Thus, the repositories with C++, R and LaTeXcode for both research projects (six repositories) were created. These projects were defined as submodules of main repository joining the whole dissertation research as an united single entity. This allowed to manage and recreate the state of the project on certain milestones and reproduce the state of the projects on them.

Summarizing, we can say, that, as we could see from the explanation of both projects, they are ambivalent not only in the context of general mathematical modeling, but also in the context of specific stochastic models: the agent-based model from the first dissertation project is considered as a continuous-time stochastic process (Markov chain), while the asymmetric Markov multifrequency model of log-returns from the second project is modeled as a discrete-time stochastic process.

The first project of this dissertation provides new mathematical basis for this kind of agent-based models, considering the variant of Weidlich's agent-based model belonging to a relatively rare class (in economics and finance) of structural form models as a continuous-time Markov chain with a finite state space. This basis allowed to suggest three new approaches to estimate the agent-based sentiment dynamics process of Weidlich, aimed to mimic sentiment-based business cycles indices. The most efficient method among these three approaches is approved using the real data for the ZEW Indicator of Economic Sentiment.

The second project is also aimed to take into account business cycles by the assumption of existing multiple frequencies modeling log-returns of assets like the Markov switching multifrequency model (AMSM). The new AMSM model variant is suggested in this work and investigated together with the one suggested by A. Lëove [59]. The desired properties such as the non-normality of the log-returns distribution and the stylized facts associated with the notion of clustering of volatility are proven theoretically or with the use of simulations. The theory of option pricing for both variants of the AMSM model is developed

[21] https://www.r-project.org/

[22] https://git-scm.com/

and based on adoptions of the approach of Duan that had constructed the risk-neutral measure for the GARCH model. The rich set of techniques of calibration and estimation of not only the model parameters, but also the equity risk premium for both AMSM model variants, using option prices, historical asset returns or both, are developed, tested and approved for the real financial data. The superiority of the AMSM model is shown in a series of goodness-of-fit tests based on the real data with a few popular models as a benchmark.

The structure of this dissertation is as follows: two chapters, corresponding to two projects; sixteen sections; sixteen appendices; code listings for both projects; bibliography; sixty plots; forty two tables.

In particular, the first chapter consists of eight sections. Section 2.1 describes the model of business climate changes and the probabilistic features of the underlying Markov process with finite state space. In addition, this section is dedicated to constructing both the continuous-time and discrete-time likelihood functions. In Section 2.2 the model sensitivity with respect to the model parameters is analyzed and the Monte Carlo simulation technique is also described. In order to estimate the model parameters, the EM algorithm is applied. Its technique, properties and application are described in Section 2.3. The second approach of the model estimation is defined and tested in Section 2.4, while Section 2.5 considers and then verifies the third approach in the case of simulated data. Section 2.7 is dedicated to computing the efficiency of all three approaches. Finally, Section 2.8 presents the implementation of the approaches to the real data (ZEW index).

The second chapter consists of six sections. Section 3.1 is dedicated to the definition of both variants of the AMSM model and the ability of this model to mimic known stylized facts of returns and volatility. In Section 3.2 a mathematical basis for option pricing using the AMSM model is developed. Section 3.3 describes Monte Carlo simulation technique used for option pricing. The calibration and estimation methods based on option prices data and historical log-returns data are presented in Section 3.4 and Section 3.5. The final Section 3.6 illustrates in-sample and out-of-sample performance of the AMSM model variants.

2

Three Approaches For Estimation Of Agent-based Model Of Experts' Sentiment Index

2.1. Theoretical basis

2.1.1. Jump processes vs. Diffusion processes

As was mentioned above, the inputs of the model are regular surveys of relevant experts' from throughout society, who give their opinions about the current business climate. These experts' assessments are used to build a business climate index (and also can be used for modeling similar agent-based indexes). In general, this index can be modeled using the Weidlich model. A simple case study of this model is given further.

Further, it is assumed, that the negative pole of assessments corresponds to the value "-1", while the positive pole of assessments to the value "1". Thus, the business climate index (which is, in fact, a Weidlich's opinion dynamics process) is a stochastic process with discrete values in the interval $[-1,1]$. However, what kind of stochastic process is it?

The second assumption is that the opinion dynamic process is a continuous-time process with a Markov property. This leads us to the class of continuous-time Markov processes. There are two main concepts regarding their modeling: as a diffusion process or as a jump process. In the first case, it is necessary to assume that at least one transition (a small one) occurs at any short period of time with the probability 1. In the second case, it is assumed that no transition occurs in a short enough period of time with high probability, but if any transition does happen, it has large amplitude. In the presented dissertation, the second concept is used, while T. Lux [66] instead follows the first concept and models opinion dynamics as a diffusion process.

In addition, the following is assumed: agents are homogeneous; at each moment of time just one agent can change their own opinion; and the time between opinion changes is not a constant. Further, the probability of opinion changes is defined by transition probabilities that depend on three parameters: the parameter ν is a time scale parameter and,

specifically, it means the frequency of opinion changes; the parameter α_0, depending on its sign, explains a shift to the optimistic (positive sign) or pessimistic (negative sign) side; the parameter α_1 expresses a power of social pressure towards a common opinion (positive sign) or against it (negative sign). Thus, we assume that there are N agents and, each of them at time t has either an optimistic ("+") or pessimistic ("-") opinion; therefore, there are two types of agents. Let

- $n_+(t)$ be the number of optimistic agents,
- $n_-(t)$ be the number of pessimistic agents.

Let us define the process of agents' opinion dynamics $\mathbf{X} = (X_t)_{t=0}^{\infty}$ on the probability space $(\Omega, \mathscr{F}, P)$ as follows

$$X_t = \frac{n_+(t) - n_-(t)}{N}, \tag{2.1}$$

where the state space is

$$\Omega = \left\{-1, -\frac{(N-1)}{N}, \ldots, 0, \ldots \frac{(N-1)}{N}, 1\right\}. \tag{2.2}$$

Note, the state space Ω has $2N+1$ states with an indices i running over the index set $I = \{-N, -(N-1), \ldots, 0, \ldots, (N-1), N\}$. Therefore, any values of $\mathbf{X}$ have the form i/N for every t, where $i \in I$. Further, the formulations "the process X is in the state i" and "the process X at the time t has the value i/N" will be used interchangeably. The process trajectory example is depicted in large scale in Figure 2.1.

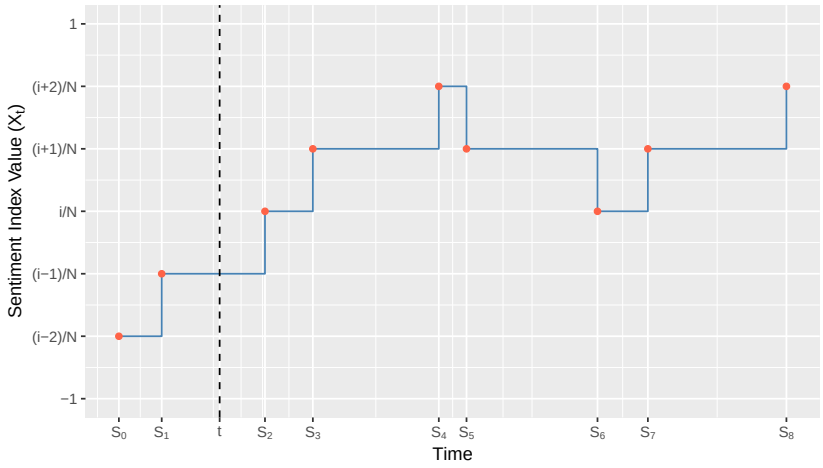


Figure 2.1: The sentiment index process $\mathbf{X}$ dynamics on the state space Ω (y-axis).

Let us proceed from the intuitive formulation of the model to a stricter one based on Gallager's book [37].

2.1.2. Countable-state continuous-time Markov chain

In the relevant chapter the process of opinion formation $\mathbf{X} = (X_t)_{t=0}^{\infty}$ is modeled as a continuous-time countable-state Markov process of the Poisson type (further Markov process) on the given filtered probability space $(\Omega, \mathscr{F}, \{\mathscr{F}_t\}_{t=0}^{T}, P)$ satisfying the *usual conditions*, adopted to the filtration $\{\mathscr{F}_t\}_{t=0}^{T}$. Therefore, trajectories of the process $\mathbf{X}$ are assumed

to be right-continuous step functions with values from the state space Ω. Besides, the Markov property holds.

Definition 1. *If the following expression holds for any states $i, j \in I$*

$$P\left(X_{t+\Delta} = \frac{j}{N} \middle| X_t = \frac{i}{N}, X_{s_1} = x_1, \ldots, X_{s_m} = x_m\right) = P\left(X_{t+\Delta} = \frac{j}{N} \middle| X_t = \frac{i}{N}\right), \tag{2.3}$$

where $0 < s_l < \ldots < s_1 < t$, then the process $\boldsymbol{X}$ has the Markov property.

This definition states that future values of the process **X** depend only on a current value of the process; in other words, **X** is a memoryless process.

Definition 2. *Let $S_1 < S_2 < \ldots$ denote all moments of time at which transitions occur, then the sequence $(X_n)_{n=0}^{\infty} = (\boldsymbol{X}(S_n))_{n=0}^{\infty}$ or equivalently $X_n = X_t$ for $S_n \le t < S_{n+1}$ forms a Markov chain on the state space Ω, called an embedded Markov chain (in the literature, it is also called "jump process" [77]) with the probabilities of transition from any state $i \in I$ to $j \in I$ denoted as P^e_{ij}.*

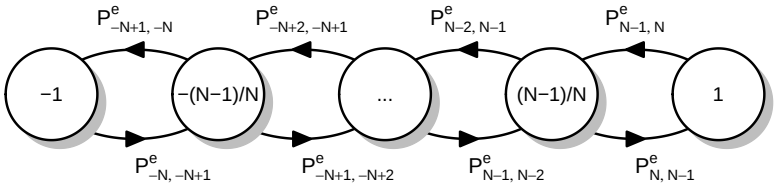


Figure 2.2: Embedded Markov Chain.

Note, it is assumed $P^e_{ii} = 0,\ i \in I$ for simplicity. So, there is no self transitions assumption. Figure 2.2 illustrates the embedded Markov chain for the considered model.

Definition 3. *Let time intervals between two **successive** transitions be denoted as $U_n = S_n - S_{n-1}$ for any $n > 0$. Then, the distribution of holding intervals U_n is given by*

$$P^U_i(x) = P\left(U_n \le x \middle| X_{n-1} = \frac{i}{N}\right) = 1 - \exp(-q_i x),\ i \in I, \tag{2.4}$$

where q_i is called a transition intensity (also a transition rate, in literature) from the state $i \in I$.

Note, according to the Markov property (2.3), U_n is independent of $X_{n-2}, X_{n-3}, \ldots, X_0$ and $U_{n-1}, U_{n-2}, \ldots, U_0$.

Definition 4. *Let the current state be i and the moment of time t (not necessarily the time of previous transition), then the time until the next transition to the state j occurs is $Y(t)$.*

Therefore, taking into account the Markov property, the distribution of transition time $Y(t)$ from i to j is the following

$$\begin{aligned} P_{ij}^{Y}(x) &= P\left(Y(t) \le x, X(t+Y(t)) = \frac{j}{N} \middle| X(t) = \frac{i}{N}\right) \\ &= P\left(Y(t) \le x \middle| X(t+Y(t)) = \frac{j}{N}, X(t) = \frac{i}{N}\right) P\left(X(t+Y(t)) = \frac{j}{N} \middle| X(t) = \frac{i}{N}\right) \\ &= P_i^U(x)\, P_{ij}^e \\ &= \left[1 - \exp(-q_i x)\right] P_{ij}^e. \end{aligned} \tag{2.5}$$

In order to summarize all details above, let us give a strict definition of the countable-state Markov process[1] (see also Figure 2.1).

Definition 5. *Any continuous-time stochastic process $\boldsymbol{X} = (X_t)_{t=0}^{\infty}$ taking values on countable set on the given filtered probability space $(\Omega, \mathscr{F}, \{\mathscr{F}_t\}_{t=0}^{T}, P)$ is called a continuous-time Markov chain (CTMC), if it has the Markov property and for each $t \ge 0$*

$$X_t = X_n \ \text{ for } S_{n-1} \le t < S_n;\ S_0 = 0;\ S_n = \sum_{m=1}^{n} U_m \ \text{ for } \ n \ge 1, \tag{2.6}$$

where $(X_n)_{n=0}^{\infty}$ is a discrete-time (embedded) Markov chain defined by the transition probabilities P_{ij}^e on a countable or finite state space, U_n is a holding interval with exponential distribution (given X_{n-1} is in state $i \in I$) with the transition rate $q_i > 0$.

It is easy to see that such a Markov process is in fact specified by quantities q_i and P_{ij}^e. So, the product $q_i P_{ij}^e$ can be interpreted as a transition intensity from any state i to any state j, namely

$$q_{ij} = q_i P_{ij}^e,\ i \ne j,\ i, j \in I \tag{2.7}$$

then a summation of them over all $j \ne i$ gives

$$q_i = \sum_{j \ne i} q_{ij}. \tag{2.8}$$

Hence, the transition probabilities of the embedded Markov chain can be defined as

$$P_{ij}^e = \frac{q_{ij}}{q_i}. \tag{2.9}$$

Therefore, let us introduce the *intensity matrix* $Q = \{q_{ij}\}_{i,j \in I}$ with the non-diagonal entries q_{ij} and the diagonal entries $-q_i$, then the whole Markov process $\mathbf{X}$ can be defined by the intensity matrix. In the related literature, this type of matrix is also called an *infinitesimal generator*.

[1] This formulation is closely related to the definition of R.Gallager in [37]

In considered adaptation of the Weidlich sociological model the intensity rates of process **X** are given by

$$\begin{aligned} q_{ii+1}(\theta) &= \omega_+(i/N;\theta), \\ q_{ii-1}(\theta) &= \omega_-(i/N;\theta), \\ q_i(\theta) &= \omega_+(i/N;\theta) + \omega_-(i/N;\theta), \\ q_{ij}(\theta) &= 0 \text{ for } |i-j| > 1,\ i,j \in I \end{aligned} \tag{2.10}$$

where

$$\begin{aligned} \omega_+(x;\theta) &= (1-x)\nu \exp(U(x;\alpha_0,\alpha_1)), \\ \omega_-(x;\theta) &= (1+x)\nu \exp(-U(x;\alpha_0,\alpha_1)), \\ U(x;\alpha_0,\alpha_1) &= \alpha_0 + \alpha_1 x. \end{aligned} \tag{2.11}$$

Thus, the transition probabilities of the embedded Markov chain $(X_n)_{n=0}^{\infty}$ for $i, j \in I$ are constructed as

$$\begin{aligned} P^e_{ii-1}(\theta) &= \frac{q_{ii-1}(\theta)}{q_i(\theta)} = \frac{\omega_-(i/N;\theta)}{\omega_-(i/N;\theta)+\omega_+(i/N;\theta)}, \\ P^e_{ii+1}(\theta) &= \frac{q_{ii+1}(\theta)}{q_i(\theta)} = \frac{\omega_+(i/N;\theta)}{\omega_-(i/N;\theta)+\omega_+(i/N;\theta)}, \\ P^e_{ij}(\theta) &= 0 \text{ for } |i-j| > 1,\ i = j. \end{aligned} \tag{2.12}$$

Altogether, the main inputs of the model were denoted and their mathematical sense was described. Let us give an explanation for the model parameters: ν is a time scale parameter and, specifically, it means a frequency of opinion changes; the parameter α_0, depending on its sign, explains a shift to the optimistic (>0) or pessimistic (<0) side; and the parameter α_1 expresses the power of social pressure towards the common opinion (>0) or against it (<0).

2.1.3. Kolmogorov equations

In the case of the considered ABM model, a transition from the current state i to any state rather than $i-1$ or $i+1$ has to be done within a few transitions (see Figure 2.1). For example, the process **X** has to make at least $2N$ transitions within time t in order to reach the state $N \in I$ from the state $-N \in I$. Now, it is necessary to define the corresponding probability of successive and non-successive transition (series of successive transitions in fact) from any state i to any j within time t, as depicted in Figure 2.1. These probabilities are important for practical use of the model; in particular, for a likelihood function construction.

Theorem 1. *The probabilities $P^X_{ij}(t)$ of the process **X** transition between any states i and j (through an unknown number of successive transitions) and within time t are defined by the Kolmogorov forward and backward equations in the case of CTMC, correspondingly*

$$\begin{aligned} \frac{dP^X_{ij}(t)}{dt} &= \sum_{k\neq j}\left(P^X_{ik}(t)q_{kj}\right) - P^X_{ij}(t)q_j, \\ \frac{dP^X_{ij}(t)}{dt} &= \sum_{k\neq j}\left(q_{ik}P^X_{kj}(t)\right) - q_i P^X_{ij}(t), \end{aligned} \tag{2.13}$$

or in the matrix form

$$\begin{aligned} \frac{dP^X(t)}{dt} &= P^X(t)Q, \\ \frac{dP^X(t)}{dt} &= QP^X(t), \end{aligned} \tag{2.14}$$

where Q is a matrix of intensity rates with q_{ij} as the off-diagonal elements and $-q_i$ as the diagonal elements.

Proof. The proof is given in Appendix A.1. □

The following corollary is dedicated to the solution of Kolmogorov equations under certain assumptions, but firstly it is necessary to give a definition.

Definition 6. *Let X be an $N \times N$ matrix, then the $N \times N$ matrix*

$$\exp(X) = \sum_{k=0}^{\infty} \frac{X^k}{k!}, \; X^0 = I \tag{2.15}$$

is called a matrix exponential of X.

Corollary 1. *If the initial condition $P^X(0) = I$ holds, then the matrix differential equation (2.14) has the following solution*

$$P^X(t) = \exp(tQ) \tag{2.16}$$

where $\exp(\cdot)$ *is a matrix exponential.*

So, if there is an efficient computation method of the matrix exponential on the r.h.s. of (2.16), then the whole matrix of transition probabilities $P^X_{ij}(t)$ can be calculated.

Another corollary reveals how the structure of the considered ABM model affects the Kolmogorov equations.

Corollary 2. *Recall, the intensity rates matrix Q is a tridiagonal in the case of the ABM model.*

$$Q = \begin{bmatrix} \bullet & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & 0 & 0 \\ 0 & \bullet & \bullet & \bullet & 0 \\ 0 & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & \bullet \end{bmatrix}$$

Therefore, the sum in the Kolmogorov system of equations (2.13) reduces to the terms containing nonzero intensities, namely q_{ii+1}, q_{ii-1},q_k. Thus, the Kolmogorov forward equations reduce to

$$\begin{aligned} \frac{dP^X_{i,-N}(t)}{dt} &= P^X_{i,-N+1}(t)q_{-N+1,-N} - P^X_{-N,j}(t)q_{-N}, \\ \frac{dP^X_{i,j}(t)}{dt} &= P^X_{i,j-1}(t)q_{j-1,j} + P^X_{i,j+1}(t)q_{j+1,j} - P^X_{i,j}(t)q_j, \; for -N < j < N, \\ \frac{dP^X_{i,N}(t)}{dt} &= P^X_{i,N-1}(t)q_{N-1,N} - P^X_{i,N}(t)q_N. \end{aligned} \tag{2.17}$$

The Kolmogorov backward equations reduce to

$$\begin{aligned}
\frac{dP^X_{-N,j}(t)}{dt} &= q_{-N,-N+1}P^X_{-N+1,j}(t) - q_{-N}P^X_{-N,j}(t),\\
\frac{dP^X_{i,j}(t)}{dt} &= q_{i,i-1}P^X_{i-1,j}(t) + q_{i,i+1}P^X_{i+1,j}(t) - q_i P^X_{i,j}(t),\ -N < i < N,\\
\frac{dP^X_{N,j}(t)}{dt} &= q_{N,N-1}P^X_{N-1,j}(t) - q_N P^X_{N,j}(t).
\end{aligned} \tag{2.18}$$

Altogether, the ABM model is determined by the vector of parameters $\theta = (\nu, \alpha_0, \alpha_1)$, which in turn specifies the transition intensity matrix $Q(\theta)$ and the transition probabilities matrix $P^X(t;\theta)$.

□

2.1.4. Summary of the results on the theoretical basis of the agent-based model.

There are a few probabilities defined above, which have to be clearly distinguished. Note, it is important to understand that only the probability $P^X_{ij}(t)$ is defined for the case of non-successive transitions. In contrast, it is assumed that there are an arbitrary number of intermediate transitions between i and j for $P^X_{ij}(t)$. Let us summarize:

1. P^e_{ij} is a *successive* transition probability of the embedded Markov chain of Markov process $\mathbf{X}$ from the state i to j (see Definition 2);
2. $P^U_i(x)$ is a *distribution* of time between any two *successive* transitions from the state i (see Definition 3);
3. $P^Y_{ij}(x)$ ($f^Y_{i,j}(x)$) is a *distribution (density)* of time $Y(t)$ until the next *successive* transition from the state i at time time t to and state j (see Definition 4);
4. $P^X_{ij}(t)$ is a probability of *non-successive* transition from the state i to j within the exact time t (see Theorem 1).

The latter two probabilities allow us to construct two different likelihood functions. Thus, the obvious choice of the model estimation approach is the widespread maximum likelihood method.

2.1.5. Likelihood functions

In order to estimate parameters $\theta = (\nu, \alpha_0, \alpha_1)$, the algorithms that are based on maximization of likelihood functions were used. Therefore, it is necessary to construct a likelihood function. It is possible to do this in two ways:

- In the case when the complete data sample (non-discrete observations) of the process $(X_t)_{t=0}^{\infty}$ is available, the complete likelihood function $L^c(\theta|X)$ can be constructed. As will be shown later on, this function is expressed through the intensities q_{ij} and depends on information about all transitions and the time that the process spent in different states;

- In the case when there is only a discrete data sample y of the continuous-time process $(X_t)_{t=0}^{\infty}$, the incomplete likelihood function $L^d(\theta|y)$, which is expressed through the transition probabilities $P^X(\Delta t)$ (Δt is an interval between observations), can be used.

It is necessary to construct both the complete and the incomplete likelihood functions to proceed further. Note, these two likelihood functions are constructed also in [12] and [71].

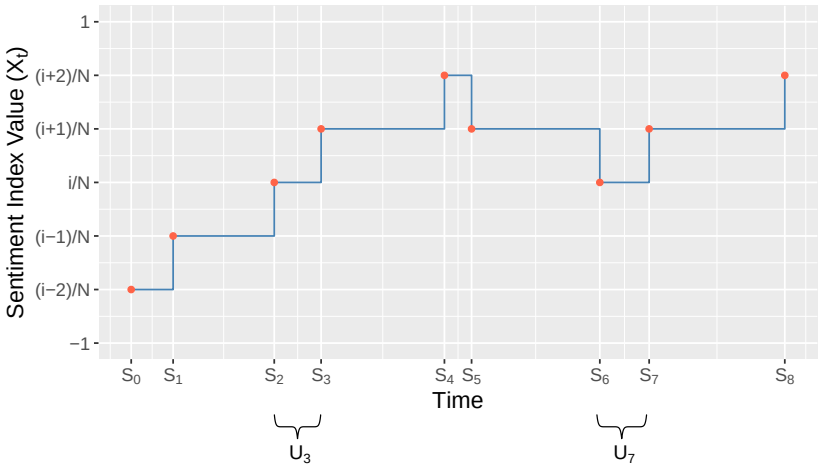


Figure 2.3: Holding time $R_i(T) = U_3 + U_7$ of the process $\mathbf{X}$.

Let us start with construction of the likelihood function for the case when the process $(X_t)_{t=0}^{\infty}$ is observed continuously. This means all the moments of transition S_k are known, as are all the corresponding states of the process $\mathbf{X}$ in these moments. Then, let M be the total number of transitions $\mathbf{X}$ made during the whole observation time T and the variable $R_i(T)$ is the total time that the process $(X_t)_{t=0}^{\infty}$ spent in any state i. So, the formal definition is the following

$$R_i(T) = \int_0^T I\{X_s = i/N\}ds = \sum_{k=1}^{M} U_{k+1} I\{X_{S_k} = i/N\}, \tag{2.19}$$

where $I\{\cdot\}$ is an indicator function, U_{k+1} is an interval until the next transition occurs from the time moment S_k of the k-th transition. This illustrated in Figure 2.3; there, the process $\mathbf{X}$ holds state i for two periods, U_3 and U_7, therefore $R_i(T) = U_3 + U_7$ for this example.

Next, the number of transitions from any state i to any state j during the time T is denoted further as $N_{ij}(T)$. In the example depicted on Figure 2.3, the process $\mathbf{X}$ made two transitions from i to $i+1$, therefore $N_{i,i+1}(T) = 2$.

In order to construct the likelihood function, it is also necessary to define the corresponding density function.

Lemma 1. *The probability density function of $\mathbf{X}$ successive transition time from any state i to any j is defined as*

$$f_{i,j}^{Y}(x) = P_{ij}^{e} q_i \exp(-q_i x), \tag{2.20}$$

where P_{ij}^{e} is a transition probability of embedded Markov chain, q_i is an intensity rate of $\mathbf{X}$ transitions from the state i.

From	To	Time	$N_{ij}(T)$
i-2	i-1	U_1	1
i-1	i	U_2	1
i	i+1	U_3, U_7	2
i+1	i+2	U_4, U_8	2
i+1	i	U_6	1
i+2	i+1	U_5	1

Table 2.1: Number of transitions of the process in Figure 2.3.

Proof. The proof is given in Appendix A.2. □

Note, the probability density function $f_{ij}^{Y}(x)$ allows us to calculate the probability of the event of the process **X** successive transition from the state i to j within exactly time $U_n = S_k - S_{k-1}$, rather than the distribution of time between state i and j. This is necessary for constructing the likelihood function invented by Billingsley [10].

Theorem 2. *The likelihood function based on a complete continuous-time observation sample* $\boldsymbol{x}$ *of the CTMC process* $\boldsymbol{X}$ *is expressed by (see [10], [41])*

$$L^c(\theta|x) = \prod_{i=-N}^{N} \prod_{j \neq i} q_{ij}^{N_{ij}(T)} \exp(-q_{i,j} R_i(T)), \tag{2.21}$$

where q_{ij} *is an intensity rate of* $\boldsymbol{X}$ *transitions from any state* i *to* j, $N_{ij}(T)$ *is the number of transitions from any state* i *to any state* j *during the time* T *in data sample* $\boldsymbol{x}$, $R_i(T)$ *is the total time that the process* $\boldsymbol{X}$ *spent in each state* i *during observation sample* $\boldsymbol{x}$ *at time* T.

Proof. The proof is given in Appendix A.3. □

Example

Let us consider an example of a continuous-data sample of the process **X** depicted in Figure 2.3. The process made eight transitions at the moments of time $(S_1, \ldots, S_8)$ with corresponding values $(X_{S_1} = (i-2)/N, \ldots, X_{S_8} = (i+2)/N)$.

Table 2.1 collects the data about the number of transitions that occurred during observation time T. Table 2.2 collects the data about the total holding time $R_i(T)$ during observation time T for each state.

State	$R_i(T)$
-1	0
	$\cdots$
i-2	U_1
i-1	U_2
i	$U_3 + U_7$
i+1	$U_4 + U_6 + U_8$
i+2	U_5
	$\cdots$
1	0

Table 2.2: Time the process in Figure 2.3 spent in each state.

This data allows us to construct the likelihood function $L^c(\theta|x)$ according to Theorem 2

$$
\begin{aligned}
L^c(\theta|x) &\overset{def}{=} \prod_{k=0}^{M-1} q_{X_{S_k}} \exp(-q_{X_{S_k}} U_{k+1}) \\
&= q_{i-2,i-1} q_{i-1,i} q_{i,i+1} q_{i+1,i+2} q_{i+2,i+1} q_{i+1,i} q_{i,i+1} q_{i+1,i+2} \times \\
&\exp\left(-\left[q_{i-2}U_1 + q_{i-1}U_2 + q_i U_3 + q_{i+1}U_4 + q_{i+2}U_5 + q_{i+1}U_6 + q_i U_7 + q_{i+1}U_8\right]\right) \\
&= q^1_{i-2,i-1} q^1_{i-1,i} q^2_{i,i+1} q^2_{i+1,i+2} q^1_{i+1,i} q^1_{i+2,i+1} \times \\
&\exp\left(-\left[q_{i-2}U_1 + q_{i-1}U_2 + q_i(U_3 + U_7) + q_{i+1}(U_4 + U_6 + U_8) + q_{i+2}U_5\right]\right) \\
&= \left[\prod_{i=-N}^{N} \prod_{j\neq i} q_{ij}^{N_{ij}(T)}\right] \times \left[\prod_{i=-N}^{N} \exp(-q_i R_i(T))\right] \\
&\overset{Thm.2}{=} \prod_{i=-N}^{N} \prod_{j\neq i} q_{ij}^{N_{ij}(T)} \exp(-q_{ij} R_i(T)),
\end{aligned}
$$

□

The next step after construction of the likelihood function is to formulate the system of the first order conditions necessary for a maximization of function.

Corollary 3. *The first order conditions necessary for a maximization of the log-likelihood*

function $l^c(\theta|x) = \log L^c(\theta|x)$ *in the case of the considered agent-based model are given by*[2]

$$\begin{aligned}
\frac{\delta l^c(\theta|x)}{\delta \nu} &= \sum_{i=-N}^{N} \left[(N_{ii+1} + N_{ii-1}) \frac{1}{\nu} - R_i \left(\exp\left(\alpha_0 + \alpha_1 \frac{i}{N}\right) + \exp\left(-\alpha_0 - \alpha_1 \frac{i}{N}\right) \right) \right] = 0, \\
\frac{\delta l^c(\theta|x)}{\delta \alpha_0} &= \sum_{i=-N}^{N} \left[(N_{ii+1} - N_{ii-1}) - \nu R_i \left(\exp\left(\alpha_0 + \alpha_1 \frac{i}{N}\right) - \exp\left(-\alpha_0 - \alpha_1 \frac{i}{N}\right) \right) \right] = 0, \\
\frac{\delta l^c(\theta|x)}{\delta \alpha_1} &= \sum_{i=-N}^{N} \frac{i}{N} \left[(N_{ii+1} - N_{ii-1}) - \nu R_i \left(\exp\left(\alpha_0 + \alpha_1 \frac{i}{N}\right) - \exp\left(-\alpha_0 - \alpha_1 \frac{i}{N}\right) \right) \right] = 0.
\end{aligned} \tag{2.22}$$

Proof. The proof is given in Appendix A.4. □

The solution of the system (2.22) is desired estimation of $\theta = (\nu, \alpha_0, \alpha_1)$. Unfortunately, the system (2.22) has only numerical solutions.

Further, it is necessary to mention that changes of agents' opinions cannot be observed during the time between interviews; consequently, the real data is discrete in time. So, the expressions $f_{ij}^Y(t)$ (2.20) cannot be used, because the process $(X_t)_{t=0}^{\infty}$ could make more than one transition over the time t (discrete time-step). On the other hand the use of the complete likelihood function for a discrete data leads to biased estimates. Therefore, $L^c(\theta|x)$ cannot be used in the case of available discrete data sample directly, but it will be used further for the artificially simulated (continuous) data in Section 2.3.

An alternative is an *incomplete* likelihood function.

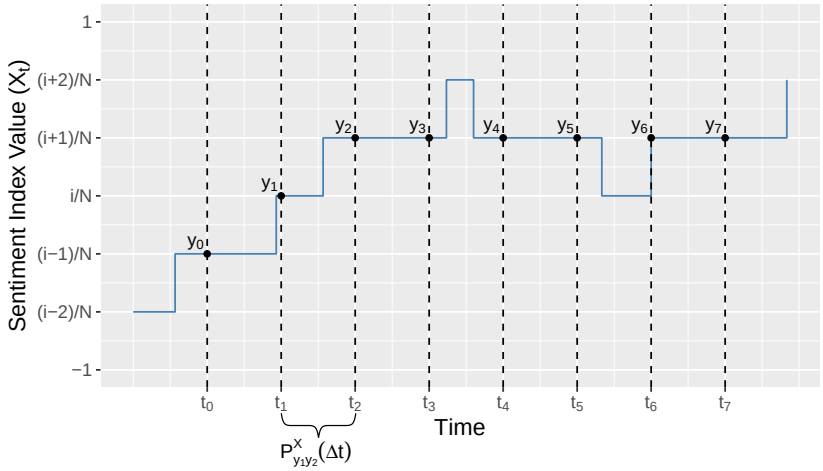


Figure 2.4: Discrete-time sample **y** of the process **X**.

Theorem 3. *Let* $\boldsymbol{y} = \{y_0 = X_{t_0}, y_1 = X_{t_1}, \ldots, y_M = X_{t_M}\}$ *be a vector of discrete observations of the process* $\boldsymbol{X}$ *at moments of time* $t_0 < t_1 < \cdots < t_M$ *(see Figure 2.4),* $t_k = t_{k-1} + \Delta t$*, then the incomplete (discrete) likelihood function based on this vector is given by*

$$L^d(\theta|y) = \prod_{i,j \in I} \left(P_{ij}^X(\Delta t; \theta) \right)^{c_{ij}}, \tag{2.23}$$

[2]The dependence on T of $N_{ij}(T)$ and $R_i(T)$ is omitted in order to increase the readability of expressions.

where $c_{ij}, i, j \in I$ is a number of transitions from any state i to any state j in the vector $\mathbf{y}$, while $P^X_{ij}(\Delta t;\theta)$ denotes the corresponding transition probabilities for the process $\boldsymbol{X}$ with parameters θ within Δt.

Proof. The likelihood function of the outcome corresponding to the series of transitions $\mathbf{y} = y_1, \ldots, y_M$ is

$$L^d(\theta|y) = \prod_{k=0}^{M-1} P^X_{y_k y_{k+1}}(t_k;\theta).$$

This product can be rearranged. Namely, the same transitions could be counted as c_{ij} and their probabilities collected as $P^X_{ij}(\Delta t;\theta)$, so

$$\prod_{k=0}^{M-1} P^X_{y_k y_{k+1}}(t_k;\theta) = \prod_{i,j\in I} \left(P^X_{ij}(\Delta t;\theta)\right)^{c_{ij}},$$

The probabilities $P^X_{ij}(\Delta t;\theta)$ are derived by solving Kolmogorov equations (Theorem 1). The solution (2.16) is given by the matrix exponential

$$P^X(\Delta t;\theta) = \exp(\Delta t Q(\theta)).$$

□

However, there is no straightforward way to calculate the elements of r.h.s. of the expression of the likelihood function in Theorem 3, because $\exp(\Delta t Q(\theta))$ being a matrix exponential is not available in closed-form. It means we don't know what each $P^X_{ij}(\Delta t;\theta)$ looks like, hence we cannot take derivatives and so on. The only solution for maximization of $L^d(\theta|y)$ is a numerical calculation of matrix exponential and then numerical optimization of $L^d(\theta|y)$, which is provided in Section 2.4.

Before we begin with estimation techniques for both likelihood functions, it is necessary to highlight the simulation technique used for verification of the methods described further.

2.1.6. Monte Carlo simulation technique

It is important to note the following fact: the process $\mathbf{X} = (X_t)_{t=0}^{\infty}$ is not able to make two transitions at one moment of time by definition. In particular, only one transition occurs at each moment of time, either to a state $i-1$ or to a state $i+1$. A direction of transitions being a discrete-time random variable is defined by the following probabilities

$$P^e_{ii+1} = \frac{q_{ii+1}}{q_i},\ P^e_{ii-1} = \frac{q_{ii-1}}{q_i},$$

where i is a current state of process $\mathbf{X}$. The following well-known procedure was used to generate it. An uniform random variable u is generated by one of the pseudo-random number generators. Then, if $u < P^e_{ii+1}$, then $X_{t_{k+1}} = X_{t_k} + 1/N$ (step up), otherwise $X_{t_{k+1}} = X_{t_k} - 1/N$ (step down). From the previous Section 2.1.2 (equation (2.4)), it is known that

the distribution of time interval U_n up to the next successive transitions from any state $i \in [-N, N]$ is given by

$$P_i^U(x;\theta) = P\left(U_n \leq x \middle| X_{t_{n-1}} = \frac{i}{N}, X_{t_n} = \frac{j}{N}\right) = 1 - \exp\left(-q_i(x;\theta)x\right) = f(x).$$

In order to generate U_n, an inverse of function $f(x)$ is derived, then the uniform random variable $u \in (0,1)$ generated by a computer[3] is plugged into $f^{-1}(x)$. So, it is assumed $U_n = f^{-1}(u)$.

In order to simulate a whole continuous-time sample $\mathbf{x}$ of the process $\mathbf{X}$, $X_0 = 0$ is assumed as an initial state first, then an interval of time to the next successive transition U_n and its direction (from i to $i-1$ or $i+1$) are generated. This procedure is continued while $S_n = \sum_{i=1}^{k} U_i <= T$.

In the next stage, M discrete-time observation values collected as $\mathbf{y} = (y_0, \ldots, y_M)$ are extracted from the artificial continuous-time data sample $\mathbf{x}$ at the time moments $t_k = k \times \Delta t$, where the discretization step $\Delta t = 1$. This means it is necessity to evaluate a state of generated trajectory $\mathbf{x}$ in each of time points $t = (0, 1, \ldots, T)$. As a result, a sample of discrete-time observations $\mathbf{y} = y_1, \ldots, y_T$ is derived, where $y_i = X_{t_i}$. The sample $\mathbf{y}$ is used further as an input for estimation procedures. This allows us to simulate real world circumstance characterized by accessibility of discrete observations of any sentiment index only, rather than continuous-time observation.[4]

The following parametric sets were used further for testing purposes, analogous to T. Lux [66]:

- Set 1: $\theta = (3, 0, 0.8)$;
- Set 2: $\theta = (3, 0.2, 0.8)$;
- Set 3: $\theta = (3, 0, 1.2)$;
- Set 4: $\theta = (3, 0.2, 1.2)$.

Note, you can see the example of simulated discrete-time data sample in Figure (2.6e) and the corresponding continuous-time trajectory path (see Figure 2.6d).

□

Prior to proceeding to the estimation of the model, it is important to investigate the influence of parameters on the model. This will help us to understand the performance of the estimation techniques for different parametric sets.

[3] A high quality random number generator from AlgLib library [13] is used for the generation of uniform random numbers. This procedure requires two seed numbers; two prime numbers for better quality of generated random numbers were used, namely 3715061396 and 2984140826. The idea is to make sequences necessary for generation of direction and intervals between successive transition less correlated, it ensures good properties of Monte Carlo simulations.

[4] All Monte Carlo simulation experiments share **only two** long sequences of uniform random numbers. The first one is for the generation of transition directions; the second one is for the generation of intervals between successive transitions.

2.2. Sensitivity analysis w.r.t. model version and its parameters

It is not evident what kind of inner dependencies the model has with respect to its parameters $(\nu, \alpha_0, \alpha_1)$ and number of agents N, if considering only its construction from a mathematical aspect, as described in Section 2.1. In this section such an analysis is provided.

Recall, the probability distribution of transition time from the state i to j is given by (2.5), namely

$$P^Y_{ij}(t;\theta) = P^e_{ij}(\theta) P^U_i(t;\theta).$$

The probability above is defined as the product of two probabilities: the distribution of time between successive transitions from i to j (holding time) given by $P^U_i(t;\theta)$ (see (2.4)) and the transition probability of the embedded Markov chain $P^e_{ii+1}(\theta)$ or $P^e_{ii-1}(\theta)$[5] defining the probabilities of transition directions. The analysis is aimed at investigation of properties of these two probabilities.

Also, the numbers of plots' sets provide certain evidence useful for understanding the connections among parameters. Figures 2.5a-d display various plots of the transition time distributions $P^U_i(x;\theta)$ and $P^U_i(0.2;\theta)$ (z-axis) that occur during time horizon 0.2 with respect to parameters ν, α_0, α_1 (y-axis) and state i (x-axis). Figures 2.6a-e display various simulated continuous-time sample paths and one discrete-time observation of the process $\mathbf{X}$ (y-axis) for different agents numbers N and parameters α_0, α_1. Figures 2.7a-d depict simulated examples of distributions of holding time for each state and parameters α_0, α_1. Finally, Figure 2.8 shows the dependencies of transition probability of the embedded Markov chain P^e_{ij} for each state i with respect to the parameters α_0, α_1.

2.2.1. Number of agents N

Firstly, let us consider the number of agents N that is not a parameter, but rather a model-defining constant. This defines the number of states and, therefore, the state space Ω of the process $\mathbf{X}$.

An analysis of Figures 2.6a-c shows the tendency of holding time decreasing with upgrowth of N. The process looks almost smooth and the transitions are almost invisible, while in the case of $N = 5$ there are clear periods with no transitions at all.

Another clear dependency is that, the paths become more narrow with a growth of N; the evidence of this is shown in Figures 2.6a-c and Figures 2.7. There, the range of sentiment index values is equal to $[-1,1]$ for $N = 5$ and the process reaches the borders a few times and twice moves from the one border to another. In contrast, the process oscillates around the trend line for $N = 150$, never touching the borders. The reason is that the step size $1/N$ becomes twice as small with a twice as large N. At the same time, the probability of transition

$$P^e_{ii+1}(N,\alpha_0,\alpha_1,\nu) \geq P^e_{2i,2i+1}(2N,\alpha_0,\alpha_1,\nu) P^e_{2i+1,2i+2}(2N,\alpha_0,\alpha_1,\nu)$$

[5] All other P^e_{ij} are equal to zero in the considered model.

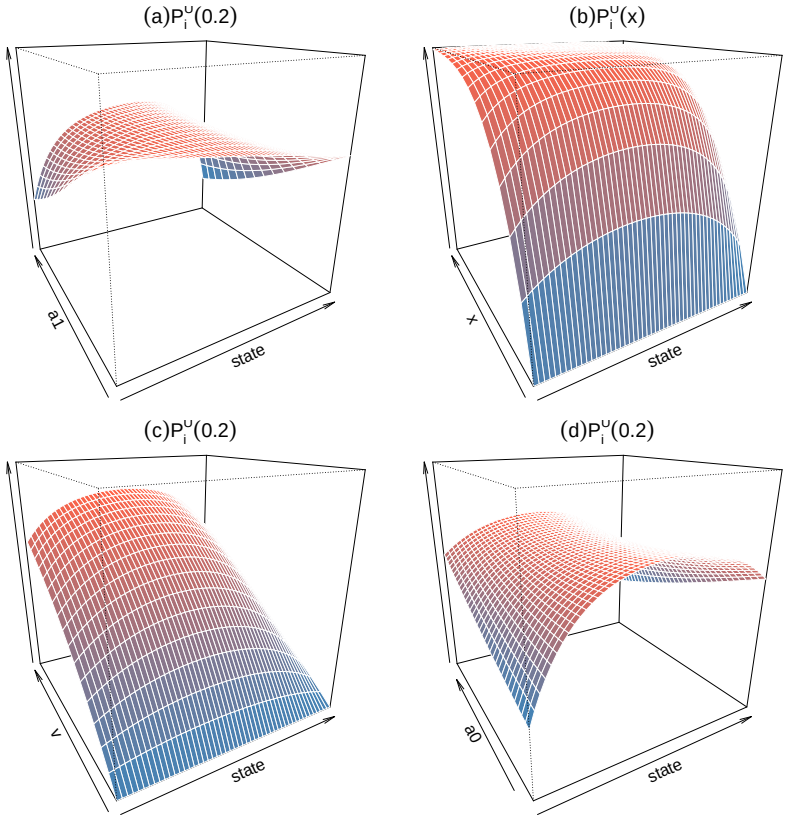


Figure 2.5: Probability distribution of time between transitions w.r.t. model parameters. As the fixed values of the parameters are taken, the vector $\theta = (N, \alpha_0, \alpha_1, \nu) = (50, 0, 0.8, 3)$

for the same step size $(1/N)$, because

$$\begin{aligned} P^e_{2i,2i+1}(2N, \alpha_0, \alpha_1, \nu) &= \frac{\exp(\alpha_0 + \alpha_1 2i/2N)}{\exp(\alpha_0 + \alpha_1 2i/2N) + \exp(-\alpha_0 - \alpha_1 2i/2N)} \\ &= P^e_{ii+1}(N, \alpha_0, \alpha_1, \nu), \end{aligned}$$

therefore

$$\begin{aligned} P^e_{2i+1,2i+2}(2N, \alpha_0, \alpha_1, \nu) \leq 1 &\overset{def}{=} P^e_{2i,2i+1}(2N, \alpha_0, \alpha_1, \nu) P^e_{2i+1,2i+2}(2N, \alpha_0, \alpha_1, \nu) \\ &= P^e_{i,i+1}(N, \alpha_0, \alpha_1, \nu) P^e_{2i+1,2i+2}(2N, \alpha_0, \alpha_1, \nu) \\ &\leq P^e_{i,i+1}(N, \alpha_0, \alpha_1, \nu). \end{aligned}$$

because a probability by definition is less or equal to unity.

This means that the extreme points become less likely and the amplitude becomes smaller.

2.2.2. Parameter α_1

Figure 2.8 provides a prediction that the simulated process **X** sample paths should have "points of attraction". It is clear that the process should fluctuate around 0 for $\alpha_0 = 0$, $\alpha_1 = 0.8$ (see Figure 2.8b), but for $\alpha_0 = 0$, $\alpha_1 = 1.2$ there should be three "points of attraction"

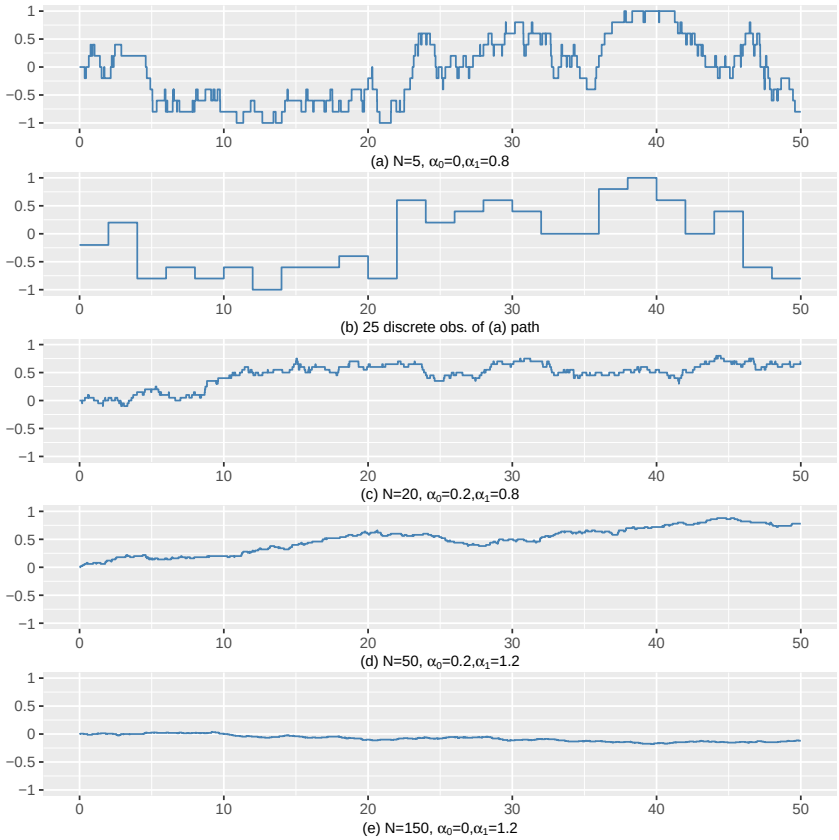


Figure 2.6: Changing of trajectories (simulated) for different N. Horizontal axis is real time.

($x = -0.7,\ 0,\ 0.7$). The simulations pictured in Figure 2.7e confirms this. In general, α_1 changes the curvature of transition probabilities P^e defining transition direction; namely, the curve of P^e plots raises for larger α_1 and becomes non-monotonic for $\alpha_1 > 1$. This fact leads to higher transition probabilities pushing the process from borders for higher α_1, due to steeper P^e (larger difference between P^e_{ii+1} and P^e_{ii-1}), while upward/downward probabilities are close around zero-state. As a result, the trajectories become narrower for larger α_1 (see Figure 2.7).

Also, the probability of there being transitions becomes lower for the same period of time with an increase of α_1, especially for the states close to the extreme ones. The evidence for this is presented in Figure 2.5a. It clear that the surface monotonically decreases with a growth of α_1, meaning lower probability of transition. Further, the plot is folding along the α_1-axis, making a transition from the polar states less likely for larger α_1. This kind of dependency effectively explains the real world, where a change of opinion is less likely with the higher pressure of a majority, especially in cases where most of people have the same opinion (the polar states -1 and 1).

2.2.3. Parameter α_0

The influence of α_0 on a shift of trajectory presents the opposite picture. It can be traced back easily in Figures (2.6a)-(2.6d). The Figures in 2.8 are instructive for understanding of dependency with respect to parameter α_0 . For instance, the process should fluctuate around the state 0.9 according to Figure 2.8f, where the curves of $P^e_{ii+1}(\theta)$ and $P^e_{ii-1}(\theta)$

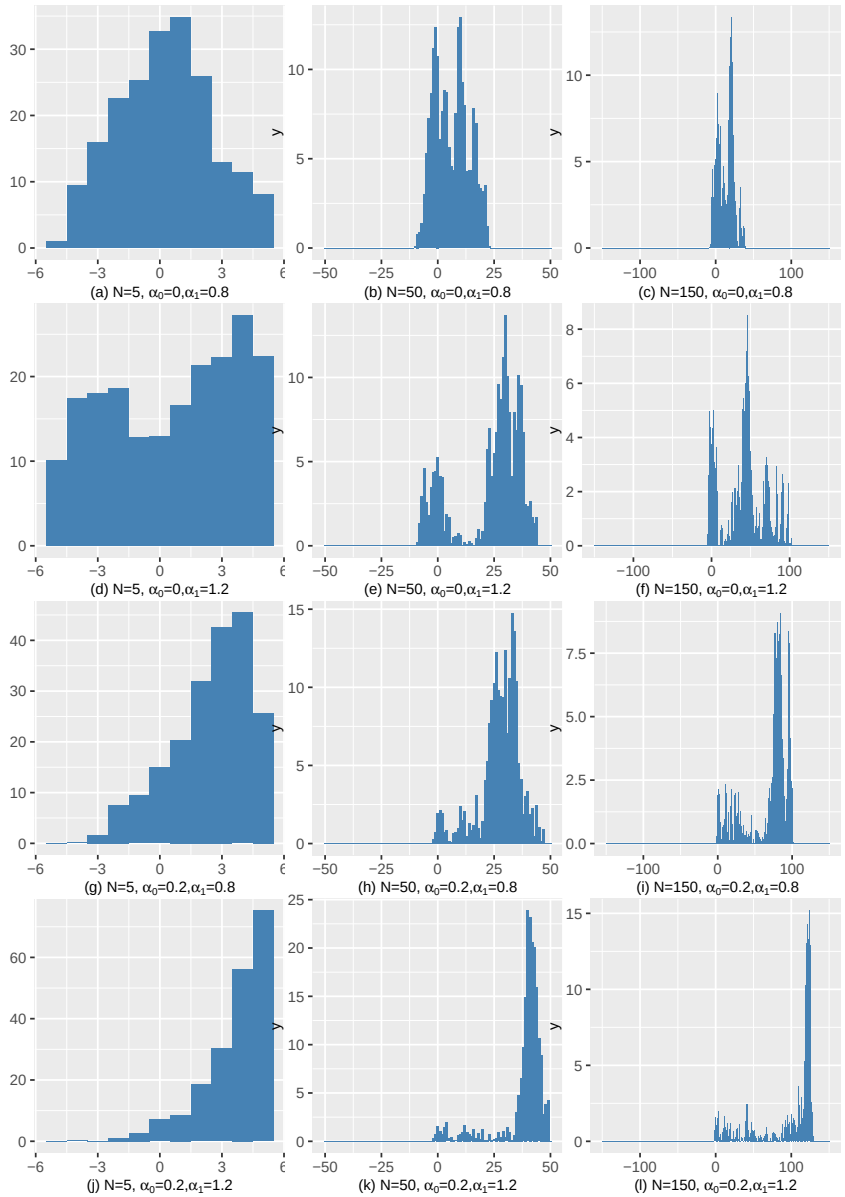


Figure 2.7: Simulated distribution of time the process spends in each state $i = 1,\dots,2N+1$ (equal to $i = -N,\dots,N$) for different parameters (N,α_0,α_1), where the horizontal axis is a state i, the vertical axis is a time spent in corresponding state. $\nu = 15$.

are intersected. This fact is confirmed by Figure (2.6c) of the sample path. Similarly, Figure (2.8c) shows that the point of attraction migrates from the point $x = 0$ (for $\alpha_0 = 0$) to $x = 0.7$ (for $\alpha_0 = 0.2$), and Figure (2.6b) of the sample path also asserts this fact. In general, any positive value of α_0 leads to a shift of P^e_{ii+1} to curve upward, P^e_{ii-1} downward, vica versa for any negative α_0. In turn, this leads to the movement of the equilibrium point to the right for $\alpha_0 > 0$ and to the left for $\alpha_0 < 0$ (see Figures 2.8). The above analysis demostrates that the model can easily mimic a shift oin the real sentiment index if it takes place.

The model parameter α_0 also shows clear influence on holding time probability distribution $P^U_i(x;\theta)$ (2.4). This influence is controversial according to Figure 2.5d, which has an asymmetric saddle-shape – the probability of transition within 0.2 units of time for any

states $i < 0$ increases for larger $\alpha_0 > 0$ and decreases for states $i > 0$ for smaller $\alpha_0 > 0$; in the case of negative α_0, the relationship is the opposite. Although the biases in the distribution of holding time for each state are clear from the plot of theoretical probability $P_i^U(0.2)$, it is not easy to detect an influence of such biases on simulated sample paths Figures 2.6a-2.6d and on the distribution of holding time in states (Figures (2.7)). However, Figure 2.6c, depicting the simulated sample path with $\alpha_0 = 0.2$, seems to have significantly more motionless periods when the process is closer to 1. In the case of Figure 2.6d, the trajectory is quite volatile, thus, it is difficult to recognize an influence, at least for the current scale.

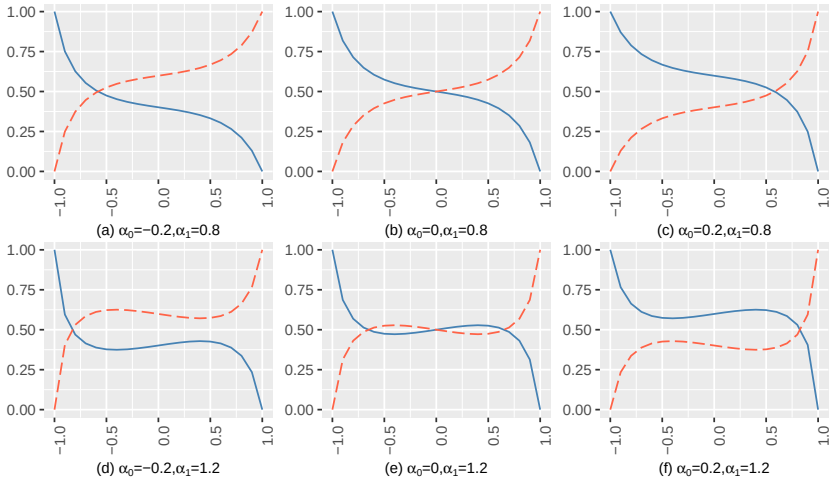


Figure 2.8: Changing of probabilities P^e_{ii+1} (solid line) and P^e_{ii-1} (dash line) for different (α_0, α_1). The horizontal axis is a state of process.

2.2.4. Parameter ν

There is evident appearance of dependency of the last parameter ν in Figure 2.5d. The probability of transition during 0.2 units of time dramatically rise with an increase of ν. As an example of sample paths for different ν is not provided, let us just mention that, in case of simulations for $\theta = (\nu, \alpha_0, \alpha_1) = (3, 0, 1.2)$, $N = 50$ for time horizon $T = 200$, the process makes around 1000 transitions. Meanwhile, for $\theta = (50, 10, 0, 1.2)$, it makes around 3300 transitions.

Concerning direction of the process transitions with respect to parameter ν, it does not depend on it, because

$$P^e_{ii+1}(\theta) = \frac{q_{ii+1}(\theta)}{q_i(\theta)} = \frac{\omega_-(i/N;\theta)}{\omega_-(i/N;\theta)+\omega_+(i/N;\theta)} = \\ = \frac{\exp(-\alpha_0 - \alpha_1 i/N)}{\exp(-\alpha_0 - \alpha_1 i/N) + \exp(\alpha_0 + \alpha_1 i/N)}. \tag{2.24}$$

So, ν is purely in charge of the time scale: in other words, for the time between transitions (aka holding time).

□

Let us proceed to estimation of the model parameters. There are three estimation approaches considered: the EM algorithm based on likelihood function $L^c(\theta|x)$ which is defined for a continuous-time sample and two approaches based on maximization of likelihood function $L^d(\theta|y)$ for a discrete-time sample.

2.3. EM algorithm approach

The first examined approach is the EM algorithm suggested by Dempster in 1977 [27], which was also very well described with a lot of theoretical details in McLachlan's book [69]. The idea of implementation of the EM algorithm in the case of incomplete data (discrete observations) for a continuous-time Markov Chain is based on M.Bladt and M.Sorensen's 2005 paper [12]. Further, Metzner, Horenko, and Schütte 2007 [71] developed the approach of Bladt and Sorensen and suggested a more efficient expectation-step. In both cases, the authors considered and estimated the elements of infinitesimal generator $Q = \{q_{ij}\}$ with constant intensity rates q_{ij}, while the ABM model has intensity rates $q_{ij}(\theta)$ parameterized by three parameters $\theta = (\nu, \alpha_0, \alpha_1)$, which are necessary to estimate. In this section, the approach from [71] adapted to the ABM model of sentiments dynamics is considered.

2.3.1. Theoretical description

The idea of the EM algorithm is to augment missing data with expected data. It is an iterative algorithm, based on the iterative maximization (M-step) of the expectation of a complete data based likelihood function $L^c(\theta|x)$ conditional on an incomplete data sample y (E-step).

So, let there be a vector of parameters $\hat{\theta}_m$ obtained on the iteration m of the EM algorithm, which is then used to find the conditional expectation (E-step)

$$E_{\hat{\theta}_m}\left[\log L^c(\theta; x)\middle| y\right], \tag{2.25}$$

where **x** is a complete data sample, **y** is an incomplete data sample (data between observations is missing), $E_{\hat{\theta}_m}[\cdot]$ is an expectation substituting the parameters' values $(\alpha_0, \alpha_1, \nu)$ with $\hat{\theta}_m = (\hat{\alpha}_0, \hat{\alpha}_1, \hat{\nu})$ when it is necessary for calculation of this expectation. The expectation (2.25) is maximized in order to obtain the next approximation $\hat{\theta}_{m+1}$ (M-step)

$$\hat{\theta}_{m+1} = \arg\max_{\theta}\left(E_{\hat{\theta}_m}\left[\log L^c(\theta; x)\middle| y\right]\right). \tag{2.26}$$

Before we formulate crucial property of the algorithm, monotonicity, let us begin with the preliminary result connecting the two likelihood functions defined in Section 2.1.5.

Lemma 2. *The incomplete-data likelihood function $L^d(\theta|y)$ is connected with its complete-data counterpart in the following way*

$$\log L^d(\theta|Y = y) = \log L^c(\theta|X = x) - \log f_X(x|Y = y; \theta) \tag{2.27}$$

*where **x** is a complete data (all transitions occur and intervals between them) about the process $\boldsymbol{X} = (X_t)_{t=0}^{t=\infty}$ behavior, **y** is a discretization of **x** with a constant time step (not all transitions fixed and real intervals are unknown). In other words, it is a vector with states of $\boldsymbol{X} = \boldsymbol{x}$ recorded each Δt step and $f_X(x|Y = y; \theta)$ is a conditional distribution of **X** given $\boldsymbol{Y} = \boldsymbol{x}$.*

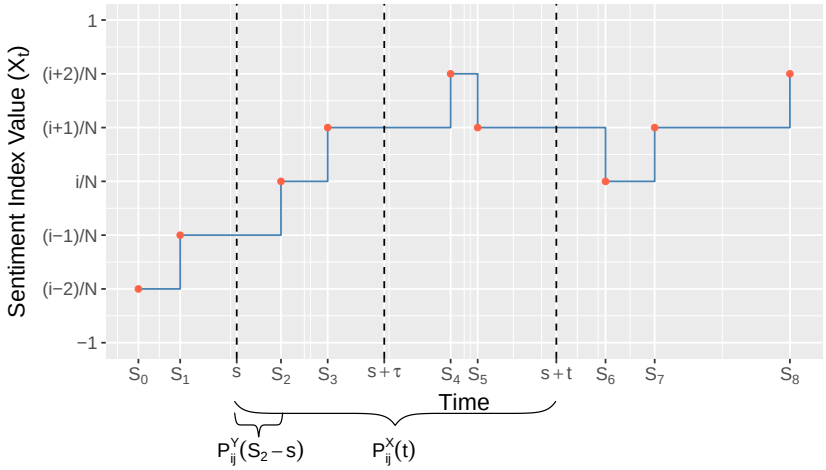


Figure 2.9: Transition probabilities of the process **X**.

Proof. Any complete data sample **x** incorporates any correspondent incomplete data **y** obtained as a discretization of **x**. Thus,

$$P(Y = y) = \frac{P(X = x, Y = y)}{P(X = x|Y = y)} = \frac{P(X = x)}{P(X = x|Y = y)} \tag{2.28}$$

or equivalently

$$f_Y(y) = \frac{f_{X,Y}(x, y)}{f_X(x|Y = y)} = \frac{f_X(x)}{f_X(x|Y = y)}, \tag{2.29}$$

where $f_Y(y)$ is a probability density function for an incomplete data sample. Namely, it is defined as the following density for the considered ABM model

$$f_Y(y) = f^d(y;\theta) = \prod_{k=0}^{M-1} P^X_{y_k y_{k+1}}(\Delta t;\theta) \tag{2.30}$$

and $f_X(x)$ is its complete data counterpart, which is defined in Section 2.1.5 for the considered model

$$f_X(x) = f^c(x;\theta) = \prod_{k=0}^{M-1} f^Y_{X_{S_k}, X_{S_{k+1}}}(U_{k+1}) = \prod_{i=-N}^{N} \prod_{j\neq i} q_{ij}^{N_{ij}(T)} \exp(-q_{i,j} R_i(T)). \tag{2.31}$$

Then, taking the log function from both sides of (2.29), one led to the proposition of lemma.

□

The above result allows us to prove the crucial property of the EM algorithm in ensuring monotonic enhancement of estimates from a single iteration to a successive iteration.

Theorem 4. *Let a vector of parameters obtained on the m-th iteration of the EM algorithm by maximization of expected complete (continuous-time) likelihood $L^c(\theta; x)$ (2.26) be denoted as $\hat{\theta}_m$, then the following inequality holds (the property of monotonicity)*

$$L^d(\hat{\theta}_{m+1}) \geq L^d(\hat{\theta}_m), \; m > 0, \tag{2.32}$$

where $L^d(\theta; y)$ is an incomplete (discrete-time) likelihood functions defined in (2.23), (2.27).

Proof. See the proof in [27] and [69] (Section 3.2, "Monotonicity of the EM algorithm"). □

Let us describe the computation of conditional expectation (2.25) in detail. It could be said that usually the E-step is the most difficult part of the algorithm implementation, both – theoretically and numerically. To begin with, the complete likelihood (2.21) is plugged into the expectation (2.25) conditional on incomplete data sample $\mathbf{y}$ given the parameters' values $\hat{\theta}_m$ obtained on the previous iteration

$$E_{\hat{\theta}_m}\left[\log L^c(\theta)|y\right] = \sum_{i=-N}^{N}\sum_{i\neq j}\left[\log(q_{ij}(\theta))E_{\hat{\theta}_m}\left[N_{ij}(T)|y\right] - q_{ij}(\theta)E_{\hat{\theta}_m}\left[R_i(T)|y\right]\right]. \tag{2.33}$$

So, the goal is to calculate $E_{\hat{\theta}_m}[N_{ij}(T)|y]$ and $E_{\hat{\theta}_m}[R_i(T)|y]$ in the case of the ABM model.

Theorem 5. *Let $\mathbf{y} = \{y_0, \ldots, y_M\}$ be M discrete observations of agent-based process $\boldsymbol{X}$ with a discretization step Δt, T be a total observation time ($T = M \times \Delta t$), so then*

$$\begin{aligned} E_{\hat{\theta}_m}[R_i(T)|y] &= \sum_{k,l\in I}\frac{c_{kl}}{P^X_{kl}\left(\Delta t;\hat{\theta}_m\right)}\int_0^{\Delta t} P^X_{ki}\left(s;\hat{\theta}_m\right)P^X_{il}\left(\Delta - s;\hat{\theta}_m\right)ds, \\ E_{\hat{\theta}_m}[N_{ij}(T)|y] &= \sum_{k,l\in I}\frac{c_{kl}q_{ij}(\hat{\theta}_m)}{P^X_{kl}\left(\Delta t;\hat{\theta}_m\right)}\int_0^{\Delta t} P^X_{ki}\left(s;\hat{\theta}_m\right)P^X_{jl}\left(\Delta - s;\hat{\theta}_m\right)ds, \end{aligned} \tag{2.34}$$

where $R_i(T)$ is a time process $\boldsymbol{X}$ holden any state i during observation time T, $N_{ij}(T)$ is a number of transitions from arbitrary state i to arbitrary state j, $q_{ij}(\hat{\theta}_m)$ are entries of the generator matrix $Q(\hat{\theta}_m)$, while $P^X_{kl}(\Delta;\hat{\theta}_m)$ are elements of the transition probability matrix $P^X(t;\hat{\theta}_m)$, c_{kl} is a number of transitions from any state $k \in I$ to any state $l \in I$ in the data sample $\mathbf{y}$, both matrices are obtained on the m-th iteration of the EM algorithm.

Proof. The proof is given in Appendix A.5. □

The next corollary allows us to calculate the integrals (2.34) obtained in Theorem 5. This method of computation was suggested by Metzner [71]. In order to implement the formula given in the corollary below, the generator matrix $Q(t;\theta_m)$ defined by (2.10) has to be calculated for the fixed time t and parametric set θ_m obtained on m-th iteration of the EM algorithm. Then the eigenvalues and eigenvectors of $Q(t;\theta_m)$ have to be extracted using a numeric method like QR-decomposition, for example. This procedure has to be repeated for each m-th iteration of the EM algorithm with a new set of model parameters θ_m.

Corollary 4. *Assume the dynamics of the CTMC process $\boldsymbol{X}$ is defined by transition probability matrix $P^X(t;\theta) = \exp(tQ(\theta))$, then the following expression holds*

$$\int_0^t P^X_{ab}(s;\theta)P^X_{cd}(t-s;\theta)ds = \sum_{i\in I}u_{ai}u^{-1}_{ib}\sum_{j\in I}u_{cj}u^{-1}_{jd}\Phi_{ij}(t), \tag{2.35}$$

where

$$\Phi_{pq}(t;\theta) = \begin{cases} te^{t\lambda_i} & : \lambda_i = \lambda_j \\ \dfrac{e^{t\lambda_i} - e^{t\lambda_j}}{\lambda_i - \lambda_j} & : \lambda_i \neq \lambda_j, \end{cases} \tag{2.36}$$

λ_k are eigenvalues of $Q(t;\theta)$, u_{lr} are elements of matrix U with corresponding eigenvectors of $Q(t;\theta)$, $\theta = \{\nu, \alpha_0, \alpha_1\}$ are the model parameters.

Proof. In order to compute the matrix $P^X(t;\hat{\theta}_m)$ we use a decomposition of $(2N+1)\times(2N+1)$ matrix $Q = Q(\hat{\theta}_m)$ as follows

$$Q = UD_\lambda U^{-1}, \tag{2.37}$$

where U is a matrix, each row of which is a certain eigenvector, and D_λ is a diagonal matrix that consists of corresponding eigenvalues of the matrix Q. This decomposition has the following well-known property (see p. 195 in [36], [72])

$$P(t) = \exp(tQ) = U\exp(tD_\lambda)U^{-1}, \tag{2.38}$$

where $\exp(\cdot)$ is a matrix exponential. The elements $P_{ab}(s)$ and $P_{cd}(t-s)$ of this matrix, according to the matrix multiplication rules, are given by

$$P_{ab}(s) = \sum_{i\in I} u_{ai}u_{ib}^{-1}e^{s\lambda_i}$$
$$P_{cd}(t-s) = \sum_{j\in I} u_{cj}u_{jd}^{-1}e^{(t-s)\lambda_j}$$

Therefore,

$$P_{ab}(s)P_{cd}(t-s) = \sum_{i\in I} u_{ai}u_{ib}^{-1}\sum_{j\in I} u_{cj}u_{jd}^{-1}e^{\lambda_i s+\lambda_j(t-s)}$$
$$\int_0^t p_{ab}(s)p_{cd}(t-s)ds = \sum_{i\in I} u_{ai}u_{ib}^{-1}\sum_{j\in I} u_{cj}u_{jd}^{-1}\int_0^t e^{\lambda_i s+\lambda_j(t-s)}ds,$$

where

$$\int_0^t e^{\lambda_i s+\lambda_j(t-s)}ds = \begin{cases} te^{t\lambda_i} & : \lambda_i = \lambda_j \\ \dfrac{e^{t\lambda_i}-e^{t\lambda_j}}{\lambda_i-\lambda_j} & : \lambda_i \neq \lambda_j. \end{cases}$$

□

As a result of m-th iteration the vector $\hat{\theta}_{m+1}$ is obtained and, consequently, the implicit matrix $Q(\hat{\theta}_{m+1})$ and $P^X(t;\hat{\theta}_m)$ can be calculated, which are inputs for the next iteration of the EM algorithm.

Summing up, Theorem 5 provides us with the theoretical basis of the E-step calculation. Note, this theorem does not provide an analytical solution; it is necessary to use numerical methods for eigendecomposition of $Q(t;\hat{\theta}_m)$ on each E-step of each EM iteration. Further, the M-step also has to be made using one of the numerical optimization methods, such as the BFGS method named after its authors Broyden—Fletcher—Goldfarb—Shanno. Theorem 4 guarantees improvements of estimation in the regards to improvement of the likelihood function L^d.

2.3.2. Application

As mentioned previously, it is impossible to use the continuous-time likelihood function $L^c(\theta|x)$ from (2.21) *directly* due to the lack of continuous-time sample $\mathbf{x}$ in the real world, since the use of a discrete time (incomplete) sample $\mathbf{y}$ with $L^c(\theta|x=y)$ would causes biased estimates. Meantime, the complete-data likelihood $L^c(\theta|x)$ and incomplete-data likelihood $L^d(\theta|y)$ are tightly connected as it was proven in Lemma 2. The presence of this connection is allowed by iterative EM maximization[6] of expectation $E[L^c(\theta|x)|y]$, to maximize monotonically its discrete-time counterpart $L^d(\theta|y)$ according to Theorem 5.

As a starting point, the following initial parameters values for testing of algorithm performance for the EM algorithm are used: $\hat{\theta}_0 = (1,0,1)$. The E-step requires an eigen decomposition of the intensity matrix $Q(\Delta t;\theta_m)$ from (2.37). An eigendecomposition is the most crucial step of the whole algorithm, because a robustness of the algorithm depends very strongly on quality of this decomposition. In order to calculate a matrix with of eigenvector U, matrix D with eigenvalues on its diagonal and the inverse of matrix U from the Corollary 4 used AlgLib library [13]. A calculation of the transition probability matrix $P^X(\Delta t;\theta_m)$ by QR decomposition of $Q(\Delta t;\theta_m)$ is formalized as the subroutine named *Matrix_P_eigen_decomposition*, as well as all further steps and sub-steps according to the logic of the procedural programming paradigm. Next, all values $\Phi_{pq}(\Delta t;\theta_m)$ defined in Corollary 4 are calculated in the corresponding subroutine *Matrix_Phi* and stored in a memory. Unfortunately, they depend on the eigenvalues matrix D derived from $Q(\Delta t;\theta_m)$. Therefore, we have to repeat calculation of them on each E-step. Having matrices U, D, $\Phi_{pq}(\Delta t;\theta_m)$ and $P^X(\Delta t;\theta_m)$, it is possible to proceed to the calculation of $E_{\hat{\theta}_0}[R_i|y]$ and $E_{\hat{\theta}_0}[N_{ij}|y]$ from (2.34) using Corollary 4. As a result, the expectation of likelihood function $E_{\theta_m}[L^c(\theta|x)|y]$ defined in (2.33) can be calculated.

The next stage of an iteration is the M-step defined by equation (2.26). Namely, we need to maximize $E_{\theta_m}[L^c(\theta|x)|y]$ with respect to θ. A non-linear congruential optimizer (BLEIC optimizer - Bound and linear equality/inequality constrained optimizer - from AlgLib library) is used for this purpose with the following settings: starting point $\theta = (1,0,1)$, lower boundary conditions $(0.01,-3,-5)$, and upper boundary conditions $(7,3,5)$ corresponding to parametric vector $\theta = (\nu,\alpha_0,\alpha_1)$. The maximum is denoted by $\hat{\theta}_{m+1}$ and used on the next iteration instead of $\hat{\theta}_m$. The process is continued till the stopping criteria are not fulfilled. Shortly, the algorithm can be sketched as compact scheme.

1. $\hat{\theta}_0 := (1,0,1)$
2. E-step:
   (a) Call *Matrix_P_eigen_decomposition*$(Q(\hat{\theta}_m)) \rightarrow U$, D, P^X;
   (b) Call *Matrix_Phi*$(D) \rightarrow \Phi$;
   (c) Call *Matrix_E_R*$(U, P^X, \Phi, Q, C) \rightarrow$ vector of $E_{\hat{\theta}_m}[R_i|y]$ for $i \in I$;
   (d) Call *Matrix_E_N*$(U, P^X, \Phi, Q, C) \rightarrow$ matrix of $E_{\hat{\theta}_m}[N_{ij}|y]$ for $i, j \in I$;
3. M-step:

[6] In order to use the EM algorithm C++ programme was created, where by the AlgLib [13] and TNT [79] libraries were used for complicated mathematical calculations.

(a) $\arg\max_\theta E_{\theta_m}[L^c(\theta|x)|y] \rightarrow \hat{\theta}_{m+1}$;

4. IF (stopping criteria is not true) THEN $\hat{\theta}_m = \hat{\theta}_{m+1}$, go to (2) ELSE $\hat{\theta}^* = \hat{\theta}_m$.

where C is a matrix of transitions in the incomplete sample data y.

□

In order to check the performance of the presented EM algorithm based estimation approach, synthetic continuous-time trajectories of the opinion dynamics index process $\mathbf{X}$ defined in the Section 2.1 for the time-horizon $T = 200$ were simulated using the Monte-Carlo technique. Each synthetic continuous-time trajectory consists of around 1200 transitions, which is the number of transitions the process $\mathbf{X}$ makes until T. Further, the discrete-time sample was extracted from each synthetic continuous-time trajectory by selecting values of it with the discretization step $\Delta t = 1$. So, the result of such a procedure is a discrete-time trajectory with 200 transitions. These trajectories were used as an input for the tested estimation procedures. Then, the statistical properties of the obtained parameter estimates were investigated, to analyzed the advantages and disadvantages of the approaches.

2.3.3. Error metrics

In order to compare estimates obtained by the EM algorithm and other approaches we use:

- Mean value of $\bar{\hat{\theta}}$ – measure of accuracy. It shows whether an estimator is biased or not;
- Corrected sample standard deviation (SD) – measure of estimation precision:

$$SD = \sqrt{\sum_{i=0}^{n} (\hat{\theta}_i - \bar{\hat{\theta}})^2 \Big/ (n-1)},$$

  where n is a number estimates obtained during n Monte Carlo experiments;
- Relative Standard Error (RSE) - is a standard error of the mean (or SEM) divided by value of the mean:

$$RSE = \frac{SEM}{\bar{\hat{\theta}}} \times 100\% = \frac{SD}{\sqrt{n}\bar{\hat{\theta}}} \times 100\%,$$

  where n is a number estimates obtained during n Monte Carlo experiments. An estimate is good enough in terms of precision if RSE is less than around 30%;
- Median – is also a measure of accuracy, but less sensitive to extreme values than a mean;
- Root-Mean Square Error (RMSE) – measure of both accuracy and precision, because it incorporates both, variance of estimator and its bias:

$$RMSE = \sqrt{\sum_{i=0}^{n} (\hat{\theta}_i - \theta)^2 \Big/ n},$$

  where n is a number estimates obtained during n Monte Carlo experiments.

As we can see standard deviation and root-mean square error are very close in the case of unbiased estimator.

2.3.4. Numerical results

Here we will discuss the outcomes and performance of the EM algorithm. The results of 100 repeated estimations of 100 Monte-Carlo simulated data samples for ABM models with the number of agents N equal to 5, 10, 15, 20 and different parameters vectors θ based on the methodology from Section 2.1.6 are presented in Tables 2.3 and 2.4 and Figures 2.10 and 2.11. To compare the results, the maximum likelihood estimates based on the continuous-time $L^c(\theta|x)$ are also presented.[7] Let us analyze these results.

		L^c-MLE			EM algorithm		
		$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$	$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$
S	Mean	3.004	0.015	0.747	3.025	0.015	0.747
E	Median	3.000	0.008	0.768	3.037	0.008	0.767
T	SD	0.086	0.021	0.133	0.297	0.021	0.130
1	RMSE	0.086	0.026	0.142	0.296	0.026	0.140
S	Mean	3.005	0.253	0.711	3.038	0.257	0.701
E	Median	3.012	0.237	0.724	3.067	0.237	0.706
T	SD	0.094	0.100	0.179	0.278	0.103	0.181
2	RMSE	0.094	0.272	0.199	0.279	0.277	0.206
S	Mean	3.009	0.043	1.131	3.042	0.045	1.125
E	Median	3.012	0.009	1.163	3.030	0.010	1.158
T	SD	0.097	0.065	0.109	0.293	0.068	0.113
3	RMSE	0.097	0.078	0.348	0.294	0.081	0.344
S	Mean	2.999	0.286	1.096	2.979	0.321	1.040
E	Median	2.997	0.242	1.129	2.971	0.275	1.073
T	SD	0.114	0.180	0.233	0.329	0.193	0.249
4	RMSE	0.113	0.338	0.376	0.328	0.374	0.345

Table 2.3: The results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each. The L^c-MLE estimates based on the original continuous-time sample paths (as a benchmark) and EM estimates based on the corresponding discrete-time sample paths. ABM model with 20 agents.

Experiments settings

The settings of the Monte Carlo simulation experiments are the following for the EM algorithm, if not mentioned otherwise:

- 100 discrete-time paths obtained from discretization of 100 simulated continuous-time sample paths[8];

[7] It is possible to use complete-data likelihood function $L^c(\theta|x)$ in Monte Carlo simulations, because we simulate firstly the continuous-time (complete) sample path x, and the discrete-time sample path y using discretization of x, while the only discrete-time sample y is usually available in real world examples.

[8] Recall, a continuous-time data for the model is unavailable in real world.

- Time-horizon for each continuous-time sample $T = 200$[9];
- The step of discrete "observation" of corresponding continuous-time sample paths $\Delta t = 1$, therefore there are 200 discrete observations;
- ABM-5,10,15,20 model versions with the number agents $N = 5, 10, 15, 20$, correspondingly;
- The initial point of optimization subroutines $\hat{\theta}_0 = (4, 0.5, 15)$;
- The parametric sets:
  - Set 1: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 0.8$,
  - Set 2: $\nu = 3$, $\alpha_0 = 0.2$, $\alpha_1 = 0.8$,
  - Set 3: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 1.2$,
  - Set 4: $\nu = 3$, $\alpha_0 = 0.2$, $\alpha_1 = 1.2$.
- The continuous-time sample paths (complete-data) in such settings consist of about 1200 transitions comparing to only 200 "observed" transitions in the corresponding discrete-time sample paths (incomplete-data).

The simulation procedure of continuous-time sample paths described in Subsection 2.1.6 starts from point $x_0 = 0$ and provides the vector of process $\mathbf{X}$ states $\mathbf{x}$, the time intervals between transitions U_n, the matrix with transitions number $N(T) = \{N_{ij}(T)\}$, the vector with occupation times $R(T) = \{R_i(T)\}$. This data is enough to calculate the likelihood function for the L^c-based MLE method. Further, this data allows us to construct a discrete-time sample, namely to obtain the vector with states $\mathbf{y} = \{y_i\}_{i=0}^{200}$ obtained with the observation step $\Delta t = 1$. Further, the $\mathbf{x}$ and $\mathbf{y}$ sample paths were estimated in parallel[10]; the results of estimation and their times were collected for further analysis and error metrics computation.

Main results

Let us begin our analysis of the EM algorithm results with the estimates of parameters α_0 and α_1 (see Table 2.3 with the results for the EM algorithm and L^c-based MLE). The median values of estimates $\hat{\alpha}_0$ for Set 1 and Set 3 are very close to zero, 0.008 and 0.01, and seem to be unbiased. The median values of $\hat{\alpha}_1$ for Set 2,4 are not so perfect, 0.237 and 0.275, but small nevertheless. In general, the EM estimates of $\hat{\alpha}_1$ show very similar tendencies to $\hat{\alpha}_0$. The only difference is a slightly higher magnitude of errors of α_1. Indirectly, unbiasness is confirmed by the values of the standard deviation and root-mean square error that are: very close for Set 1 and both α_0, α_1; close for Set 2,3 and of parameters; while for Set 4 the difference is more significant, as well as the magnitude. Thus, the level of root-mean square error and standard deviation is a result of variance of the estimator, mostly for Set 1 and less for Set 4. For instance, the root-mean square error (of α_0) has a magnitude around 0.374 for Set 4, while SD is 0.193; therefore, the root-mean square error and standard deviation

[9] See the details of simulation technique in Subsection 2.1.6.

[10] Each estimation was produced on its own core of the CPU in parallel. Thus, the performance of a simulation experiment depends on the number of CPU cores, as an estimation procedure was not parallelized.

are not so close. In this sense, it is useful to compare the values of mean and median. There is evidence that the difference between the median and mean is higher for sets with higher errors. The reason for that are the outliers of estimates for some Monte Carlo simulations, which can appear because of poor convergence of numerical optimization subroutine due to local maximums of the likelihood function or accumulation of computation errors of numerical calculation procedures. Another reason can be insignificant size of sample. Regardless, the presence of biases does not mean that they are the only explanation for the higher magnitude. The main reason is a higher variation of estimates of these sets of parameters, which seems to be a more complicated case for an estimation.

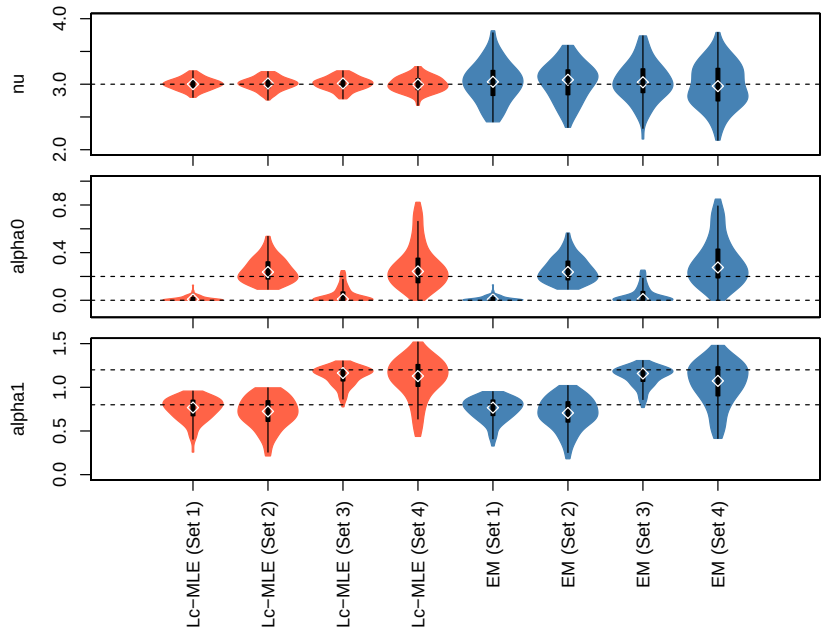


Figure 2.10: The results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each for various parametric sets by two estimation methods: L^c-MLE (complete-data) and EM algorithm (incomplete-data). ABM-20 model.

Figure 2.10 depicts the results of two methods estimating the same samples simulated with various parametric sets, as described in the Experiments Settings subsection above. The estimations distributions are represented in the figure as violins (symmetric around y-axis empirical density functions) for all investigated combinations of methods and parametric sets on three panels for the corresponding parameters ν, α_0 and α_1. An inspection of Fig 2.10 shows that the estimates $\hat{\nu}$ tend to have the low biases (around 3.042 in the worst case), but the standard deviation and root-mean square errors are significantly higher in absolute values (both around 0.3). The errors are almost the same for all parameters' sets. They are a bit larger for Set 4, but in general the parameter ν is much less sensitive to the change of parameter sets than the other two parameters. The reason for that may be in the construction of the transition probability matrix (2.16). Recall, it is given by

$$P^X(t;\theta) = \exp(tQ(\theta)).$$

The key feature of its construction is possibility to isolate parameter ν from the other two in the following way

$$\exp(tQ(\theta)) = \exp(t\nu\widetilde{Q}(\alpha_0, \alpha_1)).$$

As a result, it is evident that the transition probabilities P^X have more straightforward dependency on ν than on α_0 and α_1. This can be the reason for low sensitivity of ν with respect to different parameters' sets. At the same time, it explains the similar properties of $\hat{\alpha}_0$ and $\hat{\alpha}_1$.

In order to make a general conclusion about the quality of EM estimates it is good to check out the values of RSE. Let us provide the values of relative standard error for Set 4 which has higher errors of estimates:

- $RSE_{set4}(\hat{\nu}) = 1.13\%$[11],
- $RSE_{set4}(\hat{\alpha}_0) = 6.15\%$,
- $RSE_{set4}(\hat{\alpha}_1) = 2.44\%$.

It can be seen from Table 2.3 that the absolute errors of $\hat{\nu}$ are comparable, while the relative errors give another picture: the relative standard error of $\hat{\alpha_0}$ is significantly higher. It is, however, important to note that even 6.15% is very well and satisfactory result, because it is less than the moderate level 30%. In other words, the method shows good precision.

Note, the ML-estimates of α_0 and α_1 based on $L^c(\theta)$ and continuous-time sample have absolutely the same properties as the EM estimates; their precision and accuracy presented in first columns of Table 2.3 are also very close to the EM case. Only the estimates of ν are a little bit better. Overall, the EM algorithm using an incomplete data sample shows very close quality of estimates with respect to the classical maximum likelihood estimates based on the complete data sample.

Models comparison

Another Monte Carlo simulation experiment is dedicated to differences in estimation of ABM based on various numbers of agents N. During the experiment, the results of estimation of the models ABM-5, ABM-10, ABM-15 and ABM-20 are compared with the number of agents $N = 5, 10, 15, 20$, correspondingly. The estimation results' metrics are collected for the methods L^c-MLE (complete-data) and EM algorithm (incomplete-data) in Table 2.4 visualized in Figure 2.11.

A brief inspection of Fig. 2.11 shows two tendencies. The first one is the tendency of ν estimates to be biased for ABM-5, less biased for ABM-10,15 and unbiased for the case of ABM-20. On the contrary, the deviation of estimates α_0 and α_1 becomes worse for models with a higher number of agents. These tendencies hold for both the complete-data based L^c-MLE and incomplete-data based EM algorithm.

Deeper investigation of the estimations' results is based on the analysis of the error metrics collected in Table 2.4. The first fact confirming the evidence from Figure 2.11 is the monotonic growth of SD and RMSE for both complete-data based L^c-MLE and incomplete-data based EM algorithm for α_1, namely from around 0.068 to 0.142 and from 0.088 to 0.140, correspondingly. The error metrics of ν estimates are improving, the RMSE of EM algorithm

[11] It tells what the size of standard error (deviation) is comparing with the estimate size in percent.

		L^c-MLE			EM algorithm		
		ν	α_0	α_1	ν	α_0	α_1
A	Mean	3.004	0.014	0.780	2.847	0.014	0.740
B	Median	3.006	0.002	0.788	2.826	0.000	0.743
M	SD	0.093	0.020	0.065	0.347	0.021	0.065
5	RMSE	0.093	0.024	0.068	0.378	0.026	0.088
A	Mean	3.006	0.013	0.774	3.021	0.012	0.771
B	Median	3.010	0.003	0.782	3.055	0.004	0.788
M	SD	0.090	0.018	0.093	0.292	0.017	0.090
10	RMSE	0.090	0.022	0.096	0.291	0.021	0.094
A	Mean	3.005	0.013	0.763	3.074	0.013	0.764
B	Median	3.007	0.006	0.787	3.013	0.007	0.782
M	SD	0.088	0.019	0.112	0.312	0.020	0.111
15	RMSE	0.088	0.023	0.117	0.319	0.024	0.116
A	Mean	3.004	0.015	0.747	3.025	0.015	0.747
B	Median	3.000	0.008	0.768	3.037	0.008	0.767
M	SD	0.086	0.021	0.133	0.297	0.021	0.130
20	RMSE	0.086	0.026	0.142	0.296	0.026	0.140

Table 2.4: The results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each. L^c-MLE columns are the estimates obtained by maximization of the continuous-time sample (as a benchmark) and the EM algorithm columns are the estimates obtained by EM algorithm estimation of the discrete-time sample for models with different numbers of agents N.

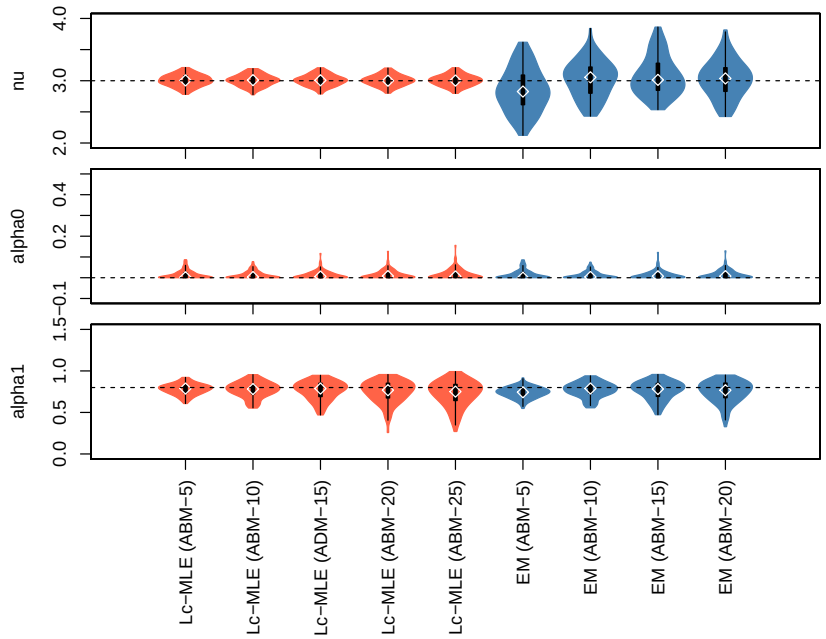


Figure 2.11: The results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each for ABM-5,-10,-15,-20 versions of ABM model and parametric Set 1 by three estimation methods: L^c-MLE (complete-data) and EM algorithm (incomplete-data).

estimates, for instance, decrease from 0.378 to 0.296. The error metrics of α_0 estimates do not indicate a significant difference.

Summing up, the EM algorithm provides a fine quality of estimates of three parameters for all considered ABM model versions (ABM-5,-10,-15,-20). The variation in the results' quality is weak for the model when the number of agents is $N \leq 20$.

2.3.5. Discussion

The results of EM based estimation are satisfactory in terms of error metrics, especially for parameters ν and α_1. As can be seen in Figure 2.10, only the ν estimates are significantly worse if they are compared to incomplete-data $\mathbf{y}$ based EM results with complete-data based results of $L^c - MLE$ method, but only in terms of the standard deviation of the obtained estimates. The rest of the behavior of the estimators is very similar – the same tendency for Set 2,4 of higher biases and higher deviation of α_0 and α_1 estimates. Moreover, the values of metrics are almost the same for α_0 α_1.

The drawback of the EM algorithm lies in another plane. An EM algorithm application to the considered ABM model is computationally intensive. It is impossible to estimate ABM models with number of agents N more than around 20 with the EM algorithm, because the time of estimation increases dramatically. The reason for that is in the highly recurrent construction of the vector $(E_{\hat{\theta}_m}[R_i(T)|y])_{i=-N}^{N}$ and, especially, matrix $(E_{\hat{\theta}_m}[N_{i,j}(T)|y])_{i,j=-N}^{N}$ from (2.34). A computation of its elements requires $(2N+1)^4$ floating-point operations. Moreover, in order to make the E-step (2.26) we have to use numerical optimization, which is based on the numerical evaluation of derivatives. It means we have

to repeat computation of (2.34) at least three times at each step of the optimization procedure. More detailed results concerning time-consumption are collected and discussed in Section 2.7. Further, the number of agents N is also bound by the eigendecomposition of the matrix Q; this problem (catastrophic cancellation) is discussed in the next section. Nevertheless, the EM algorithm is applicable on an average PC in cases without a large number of agents N.

□

Further, we consider an alternative to the EM algorithm approaches, which allow estimation based on direct maximization of $L^d(\theta)$ (2.23) for large N.

2.4. Direct computation of discrete-time likelihood function approach

A different approach may be taken in order to overcome the weakness of the EM algorithm, namely the calculation during all iterations of the following matrices with expectations

$$\left\{E_{\hat{\theta}_m}\left[R_i(T)\middle|y\right]\right\}_{i=-N}^{N}:\ 1\times(2N+1) \tag{2.39}$$

$$\left\{E_{\hat{\theta}_m}\left[N_{i,j}(T)\middle|y\right]\right\}_{i,j=-N}^{N}:\ (2N+1)\times(2N+1), \tag{2.40}$$

These expectations are necessary for the calculation of the expectation of continuous-time likelihood function $L^c(\theta|x)$ (2.34) conditionally on the observed discrete data sample $\mathbf{y}$. The approach suggests calculation of the transition probability matrix $P^X(\Delta;\theta)$ (2.16) numerically and to use it for computation of the likelihood function $L^d(\theta)$, which is maximized in the logic of the MLE method.

2.4.1. Theoretical description

The crucial part of this and the next approach is calculation of the matrix exponential $\exp(tQ(\theta))$, which is quite a complicated numerical problem. Unfortunately, it does not have a straightforward settled solution. There are plenty of ways to solve it, each with own advantages and disadvantages, including methods from different fields, such as: matrix decompositions, differential equations, and various approximations. The most widely used are described and analyzed in Moler and Loan's paper [72] (2003). The situation is complicated by the sparsity of the matrix $Q(\theta)$, which grows with the number of agents N; since $Q(\theta)$ is tridiagonal the number of zero entries is $(2N+1)\times(2N-2)+2$. Further, $Q(\theta)$ is non-negative, non-symmetric and singular by definition, with a zero eigenvalue [88] and for all others on the open left half plane, their eigenvalues are real.

As mentioned in [72], there is a group of matrix exponential $P^X(\Delta;\theta)=\exp(\Delta Q(\theta))$ computation methods based on the following property. Let Q be decomposed as

$$Q = SBS^{-1},$$

then, the definition of matrix exponential $\exp(tQ)$ implies

$$\exp(tQ) = S\exp(tB)S^{-1}.$$

In this approach, the eigendecomposition (2.37) is used, which is also used as part of the EM algorithm

$$P^X(t;\theta) = \exp(tQ(\theta)) = U\exp(tD_\lambda)U^{-1},$$

where D_λ is a diagonal matrix with eigenvalues on diagonal, U is a matrix whose columns are corresponding eigenvectors of Q. The main advantage of this decomposition is that e^{tD} is given by

$$\exp(tD) = diag(\exp(\lambda_1),\ldots,\exp(\lambda_{2N+1})).$$

Moreover, U is a nonsingular matrix, hence the inverse matrix U^{-1} exists.

Further, the transition probability matrix $P^X(t;\theta)$ of the CTMC process $\mathbf{X}$ computed the eigendecomposition as above using numerical methods. Then it is directly used for the computation of the likelihood function L^d (2.23). Recall, it is given by

$$L^d(\theta|y) = \prod_{i,j\in I}\left(P^X_{ij}(\Delta t;\theta)\right)^{c_{ij}},$$

where c_{ij} is a number of transitions from the state i to the state j in the discrete-time sample y.

On the last step, it is necessary to maximize $L^d(\theta|y)$ for given discrete observations $\mathbf{y}$. Unfortunately, it is not possible to perform analytically, because we cannot formulate F.O.C. due to the absence of the closed-form of $P^X_{ij}(\Delta t;\theta)$ In particular, we need to know partial derivatives $P^X_{ij}(\Delta t;\theta)$. So, the likelihood function $L^d(\theta|y)$ is maximized using numerical optimization methods.

2.4.2. Application

The key procedure of the method is a computation of the transition probability matrix $P^X(t;\theta)$, which consists of two stages that are necessary for eigendecomposition (2.37): first, a derivation of eigenvalues and corresponding eigenvectors; second, an inversion of the matrix U. A realization of both stages has a strong influence on the quality and speed of the whole method. Therefore, they have to be carefully, efficient and quickly computed. Thus, in order to implement the method, two C++ libraries were used for numerical computations AlgLib [13] and TNT [79]. Each of them has the necessary subroutines of eigendecomposition and matrix inversion, but as we will see further, they have different round-off errors and, as a result, different degrees of robustness.

Another important part of the method is the numerical optimization problem, that is

$$\theta^* = \arg\max_\theta(L^d(\theta)). \tag{2.41}$$

As with the previous step, there is no sense in reinventing the wheel and creating our own C++ realization of well-known optimizers. Instead, we used two optimizers from the package AlgLib: nonlinear congruential optimizer (it is called BLEIC[12] in AlgLib documentation)

[12] BLEIC – boundary and linear equality/inequality constraints.

and Levenberg-Marquart (LM) optimizer.[13] It may look a bit odd, because the Levenberg-Marquart method is known and widely used as optimizer for least squares problems, but it also can be used for general optimization problems. In this case, it acts like the Trust Region Newton method. The main advantage of LM for our purposes is that it uses a three-point finite difference stencil for numerical differentiation. Therefore, only three values of L^d have to be computed [14] on each step of this method, whereas the numerical differentiation in the BLEIC subroutine has a five-point stencil. We used it as an optimizer for $L^c(\theta|x)$, which is quite cheap computationally.[15]

In general, the method implementation can be outlined as:

1. $\hat{\theta}_0 := (1,0,1)$
2. Maximization procedure (Levenberg-Marquart or BLEIC):
   (a) Computation of L^d and its numerical partial derivatives (two evaluations of L^d for each):
      - Evaluation of L^d;
        – Computation of $Q(N;\theta_k)$;
        – Decomposing of Q by using (2.37);
        – Computation of $P^X = \exp(Q)$ by using (2.38).

   (b) LM-step;
   (c) IF (stopping criteria is not true) THEN $\hat{\theta}_k = \hat{\theta}_{k+1}$, go to (2a) ELSE $\hat{\theta}^* = \hat{\theta}_{k+1}$.

2.4.3. Numerical results

The procedure of continuous-time and corresponding discrete-time data samples simulation is described in detail in Subsection 2.3.4.

The results of the two estimation experiments using the approach of numerical computation of $L^d(\theta|y)$ are presented in Tables 2.5 and 2.6 (columns of section L^d-MLE), and corresponding Figures 2.12 and 2.13. The first experiment is dedicated to the quality of the ABM-20 model parameters estimates for various parameters' sets, while the second one is dedicated to estimation of various ABM model versions with the number of agents $N = 5, 10, 15, 20$. As the tool of analysis we continue to use the values of mean, standard deviation (SD), root-mean square error (RMSE) and median. The meanings of these were described in Subsection 2.3.4.

Let us begin with the analysis of Table 2.6. The L^d-MLE estimates of parameter ν are more robust for model $ABM-10$ with the best precision and accuracy for the parameter estimates 0.29 (both SD and RMSE), while the worst results are obtained for $ABM-5$, namely 0.366 (SD) and 0.386 (RMSE). Nevertheless, the difference between all of the results is less than 30%; only the $ABM-5$ version is significantly worse than the others. The closeness of the median and mean values indicates there is no (or at least only a few) outliers/failures of

[13] For further details about these algorithms see Numerical Recipes [42].

[14] Each evaluation of the likelihood function $L^d(\theta|y)$ is computationally expensive, because it requires the eigendecomposition of the intensity rates matrix $Q(\theta)$ to be produced.

[15] Recall, we consider ordinary MLE based on the complete-data likelihood $L^c(\theta|x)$ as perfect, but there are rare case in practice; at the same time, it is useful as a benchmark method.

		L^c-MLE			L^d-MLE		
		$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$	$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$
S	Mean	3.004	0.015	0.747	3.032	0.016	0.747
E	Median	3.000	0.008	0.768	3.044	0.006	0.767
T	SD	0.086	0.021	0.133	0.301	0.022	0.133
1	RMSE	0.086	0.026	0.142	0.301	0.026	0.142
S	Mean	3.005	0.253	0.711	3.116	0.290	0.783
E	Median	3.012	0.237	0.724	3.102	0.264	0.736
T	SD	0.094	0.100	0.179	0.478	0.126	0.309
2	RMSE	0.094	0.272	0.199	0.489	0.315	0.308
S	Mean	3.009	0.043	1.131	3.272	0.148	1.230
E	Median	3.012	0.009	1.163	3.156	0.008	1.198
T	SD	0.097	0.065	0.109	0.534	0.214	0.182
3	RMSE	0.097	0.078	0.348	0.597	0.259	0.467
S	Mean	2.999	0.286	1.096	3.959	0.497	1.489
E	Median	2.997	0.242	1.129	4.000	0.500	1.500
T	SD	0.114	0.180	0.233	0.288	0.021	0.076
4	RMSE	0.113	0.338	0.376	1.001	0.497	0.693

Table 2.5: The results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each. The L^c-MLE estimates based on the artificial continues-time data sample (as a benchmark) and L^d-MLE estimates based on eigendecomposition of P^X obtained by estimation procedure from Section 2.4. ABM model with 20 agents.

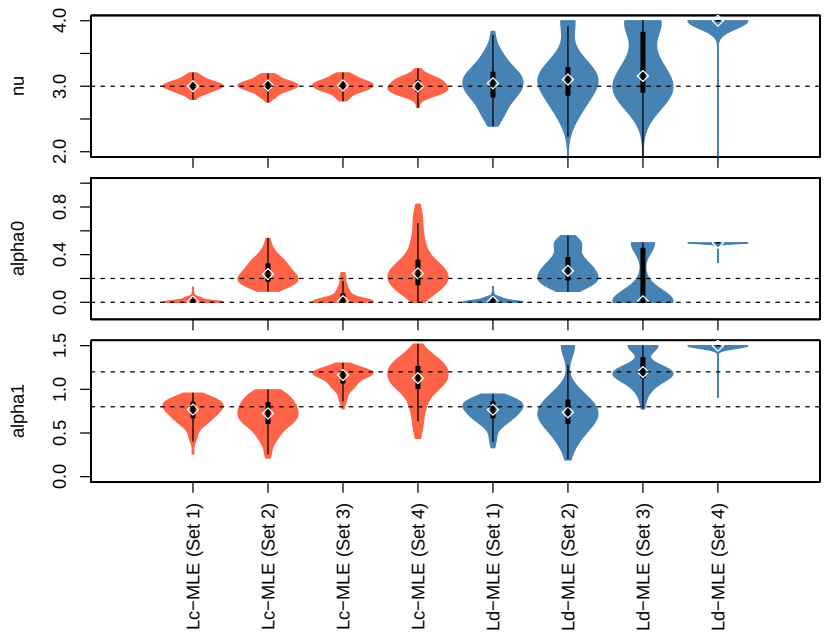


Figure 2.12: The distribution results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each for various parametric sets by two estimation methods: L^c-MLE (complete-data) and L^d-MLE (incomplete-data). ABM-20 model.

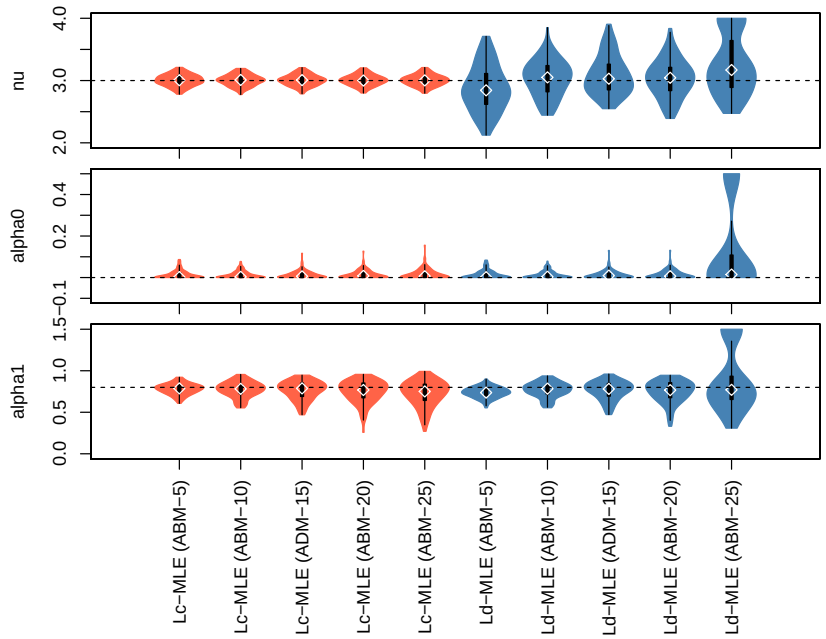


Figure 2.13: The distribution of results of estimates based on 100 Monte Carlo simulated discrete sample paths with a length of 200 discrete-time points extracted from continuous-time sample paths with around 1200 transitions each for ABM-5,-10,-15,-20 versions of ABM model and parametric Set 1 by two estimation methods: L^c-MLE (complete-data) and L^d-MLE (incomplete-data).

the estimation procedure. In particular, the standard deviation of L^d-estimate $\hat{\nu}$ is around 0.29, the root-mean square error of $\hat{\nu}$ is 0.29 for $ABM-10$ model. The L^d-estimators of parameters α_0 do not seem to vary significantly for different versions of the model, with SD and RMSE around $0.018-0.026$. The estimates of α_1 display to the opposite to ν-estimates tendency. The values of standard deviation and root-mean square error for those estimates increase monotonically from $ABM-5$ to $ABM-20$. Another tendency is underestimation of the parameter α_1, but the bias is around 5%.

Note, it is clear that the L^d-MLE and EM algorithm provide very close results with very similar quality. This effect has clear theoretical justification. As was mentioned earlier, the sequence of EM estimates $\{\hat{\theta}_k\}$ obtained on each iteration of the algorithm converges to the maximum point of discrete-time likelihood function $L^d(\theta|y)$. This property is called a monotonicity (2.32) of EM sequence and it is given by

$$L^d\left(\hat{\theta}_{k+1}\middle|y\right) \geq L^d\left(\hat{\theta}_k\middle|y\right).$$

Consequently, the L^d-MLE estimates have similar properties and features as the EM algorithm estimates that were discussed in Subsection 2.3.4. Nevertheless, there are differences the in the error metrics levels and behavior for different parameters sets, as we will see further.[16]

Also, Figure 2.13 contains results for the ABM-25 model version. It shows clear signs of instability of estimation procedure. The distributions of estimates have groups of outliers in the bottom of each plot. We will also see the presence of outliers (failures of maximization procedure convergence) in the next experiment.

Table 2.5 collects the metrics of 100 simulated sample paths estimates of the ABM-20 model for various parameters settings. There is a clear tendency of estimates to get worse from Set 1 to Set 4 for both metrics: RMSE of ν-estimates increases from 0.301 (Set 1) to 1.001 (Set 4); α_0-estimates from 0.026 to 0.497; and α_1-estimates from 0.142 to 0.693. Additional information is given in Figure 2.12. The immediately noticeable part of the figure is that the EM estimates and L^d-based estimates distribution show up very similar tendencies for various parameters sets, but there are clear "tails" of results distribution for L^d-MLE estimates. These "tails" are failures of optimization procedure, It is especially clear for Set 4 with median estimates of parameters $(4.0, 0.5, 1.5)$, which is exactly the initial point of the maximization procedure.

□

In the next section, we analyze outcomes and reasons for the method collapse for the ABM model versions with the number of agents $N \geq 25$.

2.4.4. Discussion

The main contrast to the application of the EM algorithm is the only computationally expensive operation in the presented approach, namely the eigendecomposition of the intensity rates matrix $Q(\theta)$. Due to its computational efficiency, it allows us to increase significantly the maximum number of agents N capable for estimation.

[16] The results of L^c-MLE were compared in the previous section with the EM algorithm results, which are very similar to L^d-MLE.

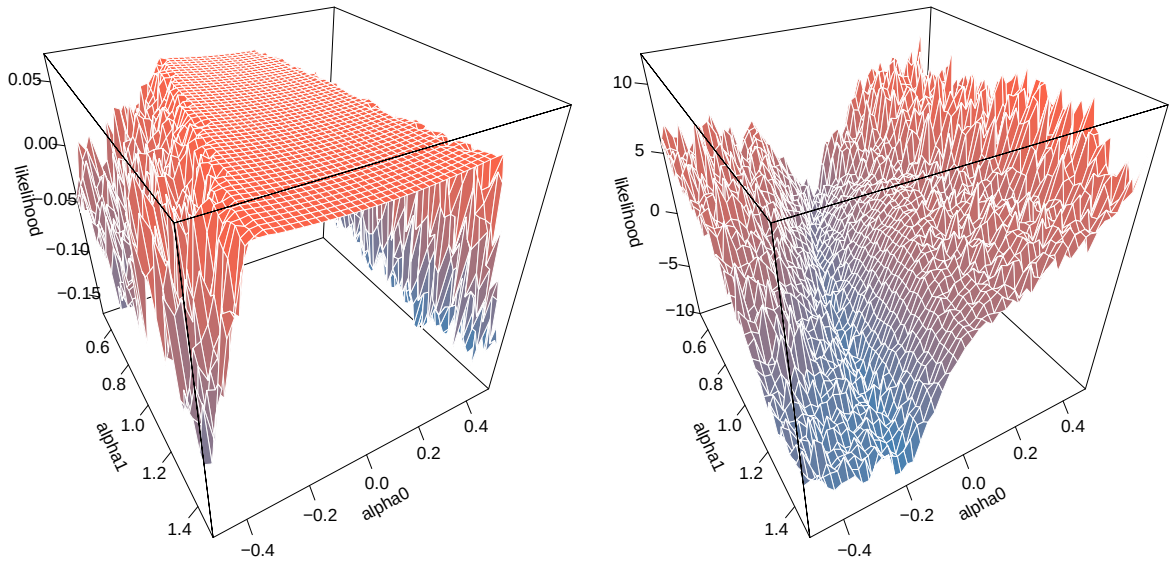
		L^c-MLE			L^d-MLE		
		ν	α_0	α_1	ν	α_0	α_1
A	Mean	3.004	0.014	0.780	2.872	0.014	0.739
B	Median	3.006	0.002	0.788	2.843	0.002	0.737
M	SD	0.093	0.020	0.065	0.366	0.021	0.064
5	RMSE	0.093	0.024	0.068	0.386	0.025	0.089
A	Mean	3.006	0.013	0.774	3.034	0.013	0.769
B	Median	3.010	0.003	0.782	3.051	0.004	0.784
M	SD	0.090	0.018	0.093	0.290	0.018	0.091
10	RMSE	0.090	0.022	0.096	0.290	0.022	0.096
A	Mean	3.005	0.013	0.763	3.084	0.014	0.761
B	Median	3.007	0.006	0.787	3.029	0.009	0.781
M	SD	0.088	0.019	0.112	0.312	0.020	0.112
15	RMSE	0.088	0.023	0.117	0.322	0.024	0.118
A	Mean	3.004	0.015	0.747	3.032	0.016	0.747
B	Median	3.000	0.008	0.768	3.044	0.006	0.767
M	SD	0.086	0.021	0.133	0.301	0.022	0.133
20	RMSE	0.086	0.026	0.142	0.301	0.026	0.142

Table 2.6: The results of averaged estimates based on 200 Monte Carlo simulations (200 discrete time-steps, around 1200 continuous time-steps for each), where L^c-MLE columns are the estimates obtained by maximization of the continues-time likelihood (as a benchmark), L^d-MLE columns are the estimates obtained by maximization of the likelihood based on eigendecomposition of P^X for different numbers of agents N.

Focusing on the general features of the methods, we can point out that the L^d-MLE approach presented in this section is simpler and faster than the EM algorithm. Both of them have subroutines for the matrix $Q(\theta)$ decomposition and matrix P^X computation, but the EM algorithm also requires the highly recursive computation of the vector $(E_{\hat{\theta}_m}[R_i(T)|y])_{i=-N}^{N}$ and matrix $(E_{\hat{\theta}_m}[N_{i,j}(T)|y])_{i,j=-N}^{N}$ from the expression (2.34). This leads to strong acceleration of computations, which in turn makes it possible to estimate agent-based models described in Section 2.1 with a greater number of agents N (up to around 30). Nevertheless, this limitation is still unsatisfactory. In order to overcome it, we should understand the reasons behind the method collapse.

The main aim of L^d-MLE is maximization of the discrete-time likelihood function $L^d(\theta)$ (2.23) as follows from its name. As a part of the method we have a subroutine for computation of $L^d(\theta|y)$. It does the following: 1. Simply simulate one trajectory path; 2. Compute the matrix $Q(\theta)$ from the expression (2.7) and decompose it by using the eigendecomposition (2.37); 3. Compute the matrix $P^X(\Delta;\theta)$, which we use in turn for computation of $L^d(\theta)$. Let us sequentially check the weaknesses of this procedure.

First of all, let us look at the 3D-surface of $L^d(\theta)$ (w.r.t. α_0 and α_1) generated by using the parameters from Set 3 for $N = 50$ for two different realizations of the optimization subroutine, one based on C++ library TNT [79] and another one based on C++ library [13]. They are displayed in Figure 2.14.



(a) AlgLib (b) TNT

Figure 2.14: Instability of likelihood function L^d based on eigendecomposition for a large number of agents. The discrete-time (incomplete) sample $\mathbf{y}$ of $ABM{-}50$ is simulated with the parameters: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 1.2$, then the likelihood function $L^d(\nu, \alpha_0, \alpha_1|y)$ is plotted w.r.t. α_0 and α_1, ν fixed to 3. L^d computed by using AlgLib library [13] eigen decomposition for the left plot, while TNT library [79] is used for the right one.

From Figure 2.14, it is apparent that the surface is strongly distorted, especially in the case of the TNT subroutine. AlgLib is robust, but in a very limited area (approximately for $-0.5 \leq \alpha_0 \leq 0.5$, $0.7 \leq \alpha_1 \leq 1.2$). So, the reason for the method collapse is not in the optimizer subroutine.

Let us check deeper into the method construction. Namely, the weakest point of the method is potentially the eigendecomposition and computation of the matrix $P^X(\Delta)$. The most descriptive and simple manner of analysis is a visual representation. So, it is shown in Figure 2.15.

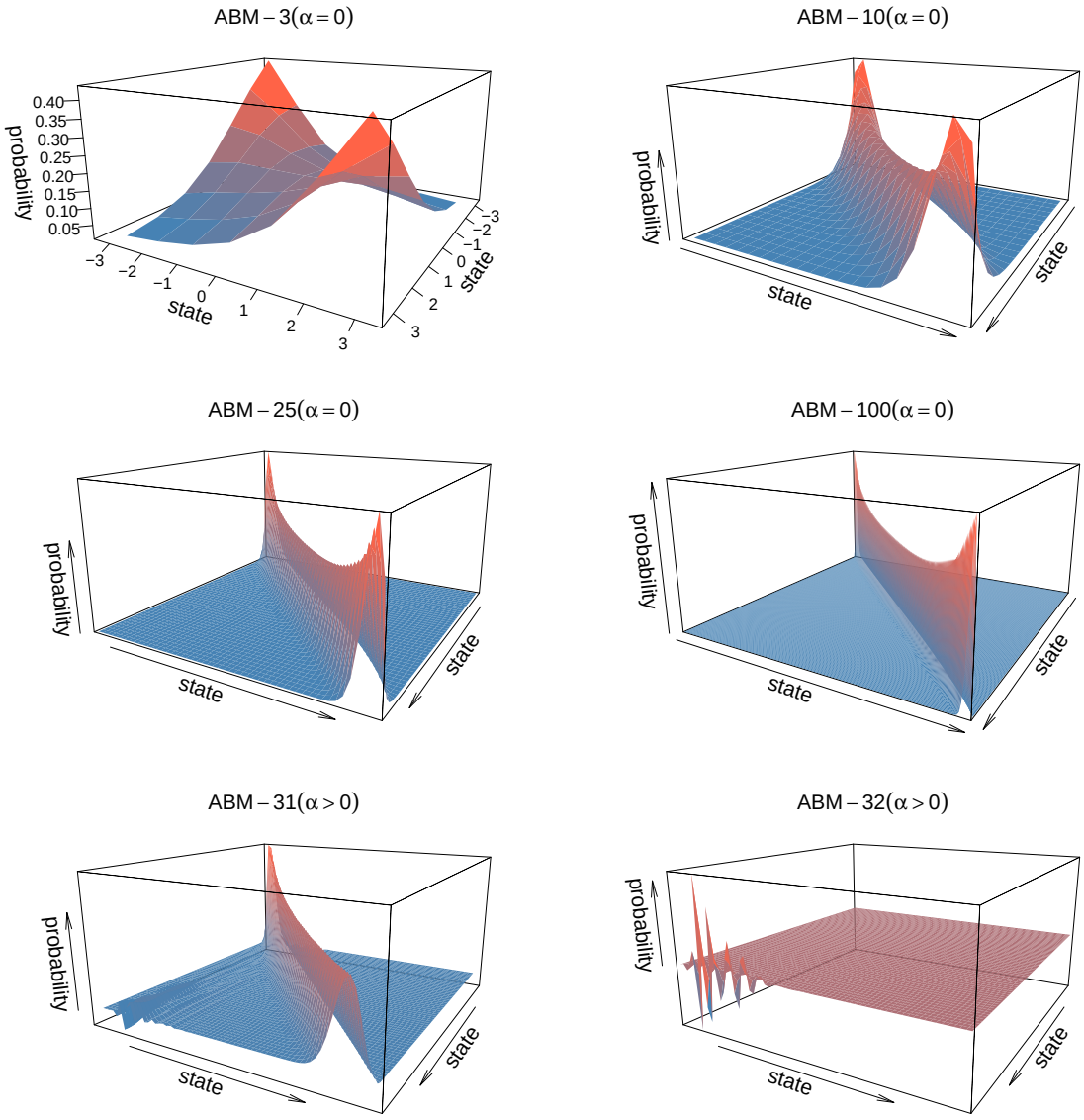


Figure 2.15: Visualization of $N \times N$ matrix $P^X(\Delta)$ entries for different N computed by using AlgLib library [13] for parameter values from Set 1: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 0.8$.

Notably, everything appears fine for ABM-5,-10,-25 and -100 ($\alpha_0 = 0$); troubles only arise for the case of nonzero parameter α_0. In particular, certain artifacts of instability show up already for $N = 30$ (Figure 2.15e). Particularly, there are clear lines on the edge of the curvature part and waves on the plain part. One can see that, with an increase of N up to $N = 32$, great distortions occur (Figure 2.15f). In the literature, this effect is called *loss of significance*.

Let us consider the matrix $P^X(\Delta)$ calculation. That is given by (2.38), namely

$$P^X(t) = \exp(tQ) = U\exp(tD_\lambda)U^{-1}.$$

There are at least two weak points regarding round off errors and loss of significance. The first one is the near confluence of eigenvalues. This problem can lead to loss of accuracy for all methods of matrix exponential computation based on the calculation of eigenvalues.[17] Another difficulty that can lead to large round-off errors is "nearly" defectiveness[18] of matrix U for certain parameter values. In Moler and Loan's (2003) paper, they defined the following measure of defectiveness

$$cond(V) = ||V||||V^{-1}||, \tag{2.42}$$

where $||\cdot||$ is a norm of matrix. This value $cond(V)$ tends to infinity for a defective matrix V; it is large for a "nearly" defective matrix V.

The value $cond(V)$ for different number of agents N and Set 2, 4 were checked out. The results are collected in Table 2.7. One can see from the table that the matrix $Q(N;\theta)$ tends to become defective extremely fast for $\alpha_0 = 0.2$ with increasing of N. This is one of the possible reasons of the described estimation method collapse for $N \geq 30$. The results in Table 2.7

	$\theta_{Set2} = (3,0,1.2)$	$\theta_{Set4} = (3,0.2,1.2)$
$N = 5$	11.2521	36.7164
$N = 25$	5.3215	2.23×10^7
$N = 32$	424.2756	1.15×10^9
$N = 40$	1380.8990	3.25×10^{11}
$N = 50$	5757.3560	1.94×10^{14}

Table 2.7: The values of defectiveness measure $cond(U)$ for the matrix of eigenvectors U for the different parameters N values.

coincide very well with Figures 2.15: both show that, for $\alpha_0 = 0$, the computation of the matrix $P(\Delta)$ is stable, but we see distortion on the plot for $N \geq 32$ for nonzero α_0. At the same time, the measure $cond(U)$ has a very high level that indicates "near" defectiveness of U.

In general, the situation with loss of accuracy in the case our method is an example of what Moler and Loan described as follows: "In practical computation with inexact data and inexact arithmetic, the gray area where the eigenvalues are nearly confluent leads to loss of accuracy". For example, the minimal distance between eigenvalues of the matrix $Q(N;\theta)$ is equal to 0.1645 and the maximum distance is 1.5652 for $N = 5$ (parametric Set 3), whereas for $N = 100$ the minimal distance is 0.00004 and the maximum distance is 0.0084. Note, from empirical observations, eigenvalues lie on the interval $(-12,0)$ for our model. As a result, eigenvalues become closer as N increases, because we have more values on the same interval (-12.0). This is a crucial difficulty for the implementation of the method for the circumstance of our study.

[17] See [72] for more details.

[18] A matrix is called defective if it does not have a complete basis of eigenvectors.

This method is better implemented in cases of symmetric matrix Q or any other matrix that guarantees non-confluent eigenvalues and orthogonal basis of eigenvectors. For our case, the method is robust only for $N \leq 20$ when $\alpha_0 \neq 0$. In fact, the same situation holds for all methods based directly or indirectly on eigenvalues.

Despite this criticism, which means the limitation on the number of agents N can be unsatisfactory for certain problems, this method gives us a great increase in speed and simplicity.

□

The next method is also based on numerical computation of the matrix exponential (2.16).

2.5. Lower Hessenberg matrix approach

The main idea of this approach to exploit the inner structure of the intensity rate matrix $Q(N;\theta)$, namely, that $Q(N;\theta)$ is tridiagonal, which is the partial case of the lower Hessenberg matrix.

2.5.1. Theoretical description

First of all, let us begin with a definition of Hessenberg matrices.

Definition 7. *A square matrix H of size $N \times N$ is called an upper Hessenberg if entries below the first subdiagonal are zeros*

$$H = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & \bullet \end{bmatrix}$$

or a lower Hessenberg matrix if entries above the first superdiagonal are zero

$$H = \begin{bmatrix} \bullet & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$$

In the case of the considered Weidlich agent-based model, the matrix $Q(N;\theta)$ is already a lower Hessenberg; moreover, it is tridiagonal. So, we do not need to decompose $Q(N;\theta)$ in order to obtain the Hessenberg form. Instead, we are able to use it directly.

Theorem 6. *Let $Q(\theta)$ be the intensity rate matrix of CTMC process $\boldsymbol{X}$ defined in Section 2.1, then the columns of the transition probability matrix $P^X(t;\theta) = \exp(tQ)$ are defined by backward iterative formula*

$$P^X_{\bullet,k-1} = \frac{1}{q_{k-1,k}}\left(QP^X_{\bullet,k} - q_{k,k}P^X_{\bullet,k} - q_{k+1,k}P^X_{\bullet,k+1}\right),\ k \in \{-N+1,\dots,N\}. \tag{2.43}$$

where $P^X_{\bullet,i}$ denotes i-th column of the matrix $P^X(\Delta;\theta)$, P^X_N is assumed to be known.

Proof. The proof is given in Appendix A.6. □

Theorem 6 allows us to calculate the transition probability matrix $P^X(t)$ assuming the very last column is known. The only missing piece of the puzzle is the last column of the matrix $P^X(t)$.

As proved earlier in Theorem 1 the transition probability matrix $P^X(t;\theta)$ can be found as a solution of the system of ordinary differential equation known as backward Kolmogorov equation, formulated as

$$\begin{cases} \dfrac{dP^X(t)}{dt} = QP^X(t), \\ P^X(0) = I. \end{cases}$$

This o.d.e. is given by the following equation for an arbitrary element of $P^X(t)$ as

$$\frac{dP^X_{ij}(t)}{dt} = \sum_{k\neq j}\left(q_{ik}P^X_{kj}(t)\right) - q_iP^X_{ij}(t),$$

where $k \in \{-N,\dots,N\}$, $q_i = \sum_{k=-N}^{N} q_{ik}$. Therefore, all the elements of j-th column of transition probability matrix $P^X(t)$ are defined by ordinary differential equations including only elements of j-th column. Thus, the very last column of $P^X(t)$ is described by the system of $2N+1$ o.d.e.

$$\begin{cases} \dfrac{dP^X_{\bullet,N}(t)}{dt} = QP^X_{\bullet,N}(t), \\ P^X_{\bullet,N}(0) = (0,\dots,1)^T. \end{cases} \tag{2.44}$$

Its solution $P^X_{\bullet,N}(t)$ can be obtained numerically using such methods of differential equations as the Runge-Kutta method and many others[19].

Summing up, it is possible to compute the last column of $P^X(t)$ solving numerical o.d.e. (2.44). Any standard o.d.e. solver[20] can be used in order to implement the method. Then it is necessary to find all the other columns of $P^X(t)$ using iterative formula (2.43) from Theorem 6. Further, the discrete-time likelihood function $L^d(\theta|y)$ from Theorem 3 can be constructed and maximized with respect to the parameters vector θ as well as in the previous approach

$$\theta^* = \arg\max_{\theta}(L^{Hes}(\theta|y)),$$

[19]The whole matrix exponential $P(t) = \exp(tQ)$ can be computed column by column, changing the initial vector $P^X(0)$ using numerical integration methods as described in [72], but it is inefficient to do so.

[20]For example, one of the Runge-Kutta type or finite-difference methods.

where **y** is a discrete-time observations of the CTMC process **X**, $L^{Hes}(\theta|y)$ has the same construction as $L^d(\theta|y)$ and is introduced only in order to distinguish results of the second and the third approaches.

2.5.2. Application

One of the advantages of this method is the relative simplicity of implementation. In fact, we just need an o.d.e. solver (we used the Runge-Kutta-Cash-Karp method realization from AlgLib library [13]). The rest of the procedure of matrix exponential $P^X(\Delta;\theta) = \exp(Q(N;\theta))$ computation is programmed in a dozen lines. As before (in subsection 2.4.2), a non-linear congruential optimizer[21] is used for maximization of the likelihood function $L^{Hes}(\theta|y)$

$$\theta^* = \arg\max_{\theta}(L^{Hes}(\theta|y)). \tag{2.45}$$

The parameters estimations based on maximization of $L^c(\theta|x)$ (also by using non-linear congruential optimizer) are still used as an idealistic case for comparison purposes.

In general, the method implementation can be sketched as:

1. $\hat{\theta}_0 := (1,0,1)$
2. Non-linear congruential optimizer:
   (a) Computation of L^{Hes} and its numerical partial derivatives (two evaluations of L^{Hes} for each iteration):
      - Evaluations of L^{Hes};
        - Computation of $Q(N;\theta_i)$;
        - Solving of the system of o.d.e. (2.44);
        - Computation of $P^X = \exp(Q)$ by using (2.43).

   (b) The optimizer does the new step;
   (c) IF (stopping criteria is not true) THEN $\hat{\theta}_i = \hat{\theta}_{i+1}$, go to (2a) ELSE $\hat{\theta}^* = \hat{\theta}_{i+1}$.

□

Now, let us discuss the numerical results and analyze their errors.

2.5.3. Numerical results for the two-parametric case

Firstly, let us consider the simplified two-parametric case. Namely, we fix the parameter α_0 to value 0 in order the investigate a performance of the method for the simpler two parametric case. It is more convenient to analyze the existence of dependence on initial conditions for the two parametric case. In particular, the two-parametric case is more computationally cheap, so it allows us to gain time. In addition, we can examine the influence of the parameter α_0 on the stability and quality of the method estimates.

[21] BLEIC optimizer from AlgLib library.

SET 3		$\hat{\nu}$	$\hat{\alpha}_1$
	L^c-MLE		
	Mean	2.99883	0.92805
	SD	0.08466	0.53078
	RMSE	0.08446	0.59521
	Median	2.99980	1.10840
	L^{Hes}-MLE		
	Mean	3.17454	0.99665
I.P. 1 =	SD	0.60156	0.85951
(2, 1.5)	RMSE	0.62492	0.88114
	Median	3.03194	1.11298
	Mean	3.69885	2.16478
I.P. 2 =	SD	0.56214	1.28756
(2, 0.5)	RMSE	0.89600	1.60634
	Median	3.99365	2.97623
	Mean	3.07588	1.00105
I.P. 3 =	SD	0.41306	0.54186
(4, 1.5)	RMSE	0.41895	0.57592
	Median	3.01856	1.13430
	Mean	3.00915	1.16871
I.P. 4 =	SD	0.40317	0.87290
(4, 0.5)	RMSE	0.40227	0.87128
	Median	3.00550	1.10598
	Mean	3.50724	1.70420
I.P. 5 =	SD	0.62708	1.34490
(2, 1)	RMSE	0.80533	1.43316
	Median	3.61332	1.40423
	Mean	3.02686	1.00673
I.P. 6 =	SD	0.41298	0.65739
(4, 1)	RMSE	0.41283	0.68361
	Median	2.99680	1.09777

Table 2.8: The results of averaged estimates based on 200 Monte Carlo simulations for different initial points and ABM-200 model version with $\theta = (3, 0, 1.2)$. During estimation α_0 is fixed to zero.

As with the previous methods, we simulated 200 trajectories of the opinion dynamics process $\mathbf{X} = (X_t)_{t=0}^{\infty}$ (see (2.1)) for the parametric Set 1,3. Then, we estimated all the trajectories by using the method described above. The results of the Monte Carlo simulation experiment in the form of mean estimated values, standard deviations, root-mean square errors, and medians are collected in Tables 2.8 and 2.9. Note, the simulated samples are the same as for the previous two estimation approaches and were explained in detail in Subsections 2.1.6, 2.3.4. Further, henceforth the BLEIC optimizer from AlgLib library [13] is used for MLE again.

Unfortunately, an analysis of the method performance by Table 2.8 shows high biases and the dependence on initial points (I.P.) for the parameterization set $\nu = 3,\ \alpha_0 = 0,\ \alpha_1 = 1.2$ of ABM-200 model for different initial points of optimization subroutine. Particularly, the biases are very significant $(0.699, 0.965)$ and $(0.507, 0.504)$ correspondingly for the initial points $(\nu^0, \alpha_1^0) = (2, 0.5)$ and $(\nu^0, \alpha_1^0) = (2, 1)$. The standard deviation and root-mean square error are also higher by 1.5 - 2 times than for the initial point $(4, 0.5)$, for instance. In contrast, the biases are relatively low for other initial points, or even close to zero. For example, the biases are less than 0.08 for ν, 0.2 for α_1 in the case of the initial points $(\nu^0, \alpha_1^0) = (4, 0.5)$, $(4, 1.5)$, $(4, 1)$. This is combined with a tolerable level of standard deviation and root-mean square errors.

Nevertheless, it turns out that the method is unstable. The reason for this is the loss of significance, as with the second method. The nature of the matrix P^X degeneration, however, is different from that of the previous methods. The evidence is shown in Figure 2.16, which depicts the probability transition matrix $P^X(\Delta;\theta)$. It has huge distortion in the columns $-100, 99, 98$ in intersection with rows -10 to 10. These columns are calculated last in the recurrent formula (2.43). So, this is the result of accumulated computational errors on the previous recursion stages.

Stabilization

It was empirically discovered that the method becomes unstable if the number of agents N are in the range between 30 and 50. This fact was the source of the idea for the stabilization of the method above. All columns of the matrix P^X are computed recursively from the last column $P^X_{\bullet,N}(\Delta;\theta)$ obtained from (2.44). However, it is possible to compute any column of the matrix $P^X(\Delta;\theta)$ by using o.d.e. representation (2.44), not only $P^X_{\bullet,N}(\Delta;\theta)$. So, in order to stabilize the method, we recompute two columns over each k column by using the o.d.e. solver. Namely, we recompute the columns for any i such that $2N + 1 - ik,\ 2N + 1 - ik + 1$ are positive. For instance, if $N = 200$ and $k = 50$, then the following columns of the matrix $P^X(\Delta;\theta)$ of size $2N + 1 \times 2N + 1$ are computed by solving o.d.e.: $401, 352, 351, 301, 302, \ldots, 51, 52$ (the rest of the columns are computed by recurrent formula from Theorem 6). By doing so, we minimize accumulated computational errors for each k columns of the P^X computation. Further, k is called *a stabilization parameter k*.

The results of the estimation after the method stabilization do not depend on an initial point of optimizer any more. Due to an insignificant difference of the estimates and corresponding estimation characteristics for the different initial points we do not present these results in form of Table 2.8. We collected the results in Table 2.9 only for one initial point $(\nu^0, \alpha_1^0) = (4, 1.5)$, but for two parametric sets. As we can see there is insignificant bias of the parameter ν estimates (with the magnitude around 10^{-2}) for both parametric cases. The standard deviation is less than the lowest standard deviation on 30% for the unstabilized method. The standard deviation of the estimates based on L^c-MLE (and the continuous-

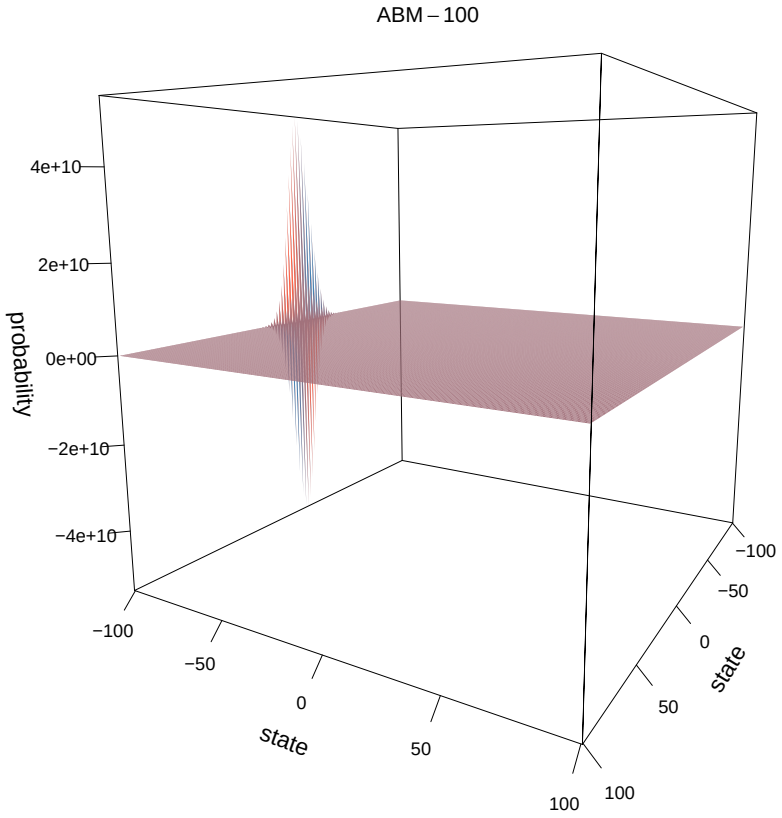


Figure 2.16: Loss of significance for three-parametric case, $\nu = 3, \alpha_0 = 0.2, \alpha_1 = 1.2, N = 100$.

	$\hat{\nu}$	$\hat{\alpha}_1$	$\hat{\nu}$	$\hat{\alpha}_1$
	L^c-MLE			
	SET 1		SET 3	
Mean	2.99989	0.50941	2.99883	0.92805
SD	0.08405	0.56114	0.08466	0.53078
RMSE	0.08384	0.63067	0.08446	0.59521
Median	2.99893	0.62834	2.99980	1.10840
	L^{Hes}-MLE			
	SET 1		SET 3	
Mean	3.00772	0.52180	3.01652	0.93083
SD	0.30687	0.56299	0.33926	0.54243
RMSE	0.30620	0.63122	0.33881	0.60433
Median	2.98873	0.60557	2.99884	1.09454

Table 2.9: The results of averaged estimates based on 200 Monte Carlo simulations of the $ABM-200$ model version for the two-parametric case (the parameter α_0 is fixed with value 0). It stabilized each $k = 100$ column.

time sample) is more than three times smaller. Regardless, the level of standard deviation is satisfactory, because if we consider the level of relative standard error, we can see that it is around 0.8%. This is a very low level. Concerning the parameter α_1, it is underestimated for both parametric sets 1,3. Curiously enough, that aligns very well with the benchmark estimates based on maximization of L^c-MLE using the continuous-time sample. Another difference is the very close level of the standard deviation and root-mean square error for the estimates based on the discrete likelihood L^{Hes} and continuous-time likelihood L^c. The magnitude is around 0.6 for both cases and the corresponding relative standard error is around 7.62% for Set 1 and 4.12% for Set 3; both are less than the acceptable level of RSE, namely 30%.

Despite the clear underestimation of the parameter α_1, the performance of the method for the two-parametric case, confirmed by the results presented in Table 2.9, are satisfactory, the method is robust.

2.5.4. Numerical results for the three-parametric case

Table 2.10 collects the results of estimation of all three parameters for all considered parametric cases (Set 1, 2, 3, 4). The first notable point is the massive underestimation of the parameter α_1 for Set 1,3 by both L^c-based and L^{Hes}-based approaches. The tendency was strong for the two-parametric case, but with the addition of parameter α_0, it became even stronger. Both methods estimate the value of parameter α_1 as negative, almost zero (around -0.09), for Set 1. That is very far away from the real value (0.8). This is confirmed by the large root-mean square error (around 1.16). More than half of this value is because of the standard deviation; its high level tells us about the instability of the α_1-estimates. The value of standard deviation, 0.75, corresponds to the level of relative standard error around 58%. That is twice as large as the tolerable 30%-level. It is, however, important to note a relative standard error is sensitive to the mean value of the estimated parameter; when it tends to zero ,a relative standard error tends to infinity. If we compare α_1-estimates for Set 1 and Set 3, we see the levels of standard deviation are the same (around 0.75) and the biases are also the same (around 0.8). So, we see a negative effect of the parameter α_0 addition, in that it leads to stronger underestimation of the parameter α_1 for the parametric sets 1, 3. It is important to note that this is true for both L^{Hes}-based (discrete-time sample) and L^c (continuous-time sample) MLE approaches. Hence, the reason for it is not the discrete-time data sample or the proposed method. Curiously, the effect of parameter α_1 underestimation appears only for Sets 1,3. Both methods show very similar, nearly unbiased results for the parameter α_1 for Sets 2,4. Further, the level of standard deviation is a few times lower (around 0.12) for Set 4 and 0.23 for Set 2. In terms of relative standard error, it is less than 2.2%.

If consider the estimates of the other two parameters (ν and α_0), we see that both approaches give estimates that seem to be unbiased for all four parametric sets. Therefore, the root-mean square error of the α_0 estimate has the same magnitude as the standard deviation. It is around 0.093 for Sets 1,2,3 and even smaller for Set 4 (around 0.75). It holds for both approaches. The results of estimation of the parameter ν are slightly different in this sense. The method shows a three times higher level of root-mean square error and standard deviation rather than the benchmark L^c-based method. This is the same tendency we observed above for the two-parametric case.

In order to complement the picture and to track an evolution of estimates quality, the

		$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$
		L^c-MLE		
	Mean	2.99465	0.00070	-0.08717
	SD	0.08487	0.09434	0.75158
S	RMSE	0.08483	0.09410	1.16152
E	Median	2.99357	-0.00227	-0.00227
		L^{Hes}-MLE		
T	Mean	3.03825	0.00140	-0.08988
1	SD	0.30259	0.09373	0.75492
	RMSE	0.30424	0.09351	1.16574
	Median	3.03472	-0.00388	0.09092
		L^c-MLE		
	Mean	2.99559	0.22732	0.73706
	SD	0.09360	0.09012	0.23288
S	RMSE	0.09647	0.09396	0.24067
E	Median	2.98976	0.21721	0.76706
		L^{Hes}-MLE		
T	Mean	2.98444	0.22982	0.73364
2	SD	0.31037	0.09064	0.23018
	RMSE	0.30999	0.09521	0.23900
	Median	2.97876	0.21718	0.76918
		L^c-MLE		
	Mean	2.99934	-0.00649	0.46592
	SD	0.08515	0.09452	0.73582
S	RMSE	0.08494	0.09452	1.03807
E	Median	2.99655	-0.02386	0.65713
		L^{Hes}-MLE		
T	Mean	2.98792	-0.00643	0.47176
3	SD	0.30292	0.09572	0.74437
	RMSE	0.30241	0.09569	1.04003
	Median	2.98101	-0.00228	0.65787
		L^c-MLE		
	Mean	3.00420	0.21141	1.18435
	SD	0.10695	0.06936	0.12618
S	RMSE	0.10676	0.07012	0.12684
E	Median	2.99842	0.19704	1.20463
		L^{Hes}-MLE		
T	Mean	3.00166	0.21422	1.18246
4	SD	0.34203	0.07217	0.12587
	RMSE	0.34117	0.07339	0.12678
	Median	2.99526	0.20652	1.20362

Table 2.10: The results of averaged estimates based on 100 Monte Carlo simulations for all parametric sets and model version ABM-200, stabilization parameter $k = 100$.

SET 1	$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$
L^c-MLE (ABM-50)			
Mean	2.99553	-0.00023	0.60372
SD	0.08595	0.05700	0.22680
RMSE	0.08585	0.05687	0.29951
Median	2.99260	-0.00402	0.65303
L^{Hes}-MLE (ABM-50)			
Mean	3.05232	0.00140	0.60561
SD	0.30883	0.05729	0.22996
RMSE	0.31247	0.05715	0.30068
Median	3.03412	-0.00354	0.65585
L^c-MLE (ABM-100)			
Mean	2.99265	0.00080	0.41178
SD	0.09004	0.06669	0.41253
RMSE	0.09011	0.06652	0.56573
Median	2.99028	0.00117	0.49648
L^{Hes}-MLE (ABM-100)			
Mean	3.05597	-0.00649	0.40408
SD	0.29594	0.11670	0.42082
RMSE	0.30046	0.11659	0.57702
Median	3.05833	-0.00149	0.50549
L^c-MLE (ABM-150)			
Mean	2.99978	-0.00868	0.10848
SD	0.08431	0.0745287	0.62816
RMSE	0.08410	0.07485	0.93317
Median	2.99777	-0.00645	0.26891
L^{Hes}-MLE (ABM-150)			
Mean	3.03089	-0.00858	0.10210
SD	0.28978	0.07490	0.63713
RMSE	0.29070	0.07520	0.94391
Median	3.01268	-0.00819	0.27956
L^c-MLE (ABM-200)			
Mean	2.99591	-0.00768	-0.10172
SD	0.08682	0.09193	0.79574
RMSE	0.08670	0.09202	1.2013
Median	2.98557	-0.00279	0.01131
L^{Hes}-MLE (ABM-200)			
Mean	3.03781	-0.00757	-0.10816
SD	0.32592	0.09346	0.81200
RMSE	0.32730	0.09354	1.21688
Median	3.02615	-0.00263	-0.01322

Table 2.11: The results of averaged estimates of 100 Monte Carlo paths for parametric Set 1 and numbers of agents $N = 50, 100, 150, 200$. The initial point of optimizer is (4, 0.5, 1.5), stabilization parameter $k = 100$.

results of estimation of the model parameters for the most problematic Set 1 for a different number of agents, namely $N = 50,\ 100,\ 150,\ 200$, are presented in Table 2.11. Thus, it is evident that the quality of ν-estimates does not depends on the number of agents N. The values of root-mean square error and standard deviation fluctuate around 0.087 for L^c-MLE and around 0.3 for L^{Hes}-MLE. Conversely, the α_0 estimates show a weak tendency of errors growth, of approximately 30% from $N = 50$ to $N = 200$. At the same time, α_1 shows an even stronger tendency of estimation instability. The growth is from the tolerable level of root-mean square error 2.7% (for $N = 50$) to 40% (for $N = 150$). This is complemented with the growth of systematic underestimation. Note, the version of method based on the lower Hessenberg property of matrix $Q(\theta)$ with the stabilization provides a very similar quality of estimates of the continuous-time process $\mathbf{X} = (X_t)_{t=0}^{\infty}$ (see (2.1)) using only the discrete-time sample in comparison with the "benchmark"-method of maximization of likelihood L^c based on the continuous-time sample. This means that L^c-based estimates have approximately the same biases.

An analysis of the dependence of agent numbers on biases of α_1-estimates in terms of statistical power is given in Subsection 2.5.5.

2.5.5. Statistical power of estimations analysis

In order to investigate the nature of biases, we also considered the hypothesis of small statistical power of estimation procedure, because of the small sample size. We made a series of estimations based on 200 Monte Carlo experiments for a few time horizons T to see the dynamics; the results are depicted in Figure 2.17. Then we simulated 200 longer samples (with horizon $T = 1600$ instead $T = 200$ in origin) for each of the four parametric sets to determine the real performance of the method for all parametric sets. The results of estimates for each of them are collected in Table 2.12.

Recall, the number of transitions in the continuous-time path of the process increases for a more distant horizon T in the fixed model parameters (primarily, on parameter ν). However, we can generate the transitions' direction and time between transitions during simulation. Therefore, we can simulate sample paths of a length such that the overall observed time is equal to the horizon T. Further, a discrete-time path $\mathbf{y}$ is extracted from the corresponding continuous-time sample $\mathbf{x}$ by evaluation (from continuous-time sample) of the process $\mathbf{X}$ at discrete points (the discretization step is $\Delta = 1$). So, we have 401 data points in discrete-time sample $\mathbf{y}$ in the case of $T = 400$ (at time $t = 0, 1, 2, ..., 400$). This discrete-time data $\mathbf{y}$ is only used in L^d- and L^{Hes}-based MLE approaches, while L^c-based MLE only uses a continuous-time sample $\mathbf{x}$.

Now the question is: where is the border for the horizon T that separates the insufficient size of the data sample from one that is sufficient for the ABM-200 model version, for example? As we mentioned above, in order to perform corresponding analysis, we made a series of simulation experiments with different horizons T from 200 to 1600 for the ABM-200 model version. As we can see from Figure 2.17, the increase of horizon up to $T = 400$ already made a significant difference in quality of estimates; the bias of α_1 became twice as small, at - 0.405 instead of 0.908. Further, the bias of the parameter α_1 estimator gradually decreased down to 0.285 ($T = 600$), 0.225 ($T = 800$) and, finally, 0.098 ($T = 1600$). Despite this improvement, a really good quality of estimates for the parameter α_1 (the estimates are good enough for other two parameters even for $T = 200$) is achieved for a sample size of around 1600 ($T = 1600$).

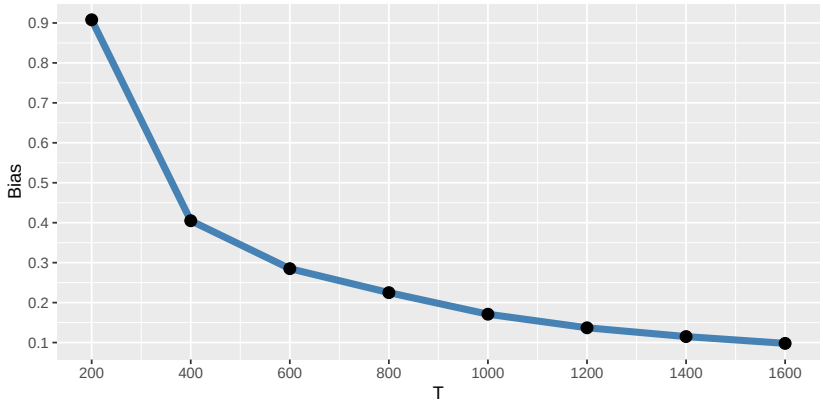


Figure 2.17: Biases for different sample sizes for parametric Set 1.

Combining the results in Subsection 2.5.4, namely the quality of estimates for models with various numbers of agents N, and the analysis in this subsection, we can conclude that the more agents, the more data needed for accurate estimation. It is reasonable, with a growth in the number of agents that the process becomes more floating because of more states it can take. As a result, it is necessary to have more data to estimate the parameters of the model. This can also explain the higher quality of L^c-MLE estimates that are based on a continuous-time sample consisting of around 1000 state changes (data points) for parametric Set 1 and $ABM-200$.

The results of estimations for the wider horizon $T = 1600$ in Table 2.12 [22] show a clear improvement of estimates and corresponding standard deviation. Compared with Table 2.10 (horizon $T = 200$), we can see that the biases of ν and α_0 estimates are still small or negligible, while the biases of α_1 estimates being significant for $T = 200$ became much smaller for $T = 1600$. For example, the bias of α_1-estimate for Set 1 (with the worst quality of estimates) is around 0.1 which is less than 10% of its absolute value. Another clear improvement is observed for values of standard deviation (SD) and root-mean square errors (RMSE), they decrease: three times for the parameter ν; around two times for α_0; seven times for Set 1; and fifteen times for Set 3. The smallest effect is observed for Set 4; that is understandable, because the best quality of estimates for $T = 200$ was obtained for this set. To summarize, the quality of estimates is absolutely satisfactory for the horizon $T = 1600$.

Now we will discuss another revealed influence on the quality of estimates: the dependence of estimates quality on the initial point of simulation. This helps to explain the especially significant difficulties for certain parametric sets (Set 1,3).

2.5.6. Dependence of estimates quality on initial point of simulation

At the moment, all simulated paths of the process $\mathbf{X}$ started in the initial point $X_0 = 0$. Now, let us go back to the analysis of the sensitivity of the model with respect to its parameters from Section 2.2. Particularly, it is important to look at plots of transition probabilities P^e_{ii+1}, P^e_{ii-1} (Figure 2.8). It is clear from them that, for each parametric sets, the process $\mathbf{X}$ has distinct points of attraction. In the case when $\alpha_0 = 0$ it is situated in zero point state

[22] In Tables 2.12 and 2.13, the results of L^c-based estimations are omitted due to their clear similarity to the L^{Hes}-based estimates.

		$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$
S	Mean	2.99788	0.00131	0.70209
E	SD	0.11328	0.01688	0.12863
T	RMSE	0.11302	0.01688	0.16140
1	Median	2.99374	0.00098	0.71709
S	Mean	2.99819	0.20873	0.78505
E	SD	0.11476	0.05347	0.09751
T	RMSE	0.11449	0.05405	0.09841
2	Median	2.9923	0.20394	0.79685
S	Mean	2.99607	0.00373	1.14047
E	SD	0.11581	0.04410	0.05393
T	RMSE	0.11558	0.04415	0.08024
3	Median	2.99737	0.00360	1.14627
S	Mean	2.99338	0.21587	1.18187
E	SD	0.10692	0.06151	0.07893
T	RMSE	0.10685	0.06338	0.08080
4	Median	2.99073	0.20977	1.18477

Table 2.12: The results of averaged estimates based on 200 Monte Carlo simulations for all parametric sets, model version $ABM-200$ and the horizon $T = 1600$.

		$\hat{\nu}$	$\hat{\alpha}_0$	$\hat{\alpha}_1$
S	Mean	2.98054	0.02479	0.73773
E	SD	0.30303	0.09334	0.19995
T	RMSE	0.30289	0.09635	0.20895
1	Median	2.97762	0.01096	0.75043
S	Mean	2.96395	0.32890	0.60430
E	SD	0.31426	0.30278	0.44972
T	RMSE	0.31554	0.32838	0.48942
2	Median	2.94704	0.32856	0.62336
S	Mean	2.99802	0.20842	0.91823
E	SD	0.32027	0.40915	0.54829
T	RMSE	0.31947	0.45827	0.61523
3	Median	3.02326	0.16079	0.90476
S	Mean	2.96286	0.65943	0.65806
E	SD	0.30795	0.75934	0.89883
T	RMSE	0.30942	0.88588	1.04764
4	Median	2.93323	0.77956	0.51360

Table 2.13: The results of averaged estimates based on 200 Monte Carlo simulations for all parametric sets, number of agents $ABM-200$, initial point of simulations $X_0 = (N-1)/N$ and the horizon $T = 200$.

(with two additional points of attraction for Set 3); it is situated far on the right (approximately at 0.75) for sets 2, 4. Then, looking at Figures 2.7 (distribution of time spent in each state), we see what it means for the process. In the case of Set 1, the process just varies around the initial point of simulation (around zero); in the case of Set 2, it also switches the attraction point to its right after variation around the attraction point (the state zero); and finally, in the case of sets 3 and 4, it makes a long migration to the attraction point on the right. All of these facts correlate well with the quality of estimates: less migration equals to worse quality of estimates. In other words, the data sample for Set 1 provides much less information.

In order to verify this hypothesis, we made another series of experiments analogical to the simulation presented in Table 2.10 with the only difference: the initial point of simulations X_0 was changed to state $N-1/N$. The results of estimation for these new simulated samples are given in Table 2.13.

There is evidence from the table that the situation with quality of estimates has changed to the opposite. The best quality of estimation is observed for Set 1,3 (at least w.r.t. biases). Note, the estimates are biased not only for α_1 (as it was for the initial point $X_0 = 0$), but also for α_0. In addition, the level of RMSE and standard deviation are significantly higher than for $X_0 = 0$. Nevertheless, this allows us to make the conclusion that the quality of estimates does not depend on the parametric set. In case of a real sample, the quality seems to be similar to Set 1 in case of $X_0 = 0$ or Set 4 in case of $X_0 = (N-1)/N$. So, in order to obtain more credible estimations, it is necessary to increase the sample size (statistical power).

It is interesting to compare our findings with the results of T. Lux [66]. The estimates calculated using the methods presented above tend to have low precision for the parameter α_1 and α_0 for certain cases. The estimates in Lux's method encounter troubles for the parameter ν and also for the parameter α_1 (but less significant). Meanwhile, the L^{Hes}-based estimates of ν have insignificant biases and stable level of RMSE and SD, at around 0.3.

□

2.6. Experiments structure

In order to make the logic of the simulation experiments clear, the structure and organization of the experiments results are outlined as a block-scheme in Figure 2.18. There are four methods that depicted as the upper four blocks in the scheme: three based on the incomplete data samples (EM algorithm, "Ld-MLE" (Matrix decomposition) and "Ld-MLE" (Recursive ODE)) and one based on complete data (denoted as "Lc-MLE"). The former method is the Maximum Likelihood Estimation (MLE) method based on estimation of synthetic complete data sample **x**. The maximization of likelihood $L^c(\theta|x)$ defined by (2.21) was used as a benchmark method for the incomplete-data counterparts. The other three methods are based on estimation of incomplete (discrete-time) data sample **y** extracted from its complete counterpart **x**. The EM algorithm iteratively maximizes a mathematical expectation $E[L^c(\theta|x)|y]$ and is described in Section 2.3. Another two incomplete data methods are the ordinary MLE methods based on likelihood $L^d(\theta|y)$ defined by (2.23). The difference between the two L^d-based methods is in the way the transition probability matrix $P^X(t;\theta)$ is calculated, namely by decomposition of infinitesimal generator $Q(\theta)$ or solving of o.d.e., and is described in Sections 2.4 and 2.5, respectively. All of the other blocks are

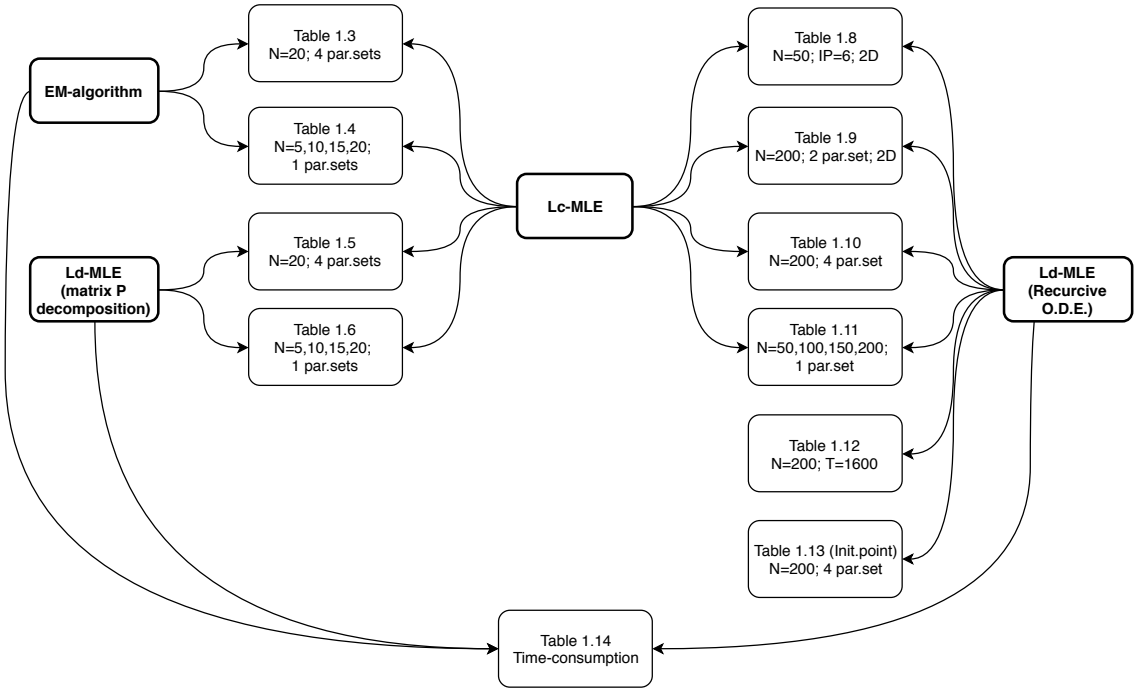


Figure 2.18: Structure of experiments and results for all three approaches.

Monte Carlo simulation experiments results presented as Tables 2.3-2.14 with computed error metrics.

Note, the EM algorithm, being an iterative procedure of maximization of L^c expectation, delivers results very similar to L^c-MLE itself. So, L^c-MLE[23] was used as a computationally faster proxy of the EM algorithm. Likewise, the methods based on the L^d likelihood maximization deliver similar results, while the L^d-MLE based on o.d.e. solving[24] is more computationally efficient. Therefore, L^c-MLE and L^d-MLE based, on lower Hessenberg matrix (o.d.e. solving), are used for comparison purposes in this research and, hence, they are the participants in most of the simulation experiments (see Figure 2.18).

2.7. Computational intensity comparison

All three approaches are based on similar constructions: they compute the transition probability matrix $P^X(\Delta;\theta)$. The crucial part of the EM algorithm and L^d-MLE approach is an eigendecomposition of intensity matrix Q. The L^{Hes}-MLE approach uses the Hessenberg property of intensity matrix Q. All three approaches are programmed as one piece of code and share the same mechanism of trajectory simulation. The L^d-based and L^{Hes}-based approaches are interchangeable on the level of matrix P^X computation. The EM algorithm and L^d-based approaches share the same procedure of matrix P^X computation[25]. This allows us to compare all three approaches directly in terms of time-consumption.

[23] Complete-data is usually unavailable in real world examples.

[24] This L^d-MLE version is significantly faster and more robust compared to the other one, as is shown later.

[25] In our application of the EM algorithm the same procedure for P^X computation as in case L^{Hes} based on Hessenberg property could also be used, but further analysis shows it to be meaningless.

N	EM (sec)	L^d (sec)	L^{Hes} (sec)
5	0.245378	0.07901	0.25547
10	5.8642	0.257703	0.425559
15	81.0639	0.709209	0.672825
20	358.404	1.73141	0.898927
25	1124.78	8.03567	1.13752
50	-	-	4.06312
100	-	-	14.4249
150	-	-	30.2157
200	-	-	58.983
250	-	-	135.98
300	-	-	195.215
350	-	-	294.828

Table 2.14: An average (across 200 simulations) time-consumption for one model estimation for the parametric Set 1: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 0.8$.

In order to make a time-consumption[26] analysis, measurements of computation time during simulation and estimation experiments were made for all three approaches. In particular, measured time was necessary to estimate the 200 simulated paths[27] for each approach, which was then averaged (divided on 200), again for each. All of the results are collected in Table 2.14. The time-consumption is visualized in Figure 2.19.

The first point of interest from Table 2.14 is that the EM algorithm is much more computationally expensive compared to the other two. In the case of $ABM-20$, it takes around 10 minutes for a single estimation, while for $N = 25$ it already takes half an hour. Nevertheless, this is a satisfactory time in practice. Further growth of N also rests on limitation of the eigendecomposition, which suffers from the catastrophic cancellation described in Section 2.4.4. The solution could be in using computation of P^X based on the Hessenberg property, but it appears more reasonable, at least with regards to computation time, to use a "pure" L^{Hes}-based approach. Comparing the performance of the L^d-based and L^{Hes}-based approaches we have a duality. The first one is more computationally efficient for a small number of agents N (for $N \leq 15$), while the second method is more efficient for large N, especially considering the instability of eigendecomposition for large values of N. In fact, the L^{Hes}-based method not only allows us to produce estimation for large N, but it also does so quite quickly: it takes on average less than a minute to estimate the model parameters for N. When N becomes greater than 200, computational time begins grow rapidly: it takes around 10 minutes to estimate the model $ABM-300$[28].

[26] All of the measurements are based on Windows API functions *QueryPerformanceCounter* and *QueryPerformanceFrequency. PC specifications: Intel i5-3210M CPU, 6Gb RAM.* allowing us to make precision measurements with accuracy to microseconds.

[27] For the parametric Set 1: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 0.8$.

[28] Recall, it means we have to work with 601-by-601 matrices Q and P^X.

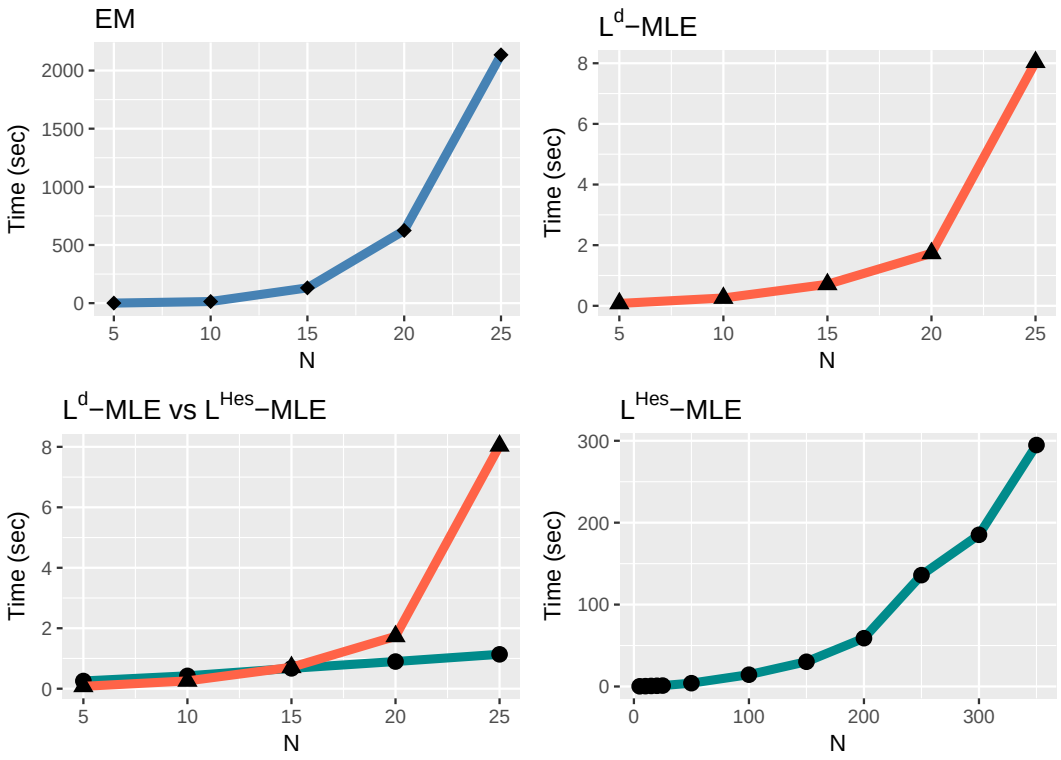


Figure 2.19: Visualization of average (across 200 simulations) time-consumption for one model estimation (values are in seconds). Horizontal line is number of agents N; vertical line is time in seconds. The parametric Set 1 was used: $\nu = 3$, $\alpha_0 = 0$, $\alpha_1 = 0.8$.

In order to visualize the dynamics of time-consumption, plots are presented in Figure 2.19. All three approaches show clear exponential growth of computation time, but the fast growth area shifts depending on the method. Thus, the critical point for the EM algorithm is $N = 10$, while it is $N = 25$ and $N = 100$ for L^d-based and L^{Hes} methods, respectively. These differences allow us to estimate the model in hundreds times faster in the case of the L^d-based method and in thousands times faster for the L^{Hes}-based case with respect to the EM algorithm. The difference between the L^d-MLE and L^{Hes}-MLE approaches is not so dramatic, but also significant. The L^d-MLE method is up to three times faster than the L^{Hes}-MLE for small N, but it is up to ten times slower for N around 25.

□

2.8. Real data estimation

The business sentiment is closely related to such lagged indicators as GDP growth. In contrast to GDP, sentiments indices allow us to estimate business conditions and the state of the economy more frequently. This section is dedicated to the estimation of real sentiment indexes using the Agent-Based model and its versions considered in the previous sections.

2.8.1. Business confidence, economic and consumer sentiment indexes

There are numerous sentiment-based indicators in the fields of financial markets, economics, business and consumer behavior. The most famous indexes are established in the US and EU:

- *Measure of CEO Confidence*™ (US) – This index is based on a survey of around 100 Chief Executives each month and is conducted by Conference Board[29], which is an independent business membership and research association working in the public interest.
- Moody's Analytics Survey of Business Confidence (US) – This is based on surveying middle and senior-level managers around the globe weekly since late 2002. It is published by Economy.com.
- YPO Global Pulse (US) – This economic sentiment indicator is based on surveys of CEOs around the globe (on a quarterly basis) regarding the economic state, sales, employment and investment activity.
- Michigan Consumer Sentiment Index (US) – A telephone survey of consumer expectations regarding overall economy conducted by the University of Michigan.
- NY Empire State Index (US) – Based on (monthly) surveys of manufacturers in New York State conducted by the Federal Reserve Bank of New York.
- Business confidence index (OECD) – Based on enterprises' current position and expectations of production, orders and stocks.

[29]The dataset with historical values of index is accessible on ChiefExecutive.net and is covered in each issue of Chief Executive magazine on a fee basis.

- Consumer confidence index (OECD) – Similar to MCSI.
- Ifo Business Climate Index (Germany) – An indicator of economic developments in Germany based on a monthly survey of 7,000 participants in the fields of manufacturing, construction, wholesaling and retailing. It has been published on a monthly basis by the Ifo Institute for Economic Research (Munich) since 1972.
- ZEW Financial Market Survey (Germany) – Introduced in 1991, it is conducted each month in order to collect expectations regarding the development of international financial markets. Participants are German analysts working in financial and industrial sectors. Overall, up to 350 participants are questioned.

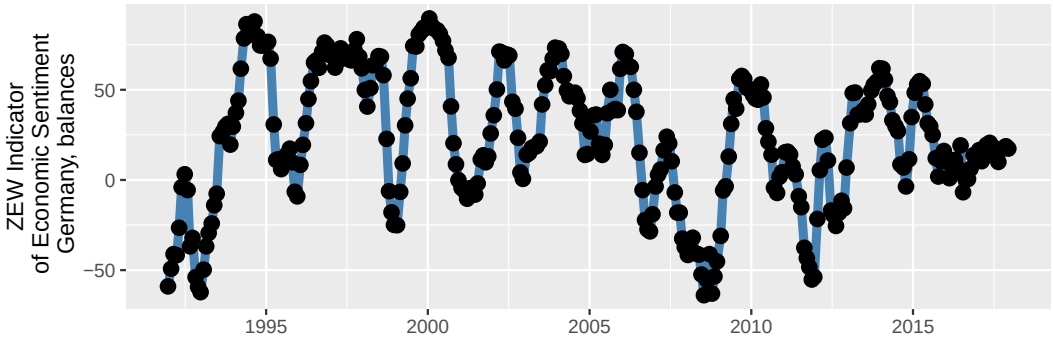


Figure 2.20: ZEW Indicator of Economic Sentiment Germany, balances.

Despite there being a number of indexes, it is not easy to test the proposed ABM model on real data. The methodology of most of the indexes publishers is not disclosed or only partially disclosed. Even the number of participants is unknown for most cases, which is a very important model-defining constant. Another obstacle is too wide a number of participants, which is especially typical for consumer confidence indexes like MCSI and CSI. Many indexes are based on rare surveys in to coincide with a short history. So, in this research we are focused on the ZEW index as well as T. Lux [66] in order to obtain comparable results of two alternative approaches of the ABM model estimation. Another reason is that the ZEW index has a known and limited (up 350) number of participants (agents). Additionally, ZEW has a long historical data series that is freely accessible, as depicted in Figure 2.20.

As stated in the description on the website of the Centre for European Economic Research (ZEW in German abbreviation), the ZEW index is constructed as the balance $s = p - n$ of percentage share p of the survey respondents with positive expectation regarding economic state and the share of negative counterparts n. Therefore, the ZEW index values belong to intervals from -100 to 100. This definition is slightly different to the one given in Section 2.1, but they are in fact equivalent, because $n_+(t)/N$ and $n_-(t)/N$ are shares of optimistic and pessimistic agents.

The input data for the methods considered above is a time-series of discrete-time observations of sentiment index $\{y_t\}_{t=0}^{T}$ supposed to take values in the state space Ω with $2 \times N + 1$ states

$$\Omega = \left\{-1, -\frac{(N-1)}{N}, \ldots, 0, \ldots \frac{(N-1)}{N}, 1\right\},$$

which is defined by a model-defining assumption about the number of agents N. So, the first necessary transformation is to normalize the ZEW index values y_t by division by 100 in order to have them in the interval $[-1,1]$

$$\bar{y}_t = \frac{y_t}{100} \in [-1,1].$$

The second step is to set the correspondence of the normalized values to the states from the state space Ω. Namely, the normalized values $\bar{y}_t$ are multiplied by the number of agents N and then rounded to the closest integer number from the set $-N,\ldots,N$, then divided again by N

$$\hat{y}_t = \frac{\text{round}(\bar{y}_t \times N)}{N} = \frac{\text{round}\big(y_t \times N/100\big)}{N}.$$

Further, the matrix c of transitions between states in the real data sample $\{\hat{y}_t\}_{t=0}^{T}$ is constructed, which is a necessary component for construction of the discrete-time likelihood function $L^d(\theta|\hat{y})$ from Section 2.1.5 and Theorem 3.

2.8.2. ZEW index estimation procedure and results

The first attempts to estimate the ZEW index data sample $\{\hat{y}_t\}_{t=0}^{T}$ with the MLE method based on the Hessenberg property described in Section 2.5 highlighted the inoperability of the method's implementation without additional modification in assumption of the large number of agents N, in particular for N larger than 100, while the C++ code developed for the method worked fine for N up to 350 in the case of simulated data. Deeper instigation using the plots of the likelihood function $L^d(\theta|\hat{y})$ surface constructed for the transformed real data sample $\hat{\mathbf{y}}$ and the corresponding transition matrix c with respect to the parameters ν and α_1 (α_0 fixed to zero), see Figure 2.21, shows that, with an assumption of a larger number of agents N, the "ridge" with the highest likelihood values move in the direction of higher ν. As a result, the estimated values of ν go up to 20, which in turn leads to large absolute values of intensity rates q_{ij}. This behavior coincides with two facts mentioned in Section 2.2: 1. A larger number of agents N means lower time between transitions; and 2. Figure 2.5 shows that transitions occur more often for higher ν. So, if we have a fixed data sample with fixed time between transitions, then an assumption of a larger number of agents should be compensated by higher ν, and that is exactly what occurred.

There is also another dependency, which is the transition probability $P^X(\Delta t)$ being the solution of Kolmogorov equations is equal to the matrix exponential $exp(\Delta t Q)$, where Δt is the interval between successive observations, Q is the intensity rates matrix with the elements of the form $\nu f(x;\alpha_0,\alpha_1)$, and therefore, the transition probabilities (due to Taylor series expansion) are functions depending on the product $\nu\Delta t$. This fact leads to a close relationship between Δt and ν, namely that higher Δt leads to lower ν and vica versa. During the estimation of real data we assumed $\Delta t = 30$, which corresponds to one month (in days) between observations.

In the sense of computational mathematics and programming, the problem nested in the o.d.e. solver used for solving the Kolmogorov system ordinary differential equations (2.13) from Theorem 1, in particular in its precision. In order to overcome an instability of the method's implementation, it was necessary to switch from the classical Runge-Kutta method (RK4, see [24]), to a higher order Runge-Kutta-Fehlberg method (RKF78, see [24]).

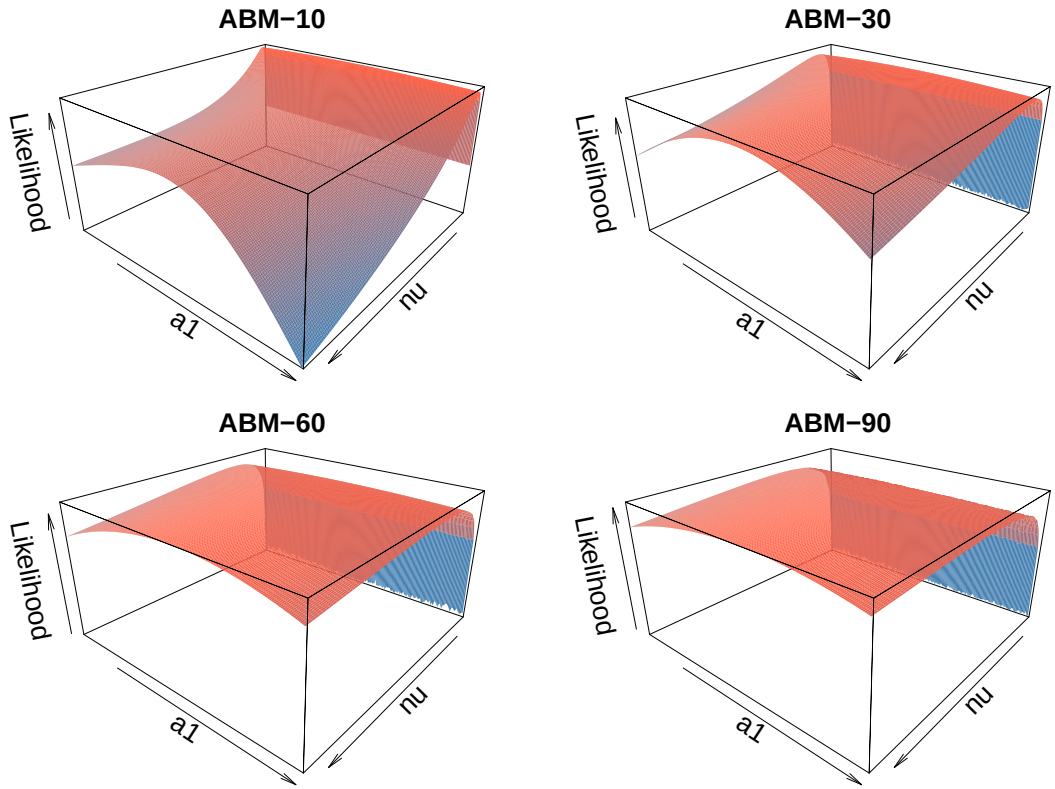


Figure 2.21: Likelihood function surface plot for ZEW index data with boundaries for $\nu = (0,30)$, $\alpha_1 = (0,2)$ and α_0 fixed to zero.

Perhaps, multiprecision o.d.e. solvers should provide even more robust solution comparing with floating point methods like RK4 and RKF78. However, the price of this improvement can be significantly higher time-consumption.

In order to control the robustness of the estimation results, the procedure was repeated for various numbers of assumed agents N. Namely, the ZEW index data was estimated firstly with the ABM-15 model version, which is characterized as the most robust, then with ABM-25, -50, -75, -100, -150, -200, -250, -300 and, finally, ABM-350. Next, the estimation procedure was repeated for each N with various initial points $\theta_0 = (\nu, \alpha_0, \alpha_1)$ of the optimization subroutine, which is the crucial part of MLE method: (2, 0.5, 1.5), (4, 0.5, 1.5), (7.5, 0.2, 0.2), (8.5, 0.2, 0.2), (12, 0.2, 0.2). This methodology should ensure continuity and convergence of results, plus exclude failures of the method's implementation.

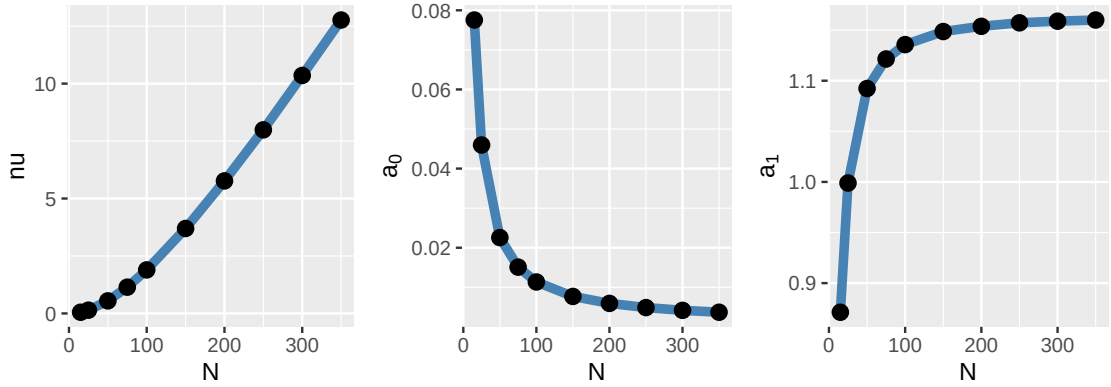


Figure 2.22: Estimation of the ZEW index by various versions of the ABM model. N is the number of agents in the ABM model version.

The results of estimation using the methodology described above are visualized in Figure 2.22 and collected in Table 2.15. An analysis of both of them convey a clear and smooth convergence of α_0 and α_1 estimates when assumption about the number of agents goes from the minimal number $N = 15$ to the real one $N = 350$, namely $\alpha_0 = 0.004$ and $\alpha_1 = 1.16$. In contrast to the estimates of α_0 and α_1, the estimates of ν display its scale nature of ν and are characterized by roughly linear dependence on the number of agents assumption. In order to gain an impression of the quality of the obtained estimates of the ZEW index, the standard deviation and bias are also estimated (see $SD^\star$ and $Bias^\star$ in Table 2.15). These metrics are obtained by Monte Carlo simulation of data paths with the parameters $\nu^\star, \alpha_0^\star, \alpha_1^\star$ estimated for the ZEW index and corresponding N. For example, $Bias^\star$ is a difference between the median of estimates of ν obtained for 50 simulated data paths in assumption $\nu = \nu^\star$ and $\nu^\star$ itself.

The results of estimation can be compared with the estimates from T. Lux's paper [66]. Recall, the results that can be compared are:

- the case of estimated number of agents ($N = 42$): $\nu = 0.15$ ($SE = 0.07$), $\alpha_0 = 0.09$ ($SE = 0.06$), $\alpha_1 = 0.99$ ($SE = 0.14$);
- the case of $N = 350$: $\nu = 0.78$ ($SE = 0.06$), $\alpha_0 = 0.01$ ($SE = 0.01$), $\alpha_1 = 1.19$ ($SE = 0.01$).

The ZEW index was also estimated using the ABM-42 model, in order to have comparable estimates. The results of the estimation are 0.39 (ABM-42) versus 0.15 (Lux) for ν, 0.03

N	$\nu^\star$	$SD^\star$	$Bias^\star$	$\alpha_0^\star$	$SD^\star$	$Bias^\star$	$\alpha_1^\star$	$SD^\star$	$Bias^\star$
15	0.052	0.004	0.002	0.077	0.061	0.024	0.871	0.123	-0.032
25	0.143	0.012	0.001	0.046	0.043	0.015	0.999	0.078	-0.043
50	0.547	0.053	-0.002	0.023	0.025	0.010	1.092	0.040	-0.029
75	1.144	0.103	0.005	0.015	0.018	0.011	1.121	0.032	-0.017
100	1.895	0.147	0.025	0.012	0.021	0.009	1.136	0.036	-0.017
150	3.698	0.354	0.017	0.008	0.024	0.011	1.149	0.032	-0.023
200	5.768	0.610	0.231	0.006	0.022	0.009	1.154	0.027	-0.025
250	7.989	0.965	0.087	0.005	0.084	0.008	1.157	0.030	-0.023
300	10.322	2.754	0.116	0.004	0.035	0.003	1.159	0.039	-0.025
350	12.768	1.071	-0.157	0.004	0.019	0.002	1.160	0.024	-0.020

Table 2.15: ZEW index data estimation.

(ABM-42) versus 0.09 (Lux) for α_0, and 1.07 (ABM-42) versus 0.99 (Lux) for α_1. Taking into account standard errors of estimates, only the estimates of ν are significantly different. In the case of the ABM-350 model version, the estimates of α_0 and α_1 are even more similar, namely 0.01 (Lux) and 0.004 (ABM-350), 1.19 (Lux) and 1.16 (ABM-350), respectively, while the estimates of ν are completely different again. The fact of different estimates of ν seems to be caused by a different assumption of discretization step Δt. In this research, Δt is assumed to be equal to 30 for monthly data, while T. Lux assumed it to be equal to unity.

3

Asymmetric Markov-Switching Multifrequency Models

A huge number of asset return models are based on the assumption of discrete time. The most popular are the ARCH-class models pioneered by Engle (1982). The original model was aimed at describing non-constant volatility stylized fact, in particular, volatility clustering. The models from this class are successfully used for forecasting a volatility, especially GARCH (Bollerslev 1986), which has the Heston model as its continuous-time limit in case of GARCH(1,1) specification. Its generalized versions are able to describe such a feature as the leverage effect (EGARCH, Threshold GARCH, GJR-GARCH, QGARCH), strong persistence with hyperbolic decay of influence – FIGARCH (Balie, Bollerslev, Mikkelsen 1996). Similarly to the continuous-time case, discrete-time models can be divided into classes of affine and non-affine models. Thus, the first result allowing implementation of ARCH-models for option price belongs to Duan (1995), who developed an option pricing methodology based on Monte Carlo simulations, to establish the option pricing theoretical basis and methodology for non-affine models. Heston and Nandi (2000) achieved another significant result: they developed an affine GARCH model that allowed them to obtain a semi-closed form expression of European call option price. Generally speaking, affine models (see Appendix A.7) have worse abilities in modeling stylized facts, being more restrictive than non-affine counterparts. At the same time, affine models are more computationally efficient and convenient. Both authors use Local Risk-Neutral Valuation Relationship (LRNVR) measures. Further efforts were targeted towards generalization of LRNVR (Duan 1999; Christoffersen et al. 2010) and alternative risk-neutral pricing approaches, such as the extended Girsanov principle (Elliot and Madan 1998) and the conditional Escher transformation (Christoffersen et al. 2009). Other discrete-time models, for example, Dallores et al. (2006), Khrapov and Renault (2014) using the CAR model, were considered.

3.1. Models and Features

In this section we describe the theoretical structure of the model, its properties and features. In particular, in Subsection 3.1.2 we describe model construction and in Subsec-

tion 3.1.3 model features.

3.1.1. An overview of discrete-time models development

First of all, let us given a filtered probability space $(\Omega, \mathscr{F}, \{\mathscr{F}_t\}_{t=0}^T, P)$ with a natural filtration $\{\mathscr{F}_t\}_{t=0}^T$, induced by the discrete-time stochastic process of asset prices, $\mathbf{S} = \{S_t\}_{t=0}^T$, satisfying the *usual conditions*[1]. Then, a general discrete-time model of asset returns is formulated as follows

$$r_t = \log\left(\frac{S_t}{S_{t-1}}\right) = \mu_t + \sigma_t \epsilon_t, \tag{3.1}$$

where $\mathbf{S} = \{S_t\}_{t=0}^{\infty}$ is an asset price process on the regular grid $t = 0, 1, 2, \ldots, \infty$, μ_t – trend, σ_t – volatility, ϵ_t are i.i.d.. In the simplest case (discrete-time equivalent of the Black-Scholes model), trend and volatility are assumed to be constant, $\mu_t \equiv \mu$ and $\sigma_t \equiv \sigma$, while $\epsilon \in N(0,1)$ or another non-Gaussian distribution (for example, Student-t or Generalized Error Distribution). More general discrete time models were developed in two main directions: deterministic volatility models, namely, autoregressive conditionally heteroskedastic (ARCH) class of models introduced by Engle [35], and stochastic volatility models. The models in ARCH-class are able to describe many stylized facts of returns and volatility dynamics, including its clustering.

GARCH(p,q) Bollerslev [15] is the most popular in the ARCH-family given by

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{q} \beta_i \sigma_{t-i}^2 \tag{3.2}$$

As mentioned above, F. Black in 1976 gave the first economical explanation of the phenomena in terms of financial leverage of companies: a decrease of equity price leads to an increase of equity-debt ratio (leverage), which in turn leads to an increase of uncertainty about the steadiness of the company. Today, the effect of asymmetrical correlation – a volatility increase in the case of negative returns is no longer linked to the financial leverage. Nevertheless, the effect is significant and incorporation of it leads to better explanation of volatility smiles. A few examples of asymmetric models in the ARCH-family are given below[2].

AGARCH (Asymmetric) of Engle (1990)

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i (\epsilon_{t-i} - \rho)^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \tag{3.3}$$

EGARCH (Exponential) of Nelson [76] also points to the model leverage effect. It is formulated using logarithms

$$\log \sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i g(z_{t-i}) + \sum_{i=1}^{q} \beta_i \log \sigma_{t-i}^2 \tag{3.4}$$

[1] A filtration $(\mathscr{F}_t)_{t>0}$ is considered to satisfy the usual conditions if it is right-continuous and complete (for more details see [57]).

[2] A comprehensive review of the ARCH-class models is given in [16].

where function g is constructed to produce asymmetric response on asset price movements, namely $g(z_t) = \theta z_t + \gamma(|z_t| - E|z_t|)$, z_t are Gaussian or non-Gaussian innovations sequence.

NGARCH (Non-linear asymmetric) of Engle and Ng [34] aimed at the modeling leverage effect

$$\sigma_t^2 = \omega + \alpha(\epsilon_{t-1} - \rho\sigma_{t-1})^2 + \beta\sigma_{t-1}^2, \tag{3.5}$$

where $\alpha,\ \beta \geq 0;\ \omega > 0$.

GJR-GARCH (Glosten-Jagannathan-Runkle, [40]) is another model with an asymmetry. The model is

$$\sigma_t^2 = \omega + \delta\sigma_{t-1}^2 + \alpha y_{t-1}^2 + \phi y_{t-1}^2 I_{\epsilon_{t-1}<0} \tag{3.6}$$

where ϵ_t is i.i.d., $y_t = \epsilon_t\sigma_t$, I is indicator function (of negativity).

Another important stylized fact is a volatility persistence, which is tightly connected with volatility clustering.

FIGARCH (Fractionally Integrated) is a generalization of GARCH by Baillie, Bollerslev and Mikkelsen (1996) allowing for the description of persistence (long memory). It defines volatility as

$$\phi(L)(1-L)^d\epsilon_t^2 = \omega + (1-\beta(L))\nu_t, \tag{3.7}$$

where $\nu_t \equiv \epsilon_t^2 - \sigma_t^2$, $0 < d < 1$, the equations $\phi(z) = 0,\ \beta(z) = 1$ have all the roots in a unit circle, L is a lag operator.

As we can see from the above, the volatility is $\mathscr{F}_{t-1}$-measured in all ARCH-class models, which means that the volatility tomorrow is not stochastic in the class, it is assumed to be known. In contrast, the volatility process in the Stochastic Volatility (SV) family of models is $\mathscr{F}_t$-measured, as it is evident from its name, and driven by its own process.

The ASV (asymmetric) model is the simplest SV model with asymmetry in discrete-time. It is defined as

$$\log\sigma_t^2 = \omega + \alpha u_t + \beta\log\sigma_{t-1}^2 \tag{3.8}$$

where u_t is i.i.d. innovation sequence with $\mathrm{corr}(\epsilon_t, u_{t-1}) = \rho$. The very last assumption defines leverage effect in the model directly.

The LMSV (long-memory) model of Breidt , Crato and de Lima (1998) is devoted to a persistence in volatility. The log-volatility process is given by

$$\begin{aligned} \sigma_t &= \sigma\exp(\nu_t/2) \\ \phi(L)(1-L)^d(\nu_t - \mu) &= \theta(L)\eta_t, \end{aligned} \tag{3.9}$$

where $\{\eta_t\}$ is i.i.d. $\mathscr{N}(0,\sigma_\eta^2)$, $\{\nu_t\}$ is ARFIMA process independent of $\{\epsilon_t\}$. Hautsch [46] describes additional stochastic volatility models with discrete time.

The stochastic volatility models are less restrictive than the ARCH-family, but an option pricing task is more convenient to solve with continuous-time versions of SV models, while Duan [29] and Heston & Nandi [47] have developed methodology for the ARCH family, which can be (and will be done for the AMSM model later) generalized for non-stochastic volatility models. There is another methodology, but it uses the same pricing formula obtained by Kallsen and Taqqu [56] for an arbitrage-free continuous-time case.

3.1.2. Asymmetric Markov Switching Multifrequency Models

In this work I consider models that are more sophisticated than ARCH-class models, but also non-stochastic volatility[3] models. Namely, I use the Markov Switching Multifractal (MSM) model created by Calvet and Fisher [19]. It can reproduce many of the stylized facts simultaneously (in contrast to the ARCH-family), as we will see further, but not the leverage effect. So, the purpose of the generalization is to create and to calibrate an asymmetric version of the MSM model, which is reflected in the name of the model – Asymmetric MSM.

Let us begin with the MSM model first. An asset price returns process is defined on the filtered probability space $(\Omega, \mathscr{F}, \{\mathscr{F}_t\}_{t=0}^T, P)$ with the natural filtration $\{\mathscr{F}_t\}_{t=0}^T$, induced by the discrete-time stochastic process of asset prices, $\mathbf{S} = \{S_t\}_{t=0}^T$, satisfying the *usual conditions*[4]. As in the previous models, the asset returns are given by 3.1, namely

$$r_t = \log\left(\frac{S_t}{S_{t-1}}\right) = \mu_t + \sigma_t \epsilon_t,$$

but the key difference compared to the ARCH-class models is in the construction of the volatility process $\sigma = \{\sigma(M_t)\}_{t=0}^{\infty}$. It is defined as

$$\sigma_t = \sigma_0 \left(\prod_{i=1}^{\hat{k}} M_{i,t}\right)^{\frac{1}{2}}, \tag{3.10}$$

or in terms of logarithmic volatility

$$\log\sigma_t^2 = \log\sigma_0^2 + \sum_{i=1}^{\hat{k}} \log M_{i,t} \tag{3.11}$$

where $\mathbf{M} = \{M_t\}_{t=0}^{\infty}$, $M_t = (M_{1,t}, \ldots, M_{\hat{k},t})$ is a random-vector of volatility components at time t switching with different frequencies, each $M_{k,t}$ is given by

$$M_{k,t} = \begin{cases} m_0, & \text{if } u_{k,t-1} \in [0, \gamma_k/2), \\ 2 - m_0, & \text{if } u_{k,t-1} \in [\gamma_k/2, \gamma_k), \\ M_{k,t-1}, & \text{if } u_{k,t-1} \in [\gamma_k, 1), \end{cases} \tag{3.12}$$

where $\{u_{k,t-1}\}$ are uniform i.i.d.. this formulation is equivalent to Calvet and Fisher's definition

$$M_{k,t} = \begin{cases} M, & \text{with probability } \gamma_k, \\ M_{k,t-1}, & \text{with probability } 1 - \gamma_k \end{cases} \tag{3.13}$$

$$M = \begin{cases} m_0, & \text{with probability } 1/2, \\ 2 - m_0, & \text{with probability } 1/2, \end{cases} \tag{3.14}$$

with the following properties for both cases

- $\hat{k}$ is number of frequencies,

[3] It can also be formulated as the stochastic volatility model.

[4] A filtration $(\mathscr{F}_t)_{t>0}$ is considered to satisfy the usual conditions if it is right-continuous and complete (for more details see [57]).

- $M_t \geq 0$, $E[M_{k,t}] = 1$,
- $M_{1,t} \perp M_{2,t} \perp \ldots M_{\hat{k},t}$ (statistically independent), as a result σ_0 is the unconditional standard deviation of log-returns.

The difference – albeit a crucial one – between Calvet and Fisher's formulation and the one considered in this research is that $M_{k,t}$ is assumed to be $\mathscr{F}_t$-measurable for (3.13), while $M_{k,t} \equiv f(u_{k,t-1})$ is assumed to be $\mathscr{F}_{t-1}$-measurable in (3.12) and further. This means $M_{k,t}$ is assumed to be known at the moment of time t by Calvet and Fisher's formulation and at $t-1$ the current research. The former fact makes option pricing using the methodology of Duan [29, 28] possible.

The authors of the MSM model, Calvet and Fisher, described the economical sense behind $M_{k,t}$ in their paper [19] in the following way:

> "Thus, the lowest frequencies might correspond to business cycles and technological shocks, while other frequencies could correspond to earnings cycles or short-lived liquidity shocks. This closely captures the economic intuition that different types of volatility shocks have different degrees of persistence." (Journal of Econometrics 105, 2001, p.40)

The transition probabilities $\gamma = (\gamma_1, \ldots, \gamma_{\hat{k}})$ for all $\hat{k}$ heterogeneous frequencies in MSM are specified as

$$\gamma_k = 1 - (1 - \gamma_1)^{b^{k-1}},\ \gamma_1 \in (0,1),\ b \in (1,\infty),\ k = 1 \ldots \hat{k}. \tag{3.15}$$

So, the MSM model is defined by four parameters ($\hat{k}$ is the matter of model specification), $\theta = (b, \gamma_{\hat{k}}, m_0, \sigma_0)$.

The original definition of γ_k above having two parameters b and $\gamma_{\hat{k}}$ seems to be excessive. The values of b and $\gamma_{\hat{k}}$ from practice and prior researches are known to be approximately 3 and 0.95, correspondingly, in the case of $\hat{k}$ from 6 to 8. In order to make the model suitable for calibration, we could simply fix these two parameters. An alternative approach is to mimic the shape of γ_k distribution (3.15) by using non-parametric expression. In particular, this approach has been used in Lux [84]. Namely, the following approximation

$$\gamma_k^\star = 2^{k-\hat{k}}. \tag{3.16}$$

The similarity of γ_k and $\gamma_k^\star$ is depicted in Figure 3.1.

Further, I consider two approaches of asymmetry incorporation.

Asymmetric MSM model, variant 1

A.E.Leövey [59] suggested the first approach. The distribution of $M_{k,t-1}$ is assumed to be a binomial distribution with two states $(m_0, 2-m_0)$, but its distribution now depends on ϵ_{t-1} innovation

$$M_{k,t} = \begin{cases} m_0, & \text{if } u_{k,t-1} \in \big[0, \gamma_k(1-\Phi(\rho\epsilon_{t-1}))\big), \\ 2-m_0, & \text{if } u_{k,t-1} \in \big[\gamma_k(1-\Phi(\rho\epsilon_{t-1})), \gamma_k\big), \\ M_{k,t-1}, & \text{if } u_{k,t-1} \in \big[\gamma_k, 1\big), \end{cases} \tag{3.17}$$

where $\Phi(\cdot)$ is a cumulative distribution function of standard normal variable, while the new parameter ρ controls an asymmetry, $m_0 \in (1,2)$. Thus, the AMSM1 model parameters vector is $\theta = (m_0, \sigma_0, b, \gamma_{\hat{k}}, \rho)$; in the simplified case, it is just $\theta = (m_0, \sigma_0, \rho)$.

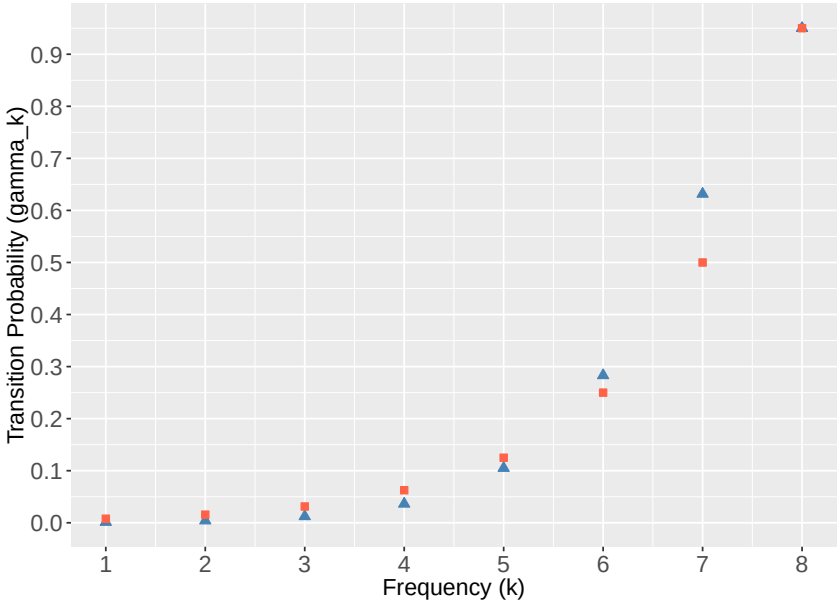


Figure 3.1: The horizontal axis is the frequency k, the vertical one is a transition probability, the triangles are γ_k, while the squares are $\gamma_k^\star$ transition probabilities, $\gamma_{\hat{k}} = \gamma_{\hat{k}}^\star = 0.95$.

Another version of the MSM model generalization is more straightforward and supposed to be more suitable for calibration. Developed by this author, its necessity is explained in detail in Section 3.4.

Asymmetric MSM model, variant 2

This variant of generalization suggests the use of an ordinary binomial distribution with equally distributed states m_0 and $2 - m_0$ for $M_{k,t-1}$, while the construction of volatility is changed as follows

$$\sigma(m_0, \sigma_0, \rho)_t = (\rho\epsilon_{t-1} - \sqrt{\sigma_0})^2 \left(\prod_{i=1}^{\hat{k}} M_{i,t} \right)^{\frac{1}{2}}, \tag{3.18}$$

or alternatively

$$\sigma(m_0, \sigma_0, \rho)_t = |\rho\epsilon_{t-1} - \sigma_0| \left(\prod_{i=1}^{\hat{k}} M_{i,t} \right)^{\frac{1}{2}}. \tag{3.19}$$

The constructions $(\rho\epsilon_{t-1} - \sqrt{\sigma_0})^2$ and $|\rho\epsilon_{t-1} - \sigma_0|$ produce an asymmetric response on a negative innovation ϵ_{t-1}.

Hereafter the first generalization variant with transition probabilities defined by (3.15) will be referred as *AMSM1 model* variant. The second variant with the construction of volatility (3.18) parameterized with $\theta = (m_0, \sigma_0, \rho)$ and the transition probabilities defined by (3.16) will be referred to as *AMSM2 model*.

Note, the models of new construction are generalizations of the ordinary MSM model; it is enough to take $\rho = 0$. Moreover, if we also take m_0 equal to 1, we obtain ordinary random walk.

□

Further, we will discuss the ability of the AMSM model (both variants) to mimic stylized facts of financial data: revealed statistical characteristics (distribution, correlation) of asset prices, its returns and standard deviation (volatility). Mandelbrot (1963, [67]) and Fama (1965) were the first to conduct notable investigations into financial data properties. The stylized facts can be divided into two tightly linked groups: stylized facts of returns (asset prices) and stylized facts of its volatility. Note, despite the stochastic nature of volatility and corresponding stylized facts being revealed in 1960s, they were ignored in modeling for a long time until Engle's ARCH model wasn't invented. The AMSM models are constructed differently to the ARCH-class models, being an alternative to them. As we will see further, this allows us to model a wider set of stylized facts of financial data, in both returns and volatility. Primarily, it is focused on volatility persistence and the leverage effect.

3.1.3. Modeling of stylized facts in returns

Historical data samples of real assets prices often have trends, periods of quietness and high volatility: rare shock events are often negative. As a result, their log-returns significantly deviate from normality in a statistical sense, which is an assumption of classical models. This fact leads to the following stylized facts of real log-returns compared to the normal ones: leptokurtic distribution (positive excess kurtosis), skewness (positive third moment) and substantial outliers. The stylized facts are closely related to each other and can be described, as real stock returns distribution is more heavy-tailed, peaked and asymmetric.

As we can see in Figure 3.2, MSM-family models allow us to generate trajectories with rare substantial outliers, switching trends and regimes. Namely, the AMSM paths on this figure have at least four clear regimes: $(0,250)$ – moderate volatility, *sideways trend* and few weak negative outliers; $(250,350)$ – high volatility, strong upward trend; $(350,450)$ is characterized by very low volatility with no outlier and moderate upward trend; $(450,520)$ – moderate volatility, downward trend and three significant negative outliers.

Another witness of the MSM family ability to mimic real market data is the histogram of the AMSM model log-returns together with DAX index (symbol GDAXI) log-returns[5] depicted in Figure 3.3 with fitted normal p.d.f., vertical lines as the mean and $\pm 2/4\sigma$ (SD). It clear that, for greater m_0 and ρ, the distribution of returns deviate from normal to distribution with heavier tails, more peaked and showing up few outliers. Note that leverage parameter ρ also contributes in non-normality, which makes the model more flexible and allows it to capture more significant deviations. The DAX index returns histogram[6] in the right bottom corner of Figure 3.3 is depicted as an example of stock returns non-normality. As we can see in the case $(m_0, \rho) = (1.4, 0.01)$, AMSM returns are quite similar to DAX return's distribution shape.

The third and the fourth moments, known as *skewness* and *kurtosis*[7], are more precise measure of deviation from normality. In order to find out the dependence between parameters m_0 and ρ on skewness/kurtosis, a simulation experiment was performed: 1000 sample paths with 2048-points of AMSM1 and AMSM2 model returns[8] for each $m_0 \in (1.0, 1.05,$

[5] https://finance.yahoo.com/quote/%5EGDAXI/

[6] AMSM and DAX share returns scaled to have the same standard deviation.

[7] Here we assume a kurtosis as the fourth moment centered by kurtosis of standard normal r.v., namely it is subtracted by 3

[8] The simulations were conduct on AWS cloud service, in particular, I used RStudio server AMI created and

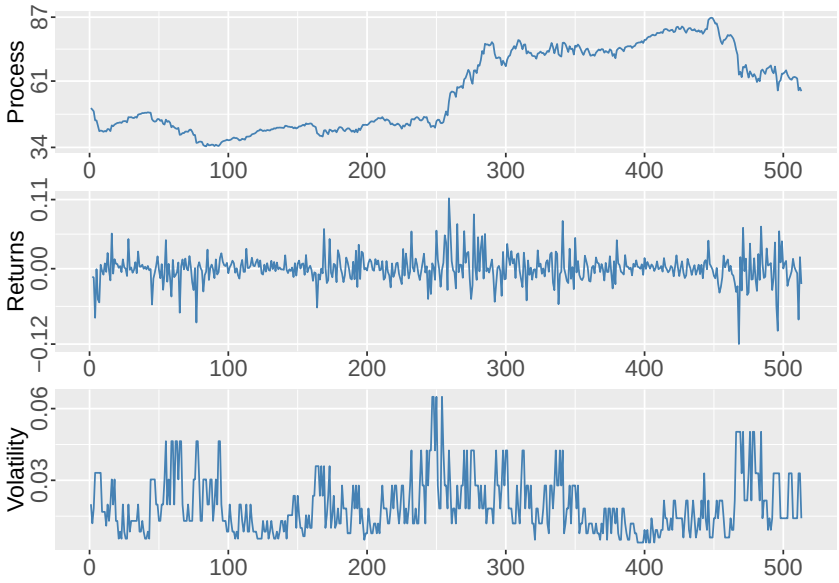


Figure 3.2: The graphical evidence of volatility clustering for the AMSM1 process. Namely, AMSM1 process sample path, its returns and the volatility process $\{\sigma(m_0, \sigma_0, \rho)_t\}$ for the parameters $\theta = (1.4, 0.02, 3, 0.95, 2)$.

$\ldots, 1.8)$ and ρ fixed to 0; $\rho \in (0.0, \ldots, 4.0)$ (AMSM1) and $\rho \in (0.0, \ldots, 0.065, 0.07)$ (AMSM2), while m_0 fixed to 1, $\sigma_0 = 0.02$, $\hat{k} = 6$ in all specifications. Then skewness and kurtosis of returns for each of 1000 sample paths were calculated and averaged for all parameters m_0/ρ values combinations. As we can see in Figure 3.4, there is clear and strong parabolic dependence of both, m_0 and ρ, on kurtosis for the AMSM2 model, but it is concave in the case of AMSM1 for ρ, while it is convex in the case of AMSM2. The skewness of returns distribution is close to zero (as predicted by the theory) with significant deviations for $\rho > 0.04$ and AMSM2, and also for $m_0 > 1.4$ and both model variants.

To summarize, the general term *fat/heavy tails* is used to refer to deviations from normality, namely leptokurtic distribution and substantial outliers. It has been shown above that the AMSM model is able to reproduce all these effects, while non-zero skewness (asymmetry of returns distribution), is not observed in the case of standard Gaussian innovations $\{\epsilon_t\}$. Using skewed ones easily solves this; for example, Hanssen (1994) uses Student t-distribution of innovations for the GARCH model. Another popular alternative is Generalized Error Distribution (GED).

3.1.4. Modeling of stylized facts in volatility

As we can see from the construction of the model, in particularly of volatility process, it is defined as a product of $\hat{k}$ elements switching with its own frequencies. The highest frequency corresponds to $k = \hat{k}$ and the lowest to $k = 1$. This is consistent with a theory of the existence of multiple scales in volatility: short-run scale corresponds to inter-day volatility, market shocks are long-run volatility and so on. This makes it possible to mimic clustering, long-range dependence, mean reversion of volatility, as well as the original MSM

maintained by Louis Aslett (www.LouisAslett.com).

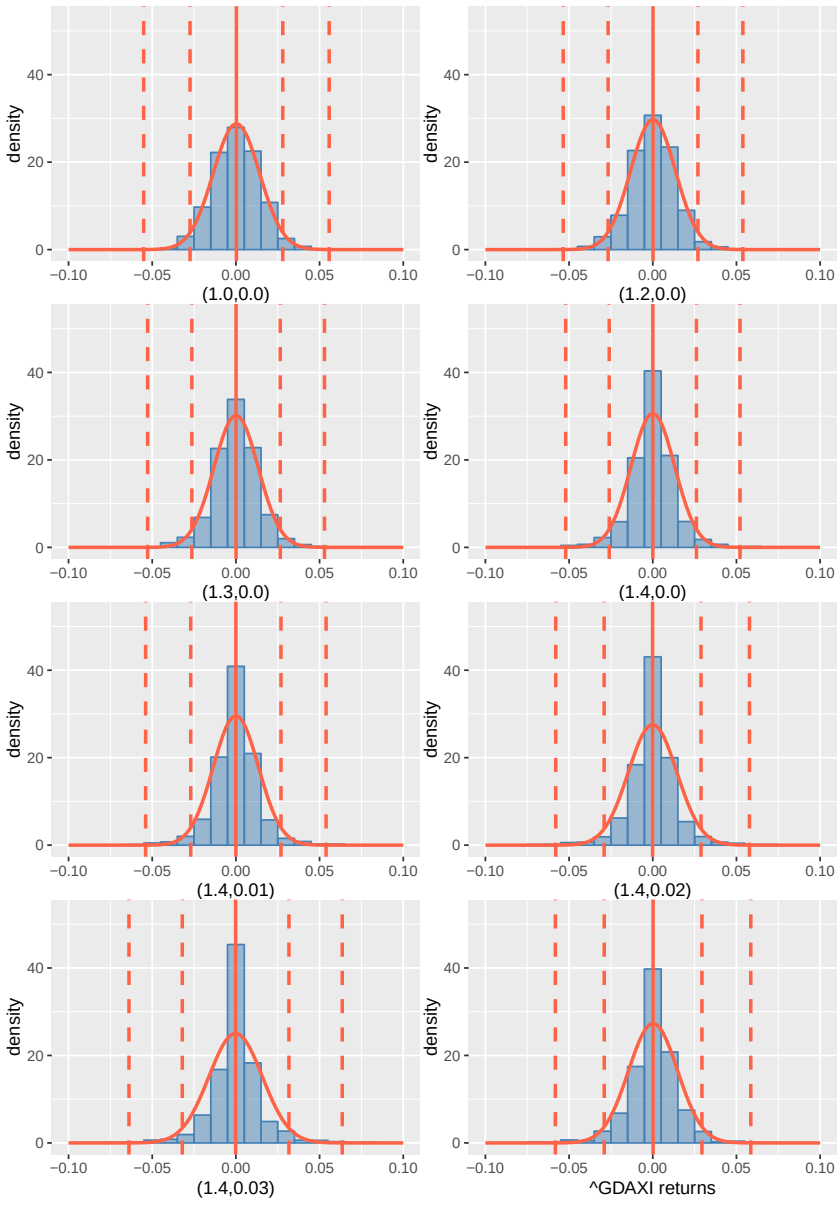


Figure 3.3: The graphical evidence of non-normality AMSM2 process returns in the case of (m_0, ρ) deviate from 1 and 0, correspondingly. The dashed lines are $\pm 2/4$ standard deviation.

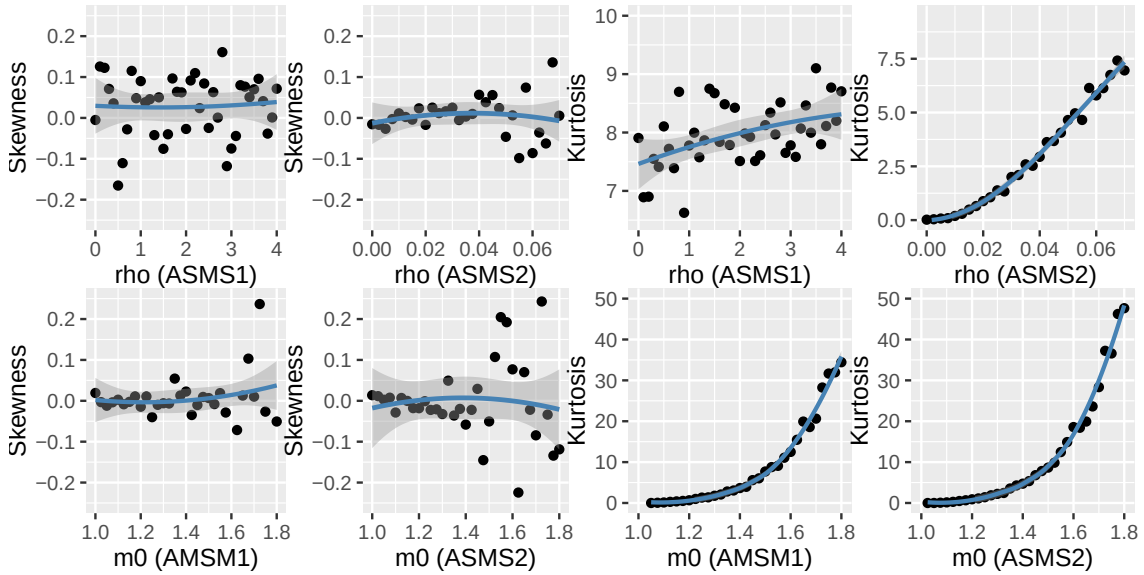


Figure 3.4: The dependence m_0 and ρ parameters on skewness and kurtosis. The points are averaged skewness and kurtosis of simulated AMSM2 returns for various parameters m_0 and ρ values. The line on the plots is either a polynomial regression or LOESS (local) regression. The shaded areas are 95% confidence bands.

model, and leverage effect – the main innovation of the AMSM models[9]. These effects and the ability of the model to reproduce them are discussed in the next subsections. All the abilities of the AMSM models are collected in Table 3.2 at the end of the subsection, as well as the abilities of other wide-spread discrete-time and continuous-times models.

The key results (mean reversion property, long memory and leverage effect) are formulated as theorems with proofs given in Appendices.

Let us start from the volatility clustering stylized fact as the most general phenomena.

Volatility clustering

The volatility clustering effect is a widely known empirical feature of stock prices data. It is described as the existence of high volatility and low volatility periods in financial data: after a large (small) return day, the next day is likely to have large (small) returns. The clustering effect was found in empirical studies, see Engle and Russell (1998). Looking ahead, we note that the AMSM models are also able to reproduce it; see[10] Figure 3.2.

A presence of clustering in the real data leads to the natural assumption of the existence of an autocorrelation structure in volatility. Due to the fact that the volatility is not a directly observed quantity, the absolute or squared returns are used as an approximation, thus

$$\gamma_n^{abs} = \mathrm{corr}(|r_t|, |r_{t+n}|), \tag{3.20}$$

$$\gamma_n^{sqr} = \mathrm{corr}(r_t^2, r_{t+n}^2). \tag{3.21}$$

The significance of the correlation coefficients γ_n^{abs} and γ_n^{sqr} is confirmed empirically and documented in many previous studies on the topic. Moreover, it was found that these cor-

[9]Cont (2001) gives an extensive and well-defined systematization explanation of the stylized facts of asset returns and their volatility.

[10]Using R package *quantmod*.

relations are persistent for long intervals of time and decay with power law speed. This effect is known as *long memory* and discussed in detail further in Subsection 3.1.4.

Another dependence structure revealed empirically for volatility is a *leverage effect*. It connects the volatility to returns. The following correlation describing a leverage effect

$$\gamma_n^{lev} = \text{corr}(|r_t|, r_{t-n}), \tag{3.22}$$

was found to be significant and negative. It means, that negative returns (decrease of underlying stock) leads to increase of volatility and, conversely, positive returns lead to decrease of volatility.

Note, the construction of (A)MSM models incorporates volatility clustering naturally. Moreover, they are able to generate volatility clustering on different time scales, which is consistent with the existence of business cycles of different frequency. Let us consider this topic more deeply.

Mean reversion of volatility

The third, slightly more technical, dependence structure is a *mean reversion of volatility*. Let us start with a definition.

Definition 8. *A mean reversion property of stochastic process is a tendency to revert to its historical mean value over time.*[11,12]

In the related literature, two kinds of mean reversion are considered: a mean reversion in prices/returns and mean reversion in volatility.

In the case of mean reversion in prices/returns, as an economical explanation, it is usually assumed that under-priced and over-priced goods/securities exist on the real market. The price of them is expected to have a tendency to increase or decrease, correspondingly. This is considered to be the market force reverting prices to their mean. Various authors have investigated this phenomenon with contradictory outcomes for different markets and horizons. So, there is no common view on its existence.

The mean reversion in volatility is closely linked to persistence in volatility (long memory) and regime switching (volatility clustering). Namely, the existence of significant correlation with lagged volatility values decaying slowly with a lag is interpreted as the tendency of perturbations (regime characterized by high volatility) damping. In other words, an influence of a high volatility period is slow decay in the sense of autocorrelation and a process that tends back to its mean values (regime characterized by low volatility).

The AMSM models possess a mean reversion property, as well as the original MSM model. The idea is that a model should either have a mechanism for damping of perturbation or should have an inner force that should push a process to its mean, for it to really have the mean reversion property. In particular, we assume "washing out" of perturbation at the current time in the course of time. This kind of mean reversion is observed in ARCH-class models. In the proposition below, we prove that the AMSM models (as well as the original MSM model) have such a mechanism.

[11] A broad description of mean reversion modeling is given in Hillebrand's dissertation [48]. In particular, the author distinguishes between three types of mean reversion modeling approaches.

[12] This property is in contrast with random walk behavior, which has no memory about the past.

Theorem 7. *The AMSM1 and AMSM2 models exhibit mean reversion of volatility property as well as ordinary MSM models.*[13]

Proof. The proof is given in Appendices A.8,A.9. □

The next and the most important stylized fact in the context of this research is a leverage effect.

Leverage effect

In the literature, as a leverage effect is defined as a negative correlation of volatility and lagged returns. The effect owes its name to the economical interpretation given by Black in 1976 (see [11], also [21]), linking it with financial leverage. To be specific: a lower value of asset price leads to higher debt/equity ratio. This in turn causes an increase in the volatility of asset price. Meantime, a rise of asset price also affects the volatility, but is weaker. In turn, it interferes with the *asymmetric*[14] nature of the phenomena (see [34], [76]). Nevertheless, some authors, for example Figlewski and Wang (2000) question this interpretation being prevalent in the literature. In particular, they mention that financial leverage cannot explain amplitude of volatility and that any other change of leverage – except for asset price change (because of debt changes or shares issue/repurchase) – does not lead to substantial change of volatility.

Despite the disagreements about the economical causes of the leverage effect in the literature, the presence of negative correlation between volatility and lagged returns is commonplace. Moreover, option prices calculated without taking into account this fact can lead to significant biases (see, [51]). Hence, the effect should be incorporated in models if possible. As we mentioned above, the modifications of the original MSM model introduced previously are aimed at embodying the leverage effect phenomenon.

First of all, in order to show that the AMSM model (both versions) can also reproduce the leverage effect, let us prove the weak stationarity property of the volatility process. It will provide us with some results necessary in the future.

Theorem 8. *Let a volatility process* $(\sigma_t)_{t=0}^{\infty}$ *of AMSM1/AMSM2 model version is defined by (3.17/3.18),* $\{\epsilon_t\}_{t=0}^{\infty}$ *is a standard white noise with distribution* $\mathcal{N}(0,1)$*, then*

$$cov(\sigma_t, \epsilon_{t-1}) < 0,$$

for both models and any $t > 0$, $\rho > 0$.

Proof. Due to complexity and length, the proof is placed in Appendices A.10/A.11. □

In order to find out the sensitivity of AMSM versions with respect to parameter ρ, which is responsible for leverage effect and possibility to model different levels of correlation (leverage effect) statistical tests were performed on the correlation of the volatilities (σ_t) and lagged shocks (ϵ_{t-1})[15]. Namely, fifty paths of length 2^{14} points each were simulated

[13] Note, the AMSM2 model has the same kind of mean reversion as ARCH-class models.

[14] This asymmetry gave the name to the models presented in this work – *Asymmetric* MSM models, namely the expressions (3.17) and (3.49), which create asymmetric responses on negative and positive lagged returns.

[15] The R software environment was used for this purpose.

ρ	Correlation	P-value	Confidence interval
		AMSM1	
0.001	-0.0019	0.4941	(-0.07488, 0.0710)
0.01	-0.0053	0.4910	(-0.0782, 0.0677)
0.1	-0.06950	0.1744	(-0.1417, 0.0035)
0.25	-0.1651	0.0013	(-0.2353, -0.0933)
0.5	-0.2988	0.0000	(-0.3638, -0.2308)
0.75	-0.3877	0.0000	(-0.4480, -0.3239)
1.0	-0.4446	0.0000	(-0.5013, -0.3841)
2.0	-0.5269	0.0000	(-0.5776, -0.4720)
3.0	-0.5410	0.0000	(-0.5907, -0.4873)
5.0	-0.5481	0.0000	(-0.5972, -0.4949)
		AMSM2	
0.0001	-0.0001	0.4963	(-0.0731, 0.0728)
0.0005	-0.0147	0.4615	(-0.0876,0.0583)
0.001	-0.0297	0.4051	(-0.1025, 0.0433)
0.005	-0.1445	0.0059	(-0.2152, -0.0723)
0.01	-0.2774	0.0000	(-0.3434, -0.2086)
0.02	-0.4858	0.0000	(-0.5397, -0.4280)
0.03	-0.6128	0.0000	(-0.6564, -0.5651)
0.04	-0.6866	0.0000	(-0.7233, -0.6460)
0.05	-0.7273	0.0000	(-0.7599, -0.6910)
0.075	-0.7599	0.0000	(-0.7892, -0.7273)

Table 3.1: Leverage-effect statistical testing for AMSM models. Null hypothesis is a zero correlation of volatility σ_t and lagged shock ϵ_{t-1}. The significance level is 5%.

with $\theta^1 = (m_0, \sigma_0, b, \gamma_{\hat{k}}, \rho, \hat{k}) = (1.4, 0.02, 3, 0.95, \rho, 8)$ and $\theta^2 = (m_0, \sigma_0, \rho, \hat{k}) = (1.4, 0.02, \rho, 8)$ model for different values of ρ for both specifications of model. The results of the testing are collected in Table 3.1. The simulations showed that, in the case of the parameters settings defined above, the approximate significance of correlation is achieved for $\rho > 0.1$ for the AMSM1 model and $\rho > 0.001$ for the AMSM2 model. This knowledge will help us later during the model calibration procedure. In addition, it gives us clues about the possible amplitude of ρ.

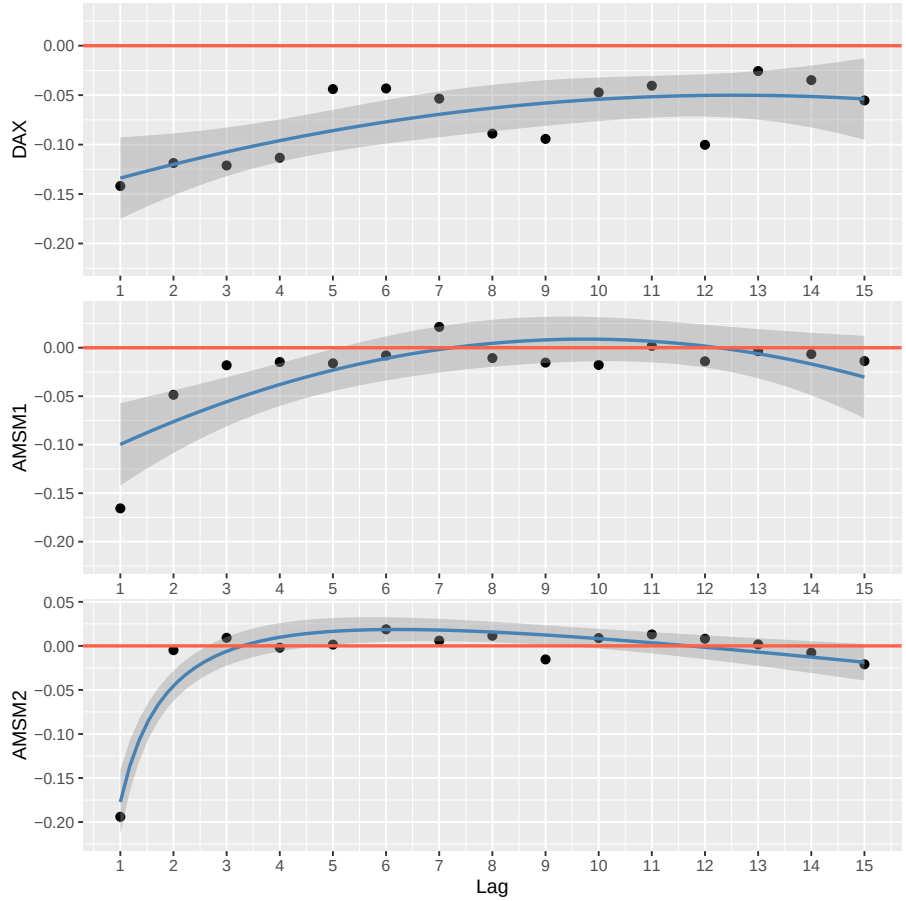


Figure 3.5: The time structure of leverage effect for the DAX index, AMSM1 ($\rho = 0.58$) and AMSM2 ($\rho = 0.38$) simulated samples paths, measured as defined in (3.22).

More evidence of the leverage effect is presented in Figure 3.5. Namely, the estimates of the correlation of absolute returns with lagged returns γ_n^{lev} (black dots) in: historical DAX index path from the 1st January 2005 to 1st January 2018 on the top plot; the simulated AMSM1 process path of length 2^{14} with $\theta^1 = (1.4, 0.02, 3, 0.95, 0.6, 8)$ on the central plot; and the simulated AMSM2 process path of length 2^{14} with $\theta^2 = (1.4, 0.02, 3, 0.95, 0.38, 8)$ on the bottom plot. Further, the blue line on all plots is a polynomial regression of γ_n^{lev} estimates, the shaded areas are 95% confidence bands and the red line is zero-correlation level.

Note, the AMSM2 model returns are not correlated already at the lag equal to 2, in contrast to the AMSM1 model, where correlation $\gamma_n^{lev} = \text{corr}(|r_t|, r_{t-n})$ tends to zero slightly more smoothly (at lag $n \geq 3$. In the case of real data, the correlation is even more persistent. This is the price for simplification and computability, which, on the other hand, leaves room for further improvements . For example, we can use AR(p) process in volatility

construction instead of AR(1) used currently. This should make persistence stronger.

Long memory

The long-range dependency is observed in financial data. Taking into account that absolute returns and squared returns are approximations for volatility, the long memory phenomena was revealed by many authors. Taylor (1986) [95] was the first who showed evidence of the long memory property of financial data series. He investigated the absolute values of stock returns and revealed that they have slowly decaying autocorrelation functions. In addition, the long-term volatility persistence was found in powers of absolute returns $|r_k|^n$ and also in squared returns by Zhuanxin, Granger and Engle (1993) [100]. Granger, Spear and Ding (2000) investigated properties of absolute daily returns for a few markets. Let us give the definition of the phenomenon.

Definition 9. *A weakly-stationary process $\boldsymbol{X} = \{X_t\}_{t=0}^{\infty}$ is said to have a long memory*[16] *(often, long-range dependent), if its autocorrelation $\gamma_k \equiv corr(X_t, X_{t-k})$ (or returns, powers of returns, absolute values) for some $\alpha \in (0,1)$ has hyperbolic decay, or equivalently power law decay, which in functional form is*

$$\gamma_k^X \sim c_{\alpha,\gamma} k^{-\alpha}, \text{ as } k \to \infty, \tag{3.23}$$

while if the process correlation satisfies

$$\gamma_k^X \sim d_{\alpha,\gamma} \alpha^k, \text{ as } k \to \infty, \tag{3.24}$$

it is said to have a short memory (exponential decay).

The long memory property is closely related to a self-similarity property, but none of these properties can guarantee another one.

Definition 10. *A process $\boldsymbol{Y} = \{Y_t\}_{t=0}^{\infty}$ is said to be a self-similar, if for any realization of it and $c > 0$ holds the following relationship by distribution*

$$Y_{ct} \stackrel{D}{=} c^H Y_t,$$

where $H \in (0,1)$ is called the Hurst exponent.

One of junctions for self-similarity and long memory properties is an autocorrelation function. It is given (if it exists) for an increments of self-similar process, $Z_t = Y_t - Y_{t-n}$, as (see Beran [8])

$$\gamma_k^Z = H(2H-1)k^{2H-2} = c_H k^{2H-2}. \tag{3.25}$$

So, there could be a suggested intersection of self-similarity and long memory, through (3.23) and (3.25), as $H = 1 - \alpha/2$. Note, the self-similar processes itself (**Y**) is not stationary; only its increment process (**Z**) can be stationarity. Moreover, it has to be, at least,

[16] A comprehensive insight about these kinds of processes is provided by Beran [8], [9].

weak stationary in order to have the long memory property as determined in the definition above. Hence, the increment process **Z** has long memory property when $H \in (1/2, 1)$; it is either memoryless or has a *short memory* when $H = 1/2$ and it is an *anti-persistent process* when $H \in (0, 1/2)$. The most well-known example of the self-similar process with the long memory property of increments process is Fractional Brownian Motion (FBM) with its increment process called Fractional Gaussian Noise (FGN), which we use further as a trial process.

Returning to our models, the first evidence of a presence of long memory can be given by an inspection of autocorrelation function. In the case of the AMSM1 process, it is given in Figure 3.6 (there is the small jitter of the surface due to estimation errors during simulation). In the theoretical sense, Calvet and Fisher [20] proved that an ordinary MSM process has hyperbolic decay of autocovariance of absolute returns powers. Further, it has an important additional feature: the rate of decay differs for various powers, which coincides with the empirical behavior of real financial time series. In addition, we will show that the AMSM1 process enjoys the same features theoretically and testify to the strength of the effect by simulations.

Before we formulate and prove the long memory property of AMSM1/AMSM2 models, we require the following auxiliary assertion.

Lemma 3. *The following expression holds for the components $M_{k,t}$ of AMSM1 and AMSM2 model versions*

$$E[\sigma_t] = \hat{\sigma}\mu^{\hat{k}}, \tag{3.26}$$

$$cov(\sigma_t, \sigma_{t+\tau}) = \hat{\sigma}^2\mu^{2\hat{k}}\left(\prod_{k=1}^{\hat{k}}\left(1 + a_1(1-\gamma_k)^\tau\right) - 1\right), \tag{3.27}$$

where $\tau > 0$ and

$$\mu = E\left[M^{1/2}\right]$$
$$\hat{\sigma} = \begin{cases} \sigma_0 & (AMSM1) \\ \sigma_0 + \rho^2 & (AMSM2) \end{cases}$$
$$a_1 = \frac{1}{\mu^2} - 1.$$

Besides,

$$E\left[M_{k,t}^{q/2} M_{k,t+\tau}^{q/2}\right] = \left[E\left(M^{q/2}\right)\right]^2\left(1 + a_q(1-\gamma_k)^\tau\right), \tag{3.28}$$

where

$$a_q = \frac{E(M^q)}{E(M^{q/2})^2} - 1, \tag{3.29}$$

M - is a binary variable taking values m_0 and $(2 - m_0)$ equiprobably in the case of AMSM2 model and with the probabilities $(1 - CDF(\rho\epsilon_{t-1}))/CDF(\rho\epsilon_{t-1})$ in the case of AMSM1.

Proof. The proof is given in Appendix A.12. □

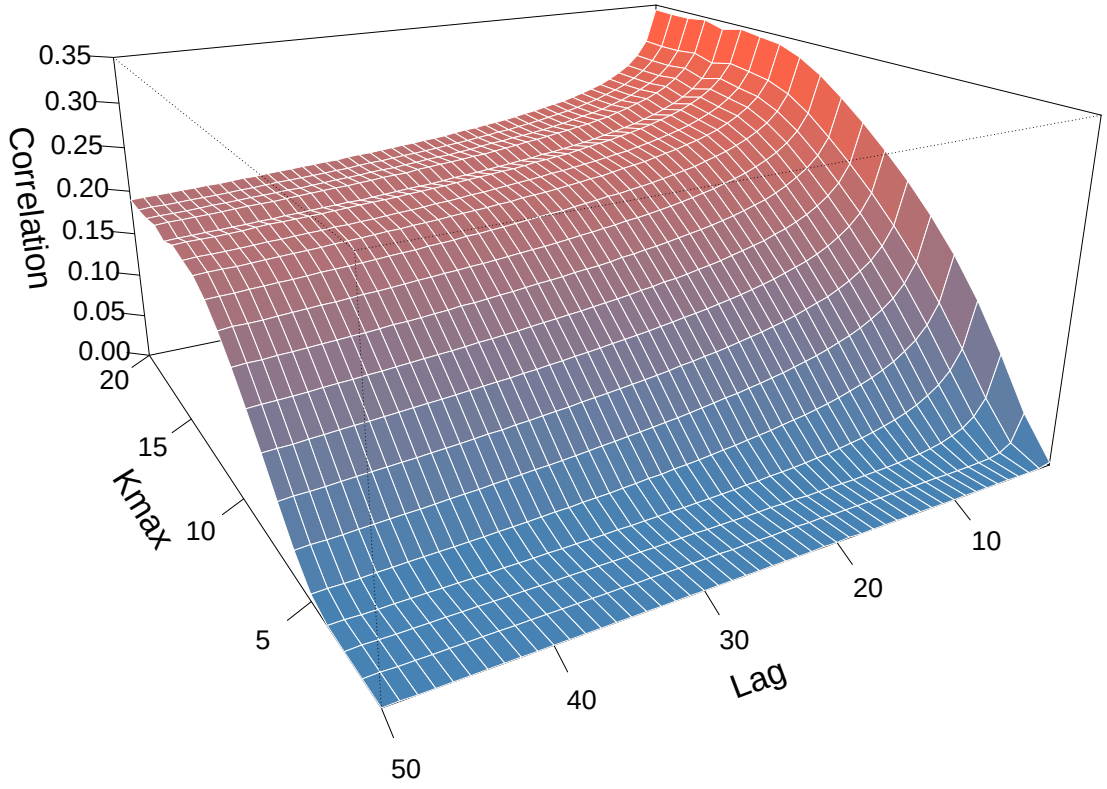


Figure 3.6: The graphical evidence of long memory for the AMSM1 process absolute returns. Namely, 3D-profile of ACF w.r.t. $\hat{k}$ (*Kmax* on the plot) for the parameters $\theta = (1.4, 0.02, 3, 0.95, 2)$.

The main theoretical result formulated in the theorem below is closely aligned with the results for the ordinary MSM model in [20], but the proof requires further explanations due to differences in the models.

Theorem 9. *Let $\psi_q(n) = corr(|r_t|^q, |r_{t+n}|^q)$, then the following holds*

$$\sup_{n \in I_{\hat{k}}} \left| \frac{ln(\psi_q(n))}{ln(n^{-\delta(q)})} - 1 \right| \to 0, \tag{3.30}$$

on the interval $I_{\hat{k}} = \{n : \ \alpha_1 \log_b(b^{\bar{k}}) \le \log_b(n) \le \alpha_2 \log_b(b^{\bar{k}});\ \alpha_1, \alpha_2 \in (0,1)\}$, where $\{r_t\} = \{\epsilon_t \sigma_t\}$ are returns of the AMSM1/AMSM2 process,

$$\delta(q) = \log_b(E(M^q)/E(M^{q/2})^2),\ q \ge 0. \tag{3.31}$$

Proof. The only difference of the AMSM1 model from the ordinary MSM model is the construction of distribution M. Hence, the expectations $E(M^q)$, $E(M^{q/2})$ are different from their counterparts in the case of ordinary MSM, and they also depend on the parameter ρ, but they are still constants for any fixed ρ. Besides, $E(M_{k,t}^{q/2} M_{k,t+n}^{q/2})$ are defined by the same expression as for the ordinary MSM, as we have proven in Lemma 3. So, there are no changes in the major steps of the proof in [20], only minor ones.

The proof of theorem in the case of AMSM2 model is given in Appendix A.13. □

Theorem 9 establishes the existence of the long memory in volatility for the AMSM models, but we would like to estimate how strong it is in terms of the Hurst exponent for different parameters of the model

$$\theta = (m_0, \sigma_0, b, \gamma_{\hat{k}}, \rho) = (1.4, 0.02, 3, 0.95, 2)$$

and their neighborhoods. In particular, we are interested in an investigation of the model defining constant $\hat{k}$ influence and the strength of the long memory for various powers of absolute returns.

The problem is that the Hurst exponent, being a clear mathematical object, provides the non-trivial task of estimation of its value in the practical sense. Hurst proposed the very first method in 1951 [52], namely the rescaled range analysis (also known as R/S analysis). Later, two alternatives were suggested: Detrended Fluctuation Analysis [78], which is developed to be less sensitive to a violation of stationarity; and the approach of Geweke and Porter-Hudak [38], which is based on a periodogram regression. All three approaches are based on construction of a linear regression model, where H acts as slope parameters of it. This, evidently, leads to the low precision of these methods. Later on, the modified R/S by Lo [65] was aimed at distinguishing a short-range and a long-range dependence. In this sense, Lo's method could be especially useful in our case, but it suffers from the same issues as the original R/S. Further, it requires the selection of the truncation lag (in order to exclude a short memory effect). In addition, it was criticized for biases towards rejection of long-dependence (see [96]). Another group of regression-based methods in the time domain were suggested in the 1990s, namely Absolute Values of the Aggregated Series, Aggregated

Variance, Differencing the Variance (see, Taqqu and Teverovsky [94, 93]) and Peng's estimator [78]. In the frequency domain, a method was developed based on a regression of periodogram on frequency, namely the Periodogram method, Boxed Periodogram [94], Reisen's periodogram method [82] and the Geweke and Porter-Hudak method. The Whittle estimator [93, 8, 9] is also based on a periodogram, but it is not a graphical (regression) method. The second non-graphical estimator is based on a wavelet analysis discussed in [2]: it is an unbiased semi-parametric efficient estimator. Further, there is a Haslett-Raftery estimator [45], which is an approximation of the maximum likelihood estimator for the *fractional difference parameter d* of ARFIMA model (there is an expression that holds $H = 0.5 + d$)[17].

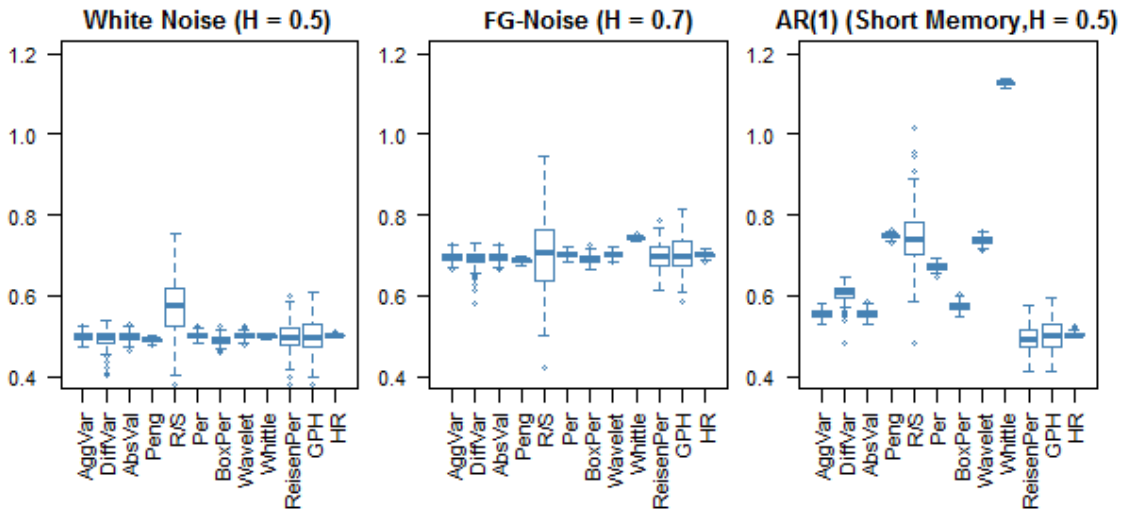


Figure 3.7: The Hurst exponent estimates of 200 paths simulated for each trial process - Gaussian white noise, fractional Gaussian noise and AR(1) process.

Before to estimate the value of the Hurst exponent for the absolute returns process with different value of $\hat{k}$ and q, we tested the performance of the twelve different approaches discussed above [18] on a few toy examples. The trial processes are: an ordinary Gaussian noise (white noise) with $H = 0.5$; a fractional Gaussian noise (fractional Gaussian noise) with $H = 0.7$ and an *stationary* autoregressive process AR(1) with the short memory and $H = 0.5$. The aim is to find out the sample properties of the estimators, as well as their a robustness to the short memory (ability to distinguish the long and the short memory properties). For this purpose 200 Monte Carlo simulations of paths for each trial process of $2^{64} = 65536$ length[19] were conducted. The estimates are presented using the boxplots in Figure 3.7.

For the next experiment 50 paths of length 2^{15} for each process type were simulated. Namely, the AMSM1 process paths are simulated with the parameters $m_0 = 1.4$, $\sigma_0 = 0.02$, $\rho = 2$, $b = 3$, $\gamma_{\hat{k}} = 0.95$. Then, each sample paths has been estimated with each estimator and the dispersion of the estimates is visualized as the boxplots.

[17] Fractional Integrated processes are a class of processes aimed at modeling a long range dependence by integrating a stationary process with a fractional number of repeats, namely d-times, where d - is called a *fractional difference parameter*. In fact, GPH, Reisen's estimator and Whittle's estimator also estimate d.

[18] R packages *fArma*, *fractal* (Whittle's estimator), *fracdiff* (Reisen's estimator, GPH) have been used.

[19] An estimation of Hurst exponent typically needs long samples in order to catch the long memory effect.

As we can see, for the white noise and the fractional Gaussian noise, most estimators show good performance in the sense of an unbiasedness and a standard deviation. The main exception is the classical R/S analysis; it shows significant bias in the case of white noise and the largest standard deviation among all estimators in all three examples. Whittle's estimators has bias in the case of fractional Gaussian noise. All other estimators seem to be unbiased and have small or moderate (Reisen's and GPH) deviation. The picture is dramatically different in the case of AR(1) stationary process[20]. Nine estimators fail to reject persistence; the only three estimators are unbiased and provide correct estimates – Reisens's periodogram, GPH and Haslett-Raftery estimator. The last one shows the best performance being based on a maximum likelihood approximation and designed to estimate AR-class processes. The robustness to the presence of short memory was discussed by Lo in 1991. He even suggested a modified version of classical R/S analysis [65], but it suffers from a number of other problems (see [96]), in particular, the tendency to overreject the persistence.

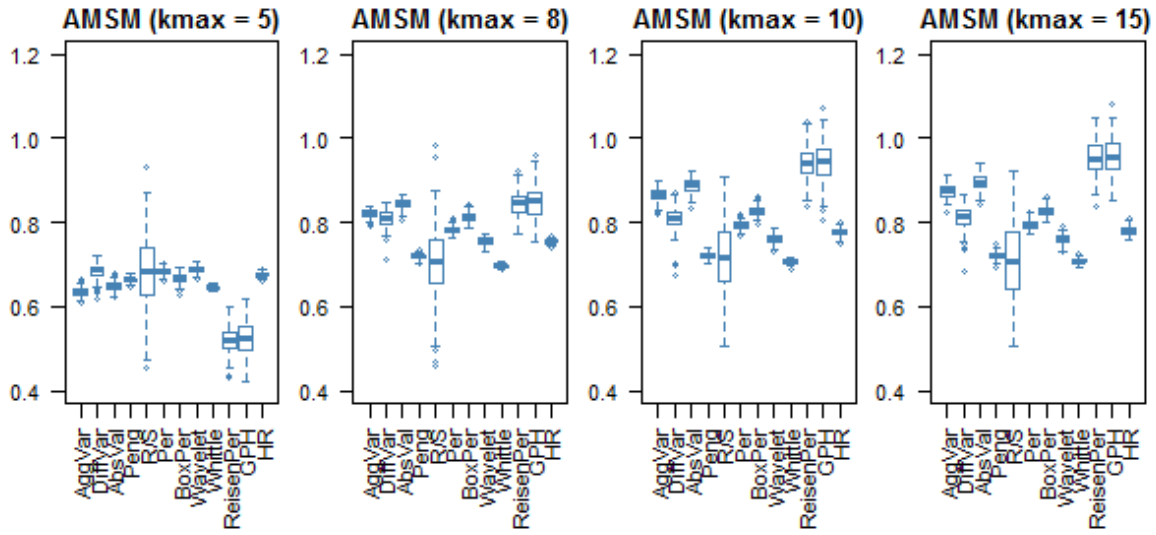


Figure 3.8: The Hurst exponential estimates for the absolute returns of AMSM1 process. Namely, 200 paths simulated with parameters $\theta = (1.4, 0.02, 3, 0.95, 2)$.

Taking into account the different performance of the Hurst exponent estimators, we need to be careful in analyzing results for the AMSM process absolute returns, which are shown in Figure 3.8. All the estimators concordantly test some level of long memory increased from around $H = 0.62$ to $H = 0.83$ with growth of number of scales from $\hat{k} = 5$ to $\hat{k} = 15$. The only exceptions are the Reisen Periodogram method and GPH method estimators in the case of $\hat{k} = 5$, as they rejected the long-memory assumption. Note, that H increases in a non-linear manner with a very slow increase and convergence after $\hat{k} = 15$.

There is a peculiarity we should pay attention to when comparing the estimates of the trial processes and the AMSM1 process: the trial processes' estimates differ significantly in the case of short memory and are concordant in the case of both pure noises. In the case of AMSM1 estimates of H, we also see the dispersion, which spreads when $\hat{k}$ increases. One hypothesis is that the leverage-effect being the short-memory effect can lead to this

[20] An AR(1) process is stationary, if its parameter $|\phi_1| < 1$. As ϕ_1 value 0.6897 was used.

mismatch, but simulations do not support this conjecture; in the case of the ordinary MSM process ($\rho = 0$), box plots of estimates look the same. Another, much more likely, hypothesis is that the reason is the multi-scale nature of the (A)MSM models. The models allow us to generate long-memory behavior for data (volatility, in our case) of different frequencies simultaneously, which in turn leads to the discrepancy of the estimates.

Thus, we have shown the ability of the AMSM model versions to mimic both the leverage effect and the long memory. In particular, the real level of correlations for different ρ (leverage effect) and the levels of persistence using Hurst exponent for AMSM1 model with different specification of $\hat{k}$ (long memory) have been measured .

Note, in Table 3.2 AMSM1/AMSM2 models are compared with a set of well-known models in terms of the possibility to generate the stylized facts.

3.2. Theoretical basis of option pricing based on AMSM models

The AMSM1 and AMSM2 models considered in this work are similar in terms of general structure (see Section 3.1) to the GARCH-class of models and its generalizations. Duan [28] was the first to develop the option pricing methodology for GARCH models, based on the earlier results of Rubinstein (1976) and Brennan (1979). Later on, Duan with his colleagues generalized this approach to implement on the Markov regime switching models [29], GARCH with jumps in returns and volatility [30]. Heston and Nandi (2000) obtained alternative results in GARCH option pricing by developing closed-form solution for the special version of the GARCH (1,1) model. The more recent results of Christoffersen et al (2009, 2012) are based on construction of the Radon-Nikodym derivative leading to the Equivalent Martingale Measure (EMM) in the case of certain conditions based on a moment generating function.

	Stylized Facts	**AMSM**	**GARCH**[21]	**AGARCH**[22]	**EGARCH**[23]	**NGARCH**[24]	**GJR-GARCH**[25]	**FIGARCH**[26]	**ASV**[27]	**LMSV**[28]
ret	Leptokurtic distr.	+	+	+	+	+	+	+	+	+
ur	Asymmetric distr.[29]	-	-	-	-	-	-	-	-	-
ns	Substantial outliers[30]	+	+	+	+	+	+	+	+	+
vo	Leverage effect	+	-	+	+	+	+	-	+	-
lat	Mean reversion[31]	+	+	+	+	+	+?	+	+	+?
ili	Long memory[32]	+	-	-	- (+[33])	-	-	+	-	+
ty	Clustering vol.	+	+	+	+	+	+	+	+	+

Table 3.2: AMSM vs Other discrete-time models.

[21] The generalized autoregressive conditional heteroskedasticity model, see [15].

The generalized (with leverage effect) and modified (non-stochastic volatility formulation) AMSM models construction allows us to implement Duan's option pricing theory for Markov switching models (see [29]). Duan's approach is based on consideration of the basic (micro-)economics and constructs a *Stochastic Discount Factor* (see also [92]), which allows us to evaluate option prices.

In Subsection 3.2.1 we show that Duan's approach is implementable to the AMSM models and define which assumptions are required for that. In Subsection 3.2.2, we define the martingale-measure and the transformation of the AMSM models under it. Next, the necessary conditions for the existence of this measure are defined.

3.2.1. Stochastic Discount Factor

Assume that an investor has a certain consumption level C_t, an utility function $u(C_t)$ and maximizes the two-stage overall utility $U(C_{t+1}, C_t)$ at time t

$$\max\{U(C_{t+1}, C_t)\} = \max\left\{u(C_t) + \exp(-\rho)E^P\left[u(C_{t+1})|\mathscr{F}_t\right]\right\}, \tag{3.32}$$

where ρ is an impatience factor. Note, the utility of consumption at time $t+1$ is uncertain and expected with respect to an information available at time t. Besides, we assume that an investor at time t also invests in I_t assets with the price S_t. Thus, the investor's payoff at time t and $t+1$ are defined as

$$Payoff_t = C_t + I_t S_t, \tag{3.33}$$

$$Payoff_{t+1} = I_t S_{t+1} = C_{t+1} + I_{t+1} S_{t+1}. \tag{3.34}$$

Then, we can substitute the last expressions in (3.32)

$$\max\{u(C_t) + \exp(-\rho)E^P\left[u(C_{t+1})|\mathscr{F}_t\right]\} \overset{(3.33)(3.34)}{=}$$
$$\max\{u\left(Payoff_t - I_t S_t\right) + \exp(-\rho)E^P\left[u(I_t S_{t+1} - I_{t+1} S_{t+1})|\mathscr{F}_t\right]\}.$$

[22] The asymmetric generalized autoregressive conditional heteroskedasticity model of Engle (1990).

[23] The exponential generalized autoregressive conditional heteroskedasticity model, see [76].

[24] The non-linear asymmetric generalized autoregressive conditional heteroskedasticity model, see [34].

[25] The Glosten-Jagannathan-Runkle generalized autoregressive conditional heteroskedasticity model, see [40].

[26] The fractionally integrated generalized autoregressive conditional heteroskedasticity model of Baillie, Bollerslev and Mikkelsen (1996).

[27] The asymmetric stochastic volatility model, see Section 3.1.1.

[28] The long-memory stochastic volatility model of Breidt , Crato and de Lima (1998).

[29] In the case of Gaussian innovations.

[30] Relatively high values of returns in all considered models are produced in the case of high volatility, rather than incorporated directly in a model by using the jump component.

[31] Many details about mechanisms of mean reversion and classification of them were provided by Eric Hillebrand in his Ph.D. dissertation [48]. I used his approach for judging the presence of the mean reversion property.

[32] It is slower (hyperbolic) than the exponential decay of autocorrelation function.

[33] FIEGARCH of Bollerslev and Mikkelsen (1996).

First order condition of maximization in the terms of I_t is

$$\begin{aligned}\frac{\partial U}{\partial I_t} &= \frac{\partial}{\partial I_t}\left(u(Payoff_t - I_t S_t)\right) + \\ &+ \frac{\partial}{\partial I_t}\left(\exp(-\rho)E^P[u(I_t S_{t+1} - I_{t+1}S_{t+1})|\mathscr{F}_t]\right) = \\ &= -S_t u'(Payoff_t - I_t S_t) + \\ &+ \exp(-\rho)E^P[S_{t+1}u'(I_t S_{t+1} - I_{t+1}S_{t+1})|\mathscr{F}_t] = 0\end{aligned}$$

After the simplification, we obtain the condition for maximization of utility, which is also known as Euler equation

$$S_t = E^P\left[\exp(-\rho)\frac{u'(C_{t+1})}{u'(C_t)}S_{t+1}\middle|\mathscr{F}_t\right], \tag{3.35}$$

where $u'(C_{t+1})/u'(C_t)$ is the marginal rate of substitution of current consumption C_t for the future consumption C_{t+1} (MRS_t).

The *stochastic discounter factor*

$$SDF = \exp(-\rho)\frac{u'(C_{t+1})}{u'(C_t)} = \exp(-\rho)MRS_t \tag{3.36}$$

is an important part of the risk-neutral measure transformation described further.

3.2.2. Martingale measure

There are two sources of randomness in the model: the normal innovations $\{\epsilon_t\}$ and the volatility regime defined by vector $\{\bar{u}_t\} = \{u_{1,t}, \ldots, u_{\hat{k},t}\}$ (see (3.12)). As a matter of theoretical convenience, we will substitute uniform variables by CDF of standard normal variable, namely, $\bar{u}_t = \Phi(\bar{\xi}_t)$, where each element of $\{\bar{\xi}_t\}$ is a i.i.d. normal sequence of $\hat{k}$ elements $\{\xi_{1,t}, \ldots, \xi_{\hat{k},t}\}$ independent from $\{\epsilon_t\}$ and independent to each other. It allows us to transform variables freely, because $u_{k,t}$ is limited to $[0,1]$, while $\xi_{k,t}$ is not limited on the real axis.

Proposition 1. *In the framework of option pricing, we assume that the asset returns* $\boldsymbol{r} = \{r_t\}_{t=0}^{\infty}$ *of the underlying asset price process* $\{S_t\}_{t=0}^{\infty}$ *are given by*

$$r_t = \ln\frac{S_t}{S_{t-1}} = r + \lambda\sigma(\epsilon_{t-1}, \bar{\xi}_{t-1})_t - \frac{1}{2}\sigma^2(\epsilon_{t-1}, \bar{\xi}_{t-1})_t + \sigma(\epsilon_{t-1}, \bar{\xi}_{t-1})_t\epsilon_t, \tag{3.37}$$

where r *is a risk-free rate,* λ *is a unit risk premium*[34], ϵ_t *is an i.i.d. standard normal sequence,* $\bar{\xi}_t$ *is a i.i.d. sequence of i.i.d. vectors distributed as* $\mathcal{N}\left(\boldsymbol{0}_{\hat{k}\times 1}, \boldsymbol{I}_{\hat{k}\times\hat{k}}\right)$ *random variables,* $\sigma(\epsilon_{t-1}, \bar{\xi}_{t-1})$ *is an* $\mathscr{F}_{t-1}$*-measurable AMSM-volatility process.*

Further, we define the new measure Q, which is equivalent to the physical measure P and prove that it is a martingale measure. This new measure is an important part of an option price valuation and, consequently, a calibration procedure.

[34]Here and later, it is assumed that if a compensation for the risk of holding an asset as a risk-free rate r could be not enough or – on the contrary – excessive, then a risk premium depending on a volatility level is necessary, namely $\lambda\sigma_t$.

Definition 11. *Let the measure Q be absolutely continuous with respect to the measure P,*

1. $\ln(S_t/S_{t-1})$ *is Q-normal conditionally on* $\mathscr{F}_{t-1}$,
2. $V^Q(\ln(S_t/S_{t-1})|\mathscr{F}_{t-1}) = V^P(\ln(S_t/S_{t-1})|\mathscr{F}_{t-1})$,
3. $E^Q[S_t/S_{t-1}|\mathscr{F}_{t-1}] = \exp(r)$,

then Q and P are called to satisfy Local Risk-Neutral Valuation Relationship (LRNVR).

It is clear from the definition that under LRNVR for both measures, Q and P, we suppose there are invariant one-period variances. Further, the returns need to be distributed log-normally under risk-neutral measure Q conditionally on $\mathscr{F}_{t-1}$ with expected log-returns equal to the risk-free interest rate r.

In the next theorem, we define the conditions that ensure LRNVR holds, before the main theorem is formulated.

Theorem 10. *Assume that an economic agent maximizes an expected utility, this utility function* $u(C_t)$ *is separable and additive, the measure Q is defined by Radon-Nikodym derivative*

$$\frac{dQ}{dP} = \exp\left((r-\rho)T + \sum_{t=1}^{T} \ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right)\right), \tag{3.38}$$

then the measures P and Q satisfy LRNVR if only one of the following contradictory[35] *conditions holds:*

1. *the coefficient of relative risk aversion is a constant, besides*

$$\ln\left(\frac{C_t}{C_{t-1}}\right)\Big|\mathscr{F}_{t-1} \overset{P}{\sim} \mathscr{N}(\mu,\sigma);$$

2. *the coefficient of absolute risk aversion is a constant, besides*

$$(C_t - C_{t-1})|\mathscr{F}_{t-1} \overset{P}{\sim} \mathscr{N}(\mu,\sigma);$$

3. *the utility function is linear*

$$u(C_t) = aC_t + b.$$

Proof. The proof is given in Appendix A.14. □

At the stage of construction of LRNVR measure Q, we need only general assumptions about the agent's utility function as we can see from the conditions and the proof of Theorem 10. These assumptions lead to normality of the logarithmic Marginal Rate of Substitution (MRS)

$$\ln(MRS_t) = \ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right),$$

[35] The first and the second condition cannot both be true.

possibility to formulate Euler equation (3.35) and define measure Q, which is defined as a logarithmic MRS discounted by the difference between an interest rate and an impatience factor.

Note, there is no use for any knowledge about volatility process $\{\sigma_t\}$ for the definition and constructing of Q, except its $\mathscr{F}_{t-1}$-measurability. Nevertheless, any change to measure Q affects the model. In particular, we need to find out the new definition of returns and volatility innovations, ϵ_t and ξ_t, under this measure. The next theorem is dedicated to this problem.

Theorem 11. *Let the underlying asset price process $\{S_t\}_{t=0}^{\infty}$ be defined by (3.37), the measure Q is defined by (3.38) and satisfies LRNVR, then under Q holds*

$$r_t = \ln\frac{S_t}{S_{t-1}} = r - \frac{1}{2}\sigma_t^\star(\epsilon_{t-1}^\star, \bar{\xi}_{t-1}^\star)^2 + \sigma_t^\star(\epsilon_{t-1}^\star, \bar{\xi}_{t-1}^\star)\epsilon_t^\star, \tag{3.39}$$

$$\begin{bmatrix} \epsilon_t^\star \\ \bar{\xi}_t^\star \end{bmatrix} = \begin{bmatrix} \epsilon_t + \lambda \\ \xi_{1,t} + \nu \\ \cdots \\ \xi_{\hat{k},t} + \nu \end{bmatrix} \Bigg| \mathscr{F}_{t-1} \overset{Q}{\sim} \mathscr{N}\left(\boldsymbol{0}_{(1+\hat{k})\times 1}, \boldsymbol{I}_{(1+\hat{k})\times(1+\hat{k})}\right), \tag{3.40}$$

the volatility process of AMSM1 model is defined as

$$\sigma_t^\star = \sigma_0\left(\prod_{i=1}^{\hat{k}} M_{k,t}^\star\right)^{\frac{1}{2}}, \tag{3.41}$$

$$M_{k,t}^\star = \begin{cases} m_0, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in \left[0, \gamma_k(1-\Phi(\rho(\epsilon_{t-1}^\star - \lambda)))\right), \\ 2-m_0, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in \left[\gamma_k(1-\Phi(\rho(\epsilon_{t-1}^\star - \lambda))), \gamma_k\Phi(\rho(\epsilon_{t-1}^\star - \lambda))\right), \\ M_{k,t-1}, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in \left[\gamma_k, 1\right). \end{cases} \tag{3.42}$$

the volatility process of AMSM2 model is defined as

$$\sigma_t^\star = \left(\rho(\epsilon_{t-1}^\star - \lambda) - \sqrt{\sigma_0}\right)^2\left(\prod_{i=1}^{\hat{k}} M_{k,t}^\star\right)^{\frac{1}{2}}, \tag{3.43}$$

$$M_{k,t}^\star = \begin{cases} m_0, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in [0, \gamma_k/2), \\ 2-m_0, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in [\gamma_k/2, \gamma_k), \\ M_{k,t-1}, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in [\gamma_k, 1), \end{cases} \tag{3.44}$$

where $\Phi(\cdot)$ is a cumulative distribution function of standard normal variable.

Proof. The proof is given in Appendix A.15. □

According to Proposition 1, there are two random processes defining the return process (3.37) under the measure P. The first one is white noise $\{\epsilon_t\}_{t=0}^{\infty}$. Theorem 11 establishes the transformation of $\{\epsilon_t\}_{t=0}^{\infty}$ under measure Q, namely it is necessary to substitute random variable ϵ_t distributed as standard normal distribution with respect to P to another variable, which should be distributed under Q (see (3.40)). In addition, the models share

the volatility regime switching construction defined by uniform i.i.d. random variables $\{u_{t,k}\}_{t=0}^{\infty} = \{\Phi(\xi_{k,t})\}_{t=0}^{\infty}$. Theorem 11 establishes corresponding transformation for the uniform random variables $u_{k,t} = \Phi(\xi_{k,t})$ to $\Phi(\xi_{k,t}^{\star} - \nu)$, where ν is the second risk-neutralization parameter (see (3.40), (3.42), (3.44)).

The theoretical difference between AMSM1 and AMSM2 models is in the leverage effect incorporation approach, namely, the way to incorporate a correlation of the volatility process $\{\sigma_t\}_{t=0}^{\infty}$ and the lagged innovation process $\{\epsilon_{t-1}\}_{t=1}^{\infty}$ of the returns process definition. In the case of the AMSM1 model the leverage effect, namely ϵ_{t-1}, is incorporated in the volatility regime switching process components $M_{k,t}$. Therefore, the measure change from P to Q leads to another interval defining the value of $M_{k,t}^{\star}$ (see (3.42)). At the same time, the leverage effect is incorporated in the volatility process definition directly in the case of AMSM2. Therefore, it is necessary to correct this definition under measure Q (see (3.43)), while the intervals defining $M_{k,t}^{\star}$ are the same as under the physical measure P (see (3.44)).

Corollary 5. *The discount price process* $\{\hat{S}_t\}_{t=0}^{\infty} = \{S_t \exp(-r\,t)\}_{t=0}^{\infty}$ *is a martingale under* Q*, thus* Q *is a martingale measure.*

Proof. The proof is given in Appendix A.16. □

The results of Theorem 11 and its Corollary 5 allow us to price derivatives, such as vanilla options, defined in the form

$$C_T = E^Q\left[f(S_T)\right],$$

by using the Monte Carlo technique. As a consequence, we are able now to calibrate the model parameters $\theta = (m_0, \sigma_0, \rho)$ from real option prices from the financial market, as well as the risk-neutral correction parameters (λ, ν).

□

In the next section, we discuss how to implement the Monte Carlo simulation technique in the case of the AMSM model. In particular, the main aim is to make simulation computationally efficient and fast.

3.3. Monte Carlo method for option pricing based on AMSM models

In order to have reasonable model calibration time, we needed to have a fast method of performing option price computations. The fastest way is to compute option prices using a closed-form solution, but this is impossible for our complicated structure model. There are two popular alternatives for numerical solution – the Fast Fourier Transformation and Monte Carlo methods. The first one is impossible to implement, because the characteristic function is unavailable in the case of (A)MSM models. So, the Monte Carlo method is our method of choice. The main disadvantage of it is its relative slowness, especially given the fact we need to produce intense option prices computations on each step of the calibration (optimization) procedure. Hence, we need to increase the speed of the Monte Carlo option prices computation. Luckily, Monte Carlo methods are easy to parallelize and modern software/hardware is available to do this efficiently.

3.3.1. Parallel implementation of Monte Carlo methods

Historically, parallel computations were possible only on Super Computers, which are designed to use the whole power of parallel computations. For ordinary PCs, multi-core Computer Processor Units (CPU) and multi-CPU features were developed in the mid 2000s. As a result, parallel computations became affordable for a wide range of people, including scientists. Recently, a new direction has appeared: parallel computations on Graphic Processor Units (GPUs). The reason for this is the combination of a very high level of parallelism caused by the purposes of graphical 3D-modeling and the low price (relatively to multi-CPU clusters) of GPUs. In this research, the author uses a variety of GPUs, starting with the relatively old AMD4670, middle-range AMD 7730M with 8 Compute Units (512 stream cores) and AMD R7 360 (768 cores), and finally rented a remote PC[36] with powerful GPU, professional level Nvidia GRID K520 (1556 CUDA-cores) and Nvidia Tesla K80 (2496 CUDA-cores) developed specifically for parallel computations. In other words, we are able to run 512/768/1556/2496 computations (path simulations) in parallel. Today, there are two main technologies for parallel computations on GPUs: Nvidia's CUDA [25] (available only for Nvidia GPUs) and OpenCL (works with AMD and Nvidia GPUs and even with multi-core CPUs). I prefer OpenCL as a more universal technology; besides, I initially only had access to AMD GPUs (ATI 4670 and later to AMD7730M).

3.3.2. Monte Carlo simulations

A computation of option price using the Monte Carlo method requires the computation of a certain number (thousands) of payoffs at the maturity T, then taking an arithmetic average of them in the case of European Call/Put. This means we need to simulate a corresponding number of paths of underlying asset price process $\mathbf{S} = \{S_t\}_{t=0}^{\infty}$ from Section 3.1. The simulation of paths is produced in parallel on CPU/GPU. The memory of GPUs is designed to work more efficiently with the special type of variables that are 4-elements vectors. Therefore, it is more efficiently to compute 4 paths simultaneously within each stream running on a separate core. Also, the procedure for generation of Gaussian innovations gives two (actually 8 values because of storing everything in 4-elements vectors) Gaussian values for each two uniform entries. If we do not want to lose 4 Gaussian variables on generation of each point of trajectory, it is better to use them for generation of an additional 4 paths. As a result, we compute 8 paths of the process $\mathbf{S}$ on each core in parallel.

The underlying (A)MSM process has three sources of randomness on each step t of trajectory of $\mathbf{S}$ according to its definition: 1. whether there is a switch ($M_{k,t} = M$) or not ($M_{k,t} = M_{k,t-1}$), 2. if there is a switch, then in what direction it should be (m_0 or $2 - m_0$) and 3. Gaussian innovations ϵ_t. Firstly, we need to generate uniform random values, then transform them to the normal random variables ϵ_t, $\xi_{k,t}$. Secondly, we need to make the risk-neutral correction according to the Theorem 11 expressions (3.42,3.44), namely we subtract ν and λ from ϵ_t and $\xi_{k,t}$, correspondingly. The algorithm schematically described below shows n-th step of sample-path simulation of the AMSM process:

1. **Input**: independent Gaussian 4-elements vectors $NRand_1$ and $NRand_2$, 8-elements vector of asset prices $S_{i,n-1}$, $i = 1..8$;

2. Repeat from $k = 1$ to $k = \hat{k}$:

[36] Amazon AWS cloud service.

(a) Input: independent Gaussian 4-elements vectors of random numbers $NRand_3$, $NRand_4$, ϵ_{n-1} ($NRand_1$ from $(n-1)$-step);

(b) 4-elements vector $NRand_3$ is used to decide whether there is a switch for each of 4 paths on this step n. Repeat from $i=1$ to $i=4$:

- AMSM1: if $\Phi(NRand_3^i - \nu) < \gamma_k$, then $M_{k,n}^i$ draws from binomial distribution M for each i, otherwise $M_{k,n}^i = M_{k,n-1}^i$;
- AMSM2: if $\Phi(NRand_3^i - \nu) < \gamma_k$, then $M_{k,n}^i$ draws from binomial distribution M for each i, otherwise $M_{k,n}^i = M_{k,n-1}^i$;

(c) 4-elements vector $NRand_4$ is used to define new value $M_{k,n}^i$ (m_0 or $2-m_0$) for each of 4 paths. Repeat from $i=1$ to $i=4$:

- AMSM1: if $\Phi(NRand_4^i - \nu) < 1 - \Phi(\rho(NRand_1 - \lambda))$, then $M_{k,n}^i = m_0$ for i-path, otherwise $M_{k,n}^i = 2 - m_0$;
- AMSM2: if $\Phi(NRand_4^i - \nu) < 0.5$, then $M_{k,n}^i = m_0$ for i-path, otherwise assign $M_{k,n}^i = 2 - m_0$;

3. In order to obtain another 4-elements vector of volatilities, $\sigma_n^{(1)}$, repeat from $i=1$ to $i=4$:

- AMSM1: $\sigma_{i,n}^{(1)} = \sigma_0 \left(\prod_{k=1}^{\hat{k}} M_{k,n}^i \right)^{\frac{1}{2}}$
- AMSM2: $\sigma_{i,n}^{(1)} = \left(\rho(NRand_1 - \lambda) - \sqrt{\sigma_0} \right)^2 \left(\prod_{i=1}^{\hat{k}} M_{k,n}^i \right)^{\frac{1}{2}}$;

4. In order to obtain 4-elements vector of volatilities, $\sigma_n^{(2)}$, repeat 2. and 3. for ϵ_{n-1} ($NRand_2$ from $(n-1)$-step);

5. Repeat from $i=1$ to $i=4$

(a) $S_{i,n} = S_{i,n-1} \exp\left(r - \frac{1}{2} \left(\sigma_{i,n}^{(1)} \right)^2 + \sigma_{i,n}^{(1)} NRand_1^i \right)$;

(b) $S_{i+4,n} = S_{i+4,n-1} \exp\left(r - \frac{1}{2} \left(\sigma_{i,n}^{(2)} \right)^2 + \sigma_{i,n}^{(2)} NRand_2^i \right)$;

6. **Output**: 8 Monte Carlo simulated asset prices at time n: $S_{i,n}$, $i = 1..8$.

Note, we need $4 \times (4 \times \hat{k} + 2)$ independent Gaussian random values on each n-step of simulation of 8-th AMSM paths on each kernel.

Further, if the underlying 8-elements asset price vector S_T at maturity T is calculated, then we are able to calculate $Payoff_i = (S_{i,T} - K)^+$ and send it back from the parallel kernel to the host subroutine. In the host subroutine, we collect the payoffs from all the parallel kernels and take an average of them, which is the option price value C^{MC}.

□

There are a few obstacles in the way of Monte Carlo computations methods, which have a strong influence on the quality of results: uniform random numbers generation, Gaussian random numbers generation and improvement of convergence.

3.3.3. Uniform random numbers generation

Monte Carlo methods are very sensitive to the quality of random numbers generation. Therefore, we need to choose the right way to do it carefully. It is known that PC does not have an incorporated source of randomness, at least at present (we will not discuss special devices that are developed currently, for example, those based on the principle of Lava-lamp or attempts to use access time to hard drive). We have to use pseudo- or quasirandom numbers generators, which are purely deterministic by construction. Nevertheless, they pass statistical tests on randomness and they are both suitable for Monte Carlo simulations.

Firstly, let us consider pseudorandom number generators (PRNG). There are few main classes of uniform pseudorandom number generators: Linear Congruential (LCG), Lagged Fibonacci and Mersenne twister. Each has its own advantages and disadvantages. Generally speaking, the first two are simpler and faster than the third one, but they produce numbers of lower quality in the sense of randomness, because of the serial correlation and other features. So, they cannot be used for Monte Carlo simulations. Mersenne twister [68] was therefore chosen as the main PRNG in this work. It has huge period $2^{(19937)} - 1$, which is one of Mersenne primes. It produces high-quality pseudorandom numbers, passes most tests for statistical randomness and is faster than most of the linear congruential generators. Concerning disadvantages, it starts slower than LCGs and it is more sensitive for the initial seed. In order to obtain its own seed for each AMSM simulated path, we use the Linear Fibonacci Generator from the well-known C++ *Boost library* [24] to create the array of seeds. Note, in the (A)MSM-class of models we have three sources of randomness. As we

AMSM-path 1	Day 1 →	Day 2 →	Day 3 →	...	Day T
AMSM-path 2	Day 1 →	Day 2 →	Day 3 →	...	Day T
AMSM-path 3	Day 1 →	Day 2 →	Day 3 →	...	Day T
		...			
AMSM-path M	Day 1 →	Day 2 →	Day 3 →	...	Day T

Table 3.3: The scheme of parallel pseudo-Monte Carlo simulation. The arrows show the direction of random-number generation, T is a maturity, M is a number of simulated paths.

discussed above in the path simulation algorithm, all three belong to Gaussian white noise class. The pseudo-Monte Carlo is a fully parallelizable procedure. In the case of PRNG, 8 AMSM paths are simulated by using one common pseudorandom Mersenne twister sequence generated in each kernel subroutine, see Table 3.3. In other words, blocks of 8 paths independently in parallel simulate M AMSM paths. For $M = 2^{16} = 65536$, it is necessary to run 8192 independent kernel subroutines. As an initial step, it is necessary to generate M seeds and store them in the memory. Each kernel subroutine takes seed values from this array in parallel and outputs 8 terminal simulated values $S_{i,T}$, $i = 1, \ldots, 8$.

Secondly, we consider quasirandom number generators (QRNG) as a possible alternative. The main feature of QRNG is a *low-discrepancy property*. Quasirandom numbers are constructed in order to have this property. As Jäckel writes in his book ([54], 8.5, p. 88; see also Koksma-Hlawka inequality [49]): "The more 'homogeneous' the underlying number generated, the more accurate and rapidly converging will be a Monte Carlo calculation on it". In other words, it is possible to improve convergence significantly by using quasi-RNG

(with correct initialization). There are a few well-known examples of such numbers: Halton, Sobol' and Niederreiter numbers. There are two main issues of QRNG implementation in our case: dimensionality and poor parallelization.

Quasirandom numbers are very sensitive to the dimensionality of the problem. If it is necessary to compute the expectation $E[f(U_1,...,U_d)]$, where $U_1,...,U_d$ are uniform i.i.d., then according to quasi-Monte Carlo method, it is necessary to generate d-dimensional sequence of N quasirandom numbers $\mathbf{x} = \bar{x}_1,\ldots,\hat{x}_N$. As a result, the expectation is computed as

$$E\left[f(U_1,\ldots,U_d)\right] \approx \frac{1}{N}\sum_{i=1}^{N} f\left(x_i^1,\ldots,x_i^d\right),$$

where $x_i^1,\ldots,x_i^d$ are i.i.d. for each $i = 1,\ldots,N$ by the construction of quasirandom numbers. Jäckel showed, that Halton, Niederreiter and Sobol' (with unit initialization) sequences fail to producing of low-discrepancy numbers in the case of large dimensions. Meanwhile, Sobol' numbers with regularity breaking initialization are comparable in quality to pseudorandom numbers (see, Jäckel sections 8.5, 8.6). So, the only choice for high dimensional problems are Sobol' numbers. In the case of the AMSM model, it is necessary to generate $4\times\hat{k}+2$ quasirandom sequences with a dimension equal to maturity T (up to 720) in each kernel subroutine described in Section 3.3.2. Theoretically, we could use a smaller dimension, for example 12 (monthly monitoring), but in doing so, we would neglect all advantages of the MSM model, because for such a small number of steps, the model would not generate stylized facts, due to a lack of any different frequencies. We therefore use the the generator of Sobol' quasirandom numbers written with C++ language by John Burkardt of the original Fortran77 version by Bennett Fox [7] library. This generator computes elements of the Sobol quasirandom sequence with dimension up to 1111.

Another issue is the impossibility to fully parallelize the Monte Carlo approximation procedure based on quasirandom numbers. Unlike to pseudorandom numbers, only one T-dimensional sequence $\mathbf{x}$ of quasirandom numbers theoretically is able to generate a whole set of uniform random numbers so all necessary paths are long enough, see Table 3.4. In other words, each T-dimensional element $\bar{x}_i$ of quasirandom sequence $\mathbf{x}$ is a set of uniform numbers for random variable in AMSM path simulation. This means we should generate huge a T-dimensional QRN-sequence of length $M\times(2\hat{k}+1)$, where M is a number of simulated AMSM paths. For $M = 2^{16} = 65536$ and $\hat{k} = 5$, the length of generated T-dimensional sequence $\mathbf{x}$ equal to $65'536\times(2\times5+1) = 720'896$. This means it is necessary to store the array with $720'896\times720$ elements in the case of maturity (dimension) $T = 720$. This huge array is computed as a part of quasi-Monte Carlo initialization step, then it stored in the memory (it is necessary to allocate almost 3Gb of memory). Each kernel subroutine takes random values from its own rows of this array in parallel and produces 8 Monte Carlo simulated values $S_{i,T}$, $i = 1,\ldots,8$. As a result, this quasi-Monte Carlo simulation procedure is slower than its pseudo-Monte Carlo counterpart, because of the vast number of read operations from the memory by the kernels running on GPU in parallel.

The conceptual difference between PRNG and QRNG approaches is outlined in Tables 3.3 and 3.4. The results for computation time of vanilla European Call-option price are summed up in Table 3.5. The first column is the number of AMSM paths simulated to compute each option price. An average computation time on CPU/GPU based on pseudo-/quasi- random number generation for the various length of paths (maturity) and number

	Dimension 1	Dimension 2	Dimension 3	...	Dimension T
AMSM-path 1	Day 1	Day 2	Day 3	...	Day T
	↓	↓	↓	...	↓
AMSM-path 2	Day 1	Day 2	Day 3	...	Day T
	↓	↓	↓	...	↓
...	...	...	...	...	...
	↓	↓	↓	...	↓
AMSM-path M	Day 1	Day 2	Day 3	...	Day T

Table 3.4: The scheme of quasi-Monte Carlo simulation. The arrows show the direction of generation of T-dimensional sequence of quasirandom numbers, T is a maturity, M is a number of simulated paths.

of paths is collected in the 2nd, 3rd, 4th and 6th columns. An acceleration achieved on the GPU by comparing to the CPU[37] for PRNG and QRNG is collected in the 5th and 7th columns.

As we can see from Table 3.5, the use of GPU for the parallel computations of option prices with (A)MSM model yields great benefit for the time-consumption. An computation on GPU provides acceleration from six times for fewer short paths up to eighteen times for more numerous and longer paths. Note, there are different relationships for PRNG and QRNG cases: the computation efficiency of PRNG-based computations grows exponentially for longer paths, while the QRNG-based computations efficiency is relative stable. Meanwhile, the PRNG-based computations are less time-consuming than the QRNG-based ones; they are from 2.5 to 9 times faster depending on the number and the length of the AMSM paths. This is caused by an intensive use of relative slow read/write operations from/to the memory, compared to direct generation on the fly in the case of PRNG. Additionally, the QRNG-based computations are very memory consuming; more than 1Gb of memory is necessary in the case of 65536 paths of length 240 points. On the other hand, QRNG provides better quality of random numbers, which leads to a more robust Monte Carlo method use.

Note, sample-path simulations of the AMSM model in fact require random variables with normal distribution rather than uniform. Generating Gaussian noise that has normal distribution is less challenging. It is based on s transformation of uniform random variables, but it is also an important issue.

3.3.4. Gaussian random numbers generation

It is known that the inverse cumulative distribution method is a preferable method for generation of non-uniform distributions. Random numbers produced using the inverse cumulative distribution function (CDF) have better statistical properties; for instance, they have lower discrepancy rather than numbers produced using the Box-Muller technique [54], however, in the case of normal distribution, a closed-form of inverse CDF is unavailable. As a solution, we have to use either a numerical approximation of the inverse CDF of normal distribution or choose another technique. Both approaches are tested in this re-

[37] The computation subroutine uses all cores of CPU for parallel computations.

No.of sample-paths	CPU (PRNG)	CPU (QRNG)	GPU (PRNG)	(Accel.)	GPU (QRNG)	(Accel.)
			Maturity 30			
16384	0.0090	0.0214	0.0014	(x6.43)	0.0036	(x5.94)
32798	0.0159	0.0458	0.0018	(x8.83)	0.0064	(x7.15)
65536	0.0297	0.0911	0.0025	(x11.88)	0.0128	(x7.12)
			Maturity 60			
16384	0.0145	0.0464	0.0017	(x8.52)	0.0066	(x7.03)
32798	0.0257	0.0912	0.0022	(x11.68)	0.0124	(x7.35)
65536	0.0500	0.1823	0.0031	(x16.13)	0.0246	(x7.01)
			Maturity 90			
16384	0.0188	0.0677	0.0021	(x8.95)	0.0088	(x7.69)
32798	0.0345	0.1603	0.0026	(x13.27)	0.0162	(x9.89)
65536	0.0665	0.3227	0.0036	(x18.47)	0.0323	(x9.99)

Table 3.5: An average computation time (secs) of European call option prices with various strike prices and maturities by using AMSM2 model on CPU (8-core AMD 8320) and GPU (2496-cores Nvidia Tesla K80) in the case of pseudo- and quasirandom number generator and different number of used sample paths.

search. As the main approach, the classical Box-Muller transformation ([17]) is used. As an alternative Moro's interpolation formula of the inverse Gaussian CDF [73] is used. Another possible choice for the inverse CDF, as advised by Jäckel, is Acklam's interpolation formula [3], which is much more sophisticated for programming.

The Box-Muller transformation has at least two known undesirable outcomes: 1. the Neave effect with congruential pseudorandom number generators *Ran0* (see, [74], [54]); 2. the loss of equidistant property and low-discrepancy in a combination with QRNGs (see 3.5.2, p.104 in Seydel [86]). There are also sometimes collapses in generation of random-numbers (see Figure 9.5 in [54]). Hence, theoretically inverse CDF is a preferable and safer technique, but in the sense of programming, it is slower and more sophisticated than the Box-Muller technique.

The following random numbers are used in this research: Mersenne-Twister pseudo-random numbers transformed by the Box-Muller technique [17]; and Sobol quasirandom numbers transformed by the Abramowitz and Stegun approximation formula (see, the equation 7.1.26 in [1]). Certain comparison test simulations preceded this choice; for more details, see Subsection 3.4.2.

□

There are additional methods to improve convergence of Monte Carlo integrals known as *variance reduction* techniques.

3.3.5. Variance reduction

Figure 3.9 visualizes the convergence of Monte Carlo option price values obtained for 4 different seeds with respect to the number of sample-paths. The variance of Monte Carlo estimates is quite high, and the level and the speed of convergence is not sufficient; the largest difference is around 0.04, while the convergence requires more than 100 thousands of sample-paths simulated.

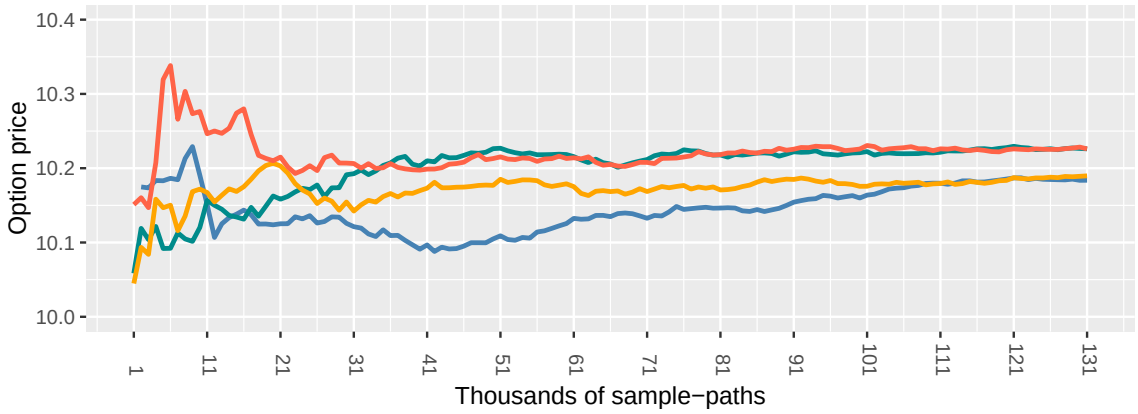


Figure 3.9: Monte Carlo integrals convergence for 4 seeds with Moro's CDF-approximation formula. X-axis is a number of AMSM sample-paths in thousands.

In order to improve convergence and decrease sensitivity to the seed *Antithetic variates* and *Control variates* variance reduction techniques have been tested (see, [86], [54], [39]). The aim of these techniques is to reduce the variance of Monte Carlo integral computation results, which leads to reduction of the standard error of estimates obtained by the Monte Carlo method.

Recall, it is necessary to compute an option price $C = E\left[f(S_T)\right]$ using the Monte Carlo method which is a mathematical expectation of payoff function $f(S_T)$, where S_T is a price of underlying asset modeled by the AMSM process in a maturity time T. This option price obtained by Monte Carlo integration C^{MC} is given by

$$C^{MC} = \frac{1}{M}\sum_{i=1}^{M} f\left(\tilde{S}_T^i\right) \overset{Q}{\to} E\left[f(S_T)\right] = C \text{ for } M \to \infty,$$

where $\tilde{S}_T^i$ are M simulated sample-path values of AMSM process at the time horizon T, Q is the risk-neutral measure constructed in Section 3.2.

The idea of the control variates technique is to use knowledge about the difference (error) between the known closed-form solution of a variable $V = E\left[g(S_T)\right]$ correlated with C and the value of V^{MC} obtained using the Monte Carlo method

$$V^{MC} = \frac{1}{M}\sum_{i=1}^{M} g\left(\tilde{S}_T^i\right) \overset{Q}{\to} E\left[g(S_T)\right] = V \text{ for } M \to \infty,$$

where $\tilde{S}_T^i$ are the same M simulated sample-path values of AMSM process at the time horizon T as used for the computation of C^{MC}. The knowledge about the error in a computation of V^{MC} helps to make correction of Monte Carlo integral C^{MC}. The option price value

C^{CV} corrected by control variate is given by

$$C^{CV} = C^{MC} + b\left(V - V^{MC}\right), \tag{3.45}$$

where C^{MC} is an original Monte Carlo integral value, V is the value of the expectation $E\left[g(S_T)\right]$ known analytically, b is defined as

$$b = \frac{\sigma_C}{\sigma_V}\rho_{CV} = \frac{\mathrm{cov}[C,V]}{\mathrm{var}[V]} \approx \frac{\sum_{i=1}^{N}\left(f\left(\tilde{S}_T^i\right) - C^{MC}\right)\left(g\left(\tilde{S}_T^i\right) - V^{MC}\right)}{\sum_{i=1}^{N}\left(g\left(\tilde{S}_T^i\right) - V^{MC}\right)^2},$$

where $\sigma_C^2 = \mathrm{var}[C]$, $\sigma_V^2 = \mathrm{var}[V]$, ρ_{CV} is a correlation between C and V, $\tilde{S}_T^i$ are the same M simulated sample-path values used earlier for computation of C^{MC} and V^{MC}.

There are two obvious candidates for use as a control variate. First, a spot asset price S_0 can be used as a control variate, because it is known from the theoretical background of the model that $E^Q[\exp(-rT)S_T] = S_0$. As the second candidate is a Black-Scholes option price, which has a closed-form solution, but in this case it is necessary to assume that the volatility $\sigma_t = \sigma_0 =$ constant, therefore, the inner structure of the AMSM model' volatility is neglected. Conversely, this control variate is more suitable for the correction of another option price computation in an economical sense. Further, it showed better correction results during tests. So, the second control variate has been chosen.

The control variates technique improves Monte Carlo integration, but it can be improved even further. Also, the control variates technique is less effective for out-of-the-money options, because the correlation b is much weaker for on-the-money options (see [39]). So, an additional variance reduction technique can be considered, in particular, antithetic variates.

The idea of antithetic variates is to use the property of normal distribution symmetry. Namely, if the sample-path of the process **S** is simulated using the vector of normal variables $\epsilon = \{\epsilon_1, \ldots, \epsilon_T\}$, then the sample-path simulated using the vector $\epsilon^- = \{-\epsilon_1, \ldots, -\epsilon_T\}$ is equally probable. Therefore, it is possible to increase the number of simulated paths in two times without a generation of additional random numbers. The option price C computed by the Monte Carlo method with antithetic variates, namely C^{AV}, is defined as

$$C^{AV} = \frac{1}{M}\sum_{i=1}^{M}\frac{f\left(\tilde{S}_T^i\right) + f\left(\hat{S}_T^i\right)}{2}, \tag{3.46}$$

where $\tilde{S}_T^i$ are M simulated sample-path values of the AMSM process at the time horizon T, $\hat{S}_T^i$ are antithetic variates, which are derived by using negative vectors of normal variables $\epsilon^- = \{-\epsilon_n\}_{n=0}^T$. So, the arithmetic average of payoff $f\left(\tilde{S}_T^i\right)$ and its antithetic variate payoff $f\left(\hat{S}_T^i\right)$ are used as the payoff value in each sample-path simulation.

There are three normally distributed sources of randomness in the case of the AMSM model as described in Section 3.3.2 and denoted there as NRand_1, Nrand_3 and NRand_4. Therefore, it is necessary to use $-\mathrm{NRand}_1$, $-\mathrm{Nrand}_3$ and $-\mathrm{NRand}_4$ in the Monte Carlo simulation subroutine in order to construct antithetic variates $\hat{S}_n^i$ on each step n. So, we can simulate additional eight values of underlying asset prices $\hat{S}_T^i$ at maturity T in each kernel for each of eight paths and corresponding eight payoffs $(\hat{S}_T^i - K)^+$. It is necessary to take an average of the payoffs of simulated underlying asset price and its antithetic variate, then send this average to the host sub-routine for averaging accordingly (3.46). As a result, the

antithetic variates allow us to simulate 16 paths instead of 8 with a smaller effort on each kernel (the time-consumption increases only on around 35%).

Figure 3.10 depicts the Monte Carlo option price value changes in the case of increasing the number of sample-paths used. Is is obtained with 4 different seeds using the control variates and antithetic variates variance reduction techniques. A visual comparison of Figure 3.10 with the pure Monte Carlo depicted in Figure 3.9 in the same scale shows the decrease of the variance of the Monte Carlo method. The largest difference is around 0.008 there compared to around 0.04 for the pure Monte Carlo. The increase of the speed of convergence of the Monte Carlo procedure shows the convergence after around 65 thousands sample-paths used. The drawback is that, the antithetic variates have been found to lead to significant biases of option prices obtained with this technique for the AMSM model[38]. In addition, the literature states that an implementation of the antithetic variates technique with quasirandom numbers can lead to unpredictable results (see [54]). So, the antithetic variates have been tested, but not used for calibration of the model in the next sections.

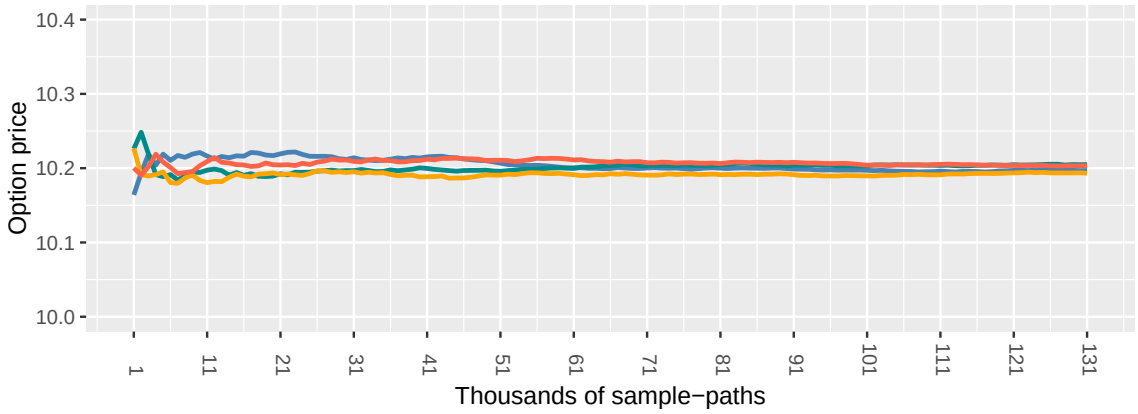


Figure 3.10: Monte Carlo integrals convergence for 4 seeds. Antithetic and control variates techniques are used. X-axis is a number of AMSM sample-paths in thousands.

3.4. Calibration of AMSM models' parameters based on option prices

In Section 3.2 the theoretical background of option pricing have been developed. The key Theorem 11 defines the asset returns process $\{r_t\}$ under the LRNVR measure by adding two risk premiums, namely λ and ν. In this section, the unit risk premium λ is assumed to be known, while the volatility risk-premium ν is assumed to be zero.

3.4.1. Theoretical background and practical obstacles

In order to use the AMSM model for computation of exotic option prices or constructing of volatility surface, we cannot use estimation techniques based on historical data about underlying asset price, because they recover parameters values w.r.t. physical probability

[38]This result was obtained empirically during simulations on artificial data and not presented here due to its minor importance.

measure P whereas we need parameters of the model w.r.t. to risk-neutral measure Q. In this case, we can use *calibration* which is a procedure of model parameters recovering from a real option price data. The idea is to find such parameter vector $\theta = (m_0, \sigma_0, \rho)$ that fits the real data in the sense of a minimal *Residual Sum of Squares* (RSS)

$$RSS = \sum_{i=1}^{N} \left(C_i^R - C_i^{AMSM}\right)^2 \tag{3.47}$$

where C_i^R is a real option price and C_i^{AMSM} is a corresponding option price computed by using AMSM model assumptions. There is an obstacle, prices of out-of-the-money options are much smaller than prices of in-the-money options, hence they would dominate the values of out-the-money options in RSS. Therefore, it is necessary to use a weighted sum for them, such as

$$WRSS = \sum_{i=1}^{N} w_i \left(C_i^R - C_i^{AMSM}\right)^2 \tag{3.48}$$

where as w_i are used $\left(1/C_i^R\right)^2$.

The theoretical problem seems to be clear, but a practical implementation leads to a number of obstacles described in the next subsection.

Narrow valley region problem

There are two issues with an optimization procedure in spite of the convex shape of the objective function being a sum of squares (WRSS). The first one is a long narrow valley on the surface of the objective function (OF). It is depicted in Figure 3.11. The surface's

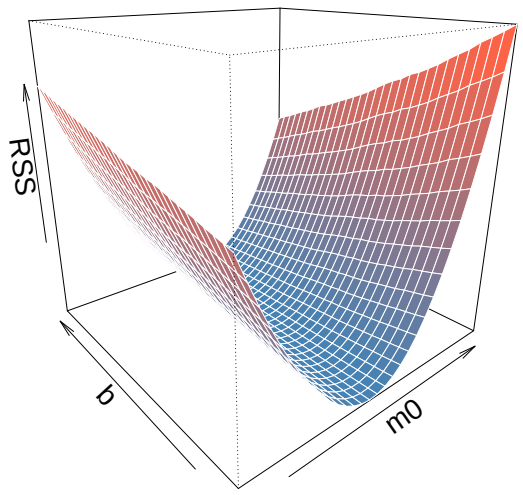


Figure 3.11: The objective function (RSS) surface w.r.t. the model' variables m_0 and b.

tilt along the valley is small. The reasons for that are the different sensitivity and scale of the parameters: m_0 is in interval $(1,2)$, σ_0 and ρ are around 0.01-0.05 (ρ is around 2 for AMSM1). The presence of such region makes many optimization procedures very slow.

Another negative factor could lead optimization procedures to fail: the noise, the source of which is in the Monte Carlo simulations that underlie the evaluation of the objective

function (WRSS). In other words, the objective function (3.48) has a stochastic nature with an amplitude of fluctuations close to the slope through the valley. As a result, the surface (that has to be smooth and convex) looks "lunar" in a small scale. The evidence is presented in Figure 3.12.

It is clear from Figure 3.12 that the surface becomes smoother for the greater number of simulated sample-paths, but it is not monotonic and non-convex anyway. The issue also relates also to a numerical analysis. All the computations were made with *single-precision* computations[39]. This means the precision of all mathematical operations is only 7-9 decimal digits, resulting in the precision of Monte Carlo being around 4 digits after the decimal point. Hence, it is another source of noise.

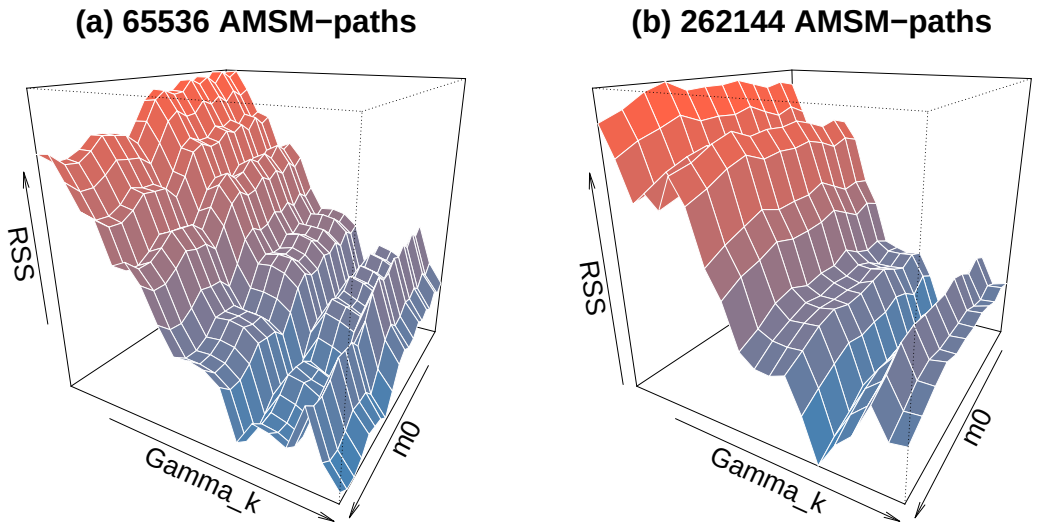


Figure 3.12: The objective function (RSS) surface in the case of 2^{16} and 2^{18} sample-paths used in Monte Carlo method for each evaluation of option price in RSS.

Further, an overview of testified optimization methods used for the minimization (optimization) of (W)RSS is given.

Optimization methods

The first class of considered methods is the class of stochastic optimization methods. In the literature, its use is often proposed for a calibration, as this class is relatively robust to noise and local minimums. One such method is the Simultaneous Perturbation Stochastic Approximation (SPSA, [87]). It is known as a cheap, robust to a noise method that is similar to gradient-based methods by the speed of convergence. Even so, it often stops reaching the bottom of the valley with the data simulated for the presented models (see Section 3.3.2) and preference has been given to another stochastic method: Simulated Annealing (SA, [85]) and its variation, Adaptive Simulated Annealing (ASA, [53]). Both are global optimization methods developed to overcome the non-convexity problem. Nevertheless, the (A)SA method has shown high dispersion of results during Monte Carlo experiments and seems to suffer from local minimums as well as other methods, as it is presented further. In order to improve the convergence of (A)SA, a two-stage approach was used: the stochastic

[39]Due to a time-consumption and the hardware limitations

global optimization method (Simulated Annealing is recommended by many authors for calibration) in order to obtain the first approximation of the minimum, then Levenberg-Marquardt gradient-based method in the second stage to obtain a more precise minimum. This approach revealed the superiority of the ASA method over SA modification in the sense of closeness to the real parameter values.

Among other testified methods is the Nelder-Mead simplex method [75] from the class of direct search zero-order methods. Methods from this class are characterized by very low efficiency (a large number of objective function evaluations). Also, the algorithm can (and does) get stuck in local minimums, as do the previous group methods. This method has showed worse performance than others and has not been used further in these experiments.

The third class consists of pure gradient-based first-order optimization methods, for example, the Levenberg-Marquardt (LM, [58]) algorithm, conjugate gradient (CG, [89]) method and the Broyden-Fletcher-Goldfarb-Shanno (BFGS, [18]) algorithm. In order to decrease probability of getting stuck in a local minimum, I added bound to the step size of these algorithms below by $\delta = 10^{-4}$ (in the case of the two-stage approach as well). In addition, I used preconditioning and scaling. These algorithms go to the bottom of valley quickly, and then go slowly through it or get stuck in the local minimums due to the non-smooth surface. Despite the convergence issues, the LM method is chosen as the second main approach for wide Monte Carlo experiments.

Errors of calibration

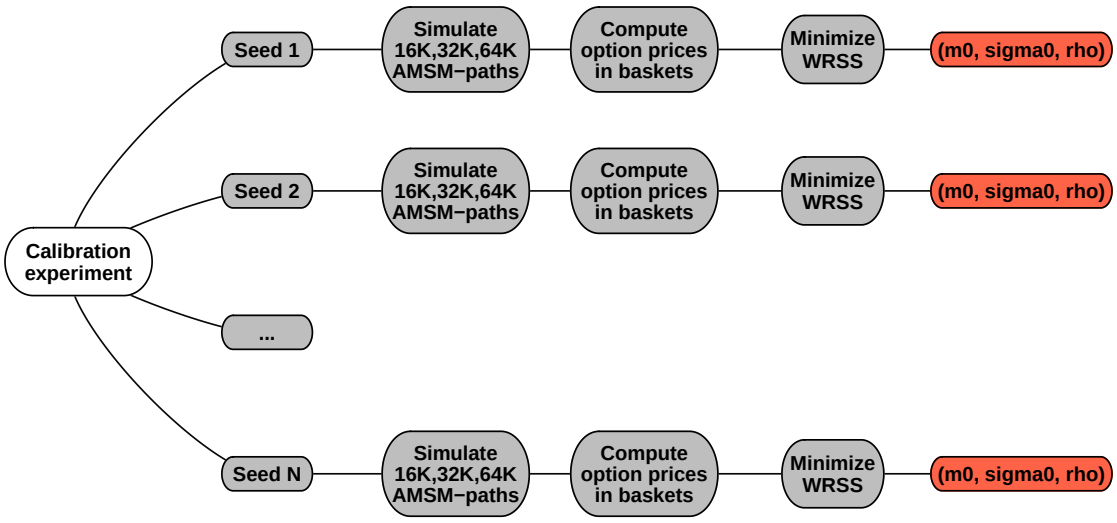


Figure 3.13: A calibration (Monte Carlo) experiment.

Before starting the model calibration, let us clarify a few terms and measures heavily used throughout the remainder of this research.

Definition 12. *In order to be able to investigate a quality of the calibrated parameters, it is necessary to repeat a calibration/estimation procedure N times (*50 *or* 100*, otherwise specified) with the different Monte Carlo seeds for each pair (#options, #paths) and for the specified settings (an optimization method, a basket structure, the model version). Further, this*

procedure is called a calibration/estimation experiment (see Figure 3.13 and the red nodes in Figure 3.14).

Definition 13. *Precision of calibration procedure means a measure of calibration results variability.*

Definition 14. *Accuracy of calibration procedure means a measure of systematic calibration errors, in other words, biases of parameters' estimates.*

As the triplets of quality measures (a center metric, an accuracy and a precision metrics) of a calibration procedure for a different number of options, sample-paths and other settings, the following metrics are used: Mean/Standard Deviation/RMSE and, the more robust triplet, Median/Median Absolute Deviation (MAD)[40]/Mean Absolute Error (MAE). As the visualization of the differences of these triplets the distribution of 100 Monte Carlo

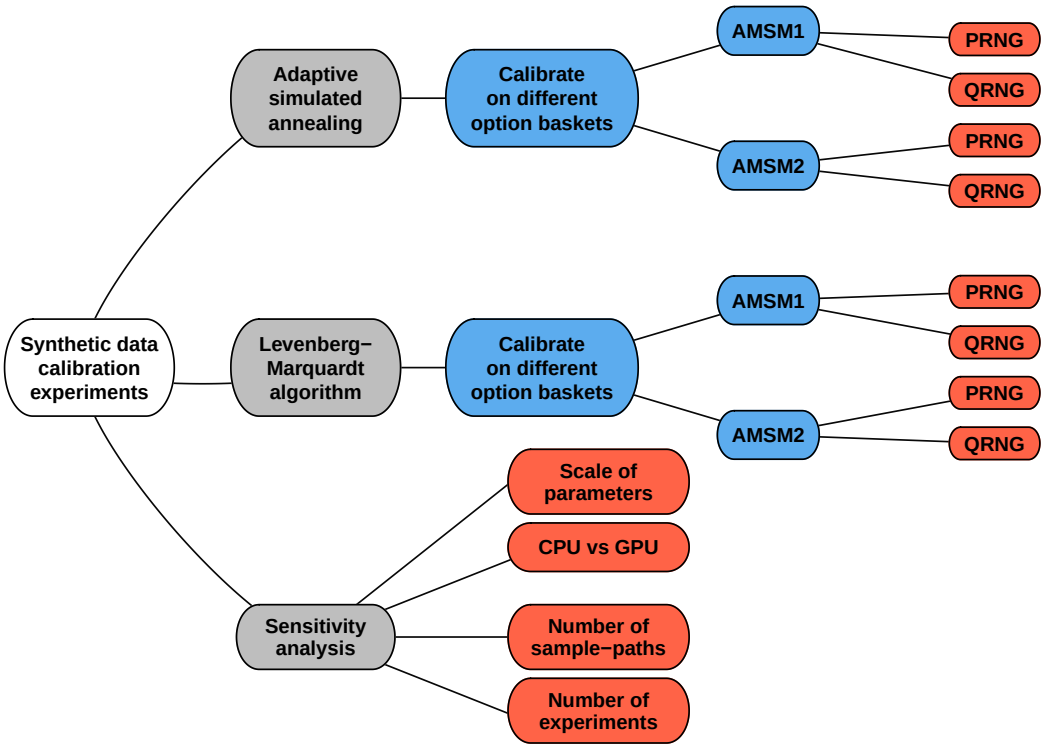


Figure 3.14: The structure of calibration (Monte Carlo) experiments.

calibration procedure repeats for the parameter m_0 are plotted in Figure 3.15, in which the red vertical lines are $Mean \pm 2 * SD$ and the blue lines are $Median \pm 2 * MAD$. It is evident that, the plot, the bi-modal distribution is corrupted by outliers[41]. The first triplet clearly gives more weight to values far from the mean (outliers), which distorts measurement of center/precision/accuracy. The Median/MAD/MAE triplet of metrics is more robust.

[40]The scale constant is calculated from the sample as 75%-quantile.

[41]The nature of outliers is a failure of the numerical procedure convergence described in Subsection 3.4.1.

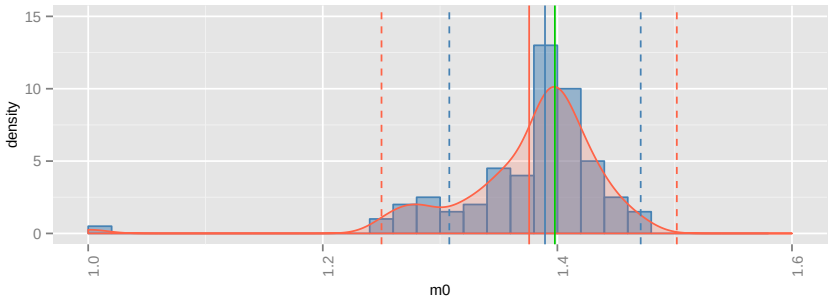


Figure 3.15: Results of 100 calibration procedure repeats with the settings: 32768 sample-paths, $\theta^{real} = (1.4, 0.02, 0.05)$, AMSM2 model. Mean/SD/RMSE – red lines, Median/MAD/MAE – blue lines, Mode – green lines.

There is an alternative solution: to *remove all outliers*. This leads to a problem of choice of calibration results, which can be recognized as a fail. If all m_0 less than some barrier are removed, say 1.2 for m_0, ρ less than 0.00075, the mean/median would depend on these barriers values. A more sophisticated solution in this circumstance is use of Expectation-Maximization clustering [83]. K-means clustering is another plausible candidate, which seems to be very natural approach. It "kills two birds with one stone"; that is, infiltrates outliers and calculates the mean value for the rest of the results of calibration. This clusterization method separates the calibration experiment data on two subsets with different Gaussian distributions. The disadvantage of this approach is the presence of an empirical rule of thumb deciding which group consists of outliers. This solution was tested, but the main direction chosen was an improvement of convergence by choosing an appropriate optimization method, smart options basket construction and an optimization of the number of sample-paths in the Monte Carlo method.

3.4.2. Sensitivity analysis

It is necessary to examine the approach and the models using simulated data before using real option price data (from the market). There are a few questions that arise before starting the calibration: how many sample-paths in each calibration should be used? How many times should the calibration procedure be repeated to get plausible results? The next subsections are devoted to these and some other questions.

First of all, it is necessary to mention that many experiments have been done for the AMSM2 model version, because of its lower time-consumption. The calibration results comparison for AMSM1 and AMSM2 are given for the best setting revealed for AMSM2 version at the end of the section. Second, in order to accelerate and to simplify the experimentation process, most of the calibration experiments in this and the subsequent sections have been done using Amazon Elastic Cloud service [4]. In particular, *g2.2xlarge* instances with the powerful Nvidia GRID K520 GPU and Intel Xeon E5-2670 CPU have been used.

A synthetic option data set is used for the calibration experiments in the next two sections. It is constructed as baskets of up to 70 European Call Option prices and computed for the vector of parameters $\theta = (m_0, \sigma_0, b, \gamma_{\hat{k}}, \rho) = (1.4, 0.02, 3, 0.95, 0.05)$ (AMSM2 model). One seed was used for the simulation of the artificial data and then distinct seeds were used for the calibration experiments. The following fixed parameters are used: the number of frequencies $\hat{k} = 5$ and the interest rate 0.00018 per day. The range of equity risk premium

(ERP) values, namely $\lambda\sigma_t$, are from 3% to 8%, depending on a historical period of time according to various references, while an annualized volatility σ_t of a stock market is usually in the range $10\% - 20\%$. Therefore, λ is in the approximate range $0.15 - 0.8$ and the value $\lambda = 0.51$ has been used as a fair possible value for the simulations in this section.

As noted above, the artificial option baskets are constructed as a set of vanilla Call options with 10 different strike prices (K): 40, 42,...,58. The maturities are: 30, 60, 90, 120, 240, 360, 720 days. The initial underlying stock price $S_0 = 50$.

Definition 15. *The Call option baskets with different strike prices (K) and different maturities (T) are henceforth called KT-baskets, in contrast to K-baskets, which includes options with one and the same maturity and only the strike price K varies.*

The structure of artificial option price data is presented in Table 3.6 for clarity. The reason for this choice is to reproduce the whole volatility surface, especially w.r.t. the maturity T, because the features of the models are more likely to be reproduced on a long horizon. Otherwise there would be only one or two frequencies for a short maturity. According to this structure, an example of K-baskets are $\{C_{31},\ldots,C_{40}\}$, $\{C_{61},\ldots,C_{70}\}$, KT-basket example is $\{C_1,\ldots,C_{30}\}$.

T \K	40	42	44	46	48	50	52	54	56	58
30	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
60	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}
90	C_{21}	C_{22}	C_{23}	C_{24}	C_{25}	C_{26}	C_{27}	C_{28}	C_{29}	C_{30}
120	C_{31}	C_{32}	C_{33}	C_{34}	C_{35}	C_{36}	C_{37}	C_{38}	C_{39}	C_{40}
240	C_{41}	C_{42}	C_{43}	C_{44}	C_{45}	C_{46}	C_{47}	C_{48}	C_{49}	C_{50}
360	C_{51}	C_{52}	C_{53}	C_{54}	C_{55}	C_{56}	C_{57}	C_{58}	C_{59}	C_{60}
720	C_{61}	C_{62}	C_{63}	C_{64}	C_{65}	C_{66}	C_{67}	C_{68}	C_{69}	C_{70}

Table 3.6: Structure of artificial datasets with 70 Call option prices computed by Monte Carlo method.

There are different numbers of sample-paths used for option pricing in the Monte Carlo method for each C_i in different experiments. The ones that have been tested are, namely:

- $8 \times 2^{10} = 2^{13} = 8192$ sample-paths denoted for simplicity as $8K$;
- $2^{14} = 16384$ denoted as $16K$;
- $2^{15} = 32768$ denoted as $32K$;
- $2^{14} + 2^{15} = 49152$ denoted as $48K$;
- $2^{16} = 65536$ denoted as $64K$;
- 2^{20} denoted as $1024K$.

The numbers of sample-paths are factors of 2 due to the reasons, in particular, the number of sample-paths simulated in each parallel subroutine (core) is $8 = 2^3$ as described in Section 3.3.2.

Hardware dependency

First of all, it is necessary to note that the round error problem plays a role in the comparison of results produced by CPU and GPU, enhanced by use of a single-precision floating-point format. In fact, a calibration procedure computes not exactly the same option price values for the same model parameters and experiments settings (including the seed), but those for a different hardware. The variation has amplitude around 10^{-4}. In turn, this means that it is impossible to reproduce exactly the same results on a different hardware (no matter its kind, CPU or GPU), but is statistically equal, as will be shown further.

The calibration experiment settings for CPU and GPU are: 100 calibration repeats; $32K$ sample-paths for each option price evaluation; 40 options with various strike prices and maturities; $\theta = (1.4, 0.02, 0.05)$; AMSM2 model. The illustration of the results is provided using violin[42], as shown plots in Figure 3.16. A visual inspection leads to the conclusion that the shape of distributions in both cases is slightly different: the GPU-based experiments seem to fail slightly more often, which leads to slightly more noisy results.

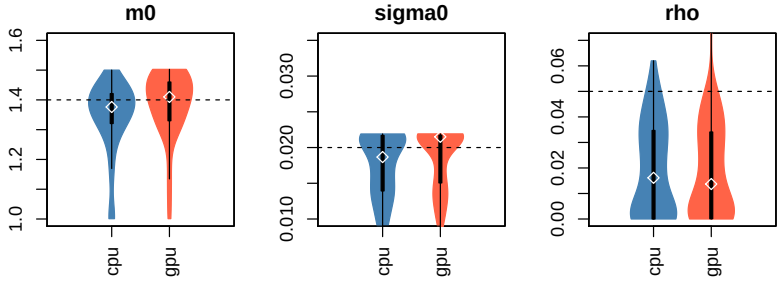


Figure 3.16: Results of 100 calibration repeats on CPU and GPU with the settings: 32768 trajectories, 40 options, $\theta^{real} = (1.4, 0.02, 0.05)$, AMSM2 model.

In addition to the distribution plots, Yuen's t-test (Yuen and Dixon (1973), Yuen (1974)) was conducted. It is more robust than the Welch t-test in the case of non-normality (including outliers, long tails). It is strongly rejected the null hypothesis (non-similarity) of the experiments results distributions based on GPU and CPU for all three parameters (see Table 3.7).

Parameter	H_0	p-value
μ	$Law(\mu^{GPU}) \neq Law(\mu^{CPU})$	$1.514451e-51$
σ_0	$Law(\sigma_0^{GPU}) \neq Law(\sigma_0^{CPU})$	$3.138822e-183$
ρ	$Law(\rho^{GPU}) \neq Law(\rho^{CPU})$	$9.853688e-108$

Table 3.7: Distribution equivalence test of Yuen for GPU- and CPU-based experiments for the same settings.

Note, due to the statistical indifference of GPU- and CPU-based calculations, all simulations for baskets of 30, 40, 50 options were made on more powerful GPUs, while CPUs

[42]This kind of plots depicts a kernel density symmetrically with respect to a vertical box plot for each dataset (labeled on x-axis). The small white circle is a median value; the upper and lower boundaries of the solid black rectangular are the 1st quartile (Q1) and the 3rd quartiles (Q3); the upper and lower boundaries of violins are $Q1 - 1.5 * IQR$ and $Q + 1.5 * IQR$, where $IQR = Q3 - Q1$ (interquartile-range).

were used for small baskets and short maturities in order to utilize computing resources efficiently.

Gaussian quasirandom number generation method

In this experiment, AMSM model parameters are calibrated on an artificial option data simulated using one of three Gaussian quasirandom number transformation methods: Box-Muller transformation, Abramowitz and Stegun approximation formula, and Moro approximation formula [73]. All the methods use uniformly distributed quasirandom numbers, transforming them to normally distributed ones. The goal is to choose a more robust approach in the sense of calibration results quality. The various errors metrics are calculated for this reason, knowing the real parameters values. The results are collected in Table 3.8 and the histograms of calibrated values are presented in Figure 3.17.

	Moro	A&S	BM
		m_0	
Mean	1.32592	1.37922	1.41051
Median	1.32770	1.37341	1.40543
SD	0.06105	0.02465	0.04013
RMSE	0.09561	0.03206	0.04109
MAE	0.07784	0.02640	0.03253
MSE	0.00914	0.00103	0.00169
		σ_0	
Mean	0.01784	0.02004	0.01939
Median	0.01848	0.01999	0.01973
SD	0.00335	0.00027	0.00246
RMSE	0.00396	0.00027	0.00251
MAE	0.00271	0.00004	0.00193
MSE	0.2×10^{-4}	0.8×10^{-7}	0.6×10^{-5}
		ρ	
Mean	0.05457	0.05471	0.05242
Median	0.05159	0.05561	0.05264
SD	0.01068	0.00405	0.00771
RMSE	0.01152	0.00619	0.00801
MAE	0.00748	0.00521	0.00616
MSE	0.00013	0.00004	0.00006

Table 3.8: The comparison of Gaussian random numbers generation methods for QRNG: Moro approximation formula, Abramowitz and Stegun formula, Box-Muller transformation.

Note, the error metrics and the figures clearly prove the superiority of the Abramowitz and Stegun approximation formula in the sense of accuracy and precision of calibration for

quasirandom numbers generation. The Box-Muller transformation provides similar quality, while the calibration based on Moro's formula provides significantly more biased and volatile results for all three parameters.

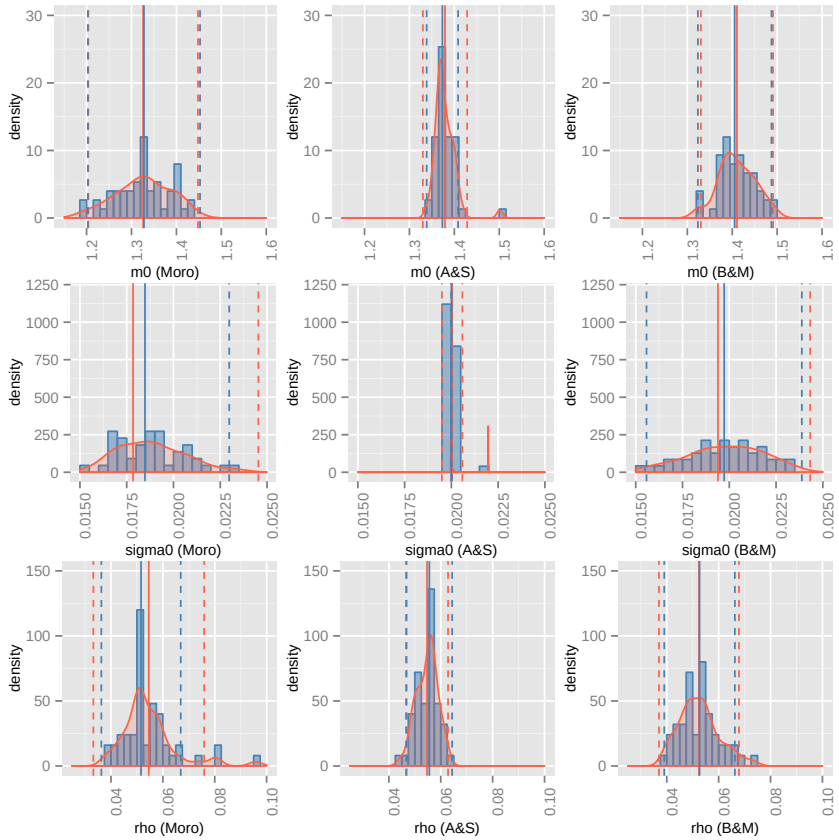


Figure 3.17: Distribution of calibration results for different Gaussian random number generation methods: Moro approximation formula, Abramowitz and Stegun formula, Box-Muller transformation.

Summing up, the Abramowitz and Stegun approximation formula is preferable for the case of Gaussian quasirandom numbers generation, while the Box-Muller transformation will be used further for Gaussian pseudorandom numbers generation.

Number of sample-paths

In the next experiments, the KT-baskets of options are used with the different maturities according to Table 3.6. For example, the basket of size 25 consists of Call options with the prices $C_1, C_2, \ldots, C_{25}$ (maturities 30, 60, 90). In this experiment various sample-paths numbers ($16384 = 16K$, $32768 = 32K$, $65536 = 64K$) are used, based on both pseudorandom and quasirandom numbers for computation of each option price.

Let us begin with Figure 3.18. It collects as bar plots the results of calibration experiments for a few combinations of option baskets and path numbers for each calibrated parameter, in both cases of pseudo- and quasirandom numbers. This set of plots should give us general information on the sensitivity of the calibration procedure to the number of sample-paths used for each option price computation.

Firstly, the accuracy of calibration is considered using Figure 3.18. The most important outcome is almost absent biases for a large number of sample-paths ($64K$), especially in

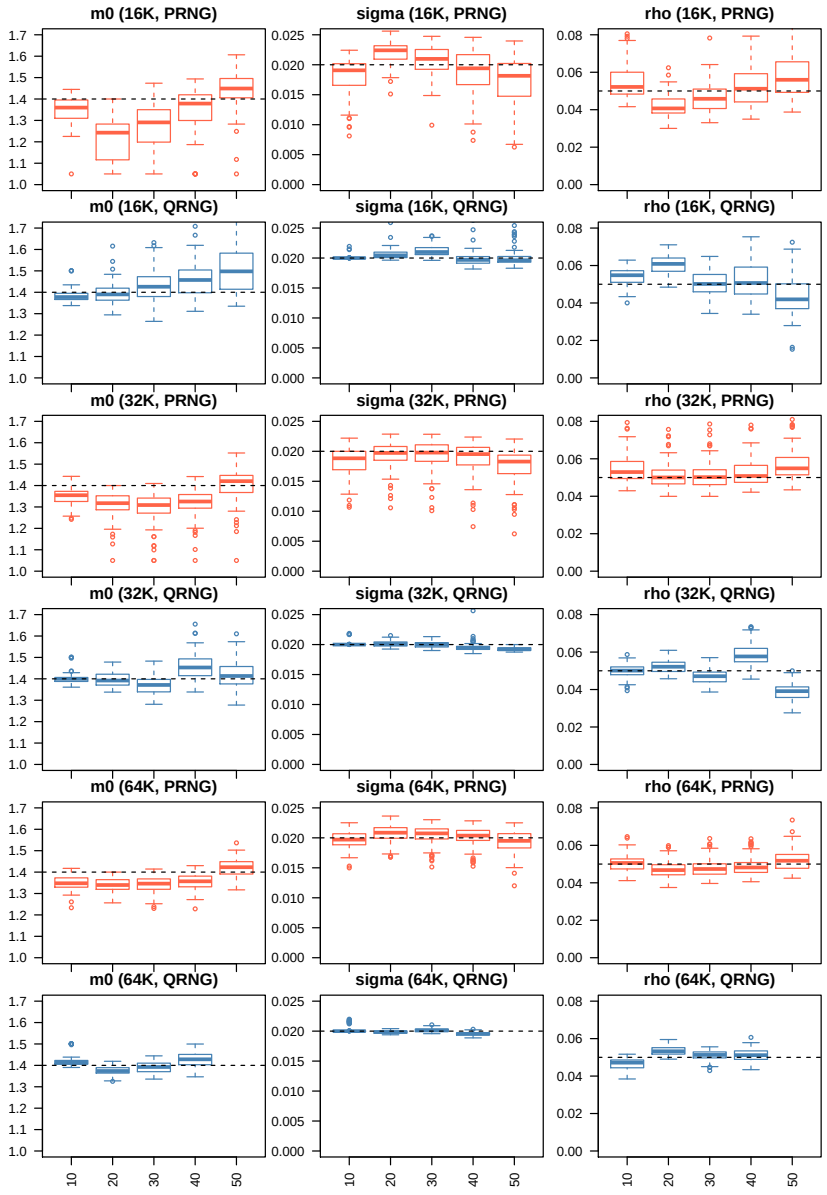
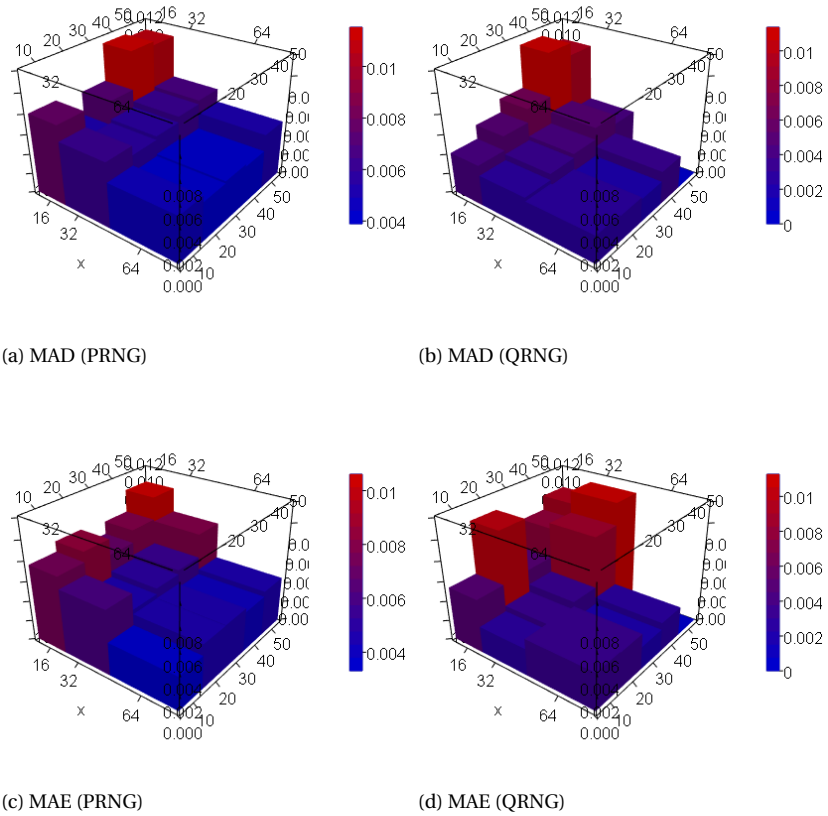


Figure 3.18: Levenberg-Marquardt optimization method. KT-baskets with 10 to 50 options. AMSM2 model with $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$. X-axis is a number of options in basket, Y-axis is distribution of calibration results for one of the parameters, the number of sample-paths and the kind of generator are noted in the brackets.



(a) MAD (PRNG)

(b) MAD (QRNG)

(c) MAE (PRNG)

(d) MAE (QRNG)

Figure 3.19: Error metrics (z-axis) of ρ estimates in cases of different size of option baskets (y-axis) and the number of sample-paths (x-axis).

the case of ρ estimates. Note, there is a bias of median ρ for the large baskets (40, 50 options) and the medium or low number of sample-paths ($16K$, $32K$). Comparing the results for PRNG and QRNG, it is easy to identify different patterns: the estimates of σ_0 based on QRNG are almost perfect in all three cases, while the PRNG-based estimates strongly depend on the number of sample-paths; therefore, the biases have different directions. In general, QRNG exceeds PRNG results in the sense of accuracy for all numbers of sample-paths, while the preferable setting is $64K$ sample-paths.

Secondly, the precision of the calibration is considered using Figure 3.18. It is clear that increasing of the sample-paths number leads to a decrease of the calibration results dispersion for all parameters, especially for m_0. The deviation is moderate for $32K$ sample-paths in both cases of random number generators; it improves even more for $64K$.

Summing up, quasirandom numbers based experiments show up superior calibration experiments results for $64K$ sample-paths in the sense of accuracy and precision, especially because of ρ estimates. Note, the quasi-Monte Carlo method has superior results even for a small number of sample-paths in the case of small 10-options K-baskets with maturity $T = 30$[43]. Figure 3.19 confirmes these conclusions, showing the values of metrics MAD (precision) and MAE (accuracy) for different numbers of sample-paths and KT-baskets in the case of ρ parameter estimates.

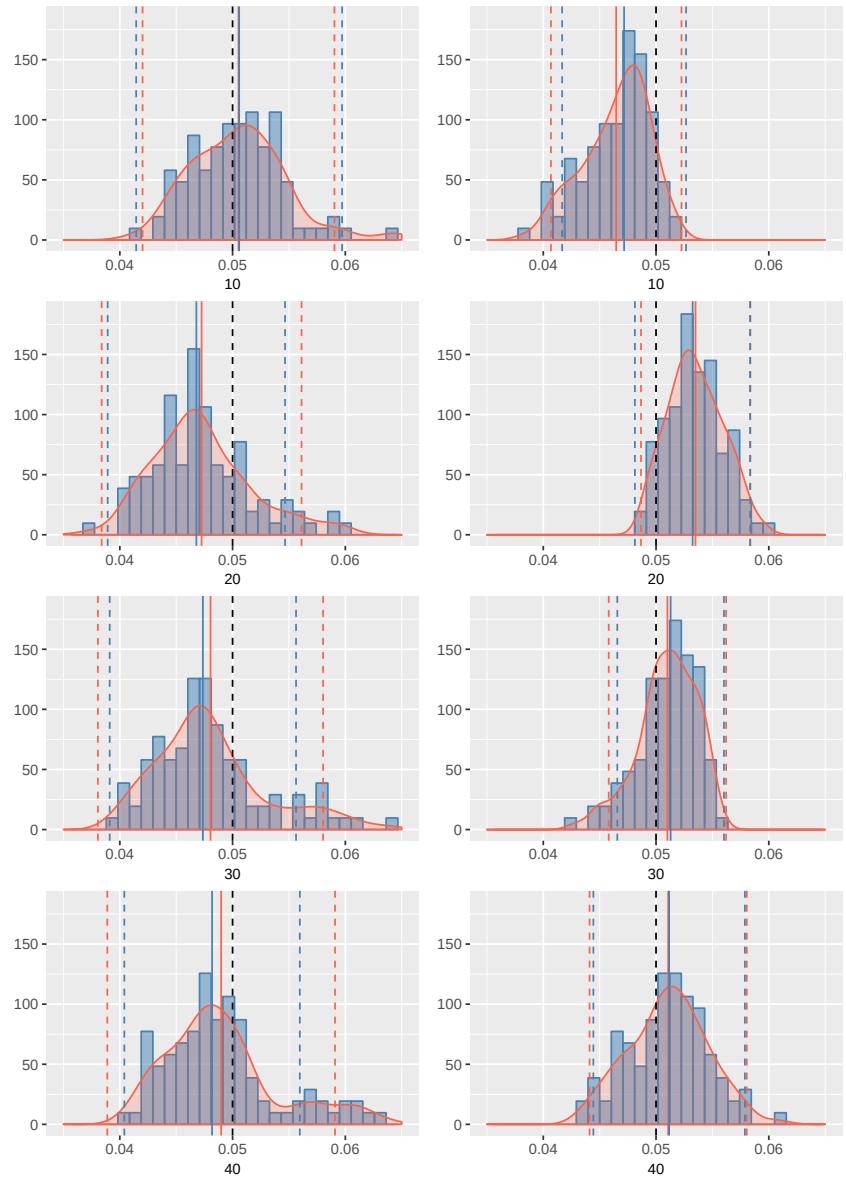
Let us look deeper and compare the results for pseudorandom and quasirandom numbers by the distribution (histograms) of ρ estimates in the case of $64K$ sample-paths in Figure 3.20. The distribution looks like a normal distribution with a clear mean, median and mode approximately equal to 0.05 for the case of 10 options baskets and PRNG. However, there is a tendency for outliers to appear with the growth of baskets size. The most clear tail with outliers has the basket with 40 options, the maturities $T = 30, 60, 90, 120$ and the strikes $K = 40, 42, ..., 58$. In the case of QRNG, the concentration of mass tends to be tighter. The median/mean central metrics are skewed more for small baskets, while there is almost no bias and the majority of results concentrate around the real $\rho = 0.05$.

Note, there is no tails in the case of QRNG, which is formed by outliers (fails of convergence, in fact). That makes the results based on quasi-Monte Carlo more reliable.

Number of calibration repeats

Another factor that could influence analysis results is a number of calibration procedures repeats. For most of the tests, 50 was chosen as sufficient, being limited by the performance of the Monte Carlo method and the necessity of a large number of various experiments. The illustration of convergence is depicted in Figure 3.21, where the median of cumulative distribution of results is given. This allows us to see how adding each calibration result changes the median. It is clear that, after 40 calibration repeats, relative convergence is observed. The settings are $64K$ sample-paths and various KT-baskets according to the Table 3.6. In these circumstances, 50 repeats looks like a fair trade-off between the reliability of results and the computation time of the array of tests, while it can be useful to increase the number of calibration repeats in practical cases.

[43] In the case of QRNG, the method of Gaussian numbers generation — as mentioned earlier — is also a crucial factor affecting the quality of calibration. The Box-Muller approach in combination with quasirandom numbers may provide unpredictably poor results.



(a) $64K(PRNG)$ (b) $64K(QRNG)$

Figure 3.20: Distribution of 100 calibration repeats results for 10, 20, 30, 40 options in a basket and $64K$ paths. AMSM2 model with $\rho^{real} = 0.05$. Red lines – Mean, $\pm 2SD$; blue lines – Median, $\pm 2SD$; black line – real parameter value.

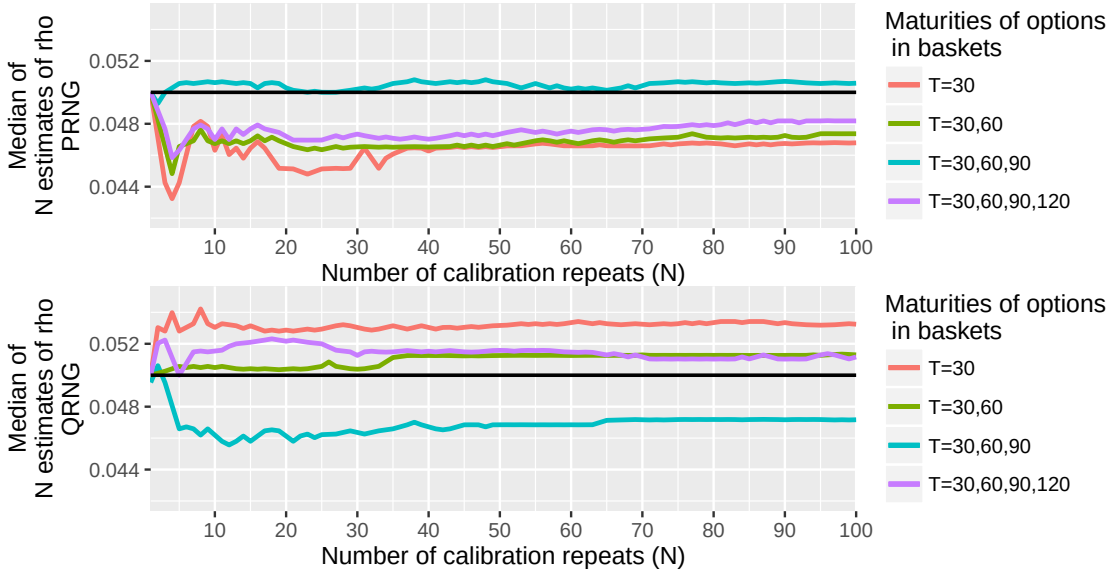


Figure 3.21: AMSM2 model with $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$, KT-baskets with 10 to 40 options. X-axis is a number of calibration results used for the median calculation (N), Y-axis is a median of N calibration results of ρ parameter in the case of PRNG/QRNG.

Parameters' scale

In this experiment, the sensitivity of calibration quality depending on a scale of leverage parameter ρ is tested. The settings are: AMSM2 model; LM optimization algorithm; KT-basket with maturities $T = 30, 60, 90, 120$. During the experiment, the AMSM2 model is calibrated with the true values of $\rho = 0.01, 0.02, 0.03, 0.04, 0.05$, while other two parameters are $\theta^{real} = (m_0, \sigma_0) = (1.4, 0.02)$. The results are plotted in Figure 3.22.

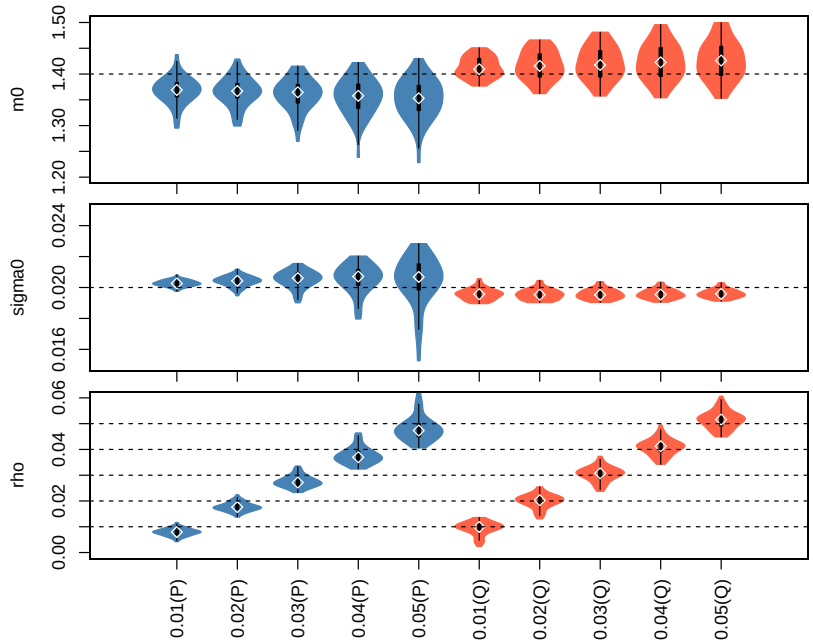


Figure 3.22: Sensitivity of calibration experiment's distribution depending on the size of ρ for Levenberg-Marquardt optimization method and AMSM2 model.

The result of the experiment demonstrates that the calibration procedure turns out to be even more stable in the case of smaller ρ. Additionally quasi-Monte Carlo (red violins on the plot) reaffirms its excellence, being less biased, especially with respect to the parameter ρ, as well as less deviating. The best results in the experiments are achieved for $\rho = 0.01$ and QMC.

3.4.3. Preliminary calibration results for λ fixed and $\nu = 0$ case.

In this subsection, the results of calibration experiments based on two optimization methods are presented: the local gradient-based Levenberg-Marquardt (LM) method and the global stochastic Adaptive Simulated Annealing (ASA) method.

The settings for the following experiments are similar to the settings in Section 3.4.2: $64K$ AMSM paths; the AMSM1 model parameters $\theta^{\text{real}} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.25)$; the AMSM2 model parameters $\theta^{\text{real}} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$; $\hat{k} = 5$, $S_0 = 50$, $r = 0.00018$, risk-neutralization parameter $\lambda = 0.51$[44] and $\nu = 0$[45]. The option prices data set consists of: four K-baskets with 10 options each, the corresponding four maturities are 30, 60, 90, 120; three KT-baskets with the corresponding maturities $T = 30, 60$, $T = 30, 60, 90$, $T = 30, 60, 90, 120$. The maximum number of objective function (WRSS from (3.48)) evaluations is limited to 1000 times, the initial point of optimization method is $[1.5, 0.05, 0.05]$, the search region lower boundary is $[1.05, 0.001, 0.000001]$, the upper boundary is $[1.85, 0.1, 1.0]$.

All of the collected results were grouped by experiments settings and prepared for an analysis, which was conducted using R programming language and software environment for statistical computing and visualizations. Diverse plots assisted to estimate the general precision and accuracy of the calibration techniques for different settings in a reader-friendly manner. Another stricter tool of analysis are the error metrics collected in Tables 3.10,3.11: mean and median value, Median Absolute Deviation (MAD) and Mean Absolute Error (MAE).

In order to manage development and according to the idea of "reproducible research", the project (later, the whole thesis project) including C++ application and its code, R scripts, experiments data and LaTeXtext files were placed in the Git repository. This allows us to present the project as a solid object of text, code, data and their change history. Therefore, this made it possible to reproduce the full computational environment used to obtain the results according to the concept of reproducible research.

The structure and denotations of the calibration experiments in this subsection are collected in Table 3.9. For instance, the calibration experiment "30(P)" consists of calibration of the AMSM1/AMSM2 model parameters based ASA/LM[46] optimization method repeated 50 times for different seeds using the prices of K-baskets with 10 options with the maturity $T = 30$, the pseudorandom number used for Monte Carlo integration.

ASA method

In this section, K-baskets of the same size (10 options), composed from the options with common maturity, but different strikes (40, 42,...,58), and KT-baskets with various maturities in the same basket, for example the basket "30-90" consists of Call options with the

[44] The choice of λ is discussed in more details in Section 3.4.2 and 3.5.

[45] The simplified version is considered in this section.

[46] The experiments were repeated for both model versions and both optimization methods in order to have comparable results.

Calibration Experiment	Number of Options	Maturity	Random Number Generator	Basket Type
30(P)	10	30	PRNG	K-basket
30(Q)	10	30	QRNG	K-basket
60(P)	10	60	PRNG	K-basket
60(Q)	10	60	QRNG	K-basket
90(P)	10	90	PRNG	K-basket
90(Q)	10	90	QRNG	K-basket
120(P)	10	120	PRNG	K-basket
120(Q)	10	120	QRNG	K-basket
30,60(P)	20	30,60	PRNG	KT-basket
30,60(Q)	20	30,60	QRNG	KT-basket
30-90(P)	30	30,60,90	PRNG	KT-basket
30-90(Q)	30	30,60,90	QRNG	KT-basket
30-120(P)	40	30,60,90,120	PRNG	KT-basket
30-120(Q)	40	30,60,90,120	QRNG	KT-basket

Table 3.9: Calibration experiment's structure. It consists of 50 repeated calibrations based on pseudo- and quasi-Monte Carlo and Adaptive Simulated Annealing method with $64K$ sample-paths for each option price evaluation. The real parameters values are $\theta^{AMSM1} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.25)$.

prices $C_1, C_2, \ldots, C_{30}$ (maturities 30, 60, 90) according to Tables 3.6,3.9, are used. In all experiments in this section, $64K$ sample-paths based on pseudo- and quasirandom numbers are used for a computation of each option price.

AMSM1 model

The results of calibration based on Adaptive Simulated Annealing (ASA) method are visualized in Figure 3.23. They spread into two groups visually, corresponding to the group of K-baskets based results on the left and KT-baskets based results on the right. This subdivision is mostly characterized by results for ρ. Namely, the calibrations based on K-baskets have huge variation (low precision) for ρ and also for m_0. Meanwhile, the accuracy is moderate for K-baskets, but this fact does not allow us to speak seriously about K-baskets and AS, because of the terrible precision of the results for this combination. Besides, there is a clear difference between PRNG- and QRNG-based results. The calibration procedure fails in the case of pseudorandom numbers for m_0 – the results are both imprecise and biased, especially for the higher maturities.

The only reliable choice is the combination of KT-baskets and quasi-Monte Carlo; distribution of calibration results is concentrated densely around the real parameters' values $(1.4, 0.02, 0.25)$ characterized by slight overestimation of ρ for the baskets with $T = 30-90$ and m_0 for the basket $T = 30-120$. These results are measured in more detail with the different metrics and collected in Table 3.10. An analysis of them shows that MAD (precision metric) and MAE (accuracy metric) of KT-baskets are around two times lower for m_0 and σ_0, and around $2-5$ times lower in the case of ρ. The best performance is reached for the basket with maturities $T = 30-120$.

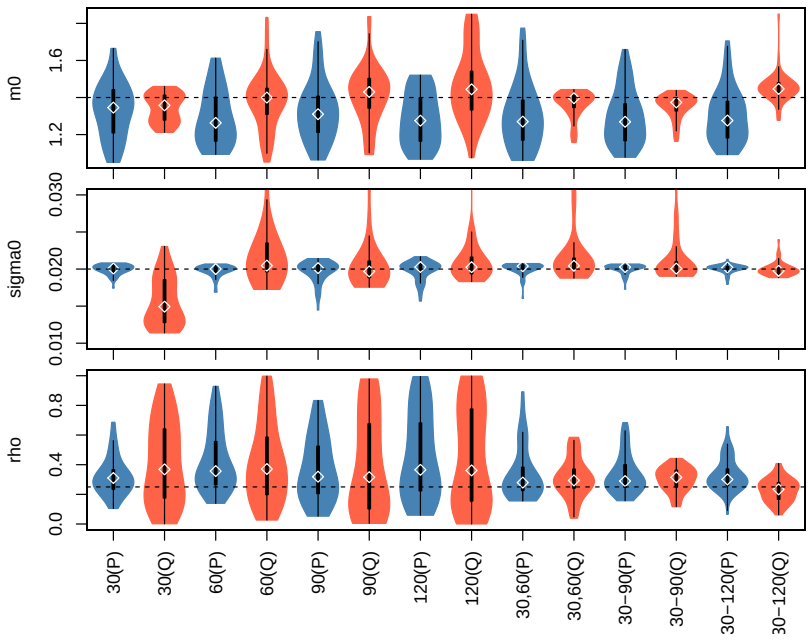


Figure 3.23: Calibration experiment's distribution for ASA and AMSM1 model. $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.25)$.

AMSM2 model

The results of the ASA optimization method for AMSM2 model are visualized in Figure 3.24. The first two violins (the experiments $30P$ and $30Q$) on all three plots (for all three parameters) show the worst performance over all K- and KT-baskets and the AMSM2 model; all three calibration results are significantly biased from the real values $(m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$ significantly. The results for the rest of the three K-baskets are also significantly biased (especially in the case of maturity 90) for the case of pseudorandom numbers. The results based on quasirandom numbers and the K-baskets with $T = 90$, $T = 120$ (experiments $90Q$ and $120Q$ correspondingly) are exceptions in the sense of accuracy not being biased, but they have very poor precision for m_0 and ρ. Similarly, the calibration experiments based on KT-baskets and the pseudo-Monte Carlo method demonstrate $(30, 60(P), 30-90(P), 30-120(P))$ less biased results, but are also very noisy and imprecise. The best results are achieved for KT-baskets and quasi-Monte Carlo $(30, 60(Q), 30-90(Q), 30-120(Q))$: moderate precision for m_0 and ρ, low precision for σ_0, and, in fact, no bias.

In addition, the results of quasi-Monte Carlo can be analyzed according to the values of errors metrics in Table 3.11. An inspection of the table provides the proof of the increasing quality of results for the larger KT-baskets, with the best choice being the KT-basket with the maturities $T = 30, 60, 90, 120$. The MADs values of (m_0, σ_0, ρ) are $(0.083, 0.000471, 0.0082)$; MAEs values are $(0.1054, 0.000821, 0.1936)$.

Levenberg-Marquardt (LM) method

The Levenberg-Marquardt (LM) optimization algorithm construction is especially suitable for a minimization of Weighted Residual Sum of Squares (3.48). Its search region was set up with the upper boundary $(1.85, 0.1, 0.1)$; the lower boundary $(1.05, 0.0001, 0.000001)$; and the initial point is $(1.5, 0.05, 0.5)$ for AMSM1 and $(1.5, 0.05, 0.05)$ for AMSM2 models. The real values of parameters are $(1.4, 0.02, 0.25)$ and $(1.4, 0.02, 0.05)$ for AMSM1 and AMSM2

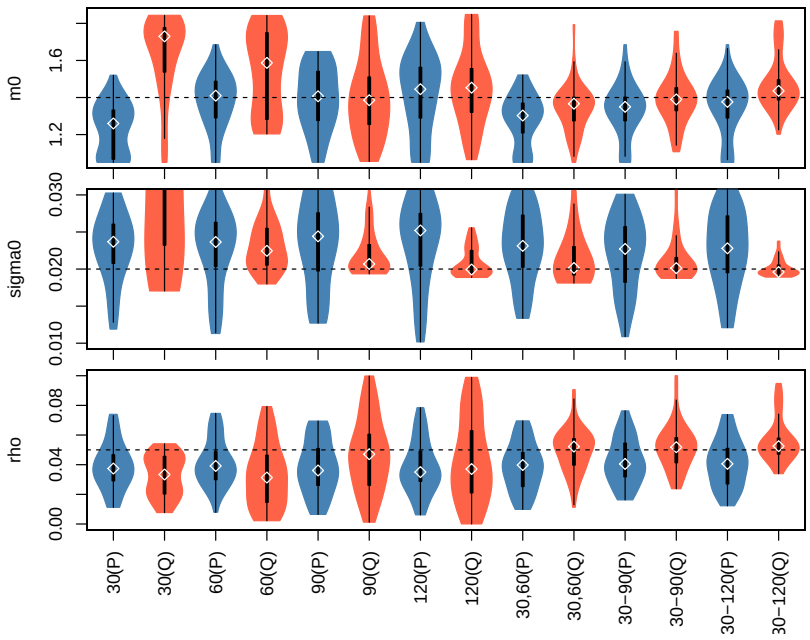


Figure 3.24: Calibration experiments distributions for ASA and AMSM2 model. $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$.

models, correspondingly. The results in the subsection are provided according to the calibration experiments structure in Table 3.9.

AMSM1 model

An inspection of Figure 3.25 plotted in the same scale as Figure 3.23, 3.24 shows that the Levenberg-Marquardt method provides significantly less noisy results. The least biased results are achieved for KT-baskets baskets and quasi-Monte Carlo as well as in the case of the ASA method. On the other hand, the calibration experiments based on LM systematically overestimate ρ for almost all baskets. The only exception is $30-120(Q)$, but it provides biased results for m_0. K-baskets are all more biased than their KT-counterparts; moreover, the calibration procedure merely sticks to the initial point in the case of $T = 120$.

The precision of the results in Figure 3.25 is characterized by a smaller range of dispersion of the experiment results' distribution for K-baskets compared to the ASA case, excepting the case of experiment $30(Q)$. Table 3.10 confirms this fact with error metrics; MAD of m_0 for the LM results is $2-4$ times lower than for ASA, and MAE is also lower. In the case of KT-baskets, the ASA and LM optimization methods have similar accuracy and precision for m_0, but the LM calibration results are significantly worse for the ρ; its MAD and MAE metrics are almost two times higher than their ASA counterparts.

In general, the distinguishing tendency of the experiments based on the LM and AMSM1 model is overestimation of ρ for all baskets.

AMSM2 model

The visualization of calibration experiments for the LM method and AMSM2 model is presented in Figure 3.26, according to the calibration experiments structure in Table 3.9. It observes uniform growth of the upward bias of m_0, σ_0 and the downward bias of ρ calibration results with the growth of maturity in K-baskets and pseudo-Monte Carlo compu-

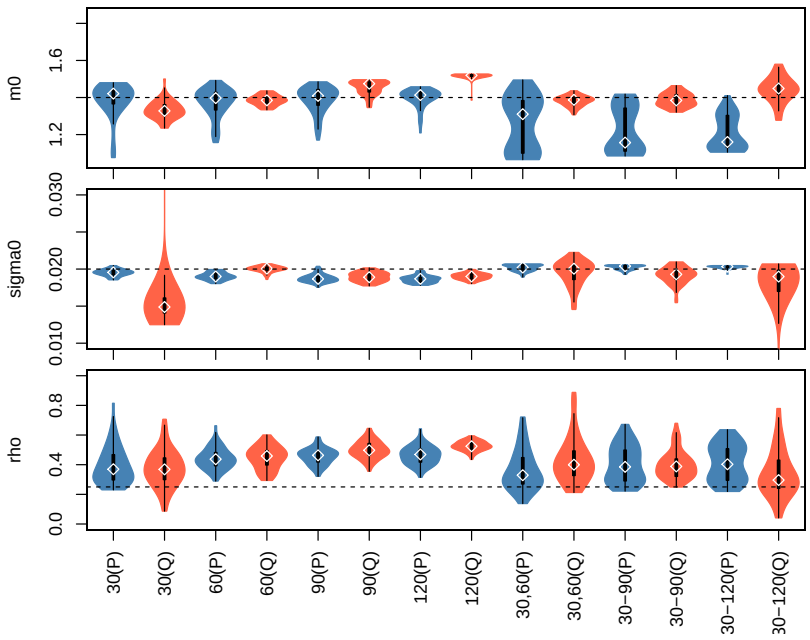


Figure 3.25: Calibration experiment's distribution for LM and AMSM1 model. The dashed lines are $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.25)$.

tations. In the case of quasi-Monte Carlo approach, there is the only significant bias of m_0 and ρ – the case of K-basket with maturity $T = 60$. In the case of KT-baskets, biases are lower for pseudo-Monte Carlo, but again quasi-Monte Carlo appears clearly preferable in this aspect. In the sense of precision the quasi-Monte Carlo wins as well, especially for the σ_0 parameter.

An inspection of Table 3.11 shows that the best results in the sense of accuracy and precision are obtained in the calibration experiments $30-90(Q)$, $30-120(Q)$ and $30(Q)$. The estimates are concentrated around the real values $(1.4, 0.02, 0.05)$: MAD and MAE equal to $0.002-0.004$ for ρ; the estimates of σ_0 are perfect in both accuracy and precision with MAD from 5.11×10^{-6} to 3.61×10^{-4}; MAE is not larger than 4.26×10^{-4}, MAD and MAE are from 0.0108 to 0.0432 for m_0. Note also, the parameter ρ is underestimated for all K-baskets and overestimated for the KT-baskets, especially in the case $T = 60$. To summarize, KT-baskets look to be a more favorable choice.

3.4.4. Comparative analysis and intermediate conclusions

The dependence structure of parameters estimates is explored deeper for K- and KT-baskets in the case of the AMSM1/AMSM2 model and ASA/LM optimization method (correspondingly) using Figures 3.27a, 3.27b. Each scatter plot in these figures depicts all 50 calibration results in each corresponding experiment $(30(Q),\ 30-60(Q),\ 30-90(Q)$ and $30-120(Q))$ according to the calibration experiments structure in Table 3.9. The figures collect the scatter plots of calibrated parameters values m_0 (1st row) and σ_0 (2nd row) against ρ, the dotted lines are real values of the parameters, namely $\theta^{\text{real}} = (1.4, 0.02, 0.25)$ for AMSM1 model and $\theta^{\text{real}} = (1.4, 0.02, 0.05)$ for the AMSM2 model. For example, the results of the calibration experiment $30-90(Q)$ for AMSM1 model based on ASA optimization procedure and consisting of 50 repeated calibrations are depicted in the third column of

Calibration Experiment	$30(Q)$	$60(Q)$	$90(Q)$	$120(Q)$	$30,60(Q)$	$30-90(Q)$	$30-120(Q)$
				m_0 (ASA)			
Mean	1.3443	1.3783	1.4092	1.4640	1.3657	1.3542	1.4524
Median	1.3576	1.3997	1.4299	1.4441	1.3962	1.3736	1.4463
MAD	0.0996	0.0974	0.1134	0.1482	0.0432	0.0396	0.0412
MAE	0.0734	0.1064	0.1162	0.1480	0.0493	0.0517	0.0650
				m_0 (LM)			
Mean	1.3327	1.3854	1.4536	1.5153	1.3818	1.3878	1.4411
Median	1.3277	1.3847	1.4745	1.5197	1.3873	1.3844	1.4487
MAD	0.0447	0.0266	0.0276	0.0067	0.0247	0.0415	0.0480
MAE	0.0725	0.0249	0.0600	0.1159	0.0251	0.0327	0.0665
				σ_0 (ASA)			
Mean	1.55e-02	2.23e-02	2.11e-02	2.10e-02	2.18e-02	2.10e-02	1.99e-02
Median	1.49e-02	2.05e-02	1.97e-02	2.03e-02	2.04e-02	2.01e-02	1.98e-02
MAD	3.87e-03	1.52e-03	1.44e-03	1.50e-03	8.54e-04	7.70e-04	5.65e-04
MAE	4.77e-03	2.98e-03	2.36e-03	1.64e-03	2.00e-03	1.47e-03	6.35e-04
				σ_0 (LM)			
Mean	1.59e-02	2.00e-02	1.89e-02	1.90e-02	1.95e-02	1.92e-02	1.72e-02
Median	1.49e-02	2.01e-02	1.89e-02	1.90e-02	2.00e-02	1.93e-02	1.90e-02
MAD	1.63e-03	3.28e-04	7.02e-04	4.91e-04	1.26e-03	8.78e-04	1.79e-03
MAE	5.33e-03	3.08e-04	1.07e-03	9.95e-04	1.28e-03	1.07e-03	2.87e-03
				ρ (ASA)			
Mean	0.4090	0.4104	0.4003	0.4513	0.3089	0.3004	0.2282
Median	0.3685	0.3707	0.3159	0.3613	0.2937	0.3145	0.2334
MAD	0.3523	0.2765	0.3944	0.4251	0.0830	0.0866	0.0896
MAE	0.2508	0.2352	0.2899	0.3125	0.1079	0.0803	0.0641
				ρ (LM)			
Mean	0.3733	0.4470	0.5005	0.5258	0.4294	0.3991	0.3391
Median	0.3683	0.4585	0.4972	0.5243	0.3997	0.3884	0.2962
MAD	0.1135	0.0778	0.0516	0.0359	0.1231	0.0913	0.1163
MAE	0.1423	0.1970	0.2505	0.2758	0.1830	0.1492	0.1279

Table 3.10: Calibration experiment's results for quasi-Monte Carlo and AMSM1 model case. The real parameters values are $\theta^{AMSM1} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.25)$.

Calibration Experiment	$30(Q)$	$60(Q)$	$90(Q)$	$120(Q)$	$30,60(Q)$	$30-90(Q)$	$30-120(Q)$
				m_0 (ASA)			
Mean	1.6355	1.5429	1.3938	1.4541	1.3425	1.3907	1.4736
Median	1.7300	1.5868	1.3854	1.4530	1.3665	1.3897	1.4357
MAD	0.1096	0.2778	0.1928	0.1651	0.0696	0.0931	0.0830
MAE	0.2927	0.2281	0.1720	0.1645	0.0956	0.0984	0.1054
				m_0 (LM)			
Mean	1.4234	1.4672	1.4293	1.3857	1.3726	1.3895	1.4263
Median	1.4122	1.4633	1.4324	1.3869	1.3707	1.3925	1.4262
MAD	0.0108	0.0186	0.0252	0.0170	0.0232	0.0337	0.0432
MAE	0.0239	0.0672	0.0327	0.0177	0.0293	0.0234	0.0363
				σ_0 (ASA)			
Mean	3.51e-02	2.32e-02	2.20e-02	2.08e-02	2.20e-02	2.13e-02	2.01e-02
Median	3.60e-02	2.25e-02	2.07e-02	1.99e-02	2.01e-02	2.01e-02	1.96e-02
MAD	1.73e-02	3.40e-03	1.40e-03	1.07e-03	1.78e-03	1.32e-03	4.71e-04
MAE	1.57e-02	3.48e-03	2.16e-03	1.40e-03	2.85e-03	1.73e-03	8.21e-04
				σ_0 (LM)			
Mean	2.02e-02	2.05e-02	2.01e-02	1.98e-02	1.98e-02	2.02e-02	1.96e-02
Median	2.00e-02	2.04e-02	2.01e-02	1.98e-02	1.98e-02	2.01e-02	1.96e-02
MAD	5.11e-06	3.22e-04	2.12e-04	9.16e-05	2.74e-04	3.61e-04	2.92e-04
MAE	1.96e-04	4.63e-04	1.79e-04	1.77e-04	2.60e-04	3.09e-04	4.27e-04
				ρ (ASA)			
Mean	0.0324	0.0328	0.0470	0.0426	0.0497	0.0516	0.0564
Median	0.0334	0.0313	0.0471	0.0371	0.0522	0.0518	0.0525
MAD	0.0184	0.0242	0.0283	0.0288	0.0127	0.0127	0.0082
MAE	0.2176	0.2172	0.2030	0.2074	0.2003	0.1984	0.1936
				ρ (LM)			
Mean	0.0462	0.0431	0.0456	0.0466	0.0535	0.0512	0.0514
Median	0.0468	0.0428	0.0455	0.0464	0.0532	0.0513	0.0516
MAD	0.0028	0.0013	0.0008	0.0016	0.0026	0.0020	0.0028
MAE	0.0039	0.0069	0.0044	0.0034	0.0036	0.0020	0.0030

Table 3.11: Calibration experiment's results for quasi-Monte Carlo and AMSM2 model case. The real parameters values are $\theta^{AMSM2} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$.

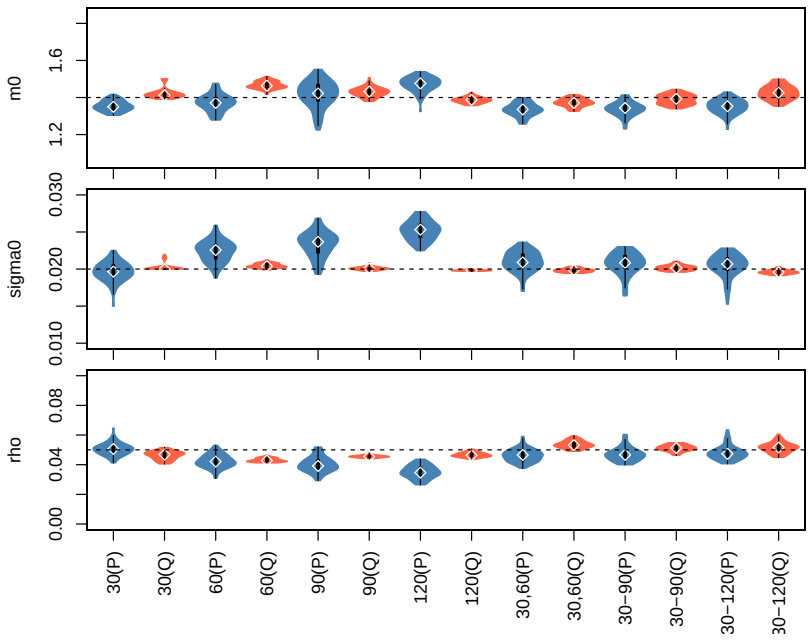


Figure 3.26: Calibration experiment's distribution for LM and AMSM2 model. The dashed lines are $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$.

Figure 3.27a (3rd and 7th plots).

In order to visualize a correspondence between the first and the second rows of plots the clustering procedure[47] was ran to colorize groups of close results. Three groups of calibration results are selected in each experiment; each of these groups is colored with three blue color tones. So, the results of experiment $30(Q)$ on the top (m_0 w.r.t ρ) and the bottom (σ_0 w.r.t ρ) plots form the line. The first group (light blue) forms the bottom part of the line, the second group (blue) is in the middle, and the third group in the top of the line.

There is a clear tendency in the scatter patterns with the growth of baskets size (from 10 to 40) and the number of various maturities (from the only $T = 30$ to the set $T = 30, 60, 90, 120$). There are clear linear dependence patterns in the $30(Q)$ experiment, while there is more or less ellipsoidal patterns in the case of the $30-120(Q)$ experiments. The results for the AMSM1 model deviates more significantly; also note that the plots between σ_0 and ρ are less scattered for both models. These dependencies are the result of volatility definition in the models. It is especially clear for AMSM2 model, where the volatility is given by

$$\sigma(m_0, \sigma_0, \rho)_t = \left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2 \left(\prod_{i=1}^{\hat{k}} M_{i,t}\right)^{\frac{1}{2}}, \tag{3.49}$$

where ρ and σ_0 play a similar role. Another tendency is the movement of results' mass closer to the real values for the experiments from $30(Q)$ to $30-120(Q)$, according to Table 3.9. Also note, there are few outliers in the case of K-basket and AMSM2 model. These are fails of convergence of the calibration procedure, where it got stuck in the initial point of optimization procedure.

The negative dependence of ρ and m_0, observed for the violin plots in Figures 3.23 and 3.26 (for K-baskets), is visualized now in the first row of Figure 3.27a and Figure 3.27b (the

[47] Different methods from the *Mclust* R package.

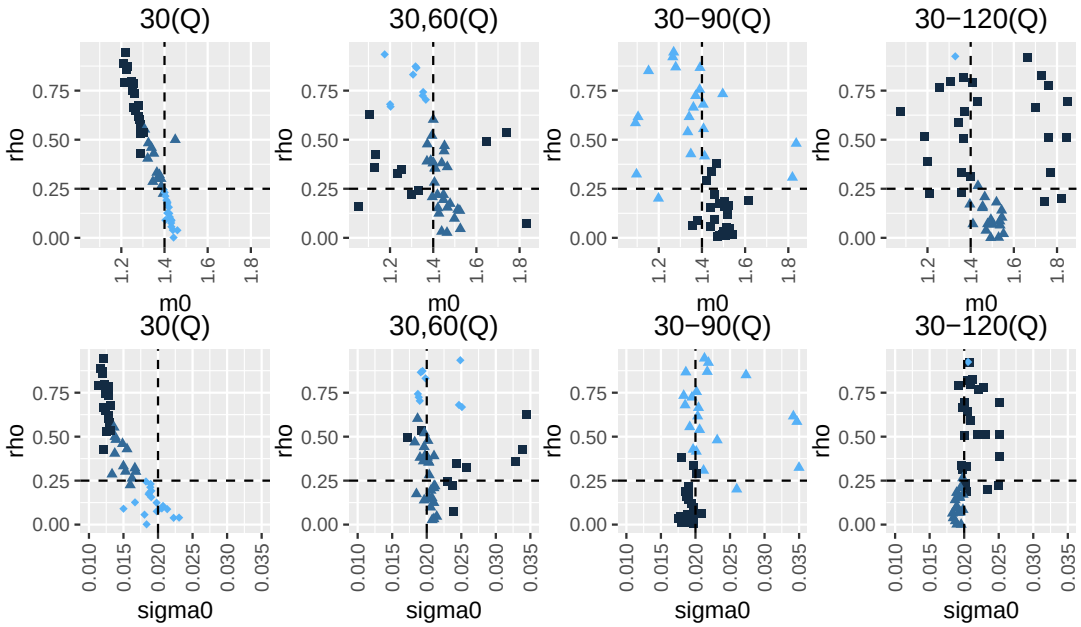
experiment 30(Q)). In order to compensate for the higher value of m_0 in (3.49) the value of ρ should be smaller (up to zero). This is what can be observed for the LM optimization procedure case and experiment 60(Q) (for example) in Figure 3.26 and ASA for the K-basket experiment 90(Q) in Figure 3.24. The values of the results just move along a certain curve on the plots, especially for the couple σ_0/ρ and the experiments 30(Q) (both models) and 30, 60(Q) (AMSM2 model). If one parameter is biased, then another one also gets biased. Notably, the effect is caused by an increase of the number of maturities in baskets. It allows us to remove or at least to decrease this dependency. There is a relatively dense cloud of results in experiments $30-120(Q)$ for AMSM1 with a lower number of outliers than for other experiments fewer maturities. For the AMSM2 model case, an absence of outliers observed, plus the round cloud for m_0/ρ and the thin ellipse for σ_0/ρ), but some bias from the crosslines of real parameters value exists.

An inspection of Table 3.10 shows the superiority of the Adaptive Simulated Annealing optimization method in combination with quasi-Monte Carlo and KT-baskets in the case of the AMSM1 model. The error metrics of the ASA results are comparable to its LM counterparts in the case of m_0 estimates, but they are better for σ_0 by up to 2.5 times. Finally, the most convincing advantage is achieved for ρ. The favorable basket is the KT-basket with maturities $30-120$ trading days. Meantime, the calibration experiments based on quasi-Monte Carlo option pricing in combination with the Levenberg-Marquardt optimization method turned out vastly superiority for the AMSM2 model. The MAD errors of the LM-based approach are lower by a factor of ten for most baskets compared to their ASA counterparts. The MAE errors are even lower by two orders of magnitude. This is a great improvement compared to ASA in the case of the AMSM2 model.

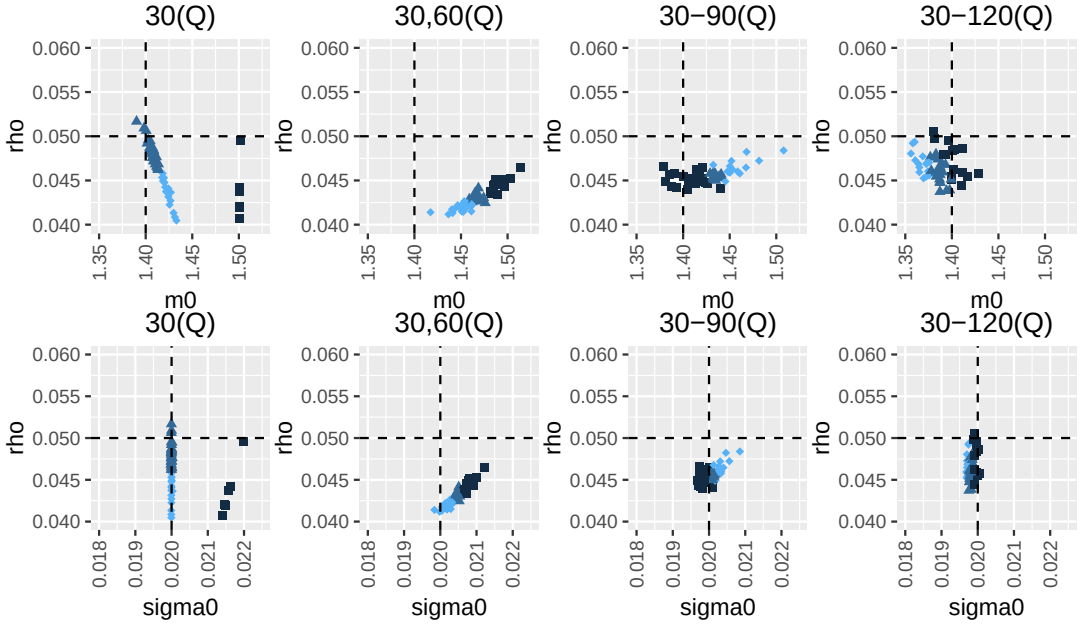
The next subsection is aimed at considering an estimation of the equity unit-risk premium λ in addition to the model parameters $\theta = (m_0\sigma_0, \rho)$.

3.5. Calibration and Estimation of AMSM model parameters and equity unit-risk premium based on option and asset prices

As was mentioned earlier in Section 3.4.2, an Equity Risk Premium (ERP) rough estimation range is usually from 0.2 to 0.8. Let us look deeper, through Aswath Damodaran's manuscript [26], which gives a comprehensive review on the topic of ERP. The author distinguishes three approaches for estimating it: a survey approach, a historical data approach and an implied approach. The first one is organized as an experts' sentiments survey about the value of the risk premium. The most challenging part of this approach is to define a representative group of experts. The second one is investigating historical values of returns and subtracting what is considered as a risk-free rate in order to determine ERP. The key question is to define the time-horizon of historical data and what to use as the risk-free asset, especially in the case of emerging markets. Finally, the third approach is forward-looking and aimed at extracting of ERP from expected cash flows or an implied volatility. The first two approaches do not take market believes into consideration, while the implied volatility approach does. Additionally, these methods can yield different values of ERP. Therefore, there is a question as to which approach to prefer. All these facts make an



(a) The results of calibration for KT-baskets, ASA optimization method, AMSM1 model version. The real values $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.25)$.



(b) The results of calibration for KT-baskets, LM optimization method, AMSM2 model version. The real values $\theta^{real} = (m_0, \sigma_0, \rho) = (1.4, 0.02, 0.05)$.

Figure 3.27: Scatter of calibration results for the experiments $30(Q)$, $30-60(Q)$, $30-90(Q)$ and $30-120(Q)$. The dotted lines are the real values of parameters.

estimation of ERP a tricky task. Therefore, it would be more feasible to estimate the equity unit equity risk premium λ (EURP[48]) jointly with the model parameters (m_0, σ_0, ρ). The joint estimation allows us to obtain a consistent value of λ.

The methods of joint estimation of model parameters and EURP in this section are based on the maximum likelihood estimation method. So, let us begin with construction of various likelihood functions that are used further for estimation purposes.

3.5.1. Likelihood functions

There are two main data sources for statistic inference concerning model parameters and EURP: a historical data about underlying asset returns and options price data. They allow us to construct various likelihood functions.

Asset returns data case

The likelihood functions of AMSM models are similar to the conventional MSM model case considered by Calvet & Fisher [20], but need to be modified due to another definition of the log-return process $\{r_t\}$ (3.37), namely it has the leverage effect modeling modifications of the volatility defined in (3.17), (3.18). Another reference is Hamilton (see [43], [44]), in which the author describes how to construct a likelihood function for Markov Switching processes in detail.

Let us begin with the log-returns $r_t = \log S_t / S_{t-1}$ for the AMSM1 model given by

$$\begin{aligned} r_t = \mu_t + \sigma_t \epsilon_t = r_f + \lambda \sigma_0 \sigma_t^M(\rho, m_0) \\ - \frac{1}{2}\left(\sigma_0 \sigma_t^M(\rho, m_0)\right)^2 \\ + \sigma_0 \sigma_t^M(\rho, m_0)\epsilon_t, \end{aligned} \tag{3.50}$$

The log-returns $r_t = \log S_t / S_{t-1}$ for the AMSM2 model are given by

$$\begin{aligned} r_t = \mu_t + \sigma_t \epsilon_t = r_f + \lambda\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2 \sigma_t^M(m_0) \\ - \frac{1}{2}\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2 \left(\sigma_t^M(m_0)\right)^2 \\ + \left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2 \sigma_t^M(m_0)\epsilon_t, \end{aligned} \tag{3.51}$$

where the sum of the risk-free rate r_f and the equity risk premium $\lambda\sigma_t$ is an expected return of asset $\{S_t\}$, $\theta = (m_0, \sigma_0, \rho)$ denotes the vector of AMSM model parameters in both cases.

It is easy to see that r_t is normally distributed conditionally on $\mathscr{F}_{t-1}$ and M_t with mean μ_t and variance σ_t^2, taking into account σ_t is $\mathscr{F}_{t-1}$-measurable and the definition of the Markov process $\{M_t\}$. The hidden Markov process $\{M_t\}$ has a finite number of states due to the binomial nature of its components, namely $d = 2^{\hat{k}}$. So, each state of $\{M_t\}$ is denoted by $m^i = \left\{m_1^i, \ldots, m_{\hat{k}}^i\right\}$, where $i = 1, \ldots, d$. Therefore, it is possible to distinguish the states of volatility process σ_t^M itself. The state of σ_t^M depends on M_t in the i-state, namely m^i, therefore, it will be denoted as σ_t^i. In general, the returns density is given by

$$\omega_t^{ij} = f\left(r_t \middle| M_t = m^j, r_{t-1}, \sigma_{t-1}^i, \ldots, r_0, \sigma_0; \theta, \lambda\right) \tag{3.52}$$

[48] An equity risk premium (ERP) is $\lambda \times \sigma$ in the AMSM model, therefore λ is an equity unit-risk premium (EURP) measured in units of volatility.

AMSM1:

$$\omega_t^j = \frac{1}{\sqrt{2\pi}\sigma_t^j} \exp\left(-\frac{1}{2}\left(\frac{r_t - \mu_t}{\sigma_t^j}\right)^2\right) \tag{3.53}$$

$$\mu_t = r_f + \lambda\sigma_t^j - \frac{1}{2}\left(\sigma_t^j\right)^2, \tag{3.54}$$

$$\sigma_t^j = \sigma_0 \left(\prod_{k=1}^{\hat{k}} m_k^j\right)^{1/2} \tag{3.55}$$

AMSM2:

$$\omega_t^{ij} = \frac{1}{\sqrt{2\pi}\sigma_t^{ij}} \exp\left(-\frac{1}{2}\left(\frac{r_t - \mu_t}{\sigma_t^{ij}}\right)^2\right) \tag{3.56}$$

$$\mu_t = r_f + \lambda\sigma_t^{ij} - \frac{1}{2}\left(\sigma_t^{ij}\right)^2, \tag{3.57}$$

$$\sigma_t^{ij} = \left(\rho\epsilon_{t-1}^i - \sqrt{\sigma_0}\right)^2 \left(\prod_{k=1}^{\hat{k}} m_k^j\right)^{1/2} \tag{3.58}$$

where

$$\epsilon_{t-1}^i = \frac{r_{t-1} - r_f - \lambda\sigma_{t-1}^i + \frac{1}{2}\left(\sigma_{t-1}^i\right)^2}{\sigma_{t-1}^i}, \tag{3.59}$$

$$\sigma_0^{ij} = \sigma_0,\ i,j = 1,\ldots,d;\ d = 2^{\hat{k}}. \tag{3.60}$$

The density $\omega_t^{i,j}$ depends on σ_t^{ij} and σ_{t-1}^i (through ϵ_{t-1}), both are measurable w.r.t. $\mathscr{F}_{t-1}$ and are known at the moment t from the previous iteration.

The transition probabilities of $\{M_t\}$ from the state m^i to the state m^j are given for $t > 0$ by

$$p_{t-1}^{ij} = P\left(M_t = m^j \middle| M_{t-1} = m^i; \theta, \lambda\right) = \prod_{k=1}^{\hat{k}} \left[\left(1-\gamma_k\right) I\left\{m_k^i = m_k^j\right\} + \gamma_k P\left(m_k^j = m_0\right)\right], \tag{3.61}$$

AMSM1:

$$P\left(m_k^j = m_0\right) = \begin{cases} 1 - \Phi(\rho\epsilon_{t-1}^i), \text{ if } m_k^j = m_0 \\ \Phi(\rho\epsilon_{t-1}^i), \text{ if } m_k^j \neq m_0, \end{cases} \tag{3.62}$$

$$\epsilon_{t-1}^i = \frac{r_{t-1} - r_f - \lambda\sigma_{t-1}^i + \frac{1}{2}\left(\sigma_{t-1}^i\right)^2}{\sigma_{t-1}^i}; \tag{3.63}$$

AMSM2:

$$P\left(m_k^j = m_0\right) = 0.5, \tag{3.64}$$

where $\Phi(\cdot)$ is a CDF of the standard normal random variable. Note, p_t^{ij} depends on ρ and λ only in the case of the AMSM1 model.

The probability of certain state m^i at the moment t has to be defined later; it is denoted by

$$\Pi_t^i = P\left(M_t = m^i \middle| r_t, \sigma_t, \ldots, r_0, \sigma_0; \theta, \lambda\right). \tag{3.65}$$

As a result, we can construct a density function of returns process $\{r_t\}_{t=0}^{\infty}$ conditional on $\mathscr{F}_{t-1}$

$$f(r_t|r_{t-1}, \sigma_{t-1}, \ldots, r_0, \sigma_0; \theta, \lambda) = \sum_{i=1}^{d}\sum_{j=1}^{d} \Pi_{t-1}^i p_t^{ij} \omega_t^j \quad \text{(AMSM1)}, \tag{3.66}$$

$$f(r_t|r_{t-1}, \sigma_{t-1}, \ldots, r_0, \sigma_0; \theta, \lambda) = \sum_{i=1}^{d}\sum_{j=1}^{d} \Pi_{t-1}^i p_t^{ij} \omega_t^{ij} \quad \text{(AMSM2)}, \tag{3.67}$$

then Π_t can be defined as

$$\Pi_t^j = \frac{\omega_t^j \sum_{i=1}^{d} \Pi_{t-1}^i p_t^{ij}}{\sum_{j=1}^{d} \omega_t^j \sum_{i=1}^{d} \Pi_{t-1}^i p_t^{ij}} \quad \text{(AMSM1)}, \tag{3.68}$$

$$\Pi_t^j = \frac{\sum_{i=1}^{d} \Pi_{t-1}^i p_t^{ij} \omega_t^{ij}}{\sum_{j=1}^{d}\sum_{i=1}^{d} \Pi_{t-1}^i p_t^{ij} \omega_t^{ij}} \quad \text{(AMSM2)}. \tag{3.69}$$

Note, Π_t is calculated recursively. The initial value for it can be chosen as [20]

$$\Pi_0^j = \prod_{k=1}^{\hat{k}} P\left(M = m_k^j\right). \tag{3.70}$$

Finally, the log-likelihood function by using (3.66) is given by

$$L^R(\theta, \lambda|r_1, \ldots, r_T) = \sum_{t=1}^{T} \log\left(\sum_{j=1}^{d} \omega_t^j \sum_{i=1}^{d} \Pi_{t-1}^i p_t^{ij}\right) \quad \text{(AMSM1)}, \tag{3.71}$$

$$L^R(\theta, \lambda|r_1, \ldots, r_T) = \sum_{t=1}^{T} \log\left(\sum_{i=1}^{d}\sum_{j=1}^{d} \Pi_{t-1}^i p_t^{ij} \omega_t^{ij}\right) \quad \text{(AMSM2)}. \tag{3.72}$$

Note, the log-likelihood has the same general form as the MSM model of Calvet and Fisher, but the components ω_t, Π_t and P are defined differently.

The computation algorithm for the likelihood based on historical returns data is the following:

1. Calculate $1 \times \hat{k}$ vector of γ_k, $d \times \hat{k}$ matrix M of $\left\{m_k^j\right\}_{k=1,\ldots,\hat{k}}^{j=1,\ldots,d}$
2. Assign $\sigma_0^i = \sigma_0$, $\sigma_{-1}^i = \sigma_0$ for $i = 1, \ldots, d$
3. Initialize Π_0 as (3.70)
4. Repeat from $t = 1$

(a) Calculate $1 \times d$ vector σ_t by (3.50,3.51)

(b) Calculate $1 \times d$ vector ω_t (AMSM1) or $d \times d$ matrix ω_t (AMSM2) by (3.52)

(c) Calculate $d \times d$ matrix P_t by (3.61)

(d) Calculate $d \times 1$ vector Π_t by (3.68)

(e) Increment $L^R := L^R + L_t^R$, where L_t^R is a likelihood of r_t obtained from its density function (3.66)

5. Until $t = T$

6. Return the value of likelihood L^R from (3.71)

Option price data case

There is another approach for construction of likelihood function based on an option price data inspired by Christoffersen's papers [22, 23]. Let us consider an option price C_i from Table 3.6, where the computation error is given by

$$\epsilon_i = \frac{C_i^R - C_i^{AMSM}(\theta, \lambda, \nu)}{w_i}, \tag{3.73}$$

where C_i^R is a real market Call option price, C_i^{AMSM} is a theoretical price of the same option computed in the assumption the AMSM model holds, as the weights w_i can be used $w_i = C_i^R$ or $w_i = BSV_i$ (Black-Scholes Vega). In particular, Black-Scholes vega weights are used by Christoffersen in [22]. Note, the sum of these squared errors is the Weighted Residual Sum of Squares (WRSS) defined in (3.48).

Assume ϵ_i are i.i.d. normal, then the log-likelihood function can be defined as follows

$$L^O\left(\theta, \lambda, \nu | C_1^R, \ldots, C_N^R\right) = \sum_{i=1}^{N} \log\left(\exp\left(-\frac{\epsilon_i^2}{2\sigma_\epsilon^2}\right) \Big/ \sqrt{2\pi\sigma_\epsilon^2}\right) \approx -\frac{1}{2}\sum_{i=1}^{N}\left(\frac{\epsilon_i^2}{\sigma_\epsilon^2} + \log\sigma_\epsilon^2\right) \tag{3.74}$$

where N is a number of option prices, σ_ϵ^2 is a variance of the error with a sample estimate $\hat{\sigma}_\epsilon^2 = \sum_{i=1}^{N} \epsilon_i^2 \Big/ N$. Let us substitute σ_ϵ^2 in the last expression, thus

$$L^O\left(\theta, \lambda, \nu | C_1^R, \ldots, C_N^R\right) = -\frac{1}{2}\frac{\sum_{i=1}^{N}\epsilon_i^2}{\sum_{i=1}^{N}\epsilon_i^2} N - \frac{1}{2}\sum_{i=1}^{N}\log\left(\sum_{i=1}^{N}\epsilon_i^2 \Big/ N\right) = \tag{3.75}$$

$$-\frac{N}{2} - \frac{N}{2}\log\left(\sum_{i=1}^{N}\epsilon_i^2 \Big/ N\right) = \tag{3.76}$$

$$-\frac{N}{2}\left(1 + \log\left(WRSS \Big/ N\right)\right) \sim \tag{3.77}$$

$$-\frac{N}{2}\log\left(WRSS \Big/ N\right), \tag{3.78}$$

then the average log-likelihood is given by

$$l^O\left(\theta, \lambda, \nu | C_1^R, \ldots, C_N^R\right) = -\frac{1}{2}\log\left(WRSS \Big/ N\right).$$

Asset returns and option prices data case

Christoffersen [22], [23] and some other authors argue for the use of both returns and options prices data in order to obtain parameters estimates consistent with physical and risk-neutral measure simultaneously. In this case, the idea is to create a mixed likelihood function. In the simplest case, it can be defined as (see [22])

$$\begin{aligned} &L^M\left(m_0,\sigma_0,\rho,\lambda,\nu|r_T,\ldots,r_1,C_N^R,\ldots,C_1^R\right)= \\ &L^R\left(m_0,\sigma_0,\rho,\lambda|r_T,\ldots,r_1\right)+ \\ &L^O\left(m_0,\sigma_0,\rho,\lambda,\nu|C_N^R,\ldots,C_1^R\right), \end{aligned} \tag{3.79}$$

an alternative formulation includes weighting the number of data points used for a computation of L^R and L^O (see [23]) is given by

$$\begin{aligned} &L^M\left(m_0,\sigma_0,\rho,\lambda,\nu|r_1,\ldots,r_T,C_1^R,\ldots,C_N^R\right)= \\ &\frac{1}{T}L^R\left(m_0,\sigma_0,\rho,\lambda|r_1,\ldots,r_T\right)+ \\ &\frac{1}{N}L^O\left(m_0,\sigma_0,\rho,\lambda,\nu|C_1^R,\ldots,C_N^R\right), \end{aligned} \tag{3.80}$$

or

$$\begin{aligned} &L^M\left(m_0,\sigma_0,\rho,\lambda,\nu|r_1,\ldots,r_T,C_1^R,\ldots,C_N^R\right)= \\ &\frac{T+N}{2T}L^R\left(m_0,\sigma_0,\rho,\lambda|r_1,\ldots,r_T\right)+ \\ &\frac{T+N}{2N}L^O\left(m_0,\sigma_0,\rho,\lambda,\nu|C_1^R,\ldots,C_N^R\right), \end{aligned} \tag{3.81}$$

where T is a log-returns path length used for calculation of $L^R(\theta,\lambda)$, N is a number of option prices used for calculation $L^O(\theta,\lambda,\nu)$.

Another alternative is a sequential estimation of model parameters, namely m_0,σ,ρ,λ from an asset price data first, whereas the volatility risk-premium parameter ν is calibrated from an option price data

$$\underset{m_0,\sigma_0,\rho,\lambda}{\arg\max}\, L^R \Rightarrow m_0^\star,\sigma_0^\star,\rho^\star,\lambda^\star \Rightarrow \underset{\nu}{\arg\max}\, L^O \Rightarrow \nu^\star \tag{3.82}$$

or

$$\underset{m_0,\sigma_0,\rho,\lambda}{\arg\max}\, L^R \Rightarrow m_0^\star,\sigma_0^\star,\rho^\star,\lambda^\star \Rightarrow \min_{\nu} WRSS \Rightarrow \nu^\star \tag{3.83}$$

or

$$\underset{m_0,\sigma_0,\rho,\lambda}{\arg\max}\, L^R \Rightarrow \lambda^\star \Rightarrow \min_{m_0,\sigma_0,\rho,\nu} WRSS \Rightarrow m_0^\star,\sigma_0^\star,\rho^\star,\nu^\star. \tag{3.84}$$

3.5.2. Preliminary estimation results for λ estimated and ν fixed.

The AMSM model parameters are estimated jointly with the equity unit risk premium λ in this section, while the volatility risk premium ν is fixed to a constant value 0.05. The reason of fixing the volatility risk premium ν is its weaker tractability in an economical sense

compared to the equity unit risk premium λ. Another reason is an increased feasibility and to shorten the calibration analysis. The author decided to use non zero ν in order to have this component of risk neutrality correction in all computations for research purposes. The value 0.05 was chosen to be small enough and to have a minor influence on option prices during sensitivity analysis omitted in the text.

In particular, two likelihood functions were tested, namely $L^R(\theta,\lambda)$, based on a stock price historical data and $L^O(\theta,\lambda,\nu)$, based on an options prices data; both data sets are simulated.

Asset returns data case

In this subsection, the case of estimators based on the simulated underlying asset's log-returns $\{r_t\} = \{S_t\}_{t=0}^{\infty}$ data is considered. Namely, the likelihood function $L^R(\theta,\lambda)$ defined as (3.71) in the previous section is used. There are two groups of 7 experiments: the first group consists of the benchmark experiments estimating the models parameters (m_0,σ_0,ρ) in the assumption λ is fixed and the second group consists of the experiments estimating the equity unit risk-premium (EURP) λ jointly with the model parameters.

The 50 asset log-returns paths of 1500 trading days are simulated in each experiment with the various parameters settings according to Table 3.12. The models are specified with $\hat{k} = 5$ frequencies, the initial underlying asset price $S_0 = 50$ and the interest rate $r = 0.00018$ a day. The maximum number of objective function evaluations in the settings of optimization procedure is limited to 1500 times, the initial point of optimization procedure is $[1.5, 0.05, 0.0005, 0.001]$, the search region of optimization procedure has the lower boundary $[1.0, 0.0005, 0.000001, 0.0]$ and the upper boundary $[1.85, 0.1, 0.7, 1.1]$.

Experiment	m_0	σ_0	ρ (AMSM1/AMSM2)	λ	ν	λ fixed/estimated?
Exp1.f	1.3	0.02	0.25/0.01	0.5	0.05	fixed
Exp1.e	1.3	0.02	0.25/0.01	0.5	0.05	estimated
Exp2.f	1.4	0.01	0.25/0.01	0.5	0.05	fixed
Exp2.e	1.4	0.01	0.25/0.01	0.5	0.05	estimated
Exp3.f	1.4	0.02	0.25/0.01	0.25	0.05	fixed
Exp3.e	1.4	0.02	0.25/0.01	0.25	0.05	estimated
Exp4.f	1.4	0.02	0.25/0.01	0.5	0.05	fixed
Exp4.e	1.4	0.02	0.25/0.01	0.5	0.05	estimated
Exp5.f	1.4	0.02	0.25/0.01	0.75	0.05	fixed
Exp5.e	1.4	0.02	0.25/0.01	0.75	0.05	estimated
Exp6.f	1.4	0.03	0.25/0.01	0.5	0.05	fixed
Exp6.e	1.4	0.03	0.25/0.01	0.5	0.05	estimated
Exp7.f	1.5	0.02	0.25/0.01	0.5	0.05	fixed
Exp7.e	1.5	0.02	0.25/0.01	0.5	0.05	estimated

Table 3.12: Estimation experiments structure. Each experiment consists of 50 repeated MLE estimations based on $L^R(\theta;\lambda)$ likelihood and Adaptive Simulated Annealing method.

In these experiments, only one optimization method is used, namely, the global stochastic Adaptive Simulated Annealing (ASA) method.

AMSM1 model

The results of the experiments for the AMSM1 model are visualized in Figure 3.28. First of all, these results do not demonstrate significant difference between the two groups of experiments (fixed and estimated λ), at least visually. The estimates of m_0, σ_0 and λ seem to be unbiased, while ρ is systematically slightly underestimated. There is no clear dependence of this bias on other parameters values and the group of experiments. The bias is approximately the same for all experiments. The ρ estimates are also significantly noisier, there was a difference of four between their minimum and maximum estimates.

An inspection of Tables 3.13 and 3.14 shows more details. The scale of m_0, ρ and λ is approximately the same, but MAD of ρ is around four times greater. It is in the range $[0.076, 0.092]$, while MAD error metrics of another two parameters are in the range [0.018, 0.023]. The MAE errors of ρ are around three times higher. The magnitude of MAD and MAE metrics of σ_0 is in the range $[4.98 \times 10^{-4}, 1.25 \times 10^{-3}]$ and $[4.00 \times 10^{-4}, 1.23 \times 10^{-3}]$. The difference of MAD and MAE errors between three-parametric and four-parametric estimations is up to 10% in favor of the three-parametric case. The best results in the sense of accuracy and precision are obtained for the experiment *Exp3.e* with $\lambda = 0.25$, while the worst was obtained for the experiment *Exp5.e* with $\lambda = 0.75$, which means the quality of estimates become lower for higher λ.

	Exp1.f	Exp2.f	Exp3.f	Exp4.f	Exp5.f	Exp6.f	Exp7.f
	m_0						
True	1.3	1.4	1.4	1.4	1.4	1.4	1.5
Mean	1.297	1.396	1.398	1.396	1.395	1.397	1.496
Median	1.299	1.396	1.400	1.396	1.396	1.396	1.497
MAD	0.024	0.023	0.028	0.024	0.022	0.024	0.020
MAE	0.020	0.019	0.020	0.019	0.018	0.019	0.017
	σ_0						
True	0.02	0.01	0.02	0.02	0.02	0.03	0.02
Mean	2.00e-02	1.00e-02	2.00e-02	2.00e-02	2.00e-02	3.00e-02	1.99e-02
Median	2.02e-02	1.01e-02	2.02e-02	2.02e-02	2.01e-02	3.02e-02	2.00e-02
MAD	6.98e-04	4.68e-04	1.02e-03	9.37e-04	9.93e-04	1.41e-03	1.24e-03
MAE	6.36e-04	4.00e-04	8.44e-04	8.02e-04	7.57e-04	1.21e-03	9.32e-04
	ρ						
True	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Mean	0.223	0.230	0.232	0.230	0.229	0.230	0.234
Median	0.216	0.222	0.232	0.223	0.218	0.225	0.225
MAD	0.082	0.073	0.076	0.072	0.091	0.073	0.076
MAE	0.071	0.061	0.057	0.061	0.066	0.061	0.055

Table 3.13: The statistics of fourteen MLE (returns-based) estimation experiments in the case of AMSM1 model and λ fixed. Each experiment consisting of 50 simulations.

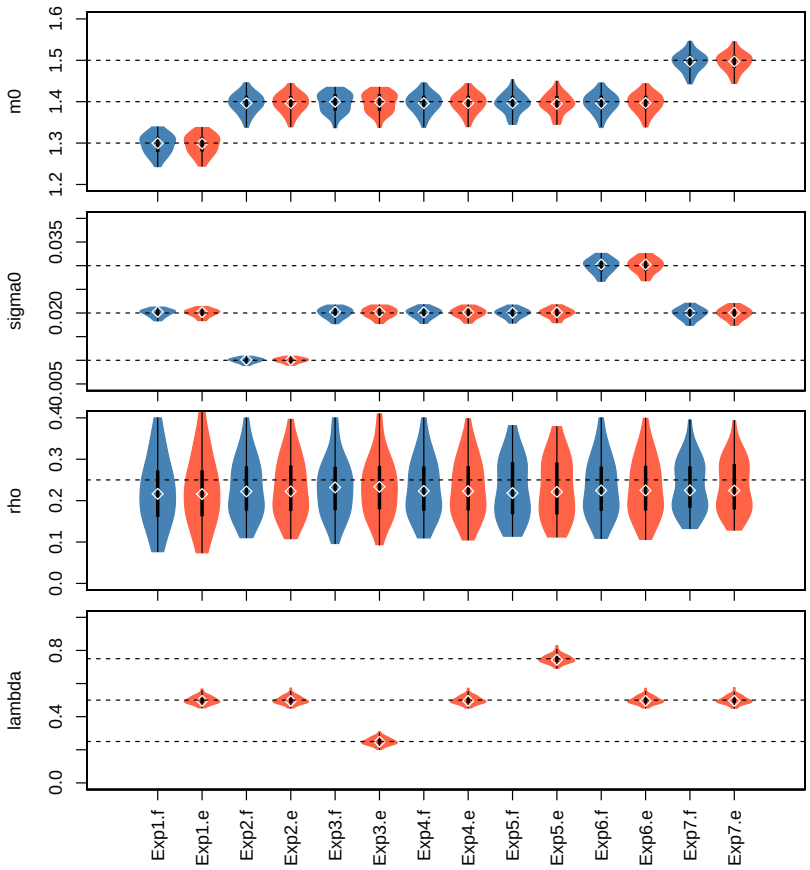


Figure 3.28: The distributions of L^R-MLE estimates of asset returns data during 14 experiments in the case of AMSM1 model. Each experiment consists of 50 simulations. Odd experiments assume λ is fixed, even experiments estimate λ. The dotted lines are the real parameters values $\theta^{\text{real}} = (m_0; \sigma_0; \rho) = (1.3, 1.4, 1.5; 0.01, 0.02, 0.03; 0.25)$ and the risk premiums real values $(\lambda; \nu) = (0.25, 0.5, 0.75; 0.05)$.

	Exp1.e	Exp2.e	Exp3.e	Exp4.e	Exp5.e	Exp6.e	Exp7.e
				m_0			
True	1.3	1.4	1.4	1.4	1.4	1.4	1.5
Mean	1.296	1.396	1.398	1.396	1.395	1.396	1.496
Median	1.299	1.396	1.400	1.396	1.395	1.397	1.497
MAD	0.024	0.024	0.028	0.024	0.020	0.024	0.021
MAE	0.020	0.019	0.020	0.019	0.018	0.019	0.017
				σ_0			
True	0.02	0.01	0.02	0.02	0.02	0.03	0.02
Mean	2.00e-02	1.00e-02	2.00e-02	2.00e-02	2.00e-02	3.00e-02	1.99e-02
Median	2.02e-02	1.01e-02	2.02e-02	2.02e-02	2.01e-02	3.02e-02	2.01e-02
MAD	7.25e-04	5.26e-04	1.11e-03	1.06e-03	1.04e-03	1.57e-03	1.24e-03
MAE	6.44e-04	4.07e-04	8.55e-04	8.17e-04	7.74e-04	1.23e-03	9.44e-04
				ρ			
True	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Mean	0.224	0.231	0.233	0.230	0.230	0.231	0.235
Median	0.216	0.222	0.233	0.223	0.221	0.225	0.224
MAD	0.083	0.078	0.076	0.076	0.092	0.075	0.077
MAE	0.072	0.062	0.058	0.061	0.066	0.061	0.056
				λ			
True	0.5	0.5	0.25	0.5	0.75	0.5	0.5
Mean	0.499	0.499	0.249	0.499	0.748	0.499	0.499
Median	0.495	0.494	0.248	0.495	0.743	0.496	0.496
MAD	0.021	0.020	0.018	0.020	0.023	0.020	0.018
MAE	0.019	0.019	0.017	0.019	0.020	0.019	0.019

Table 3.14: The statistics of fourteen MLE (returns-based) estimation experiments in the case of AMSM1 model and λ estimated. Each experiment consisting of 50 simulations.

AMSM2 model

The results of the experiments for the AMSM2 model are visualized in Figure 3.29. First of all, there is a clear difference between the three-parametric and four-parametric estimation experiments (fixed and estimated λ), but it is impossible to tell unequivocally which case has better performance anyway.

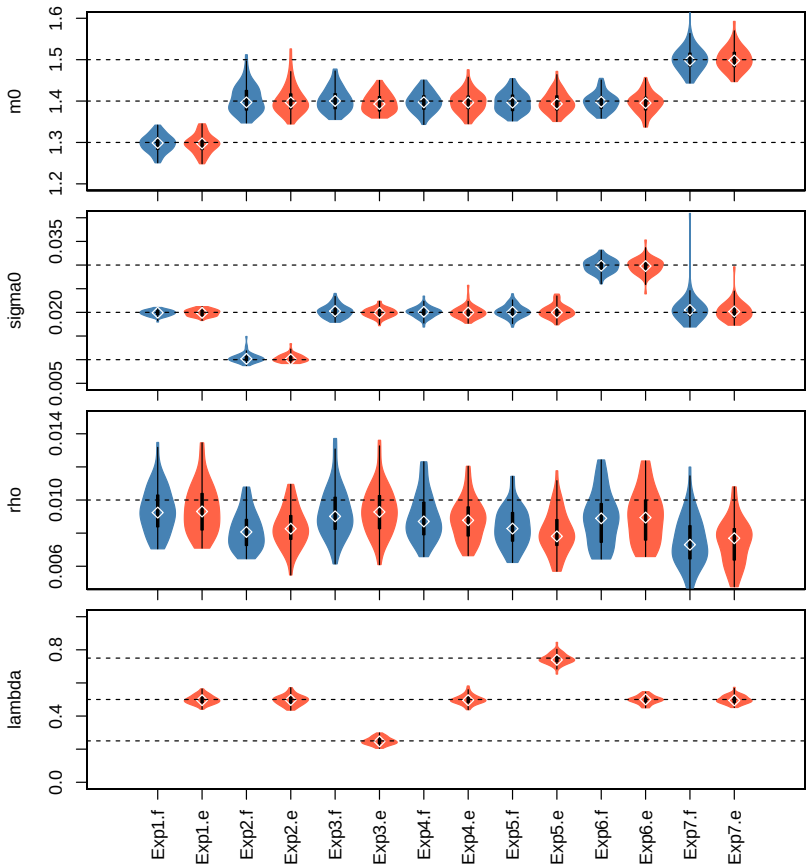


Figure 3.29: The distributions of L^R-MLE estimates of asset returns data during 14 experiments in the case of AMSM2 model. Each experiment consists of 50 simulations. Odd experiments assume λ is fixed, even experiments estimate λ. The dotted lines are the real parameters values $\theta^{\text{real}} = (m_0; \sigma_0; \rho) = (1.3, 1.4, 1.5; 0.01, 0.02, 0.03; 0.01)$ and the risk premiums real values $(\lambda; \nu) = (0.25, 0.5, 0.75; 0.05)$.

The estimates of m_0, σ_0 and λ are unbiased as well as in the case of AMSM1, while ρ is systematically underestimated and significantly in few cases (up to two times) in contrast to the case of the AMSM1 model. There is a clear dependence of these biases on λ size. Further, ρ estimates bias becomes higher for higher m_0 and lower σ_0. The ρ estimates deviation is significantly higher than for other parameters estimated.

Tables 3.15,3.16 reaffirm negative dependence between λ and σ_0 estimates and the size of λ: MAD of λ increases from 0.021 to 0.025, MAE increases from 0.018 to 0.024 for $\lambda = 0.25, 0.5, 0.75$ (the experiments *Exp3.e, Exp4.e, Exp5.e*). Also there is an increase of MAD and MAE metrics for σ_0. In addition, the real size of σ_0 effects the quality of estimates of m_0 and λ. That is smaller σ_0 leads to larger (MAD; MAE): from $(0.023; 0.018)$ to $(0.028; 0.025)$ for m_0; from $(0.021; 0.018)$ to $(0.029; 0.024)$ for λ. MAD and MAE of ρ is larger for larger m_0 and λ. The MAD and MAE metrics of m_0 estimates for the three-parametric case (the

experiments *Exp1.f,...,Exp7.f*) are larger for the experiments *Exp2, Exp3, Exp4, Exp5*, while they are lower than in the four-parametric estimations in other cases.

In general, the quality of estimates is similar for fixed λ and estimated λ, with slightly higher levels of MAD and MAE for the former case.

	Exp1.f	Exp2.f	Exp3.f	Exp4.f	Exp5.f	Exp6.f	Exp7.f
	m_0						
True	1.3	1.4	1.4	1.4	1.4	1.4	1.5
Mean	1.297	1.404	1.403	1.398	1.397	1.398	1.502
Median	1.299	1.397	1.400	1.397	1.396	1.398	1.498
MAD	0.019	0.034	0.028	0.025	0.029	0.020	0.025
MAE	0.016	0.027	0.023	0.019	0.020	0.017	0.023
	σ_0						
True	0.02	0.01	0.02	0.02	0.02	0.03	0.02
Mean	1.99e-02	1.03e-02	2.02e-02	2.00e-02	2.02e-02	2.99e-02	2.07e-02
Median	1.99e-02	1.02e-02	2.03e-02	2.01e-02	2.01e-02	2.98e-02	2.05e-02
MAD	6.51e-04	7.03e-04	1.19e-03	8.70e-04	9.12e-04	1.62e-03	1.52e-03
MAE	4.93e-04	6.67e-04	1.02e-03	7.97e-04	9.71e-04	1.16e-03	1.73e-03
	ρ						
True	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Mean	0.009	0.008	0.009	0.009	0.008	0.009	0.007
Median	0.009	0.008	0.009	0.009	0.008	0.009	0.007
MAD	0.002	0.001	0.001	0.001	0.001	0.002	0.001
MAE	0.241	0.242	0.241	0.241	0.242	0.241	0.243

Table 3.15: The statistics of fourteen MLE (returns-based) estimation experiments in the case of AMSM2 model and λ fixed. Each experiment consisting of 50 simulations.

Asset returns and options prices data case

In this subsection the performance of a wider set of estimators is considered and compared: MLE based on $L^R(\theta,\lambda)$ likelihood function from the previous subsection; the calibration procedure based on minimization of Weighted Residual Sum of Squares (WRSS) from Section 3.4, which is equivalent to a maximization of $L^O(\theta,\lambda,\nu)$ likelihood defined as (3.75) and MLE based on L^M likelihood defined as (3.79). In other words, a comparison of estimators based on underlying asset log-returns, options prices data and mixture of them is presented here.

The estimation methods applied to the simulated data in the assumption AMSM model holds with the following parametric settings:

- The parametric set I: $(m_0,\sigma_0,\rho,\lambda,\nu) = (1.4,0.02,0.25/0.01,0.25,0.05)$;
- The parametric set II: $(m_0,\sigma_0,\rho,\lambda,\nu) = (1.4,0.02,0.25/0.01,0.5,0.05)$;

	Exp1.f	Exp2.f	Exp3.f	Exp4.f	Exp5.f	Exp6.f	Exp7.f
	m_0						
True	1.3	1.4	1.4	1.4	1.4	1.4	1.5
Mean	1.297	1.403	1.396	1.398	1.396	1.395	1.502
Median	1.298	1.397	1.393	1.396	1.393	1.395	1.498
MAD	0.018	0.028	0.026	0.023	0.026	0.023	0.028
MAE	0.017	0.025	0.019	0.021	0.020	0.018	0.022
	σ_0						
True	0.02	0.01	0.02	0.02	0.02	0.03	0.02
Mean	1.99e-02	1.03e-02	1.99e-02	2.00e-02	2.02e-02	2.98e-02	2.03e-02
Median	1.99e-02	1.01e-02	1.99e-02	2.00e-02	2.00e-02	2.98e-02	2.02e-02
MAD	8.35e-04	8.18e-04	1.03e-03	9.88e-04	1.23e-03	1.47e-03	1.79e-03
MAE	5.77e-04	6.14e-04	7.51e-04	9.28e-04	1.03e-03	1.25e-03	1.43e-03
	ρ						
True	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Mean	0.009	0.008	0.009	0.009	0.008	0.009	0.007
Median	0.009	0.008	0.009	0.009	0.008	0.009	0.008
MAD	0.002	0.001	0.002	0.001	0.001	0.002	0.001
MAE	0.241	0.242	0.241	0.241	0.242	0.241	0.243
	λ						
True	0.5	0.5	0.25	0.5	0.75	0.5	0.5
Mean	0.499	0.495	0.248	0.498	0.745	0.498	0.499
Median	0.496	0.497	0.247	0.496	0.739	0.500	0.494
MAD	0.024	0.029	0.021	0.020	0.025	0.021	0.024
MAE	0.021	0.024	0.018	0.020	0.024	0.018	0.019

Table 3.16: The statistics of fourteen MLE (returns-based) estimation experiments in the case of AMSM2 model. Each experiment consists of 50 simulations.

Exper. title	m_0	σ_0	ρ	λ	ν	Estimation method	Options prices data	Asset price data path
Parametric set I								
Exp1.1	1.4	0.02	0.25/0.01	0.25	0.05	L^R-MLE	none	1500
Exp2.1	1.4	0.02	0.25/0.01	0.25	0.05	min(WRSS)	10 options ($T=30$)	none
Exp3.1	1.4	0.02	0.25/0.01	0.25	0.05	min(WRSS)	30 options ($T=30,60,90$)	none
Exp4.1	1.4	0.02	0.25/0.01	0.25	0.05	L^M-MLE	10 options ($T=30$)	750
Exp5.1	1.4	0.02	0.25/0.01	0.25	0.05	L^M-MLE	30 options ($T=30,60,90$)	750
Parametric set II								
Exp1.2	1.4	0.02	0.25/0.01	0.5	0.05	L^R-MLE	none	1500
Exp2.2	1.4	0.02	0.25/0.01	0.5	0.05	min(WRSS)	10 options ($T=30$)	none
Exp3.2	1.4	0.02	0.25/0.01	0.5	0.05	min(WRSS)	30 options ($T=30,60,90$)	none
Exp4.2	1.4	0.02	0.25/0.01	0.5	0.05	L^M-MLE	10 options ($T=30$)	750
Exp5.2	1.4	0.02	0.25/0.01	0.5	0.05	L^M-MLE	30 options ($T=30,60,90$)	750
Parametric set III								
Exp1.3	1.4	0.02	0.25/0.01	0.75	0.05	L^R-MLE	none	1500
Exp2.3	1.4	0.02	0.25/0.01	0.75	0.05	min(WRSS)	10 options ($T=30$)	none
Exp3.3	1.4	0.02	0.25/0.01	0.75	0.05	min(WRSS)	30 options ($T=30,60,90$)	none
Exp4.3	1.4	0.02	0.25/0.01	0.75	0.05	L^M-MLE	10 options ($T=30$)	750
Exp5.3	1.4	0.02	0.25/0.01	0.75	0.05	L^M-MLE	30 options ($T=30,60,90$)	750

Table 3.17: Estimation experiments structure. Each consists of 25 repeated estimations based on pseudo- and quasi-Monte Carlo and Adaptive Simulated Annealing method.

- The parametric set III: $(m_0, \sigma_0, \rho, \lambda, \nu) = (1.4, 0.02, 0.25/0.01, 0.75, 0.05)$.

Each experiment was repeated with each of the three parametric sets above and consists of:

1. The simulation of 25 asset log-returns sample paths of 1500 trading days each, for example $\{r_i\}_{i=1}^{1500}$, then the estimation of each using MLE based on $L^R(\theta, \lambda; r_1, \ldots, r_{1500})$;

2. The simulation of European Call options prices for 25 K-baskets (cross-sections) by 10 options each [49] and the only maturity $T = 30$, then the calibration of the AMSM models parameters by minimization of WRSS for each basket;

3. Similarly, the simulation of European Call options prices for 25 KT-baskets by 30 options each with the maturities $T = 30, 60, 90$, then the calibration of the AMSM model by minimization of WRSS for each basket;

4. The simulation of European Call options prices for 25 K-baskets by 10 options each with the only maturity $T = 30$, denoted as $C_1, \ldots, C_{10}$. In addition, the simulation of corresponding 25 underlying asset log-returns paths of 750 trading days each, denoted as $\{r_i\}_{i=1}^{750}$, then the estimation of the AMSM models by maximization of the corresponding likelihood $L^M(\theta, \lambda; r_1, \ldots, r_{750}, C_1, \ldots, C_{10})$ with respect to the model parameters θ and EURP λ;

5. Similarly, the simulation of European Call options prices for 25 KT-baskets of 30 options each with the maturities $T = 30, 60, 90$. In addition, the simulation of 25 asset log-returns paths of 750 trading days each, then the estimation of the AMSM models by maximization of the corresponding likelihood L^M.

The models are specified with $\hat{k} = 5$ frequencies, the initial underlying asset price $S_0 = 50$ and the interest rate $r = 0.00018$ a day. The maximum number of objective function evaluations is limited to 1500 times in the optimizer subroutines settings, the initial point of optimization method is $[1.5, 0.05, 0.0005, 0.001]$, while the search region lower boundary is $[1.0, 0.0005, 0.000001, 0.0]$ and the upper boundary is $[1.85, 0.1, 0.7, 1.1]$. The structure of the experiments is reflected in Table 3.17.

In this subsection, the Levenberg-Marquardt optimization method was used for the minimization of WRSS (calibration), while the global stochastic Adaptive Simulated Annealing (ASA) method was used for the maximization of $L^R(\theta, \lambda; \mathbf{r})$ and $L^M(\theta, \lambda; \mathbf{r}, \mathbf{C})$ likelihood functions.

AMSM1 model

Visual inspection of Figure 3.30 shows that the calibration procedure fails to estimate σ_0 and λ in the experiments *Exp2.1, Exp2.2, Exp2.3* (option chains with the maturity $T = 30$). Further, there is clear large dispersion of the estimates of ρ and, especially, λ in the case of the experiments *Exp3.1, Exp3.2, Exp3.3* (option chains with the maturities $T = 30, 60, 90$). The maximum likelihood estimators based on the likelihood functions L^R (*Exp1.1, Exp1.2, Exp1.3*) and L^M (*Exp4.1, Exp4.2, Exp4.3, Exp5.1, Exp5.2, Exp5.3*) are unbiased, excepting the slightly biased estimates of m_0 in the experiments *Exp4.1, Exp4.2, Exp4.3.* In addition,

[49] The difference between these baskets is the seed of random number generator used.

the estimates of ρ and λ based on the maximization of likelihood functions L^M overestimate the equity unity risk-premium λ in *Exp5.1* weakly.

An analysis of Table 3.18 containing the metrics of the experiments results distribution gives the following conclusions. The metric MAD (precision) of L^R-based MLE estimates of m_0 and λ is typically up to two times less compared to the calibration and L^M-based MLE, while MAD of calibration (WRSS column in the table) and L^M-based estimates of σ_0 and ρ is significantly lower than for L^R-based counterparts, mostly two to three times.

Looking deeper, the estimates obtained by the maximization of the likelihood function L^R (*Exp1.x*) are unbiased, and their mean and median values deviate from the real ones slightly, by 2% – 10%. Their precision depends on a parameter: the estimates of λ have the highest precision, namely $MAD = 0.017 \ldots 0.02$, while the lowest precision is obtained for the estimates of ρ, $MAD = 0.089 \ldots 0.1$.

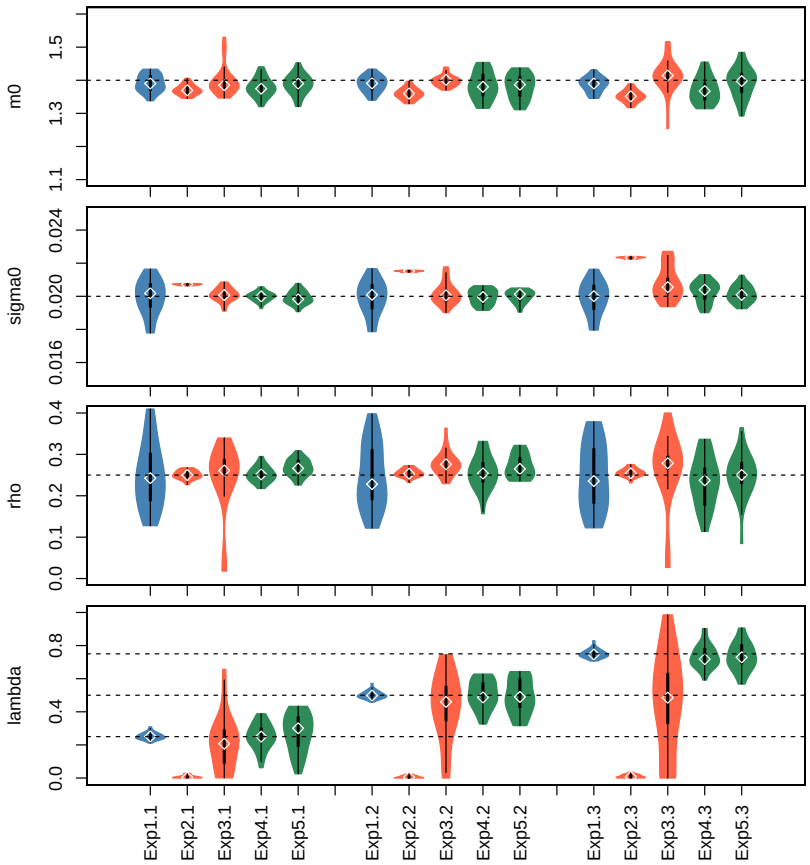


Figure 3.30: Three groups of estimation/calibration experiment's distributions depicted as violins for three $\lambda = 0.25, 0.5, 0.75$ and AMSM1 model. Each group consists of estimation experiments distributions of L^R-MLE (*Exp1.x*), calibration (*Exp2.x, Exp3.x*) and L^M-MLE (*Exp4.x, Exp5.x*) methods. The dotted lines are the real parameters values $\theta^{real} = (m_0; \sigma_0; \rho) = (1.4; 0.02; 0.25)$ and EURP $\lambda = 0.25, 0.5, 0.75$.

The calibration procedure (*Exp2.x*) using the option chain prices with the maturity $T = 30$ (strike prices $K = 40, 42, \ldots, 58$) gives very close-grouped (precise) estimates: $MAD(m_0) \approx 0.015$, $MAD(\sigma_0) \approx 5.0 \times 10^{-5}$, $MAD(\rho) \approx 0.01$, $MAD(\lambda) = 0.008$. The estimates of m_0 and σ_0 are biased; the estimates of λ are calibrated completely wrong in *Exp2.x*, as mentioned earlier. The results of calibration using the prices of KT-basket of a few option chains with

the maturities $T = 30, 60, 90$ and the strike prices $K = 40, 42, ..., 58$ (*Exp3.x*) are significantly more accurate. In particular, the median of m_0 jumped from 1.351 to 1.414 for the case $\lambda = 0.75$ (*Exp3.3*), but the results are more imprecise: $MAD(m_0) \approx 0.018$, $MAD(\sigma_0) \approx 5.0 \times 10^{-4}$, $MAD(\rho) \approx 0.025$, $MAD(\lambda) \approx 0.180$.

The performance of the MLE-estimates based on the likelihood L^M (*Exp4.x, Exp5.x*) is somewhat a mixture of L^R-MLE and the calibration results. The estimates used the single option chain (K-basket with $T = 30$) and the historical underlying asset prices data (*Exp4.x*) are slightly biased, but less than their calibration based counterparts. The estimates based on the multiple option chains (KT-baskets) and the historical underlying asset prices data (*Exp5.x*) seem to be unbiased. Meantime, L^M-based MLE estimates of σ_0 and ρ are significantly less noisy comparing to L^R-based ones, for example: $MAD(\sigma_0) \approx 3.43 \times 10^{-4}$ (*Exp5.1*) versus $1.03e{-}10^{-3}$ (*Exp1.1*), $MAD(\rho) \approx 0.043$ (*Exp5.3*) versus 0.1 (*Exp1.3*). The L^M-based MLE estimates of m_0 and λ are less precise, but the difference with L^R-based ones is not that huge, as the calibration procedure results have, for example: $MAD(m_0) \approx 0.029$ (*Exp5.2*) versus 0.051 (*Exp1.2*), $MAD(\lambda) \approx 0.072$ (*Exp5.3*) versus 0.02 (*Exp1.3*) and 0.231 for the calibration (*Exp3.3*).

AMSM2 model

An inspection of Figure 3.31 shows a similar picture for the AMSM2 model compared to the results obtained for the AMSM1 model, but there are two important differences. First of all, L^M-based MLE estimates (*Exp4.x, Exp5.x*) are no longer unbiased. Second, the results of λ calibration based on WRSS minimization turn out to fail for both baskets types (*Exp2, Exp3*).

The MLE based on the likelihood function L^R provides the best estimates of λ according to Table 3.19: $MAD \approx 0.029$, $MAE \approx 0.02$ (*Exp1.1*) comparing to L^M-based MLE estimates, MAD up to 0.107, MAE up to 0.113 (*Exp5.1*)).

The calibration procedure provides the best m_0 and ρ estimates, but it completely fails with λ. Further, the estimates of σ_0 are significantly more biased and have the highest MAE levels compared to other estimators.

In contrast to AMSM1, L^M-based MLE estimates are increasingly biased for larger λ: the median of m_0 drops down from 1.385 to 1.344 (*Exp5.x*), the median of σ_0 drops from 2.02×10^{-2} to 2.08×10^{-2} and the median of ρ drops from 0.011 to 0.007. Meantime, the bias of λ is low and similar to L^R-based MLE case, but it is significantly more deviated than its L^R-based counterpart (MAD is in the range 0.063..0.107 versus 0.021..0.029). The L^M-based MLE estimates of ρ also have large deviation (MAD 0.002), which leads – together with the significant bias of ρ estimates – to the largest MAE among the considered estimators.

3.5.3. Comparative analysis and intermediate conclusions

In general, the estimates of ρ and λ for the AMSM1 model are significantly noiser than m_0 and σ_0 estimates, especially for larger real values of λ. The estimates based on L^R likelihood are very imprecise for ρ (*Exp1.x*). The calibration on single option chain datasets (K-baskets, *Exp2.x*) is not able to estimate λ at all and it is poor for the case of multiple option chains datasets (KT-basket, *Exp3.x*). In contrast, the L^R-based MLE estimates of λ and

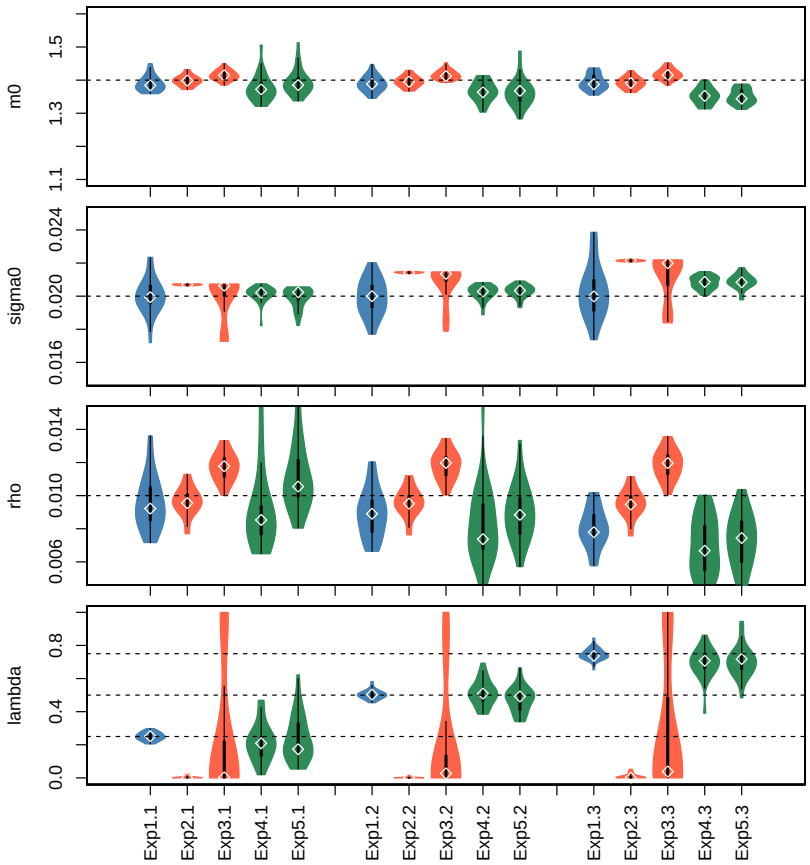


Figure 3.31: Three groups of estimation/calibration experiment's distributions depicted as violins for three $\lambda = 0.25, 0.5, 0.75$ and AMSM2 model. Each group consists of estimation experiments distributions of L^R-MLE (*Exp1.x*), calibration (*Exp2.x, Exp3.x*) and L^M-MLE (*Exp4.x, Exp5.x*) methods. The dotted lines are the real parameters values $\theta^{real} = (m_0; \sigma_0; \rho) = (1.4; 0.02; 0.01)$ and EURP $\lambda = 0.25, 0.5, 0.75$.

the calibration of ρ are both accurate and precise. In general, the MLE estimates based on the likelihood L^M in conjunction with KT-baskets (*Exp5.x*) are a preferable choice for the AMSM1 model as a good trade-off in the sense of accuracy and precision of all four parameters. At the same time, there is no perfect estimator for the AMSM2 model as well. The calibration procedure fails with λ, while the best results are achieved for ρ and m_0. The MLE based on L^M fails with an estimation of ρ and is fine with the rest parameters, but L^R-based MLE exceeds it in the sense of lower MAE of m_0, ρ and λ.

3.6. Application

The aim of this section is to apply the calibration and estimation techniques described above to real data from financial markets, to run out-of-sample performance tests and to compare the AMSM model with competitive models.

3.6.1. Real market data

First of all, it is necessary to recover a risk-free interest rate from the market. There are two main sources of borrowing on financial markets: a money market (short-range) and a capital market (long-range). In the first case, EURIBOR and other -IBOR spot rates are usually used as short-range risk free interest rates, while yield curve spot rates calculated and published by ECB are used as long-range risk free interest rates. The values of these short- and long-range interest rates were collected for various expiries with the longest expiry corresponding to the longest maturity in the options prices data. Further, these rates are used to construct interpolation using the LOESS[50] interpolation method (see Figure 3.32). The interpolation allows us to obtain interest rates for non-standard expiries equal to maturities of considered vanilla options. Note, the interpolated interest rates are used only for option pricing. They are meaningless for asset returns estimation, where fixed daily interest $r = 0.00018$ (around 6.5% annually) was used, similarly to Christoffersen, Heston and Jacobs' approach in their joint paper [22]. Also note, the interpolated annual rates are converted to daily ones using the following well-known formula

$$r_d = 1 - (1 - r_a)^{1/365}.$$

The largest European derivatives exchange is EUREX. A number of European style options are traded, including many options with German blue chips as underlying assets. In order to estimate real data options, two indices were chosen, namely Euro Stoxx 50[51] and DAX 30[52], and two shares, namely Siemens[53] and SAP[54]. The historical prices of these underlying assets were collected from the 2nd of January 2004 to the 28th of August 2018 from Börse Frankfurt. The leverage effect, being the crucial point of this chapter, is depicted in Figure 3.33 for all considered underlying assets for lags from 1 to 15. The strongest effect is observed for both indices. Note also, they appear to be closely correlated, which can be explained by the fact that they share many stocks. At the same time, SAP is clearly not affected by the leverage effect at all.

[50] LOESS is a non-parametric (local) regression model.
[51] ISIN EU0009658145, WKN 965814.
[52] ISIN DE0008469008, WKN 846900.
[53] ISIN DE0007236101, WKN 723610, symbol SIE.
[54] ISIN DE0007164600, WKN 716460, symbol SAP.

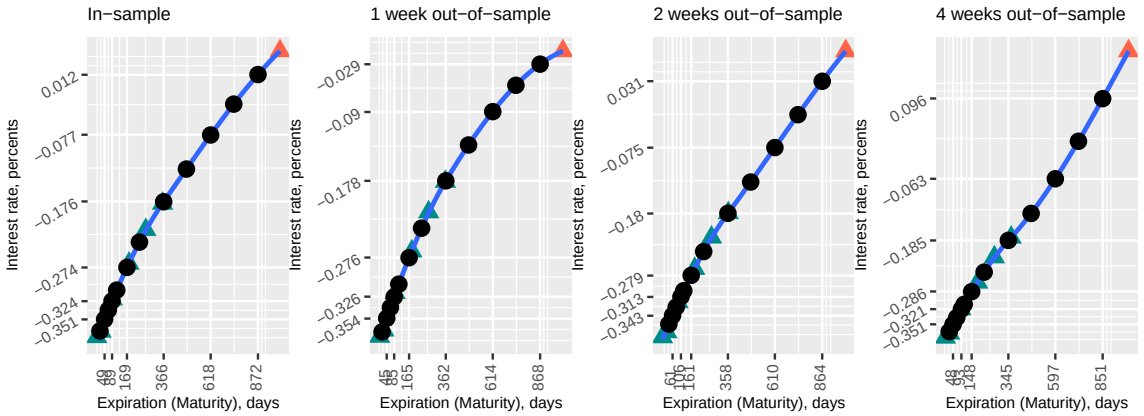


Figure 3.32: The term structure of interest rates (yield curve), where the cyan triangles are spot rates from the money market (EURIBOR), the tomato triangles are spot rates from the capital market (ECB), the blue line and the black dots are interpolated (LOESS) interest rates used for option pricing.

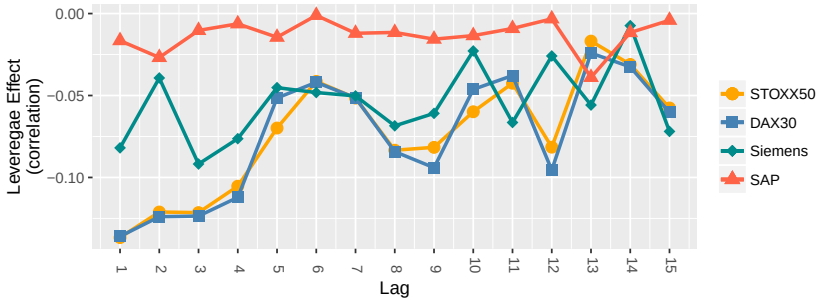


Figure 3.33: The leverage effect measured as a correlation between absolute returns and lagged returns. The dataset consists of the closing prices from 2nd January 2004 to 30th May 2018 on Börse Frankfurt.

The option chain intraday quotes data collected on the 13th of July, the 25th of July and the 13th of August 2018 for the following options: Euro Stoxx 50 Index Options (OESX)[55], DAX 30 Options (ODAX)[56], Siemens (SIE)[57] and SAP (SAP) [58] on EUREX exchange for the underlying assets described above. This data is visualized as the price surface with respect to strike prices and maturities in Figure 3.34. As option price is assumed to be an average value of ask and bid prices, therefore, options with no bid price (and zero volume of trade) are neglected. The most detailed data during the 13th of July was available for STOXX 50 index options: the maturities are from 12 to 651 and the strike prices are from 0.2 to 364.8 euros. DAX 30 index options were traded the same day for shorter maturities, namely up to around 400 business days. The datasets are even smaller for SAP and Siemens options, namely less than 200 prices; this is compared to around 600 prices in the case of Stoxx 50 and DAX 30 index options. The difference is in the number of strike prices available for trading of SAP and Siemens options. Also note that all four datasets have certain shift of strike prices with an increase of maturity, which reflects market beliefs on a future trend.

In addition, the datasets are filtered according to the level of moneyness. The definition of moneyness of an option is given by

$$\text{Moneyness} = \exp(-rT)\frac{S_0}{K},$$

where K is a strike price, T is a maturity, S_0 is a spot price of underlying in the date of signing the contract. Thus, the options with the moneyness level: between 0.96 and 1.04 are assumed further as at-the-money (ATM), less than 0.96 are assumed as in-the-money (ITM) and greater than 1.04 are assumed as out-the-money (OTM). Options with the moneyness level less than 0.9 or greater than 1.1 are filtered out from the dataset. Options with maturities less than 20 are not liquid; therefore, they were also excluded from the dataset. Further, all options in the dataset are split into three groups by maturity: short-run with maturities $T \in [20,90]$), mid-run with $T \in [91,180]$ and long-run with $T \in [181,1000]$. The resulting dataset size is 228 options in the case of Stoxx 50 index options; its structure is reflected in Table 3.20.

Moneyness	Short-run	Mid-run	long-run
ITM	21	18	23
ATM	33	22	32
OTM	27	23	18

Table 3.20: The dataset's structure and size for Stoxx 50 index options case.

The quality of real data estimates is considered in the next subsection. In this subsection, an estimation of the real data by estimators developed and testified on synthetic data in Sections 3.4 and 3.5 is considered for both model versions with both risk premiums (EURP and volatility risk premium) estimated.

[55]Product ISIN DE0009652396, Underlying ISIN EU0009658145.
[56]Product ISIN DE0008469495, Underlying ISIN DE0008469008.
[57]Product ISIN DE0007236101, Underlying ISIN DE0007236101.
[58]Product ISIN DE0007164600, Underlying ISIN DE0007164600.

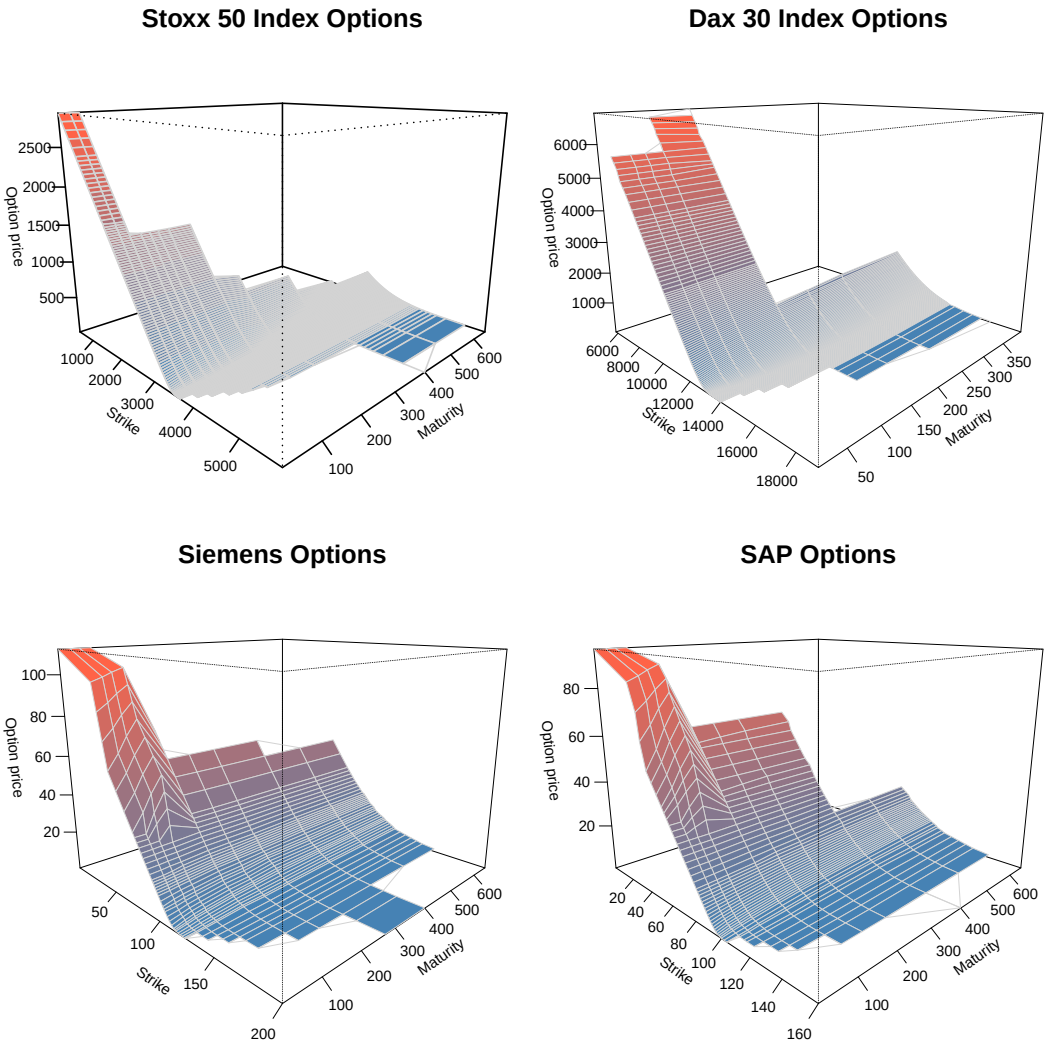


Figure 3.34: The surfaces of options quotes from EUREX exchange for various assets at 13th July 2018.

3.6.2. In-sample and out-of-sample performance

The final test for the AMSM models and three methods developed for estimation of them is in-sample and out-of-sample performance. In order to do so, the estimates are used for computation of theoretical option prices corresponding to the Euro Stoxx 50 options from Table 3.20, that are split into nine baskets by three types of moneyness (ITM, ATM and OTM) and three time horizons (SR, MR and LR). The dataset on 13th of July 2018 used for the estimation allows us to define in-sample performance measured by residuals (WRSS). Analogical datasets were created for Euro Stoxx 50 option prices at 24 July 2018 and 13 August 2018. They were filtered and prepared the same way as described in Subsection 3.6.1 in order to obtain another nine baskets, but two and four weeks out-of-sample. In order to compare the real option prices and their theoretical counterparts computed by the AMSM model, the former ones were computed using the model parameters estimates obtained for the in-sample datasets (baskets). The calibration for DAX 30, Siemens and SAP options was excluded from further analysis due to small amount of data, especially, for long horizons (DAX 30, see Figure 3.34). In addition, the liquidity of options for Siemens and SAP assets is questionable (see the artifacts of the price surfaces in Figure 3.34). Nevertheless, the estimation based on asset returns only was done for all four assets, see Section 3.6.2, in particular Figure 3.36.

As a benchmark, the Black-Scholes (BS) model, Black-Scholes Ad-Hoc (BSAH) model [33] and Heston-Nandi (HN) model [47] are estimated on each of nine baskets from Table 3.20, then the corresponding in-sample and out-of-sample theoretical option prices computed. Note, Black-Scholes ad-hoc model differs in the volatility parameter from the classic one. Namely, it is not a constant, but a nonlinear regression of strike price K and maturity T. In particular, the polynomials of K and T of order two are used as the nonlinear regression, which leads to a necessity to estimate five parameters. The closed-form GARCH option valuation model of Heston and Nandi [47] is a well-known model for cases when options prices data is not available for a calibration as it requires only an underlying asset log-returns data.

So, AMSM, BS and AdHoc models were used to estimate all nine in-sample baskets from Table 3.20 and then two and four weeks out-of-sample prices were calculated. Therefore, there are nine parametric sets obtained for the corresponding nine in-sample baskets (ATM SR and etc.) for each AMSM model version from the calibration approach, as well as from the L^M-based MLE approach, for the Ad-Hoc model, and also nine volatility parameter estimates for the Black-Scholes model. Only one parametric sets was obtained from the historical asset log-returns observed until 13th July 2018 for the AMSM models from MLE based on L^R likelihood and for the Heston-Nandi model. The parametric set consists of just the volatility parameter in the case of the BS model, while the Ad-Hoc, HN model and AMSM models have five parameters.

Calibration based on market option prices

The real data calibration results based on minimization using the Levenberg-Marquardt algorithm (LM) and Adaptive Simulated Annealing (ASA) of the Weighted Residual Sum of Squares (WRSS) defined by (3.48) are presented in this subsection. Theoretical details, the choice of optimization methods and the sensitivity analysis are given in Section 3.4. The calibration results based on synthetic data are described in Subsection 3.4.3 for the case of fixed λ and ν.

The Stoxx 50 index options real data described above. The calibration experiment consists of ten repeats of the calibration procedure with different seeds of Monte Carlo paths simulation necessary for an option price computation by the AMSM model. The theoretical option prices in the short-run (SR) and mid-run (MR) option baskets are computed in this experiment using quasirandom numbers, which showed better performance during estimation experiments based on the synthetic data. The long-run (LR) theoretical option prices are computed with pseudorandom numbers due to large memory and time consumption of computation based on quasirandom numbers. The number of sample paths during Monte Carlo option pricing were 49K quasirandom paths for each option price (SR case), 24K quasirandom paths (MR case) and, finally, 98K pseudorandom paths in the case of long-run maturities.

The seed and the parameter values corresponding to the case with the lowest WRSS among ten calibration repeats are used further as an optimal choice. These best estimates are collected in Table 3.21.

The in-sample and out-of-sample results are depicted as barplots of relative residuals for each model and baskets (moneyness is x-axis) in Figure 3.35. As relative residual is considered the expression $(C_r - C_m)/C_r$, where C_r is a real market option price, C_m is a theoretical option price obtained using one of the models. There are two situations when all the models behave poorly: out-of-the-money and short-run (OTM SR) options prices computation. In particular, the relative residuals in this case are up to 60% for the BS model, 25% for the AMSM1 model in-sample and up to $-$%120 for the BS model, $-$140% for the AMSM2 model in the case of two weeks out-of-sample. The relative residuals of OTM MR theoretical options prices are smaller, but also significant, they are up to $-$60% for the BS model, and $-$50% for the AMSM model. An economical reason for quite large residuals can be a lack of liquidity in both cases of OTM options and SR options, especially in the first case. Further, the results are surprisingly worse for the case of two weeks out-of-sample options pricing than their counterparts computed another two weeks later. This can be a result of poor estimation of the risk free interest rate, which is very important for an accurate option pricing. Another factor can be small dataset size; the baskets used for the calibration consist of around 30 options. At the same time, the main aim of this subsection is to compare the AMSM model and competitive models' performance to each other rather than perfect accuracy.

It is clear from Figure 3.35, comparing the relative residuals, that the results for the AMSM1 and AMSM2 models are visually very similar. Their precision and accuracy are so close that it is impossible to select the winner. A visual inspection aimed at comparing AMSM models with the benchmark models shows the classic Black-Scholes model is worse in most scenarios. Ad-Hoc seems to be worse for the short-run case in conjunction with ITM and OTM options, while it is better for the case of mid-run options and worse or similar for the long-run options pricing.

A more detailed picture can be found in Tables 3.24 and 3.25 with calculated WRSS values. The AMSM1 and AMSM2 model versions have close residuals with a superiority of one or another version in different cases. Both AMSM models compared to the Black-Scholes model have significantly lower errors in-sample and four weeks out-of-sample, while the BS model is surprising better than all other models two weeks out-of-sample in the case of mid-run (MR). Another benchmark model, Ad-Hoc model, has lower WRSS values than AMSM counterparts in case of mid-run options two and four weeks out-of-sample, namely:

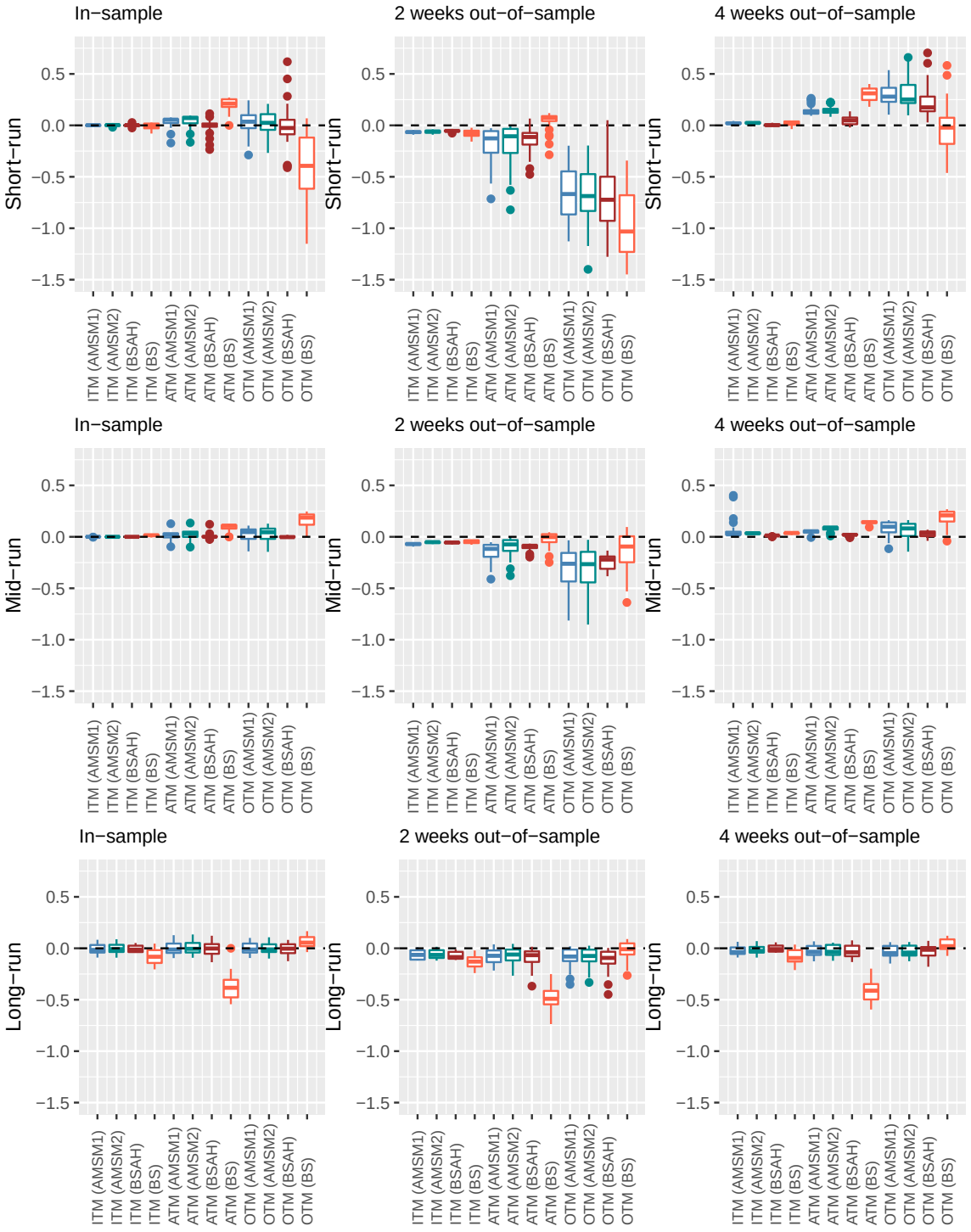


Figure 3.35: Relative standard errors (residuals) of theoretical options prices computation in the case of various maturities and moneyness for AMSM1, AMSM2, Ad-Hoc and BS models based on options prices data.

0.03 (ITM), 0.33 (ATM), 1.47 (OTM) in the case of two weeks out-of-sample and 0.00 (ITM), 0.01 (ATM), 0.04 (OTM). In general, AMSM demonstrates its superiority in the long-run, which can be explained by increasing the importance of the presense of various volatility frequencies modeled by AMSM that show up for longer sample paths.

Also, the results obtained using the LM optimization method are collected in the second part of Tables 3.24 and 3.25. Note, ASA provides slightly better optimal parameters in the sense of WRSS than the LM method in most cases, excepting few mid-run and short-run results. For example, WRSS is lower for ITM MR and ATM MR baskets in the case of LM based calibration.

Maximum likelihood estimation based on asset returns

Details about the estimator construction are given in Section 3.5.1. The simulations in Subsection 3.5.2 showed that the most precise and accurate results for MLE based on $L^R(\theta, \lambda|\mathbf{r})$ likelihood (3.71) provides the Adaptive Simulated Annealing optimization algorithm, thus it is used for the real data estimation experiments as well.

The log-returns sample paths of four assets on Börse Frankfurt (Euro Stoxx 50 index, DAX30 index, SAP and Siemens stocks) are the input data. The length of historical data sample size used for estimation is from 500 points up to 3500 data points (trading days). Note, the dataset with 3500 trading days corresponds to the time window starting from 11th October 2004 and ending 13 July 2018.

An estimation of real data on assets return with MLE does not depends on the seed, thus there is no sense to repeat it, even taking into account the stochastic nature of the ASA algorithm. Instead, the estimation was repeated for various time windows, as mentioned earlier for four underlyings. The results are presented for the AMSM1 model version with estimation of four parameters excluding ν; they are visualized in Figure 3.36. Further, the log-returns of Stoxx 50 index are depicted on the fifth panel as a scatter plot w.r.t. to the timeline, which allows us to reveal a dependence of the asset returns behavior on the estimated model parameters. The use of the datasets with various historical horizons allows us to see a stability of estimates and a certain level of convergence of the estimates if it takes place.

There is clear tendency in the estimation results presented in Figure 3.36 that, if the dataset includes high volatility periods (2008, 2011, 2014-2016), higher estimates of m_0 and σ_0 lead to lower ρ and λ. The evidence of this trend is the most clear in the jump of parameters estimates after the addition of the period from 2008-09-01 to 2009-08-24 to the dataset used for estimation. The opposite tendency is observed in the case of datasets with lower volatility, for example, in the case of the quiet period from 2016-06-16 to 2018-05-28. This dependence is clear due to m_0 and σ_0 having the greatest impact on the volatility size. It is important to note for further investigation that the estimates differ significantly depending on the time window choice.

The results for the AMSM2 model are similar, but slightly more unstable compared to the AMSM1 model version. This behavior was also observed during simulations (Figures 3.28, 3.29).

The best estimates in the sense of minimal WRSS were achieved for short historical time windows, 180 and 360 trading days. These results are collected in Table 3.22 for both AMSM model versions and for a few additional historical time windows (90, 720, 1080 days).

In order to visualize option pricing, residuals are provided Figure 3.37 with relative standard errors (residuals) and Table 3.27 collects WRSS values achieved for the optimal model

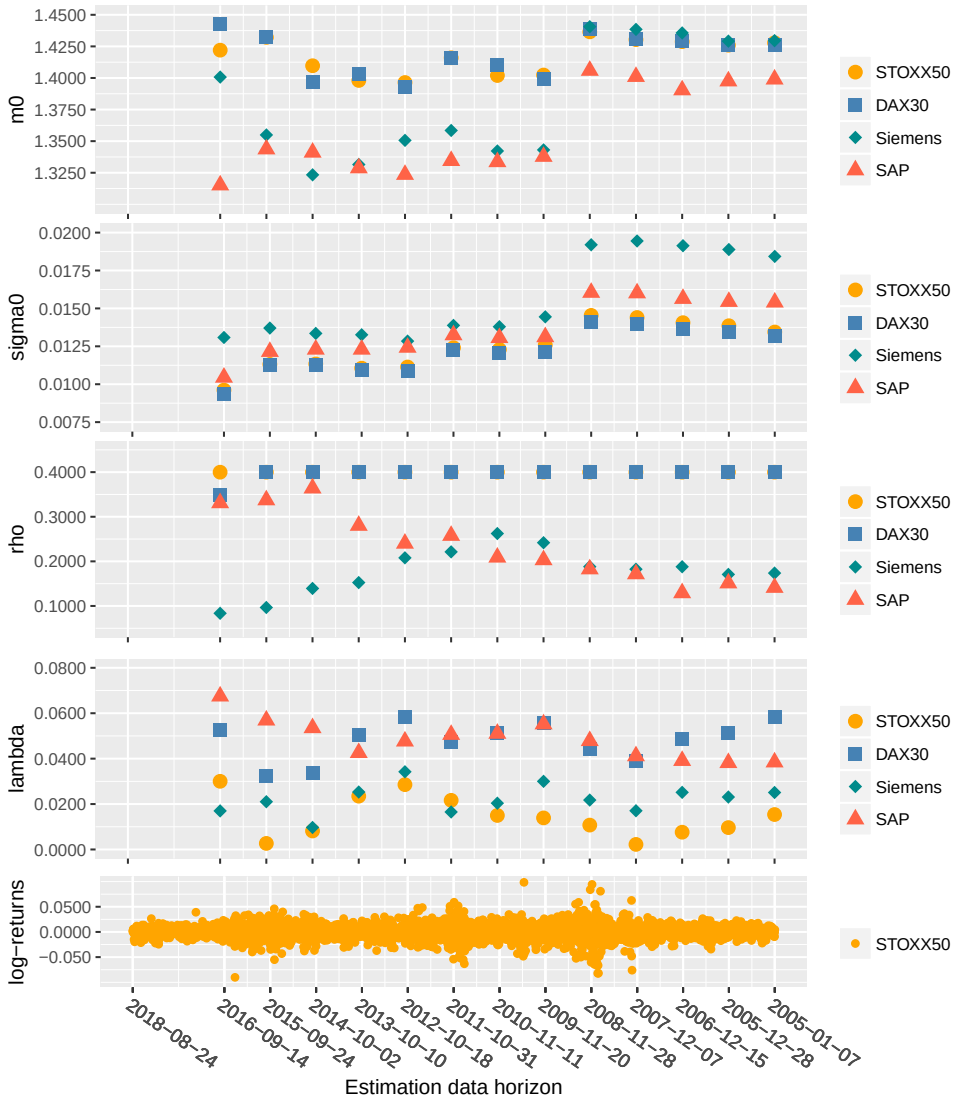


Figure 3.36: The estimates of AMSM1 model parameters. The first four panels depicts estimates of corresponding model parameter for various data horizons, namely from 500 days to 3500 trading days. The fifth panel depicts log-returns of Stoxx 50 index.

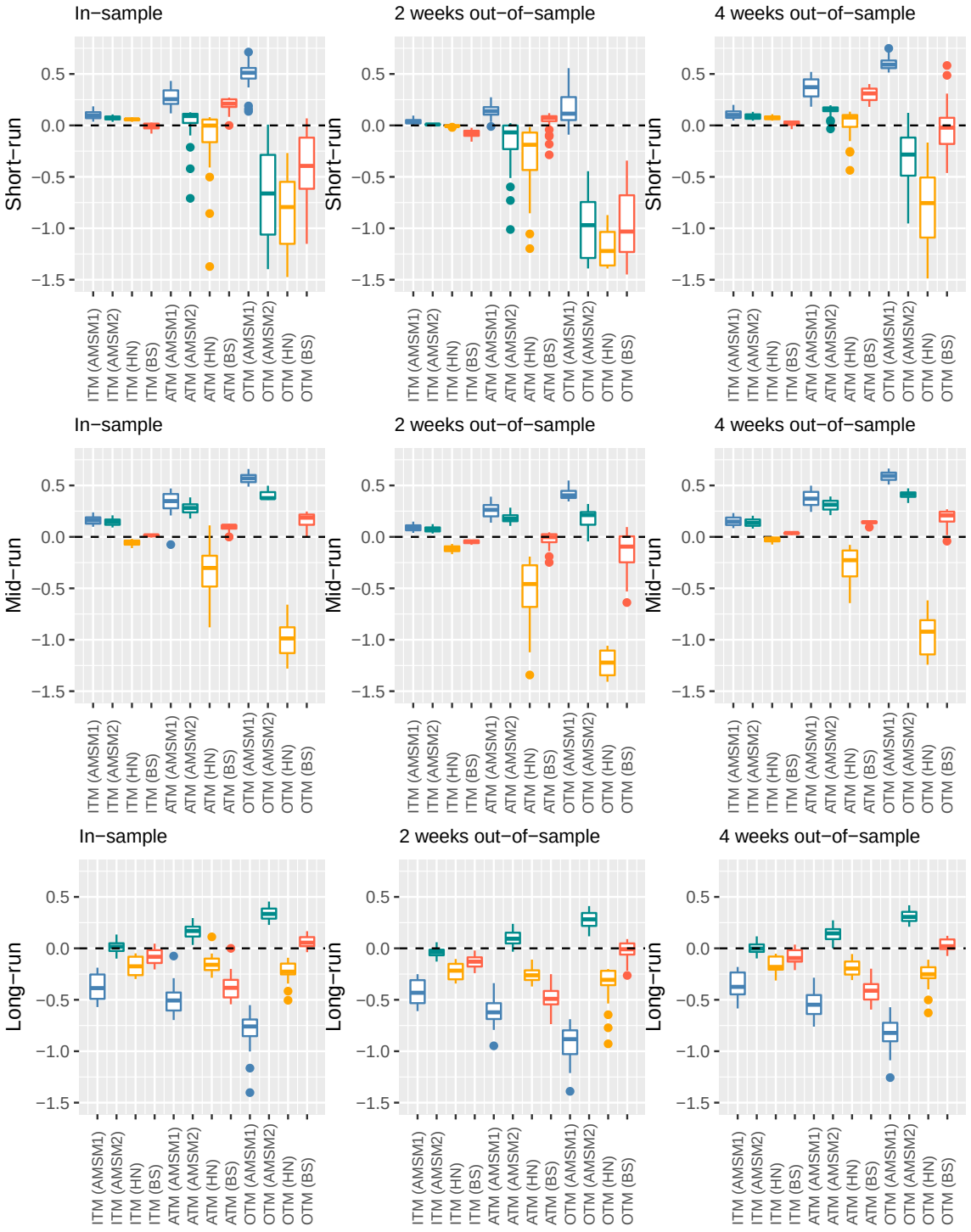


Figure 3.37: Relative standard errors (residuals) of theoretical options prices computation in the case of various maturities and moneyness for AMSM1, AMSM2, Heston-Nandi and BS models based on asset returns data.

parameters (see Table 3.22) analogical to the previous approach.

The first glance at Figure 3.37 leads to a conclusion that there is a clear growing trend of the residuals growth going from in-the-money to out-of-the-money baskets for all models in mid-run and long-run for the AMSM and HN models. The largest relative residuals are observed for short-run out-of-the-money options as well as for the previous approach, namely they are up to -150%.

Another conclusion is that the AMSM2 model outperforms the Heston-Nandi GARCH model version in most cases, especially in the case of the same two weeks out-of-sample datasets (see Table 3.27). In contrast to the results obtained during calibration of option prices data, AMSM2 model version estimated with MLE based on log-returns provides significantly more accurate and precise results than the AMSM1 model version, which is clear from both the figure and the errors size. Further, the only AMSM2 model estimated on asset returns shows up relative residuals comparable to the classic BS model results, but the BS model was calibrated on the options prices data, which is not always available. The BS model results are presented here for informational purposes only.

Maximum likelihood estimation based on asset returns and market option prices

The third and the last method implemented to the real data is MLE based on mixed likelihood functions $L^M(\theta, \lambda, \nu|\mathbf{r}; \mathbf{C})$ defined in Section 3.5.1 and investigated on simulated data in Section 3.5.2. It combines the asset returns data with the options prices data. As in the previous case, the simulated annealing algorithm (ASA) was used to maximize the likelihood. The performance of the method on the simulated data was discussed in Subsection 3.5.2.

Out-of-sample performance was measured the same way as for the first method. The data used for this method is a combination of two datasets from the previous two approaches. The options prices dataset was used exactly the same as for the first method and the datasets with historical asset returns were used of the same size as in the second method: 180 and 360 trading days.

The parameters boundaries of optimization region were extended compared to the previous two cases, since many of the preliminary estimation experiments with L^M-based MLE led to ρ close to the upper boundary. Nevertheless, the best results in the sense of WRSS are achieved not for extremal values of ρ. The best estimates for both AMSM model versions are collected in Table 3.23 for the options baskets and historical frames.

There is no evident advantage in the sense of the relative residuals plotted in Figure 3.38 for L^M-based MLE over their counterparts in Figure 3.35 obtained using only options prices data. Let us look deeper, using Tables 3.24,3.25 and 3.26. The estimates obtained during calibration provide similar WRSS of theoretical option prices for ITM and ATM cases, but the WRSS are significantly lower for the case of OTM options. For example, in the case of two weeks out-of-sample (AMSM1/AMSM2): 13.46/14.93 versus 14.98/19.38, 3.16/3.39 versus 3.81/4.11, 0.48/0.45 versus 0.54/0.6. Thus, the results this approach obtained are very close and do not exceed the ones obtained during calibration in the first approach; therefore, further analysis is omitted.

□

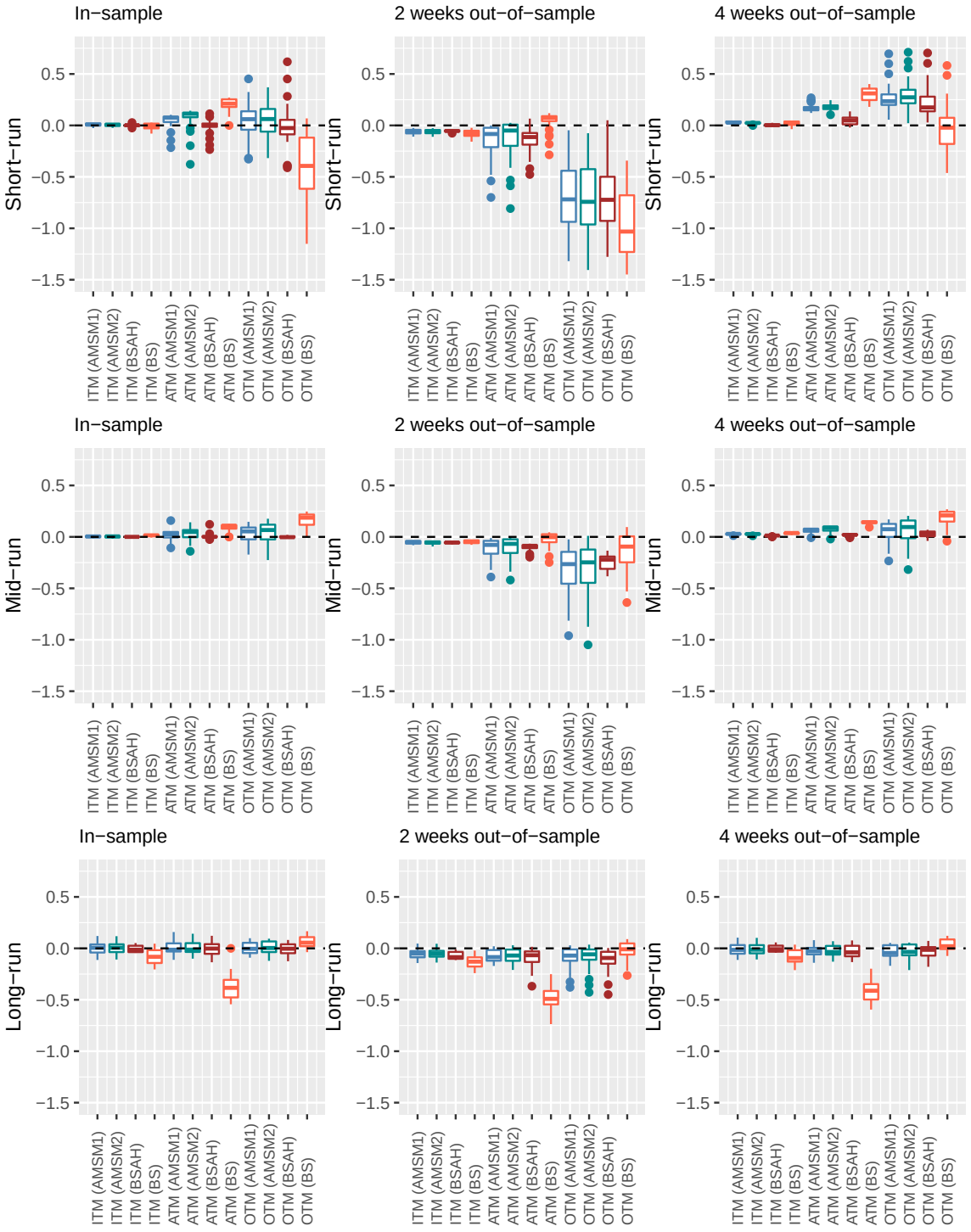


Figure 3.38: Relative standard errors (residuals) of theoretical options prices computation in the case of various maturities and moneyness for AMSM1, AMSM2, Ad-Hoc and BS models based on real market asset returns and options prices data.

Par.	Metrics	L^R	WRSS $T=30$	WRSS $T=30-90$	L^M $T=30$	L^M $T=30-90$
	Real	$(m_0,\sigma_0,\rho,\lambda,\nu)=(1.4,0.02,0.25,0.25,0.05)$				
m_0	Mean	1.391	1.370	1.399	1.379	1.388
	Median	1.390	1.369	1.385	1.375	1.390
	MAD	0.031	0.014	0.014	0.025	0.029
	MAE	0.023	0.030	0.029	0.030	0.025
σ_0	Mean	1.99e-02	2.07e-02	2.01e-02	2.00e-02	1.99e-02
	Median	2.02e-02	2.07e-02	2.01e-02	2.00e-02	1.98e-02
	MAD	1.03e-03	5.17e-05	3.30e-04	2.36e-04	3.43e-04
	MAE	8.47e-04	7.07e-04	3.08e-04	2.10e-04	3.25e-04
ρ	Mean	0.248	0.251	0.242	0.252	0.268
	Median	0.243	0.250	0.261	0.252	0.266
	MAD	0.089	0.010	0.032	0.020	0.023
	MAE	0.062	0.008	0.050	0.014	0.022
λ	Mean	0.253	0.008	0.209	0.251	0.274
	Median	0.252	0.003	0.207	0.250	0.300
	MAD	0.019	0.004	0.160	0.065	0.125
	MAE	0.016	0.242	0.125	0.059	0.098
	Real	$(m_0,\sigma_0,\rho,\lambda,\nu)=(1.4,0.02,0.25,0.5,0.05)$				
m_0	Mean	1.389	1.359	1.399	1.384	1.380
	Median	1.391	1.360	1.402	1.380	1.387
	MAD	0.029	0.015	0.016	0.043	0.051
	MAE	0.021	0.041	0.013	0.037	0.034
σ_0	Mean	1.99e-02	2.15e-02	2.02e-02	2.00e-02	2.00e-02
	Median	2.01e-02	2.15e-02	2.01e-02	2.00e-02	2.01e-02
	MAD	1.02e-03	4.73e-05	5.16e-04	5.93e-04	3.72e-04
	MAE	8.10e-04	1.51e-03	4.77e-04	3.79e-04	3.02e-04
ρ	Mean	0.247	0.255	0.277	0.257	0.271
	Median	0.228	0.253	0.276	0.254	0.265
	MAD	0.099	0.010	0.019	0.039	0.038
	MAE	0.066	0.008	0.031	0.032	0.027
λ	Mean	0.502	0.007	0.431	0.504	0.493
	Median	0.498	0.005	0.460	0.487	0.490
	MAD	0.017	0.008	0.171	0.092	0.136
	MAE	0.017	0.493	0.162	0.071	0.087
	Real	$(m_0,\sigma_0,\rho,\lambda,\nu)=(1.4,0.02,0.25,0.75,0.05)$				
m_0	Mean	1.387	1.352	1.413	1.371	1.394
	Median	1.390	1.351	1.414	1.367	1.398
	MAD	0.019	0.015	0.022	0.041	0.044
	MAE	0.019	0.048	0.034	0.041	0.035
σ_0	Mean	1.99e-02	2.23e-02	2.08e-02	2.03e-02	2.01e-02
	Median	2.00e-02	2.23e-02	2.06e-02	2.04e-02	2.01e-02
	MAD	1.10e-03	5.53e-05	7.91e-04	5.61e-04	6.05e-04
	MAE	7.81e-04	2.33e-03	9.45e-04	5.83e-04	4.10e-04
ρ	Mean	0.246	0.256	0.264	0.234	0.249
	Median	0.236	0.254	0.279	0.237	0.250
	MAD	0.100	0.009	0.026	0.074	0.043
	MAE	0.068	0.009	0.065	0.047	0.038
λ	Mean	0.751	0.011	0.449	0.736	0.733
	Median	0.746	0.008	0.485	0.718	0.729
	MAD	0.020	0.012	0.231	0.055	0.072
	MAE	0.020	0.739	0.339	0.057	0.070

Table 3.18: Three groups of estimation/calibration experiment's distribution for three $\lambda = 0.25, 0.5, 0.75$ and AMSM1 model. Each group based on asset returns (L^R), options prices (WRSS) and mixed returns/options (L^M) data. $\theta^{real} = (m_0;\sigma_0;\rho) = (1.4;0.02;0.25)$, $(\lambda;\nu) = (0.25,0.5,0.75;0.05)$.

	Metrics	L^R	WRSS $T=30$	WRSS $T=30-90$	L^M $T=30$	L^M $T=30-90$
	Real	$(m_0,\sigma_0,\rho,\lambda,\nu)=(1.4,0.02,0.01,0.25,0.05)$				
m_0	Mean	1.389	1.398	1.415	1.377	1.394
	Median	1.384	1.399	1.414	1.372	1.385
	MAD	0.022	0.015	0.016	0.030	0.026
	MAE	0.021	0.012	0.016	0.039	0.030
σ_0	Mean	2.00e-02	2.07e-02	1.99e-02	2.01e-02	1.99e-02
	Median	1.99e-02	2.07e-02	2.06e-02	2.02e-02	2.02e-02
	MAD	6.78e-04	5.44e-05	1.27e-04	3.03e-04	3.96e-04
	MAE	7.72e-04	6.91e-04	1.03e-03	3.65e-04	4.66e-04
ρ	Mean	0.010	0.010	0.012	0.009	0.011
	Median	0.009	0.010	0.012	0.009	0.011
	MAD	0.002	0.001	0.001	0.001	0.001
	MAE	0.001	0.001	0.002	0.002	0.002
λ	Mean	0.250	0.005	0.232	0.217	0.239
	Median	0.252	0.000	0.003	0.209	0.173
	MAD	0.029	0.000	0.004	0.095	0.107
	MAE	0.020	0.245	0.329	0.090	0.113
	Real	$(m_0,\sigma_0,\rho,\lambda,\nu)=(1.4,0.02,0.01,0.5,0.05)$				
m_0	Mean	1.389	1.396	1.416	1.366	1.362
	Median	1.388	1.395	1.412	1.363	1.367
	MAD	0.026	0.017	0.014	0.020	0.043
	MAE	0.022	0.014	0.016	0.037	0.048
σ_0	Mean	1.99e-02	2.14e-02	2.07e-02	2.02e-02	2.03e-02
	Median	2.00e-02	2.14e-02	2.13e-02	2.03e-02	2.03e-02
	MAD	1.01e-03	4.92e-05	1.49e-04	3.87e-04	3.38e-04
	MAE	9.01e-04	1.43e-03	1.32e-03	4.04e-04	4.12e-04
ρ	Mean	0.009	0.010	0.012	0.008	0.009
	Median	0.009	0.010	0.012	0.007	0.009
	MAD	0.001	0.001	0.001	0.002	0.002
	MAE	0.001	0.001	0.002	0.002	0.002
λ	Mean	0.503	0.002	0.201	0.512	0.473
	Median	0.503	0.000	0.027	0.509	0.492
	MAD	0.024	0.000	0.040	0.068	0.063
	MAE	0.020	0.498	0.451	0.058	0.060
	Real	$(m_0,\sigma_0,\rho,\lambda,\nu)=(1.4,0.02,0.01,0.75,0.05)$				
m_0	Mean	1.394	1.392	1.418	1.352	1.351
	Median	1.387	1.391	1.415	1.352	1.344
	MAD	0.020	0.017	0.015	0.021	0.025
	MAE	0.021	0.015	0.019	0.048	0.049
σ_0	Mean	2.03e-02	2.21e-02	2.11e-02	2.08e-02	2.09e-02
	Median	2.00e-02	2.21e-02	2.20e-02	2.09e-02	2.08e-02
	MAD	1.39e-03	4.49e-05	2.73e-04	3.64e-04	3.05e-04
	MAE	1.20e-03	2.14e-03	1.74e-03	8.24e-04	8.98e-04
ρ	Mean	0.008	0.010	0.012	0.007	0.007
	Median	0.008	0.009	0.012	0.007	0.007
	MAD	0.001	0.001	0.001	0.002	0.002
	MAE	0.002	0.001	0.002	0.003	0.003
λ	Mean	0.746	0.010	0.293	0.694	0.702
	Median	0.736	0.001	0.039	0.707	0.718
	MAD	0.021	0.001	0.057	0.070	0.083
	MAE	0.027	0.740	0.573	0.076	0.080

Table 3.19: Three groups of estimation/calibration experiment's distribution for three $\lambda = 0.25, 0.5, 0.75$ and AMSM2 model. Each group based on asset returns (L^R), options prices (WRSS) and mixed returns/options (L^M) data. $\theta^{real} = (m_0;\sigma_0;\rho) = (1.4;0.02;0.01)$, $(\lambda;\nu) = (0.25,0.5,0.75;0.05)$.

Moneyness	Maturity	m_0	σ_0	ρ	λ	ν	WRSS
			AMSM1				
ITM	SR	1.7187	0.0110	2.7675	-0.8300	1.0427	0.0005
ITM	MR	1.8484	0.0096	4.0818	0.0741	-0.9103	0.0000
ITM	LR	1.6642	0.0212	0.1881	-1.8684	-1.9118	0.0521
ATM	SR	1.4049	0.0053	4.9367	-0.2814	1.5187	0.1253
ATM	MR	1.5427	0.0126	1.1789	-1.9416	1.7478	0.0340
ATM	LR	1.5592	0.0127	1.1536	0.0495	-1.4741	0.0845
OTM	SR	1.3150	0.0115	2.4397	-1.5840	1.9484	0.4698
OTM	MR	1.3508	0.0087	3.9341	-0.9220	1.7293	0.0819
OTM	LR	1.0970	0.0059	3.9721	0.1575	0.8243	0.0880
			AMSM2				
ITM	SR	1.6575	0.0201	0.0652	-0.8723	-0.3984	0.0013
ITM	MR	1.6166	0.0366	0.0732	-1.9970	-0.1533	0.0000
ITM	LR	1.5168	0.0140	0.0018	1.0877	-1.7085	0.0499
ATM	SR	1.2303	0.0048	0.0220	-0.9640	1.5861	0.1837
ATM	MR	1.0577	0.0112	0.0452	-1.2724	0.9699	0.0578
ATM	LR	1.3544	0.0032	0.0637	-1.8556	-1.0226	0.0851
OTM	SR	1.0594	0.0083	0.0410	-0.8636	-0.1146	0.4064
OTM	MR	1.0565	0.0171	0.0348	-1.8285	-0.6712	0.0962
OTM	LR	1.0915	0.0011	0.0405	0.5522	-0.1639	0.0796

Table 3.21: Optimal estimates obtained by the calibration based on real market options prices using ASA optimization method.

Hist.horizon	m_0	σ_0	ρ	λ	ν	Likelihood
			AMSM1			
90	1.1372	0.0103	47.4035	-0.0770	1.4103	306.7150
180	1.1955	0.0097	44.2026	-0.0898	-0.9302	622.1550
360	1.2872	0.0074	0.7027	0.0009	-1.1876	1283.3400
720	1.3162	0.0099	0.6828	0.0121	0.5399	2376.2100
1080	1.2943	0.0100	0.7546	0.0063	0.9240	3456.3700
			AMSM2			
90	1.2137	0.0083	0.0039	0.0205	0.1147	305.8210
180	1.2348	0.0079	0.0045	-0.0459	-0.7980	618.4520
360	1.2770	0.0074	0.0042	0.0077	-0.0873	1279.9900
720	1.3164	0.0106	0.0051	0.0191	1.9538	2368.9800
1080	1.3019	0.0108	0.0056	0.0127	-0.2157	3445.7600

Table 3.22: Optimal estimates obtained by the estimation of asset prices using maximization of likelihood L^R with ASA optimization method.

Moneyness	Maturity	Hist.horizon	m_0	σ_0	ρ	λ	ν	Likelihood
				AMSM1				
ITM	SR	360	1.3375	0.0118	0.9682	-0.0186	-0.5541	1383.0100
ITM	MR	360	1.2854	0.0086	0.9052	-0.0622	0.1186	1324.3500
ITM	LR	180	1.2757	0.0077	0.9833	-0.0646	-1.9829	672.7520
ATM	SR	360	1.2510	0.0055	6.6845	0.0534	1.2139	1338.6800
ATM	MR	360	1.2420	0.0067	3.2521	0.0241	0.5784	1345.2100
ATM	LR	180	1.2795	0.0094	0.8531	-0.0750	-1.4864	680.3000
OTM	SR	360	1.3027	0.0073	0.7187	-0.0352	-0.2304	1330.8300
OTM	MR	360	1.2603	0.0070	1.5579	0.0403	-0.1101	1306.9500
OTM	LR	180	1.2259	0.0071	1.6467	-0.0872	-0.1455	686.3250
				AMSM2				
ITM	SR	360	1.4245	0.0160	0.0063	0.0190	-0.7765	1380.0300
ITM	MR	180	1.3412	0.0137	0.0054	-0.0437	-0.6019	657.3530
ITM	LR	180	1.2299	0.0078	0.0042	-0.0312	-1.7680	671.0360
ATM	SR	360	1.2636	0.0056	0.0057	0.0106	0.3300	1327.5000
ATM	MR	180	1.2075	0.0082	0.0059	-0.0854	-0.0961	677.4850
ATM	LR	180	1.2727	0.0103	0.0054	-0.0415	-0.9135	678.6030
OTM	SR	360	1.2813	0.0060	0.0040	0.0063	0.0787	1326.8900
OTM	MR	180	1.1965	0.0064	0.0048	-0.0451	0.1296	639.8590
OTM	LR	180	1.1210	0.0073	0.0044	-0.0415	-0.3502	680.6250

Table 3.23: Optimal estimates obtained by the estimation of real market asset returns and options prices using maximization of likelihood L^M with ASA optimization method.

Maturity	Moneyness	AMSM1	AMSM2	Ad-Hoc	BS
		In-sample			
Short-run	ITM	0.00	0.00	0.00	0.03
	ATM	**0.13**	0.18	0.15	1.52
	OTM	0.47	**0.41**	1.40	9.06
Mid-run	ITM	0.00	0.00	0.00	0.00
	ATM	0.05	0.07	**0.02**	0.26
	OTM	0.08	0.10	**0.00**	0.44
Long-run	ITM	0.05	0.05	**0.03**	0.28
	ATM	**0.08**	0.09	0.09	3.99
	OTM	0.09	**0.08**	0.08	0.20
		2 weeks out-of-sample			
Short-run	ITM	0.10	0.09	**0.06**	0.16
	ATM	2.33	2.66	1.14	**0.34**
	OTM	**13.46**	14.93	16.20	68.91
Mid-run	ITM	0.05	0.03	0.03	0.03
	ATM	0.86	0.55	0.33	**0.17**
	OTM	3.16	3.39	1.47	**1.32**
Long-run	ITM	**0.09**	0.09	0.10	0.37
	ATM	**0.32**	0.33	0.48	7.91
	OTM	0.48	0.45	0.66	**0.25**
		4 weeks out-of-sample			
Short-run	ITM	0.01	0.01	**0.00**	0.02
	ATM	0.73	0.75	**0.12**	3.07
	OTM	2.82	3.20	3.44	**1.53**
Mid-run	ITM	0.38	0.03	**0.00**	0.03
	ATM	0.06	0.17	**0.01**	0.50
	OTM	0.29	0.25	**0.04**	0.98
Long-run	ITM	0.05	0.04	**0.03**	0.27
	ATM	0.10	**0.09**	0.12	5.27
	OTM	0.11	**0.10**	0.11	0.11

Table 3.24: Weighted Residual Sum of Squares values of theoretical options prices obtained by the calibration (ASA method) based on the options chains with various maturities and moneyness, two optimization methods: Levenberg-Marquardt algorithm and Adaptive Simulated Annealing.

Maturity	Moneyness	AMSM1	AMSM2	Ad-Hoc	BS
		In-sample			
Short-run	ITM	0.00	0.00	0.00	0.03
	ATM	0.16	0.23	0.15	1.52
	OTM	0.99	13.88	1.40	9.06
Mid-run	ITM	0.00	0.00	0.00	0.00
	ATM	0.06	0.08	0.02	0.26
	OTM	0.20	0.17	0.00	0.44
Long-run	ITM	0.05	0.05	0.03	0.28
	ATM	0.08	0.09	0.09	3.99
	OTM	0.09	0.09	0.08	0.20
		2 weeks out-of-sample			
Short-run	ITM	0.10	0.08	0.06	0.16
	ATM	2.92	2.29	1.14	0.34
	OTM	17.55	**5.90**	16.20	68.91
Mid-run	ITM	**0.03**	0.04	0.03	0.03
	ATM	**0.48**	0.55	0.33	0.17
	OTM	3.87	4.34	1.47	1.32
Long-run	ITM	0.09	0.09	0.10	0.37
	ATM	0.33	0.33	0.48	7.91
	OTM	0.49	0.46	0.66	0.25
		4 weeks out-of-sample			
Short-run	ITM	0.01	0.02	0.00	0.02
	ATM	13.44	**0.74**	0.12	3.07
	OTM	3.50	9.16	3.44	1.53
Mid-run	ITM	**0.01**	**0.01**	0.00	0.03
	ATM	0.11	0.19	0.01	0.50
	OTM	0.51	0.36	0.04	0.98
Long-run	ITM	0.05	0.05	0.03	0.27
	ATM	0.10	0.10	0.12	5.27
	OTM	0.11	0.11	0.11	0.11

Table 3.25: Weighted Residual Sum of Squared values of theoretical options prices obtained by the calibration based on the options chains with various maturities and moneyness, two optimization methods: Levenberg-Marquardt algorithm and Adaptive Simulated Annealing.

Maturity	Moneyness	AMSM1	AMSM2	Ad-Hoc	BS
		In-sample			
Short-run	ITM	0.01	0.00	0.00	0.03
	ATM	0.24	0.54	0.15	1.52
	OTM	0.96	0.95	1.40	9.06
Mid-run	ITM	0.00	0.00	0.00	0.00
	ATM	0.08	0.12	0.02	0.26
	OTM	0.13	0.21	0.00	0.44
Long-run	ITM	0.08	0.07	0.03	0.28
	ATM	0.10	0.09	0.09	3.99
	OTM	0.10	0.11	0.08	0.20
		2 weeks out-of-sample			
Short-run	ITM	0.09	0.09	0.06	0.16
	ATM	1.82	2.04	1.14	0.34
	OTM	14.98	19.38	16.20	68.91
Mid-run	ITM	0.03	0.04	0.03	0.03
	ATM	0.63	0.63	0.33	0.17
	OTM	3.81	4.11	1.47	1.32
Long-run	ITM	0.09	0.08	0.10	0.37
	ATM	0.29	0.31	0.48	7.91
	OTM	0.54	0.60	0.66	0.25
		4 weeks out-of-sample			
Short-run	ITM	0.02	0.01	0.00	0.02
	ATM	0.97	1.08	0.12	3.07
	OTM	3.85	3.60	3.44	1.53
Mid-run	ITM	0.02	0.02	0.00	0.03
	ATM	0.10	0.17	0.01	0.50
	OTM	0.34	0.54	0.04	0.98
Long-run	ITM	0.07	0.06	0.03	0.27
	ATM	0.11	0.10	0.12	5.27
	OTM	0.13	0.14	0.11	0.11

Table 3.26: Weighted Residual Sum of Squares values of theoretical options prices obtained by L^M-MLE estimation (ASA method) based on the options chains with various maturities and moneyness, Adaptive Simulated Annealing is used as an optimization method.

Maturity	Moneyness	AMSM1	AMSM2	HN	BS
		In-sample			
Short-run	ITM	0.33	0.16	**0.10**	0.03
	ATM	2.68	**0.98**	3.42	1.52
	OTM	**9.99**	28.47	172.93	9.06
Mid-run	ITM	0.30	0.24	**0.04**	0.00
	ATM	3.35	**2.16**	4.55	0.26
	OTM	4.57	**2.30**	55.86	0.44
Long-run	ITM	3.60	**0.08**	0.86	0.28
	ATM	6.94	**0.82**	0.86	3.99
	OTM	21.02	3.28	**1.57**	0.20
		2 weeks out-of-sample			
Short-run	ITM	0.04	**0.00**	0.00	0.16
	ATM	**0.91**	2.97	9.27	0.34
	OTM	**1.24**	145.75	528.07	68.91
Mid-run	ITM	0.09	**0.06**	0.14	0.03
	ATM	2.07	**1.02**	10.49	0.17
	OTM	3.91	**0.98**	259.50	1.32
Long-run	ITM	3.37	**0.06**	0.97	0.37
	ATM	12.86	**0.49**	2.20	7.91
	OTM	34.70	**2.27**	4.11	0.25
		4 weeks out-of-sample			
Short-run	ITM	0.32	0.19	**0.14**	0.02
	ATM	4.54	0.76	**0.57**	3.07
	OTM	12.08	**5.21**	54.27	1.53
Mid-run	ITM	0.52	0.43	**0.02**	0.03
	ATM	3.61	**2.43**	2.50	0.50
	OTM	8.71	**4.23**	145.50	0.98
Long-run	ITM	3.11	**0.06**	0.78	0.27
	ATM	8.87	**0.68**	1.28	5.27
	OTM	23.46	2.56	**2.10**	0.11

Table 3.27: Weighted Residual Sum of Squares values of theoretical options prices obtained by L^R-MLE estimation (ASA method) based on the options chains with various maturities and moneyness, Adaptive Simulated Annealing is used as an optimization method.

4

Conclusions

This dissertation presents two very different models, but both are related to economical business cycles. The first model is the agents-based model (ABM) adopting continuous-time Markov chain (CTMC) with finite space state as a mathematical basis for describing experts' sentiments dynamics. Three approaches for estimation of this model have been developed:

1. The first approach is the EM algorithm. The estimates obtained using this approach have good quality in terms of standard deviation and root-mean square errors, especially for the estimates of parameters α_0 and α_1 (confirmed by Table 2.3), at least for the considered model versions with the number of agents $N \leq 25$. For relative standard errors, the method's estimates are also very satisfactory. The main limitation of the first approach is that it is characterized by high computational demand, which makes estimation of the model for a number of agents around 25 take quite a long time (around 20 minutes). In order to overcome this problem, parallel computation or another technique could be applied, nevertheless, the first method seems to be numerically inefficient for ABM model versions with a number of agents N more than around 10. This is due to a high level of recursion and because of the necessity of numerical maximization, which greatly increases computational efforts.

2. The second method, based on numerical computation of transition probabilities, and which could be denoted "eigendecomposition approach", was shown to be very similar to the EM algorithm in quality of estimates for precision and accuracy. That was predicted by the theory of EM algorithms. So, the estimates made by the second method have almost the same standard deviation and root-mean square errors as the EM algorithm for the same model versions. The advantage of the second method is a lower computational cost in comparison with the EM algorithm. This makes it possible to estimate the model in a quicker manner for the same range of N. As a more computationally efficient alternative, the second estimation approach was proposed, but it also has limitations. At the heart of the first and second approaches is incorporated the eigendecomposition of the intensity rates matrix Q. This decomposition

is unstable for large N (more than 30), while a robust method of estimation for N more than 200 is required for many real-world sentiment-based processes. The reason for the instability of the eigendecomposition of Q, is that the intensity matrix Q is a "nearly" defective matrix. Another possible reason is the "nearly" confluence eigenvalues of the matrix Q. There is also another less significant complication with this approach: an analysis of the estimation results revealed the problem of underestimation of the parameter α_1 that occurred with the growing of the constant N defining the number of agent in particular model.

3. Finally, the third approach is based on another method of matrix exponential computation suggested in Moler and Loan's paper, namely the fact that the intensity matrix is a lower Hessenberg matrix, moreover, it is a tridiagonal matrix. This fact, after certain tricks have been applied, enables us to use the recursive procedure of matrix exponential computation to take advantage of the tridiagonal construction of the intensity rates matrix Q of CTMC process. As a result, a robust method for the ABM model estimation was created in case a large number of agents, it could be denoted as "lower Hessenberg matrix approach". The third method uses matrix exponential computation not in the same form as proposed the original one by Moler and Loan, it has a higher computational cost, because of the use of an ordinary differential equation solver approximately $(2N+1)/50$-times for each matrix of transition probabilities computation. Regardless, the method is quite fast (one estimation takes around one minute for $ABM-200$); much faster than the pure method of matrix exponential computation based on ordinary differential equations (which requires to run the o.d.e. solver of system of $2N+1$ equations $2N+1$ times). In other words, the pure method of matrix exponential computation based on the ordinary differential equation is not that computationally efficient compared to the presented one and does not incorporate the special structure of the intensity rate matrix. In addition, the same issue, which was revealed for the second method in terms of precision and accuracy of estimates, was also obtained by the third method: an increase of biases with the growth in the number of agents N. As it was established during this work, this effect is caused by insufficiency of data, but it was shown that the biases vanish with an increase of sample size. So, there is a trade-off between the assumption on the model defining constant of the number of agents N and the sample size influencing the quality of estimates. Therefore, the important result is that the suggested third method enables us to estimate the model in a robust and quite fast manner for N equal to more than 350 at least, but such a number of agents N would require a long data sample.

As mentioned earlier, T.Lux [66] considered the same model, but his approach is based on the parabolic partial differential equation (Fokker-Plank equation), which defines the approximation of transient densities. Then the author uses the numerical solution of it to construct the likelihood function approximation. So, it was possible to compare both approaches and some discussion of this is provided. Summarizing, the method was used for estimation of the ZEW index data sample (1991 to 2017). The estimates of the ZEW index obtained by this method are similar to the ones in the alternative approach of T. Lux [66], but they differ in estimation of time scale parameter due to a different time scale.

There are a few possible directions for a future research. In order to overcome the loss

of significance effect, which is peculiar for the first and the second estimation approaches, the special subroutines based on arbitrary-precision arithmetic rather than common fixed-precision floating point arithmetic can be used. There are a few C++ libraries that can be used, for example Boost (C++ libraries) Multiprecision Library [24], but in general this way seems to be computationally expensive and complicated to apply. Also, there are a few other methods of matrix exponential computations in Moler and Loan's review [72] that used to be quite robust and computationally effective. For example, the method based on QR-decomposition, the so-called "scaling and squaring" method, the method based on Pade-approximation, or the method of matrix multiplication (which is potentially fast in combination with parallel computations on CPU or GPU).

Summing up for the first project of the dissertation, the three presented estimation approaches for the agent-based model of sentiment dynamics, namely the EM algorithm, "eigendecomposition" and "lower Hessenberg matrix" approaches, allow us to estimate the model with the same quality. The first two have limitations on the number of agents in the model, plus the EM algorithm is less computationally efficient and more difficult to implement. Comparing the second and third approaches, the eigendecomposition method is easier to implement (it is a well-known procedure included in many software and mathematical packages), and it is quite fast. The lower Hessenberg matrix approach with a modification has higher computation stability, allowing it to estimate ABM models with a larger number of agents, which is complemented with good computation efficiency, but it requires a bit more efforts to programme it.

The second project of the dissertation is dedicated to the theoretical basis, estimation techniques and goodness-of-fit of the model which belongs to the class of Markov Switching Multifractal models. One of the key features of the models in this class is an ability to describe various economic and business cycles of different periodicity. During this work the *new modification* of the original Markov-Switching Multifractal model of Calvet and Fisher was developed, and it is aimed at incorporation of a leverage effect stylized fact in the model. This model, called an Asymmetric Markov-Switching Multifrequency model, was formulated in two versions, the new one denoted in the text as AMSM2 model and Leövey's modification [59] denoted as AMSM1 model in the text. In this dissertation, in order to guarantee the properties of the original MSM model were preserved, a few lemmas and a few theorems were developed and proved. Namely, these theorems prove a mean-reversion property, a long memory of volatility and a leverage effect property, pivotal for this research for both versions of the AMSM model.

Another group of formulated and proven theorems in this dissertation was dedicated to providing a theoretical basis of option pricing with the AMSM model (both versions). First of all, the stochastic discount factor adopted from Duan's approach of GARCH option pricing gave us an economical sense of switch between physical and risk-neutral measures in terms of utility function. Besides, this approach allowed us to construct the Local Risk-Neutral Valuation Relation (LRNVR) measure for the AMSM model. As a final theoretical result, it was proved that this measure is a martingale measure in the case of AMSM model (both versions). The LRNVR measure allowed to define an option price as a mathematical expectation with respect to it. This expectation has no a closed-form solution only the numerical one in the case of both AMSM model versions. The lack of a closed-form solution for an option price leads to the use of the Monte Carlo method.

To provide the ability of practical application of the AMSM model, the technique of

the model parameters' estimation is necessary. Three estimation approaches of AMSM model parameters were proposed in this research for both model versions. The first one is the calibration of the model on real market option prices data. Another two approaches maximize the likelihood functions constructed on underlying asset returns only, or both asset returns and real market options prices. The resolution of many uncertainties took a lot of effort during the estimation stage: two model versions, two optimization methods of likelihood function maximization and loss function minimization, two kinds of random numbers for Monte Carlo sample paths simulations and three kinds of datasets (historical returns, real market option prices or both). The quantitative stage of this work was aimed at revealing the best combination of these factors and their impact, but let us summarize it step-by-step.

The first estimation approach is aimed at minimization of the weighted sum of squared residual between a real market or artificially simulated[1] cross-section of option prices and their theoretical counterparts calculated using the AMSM model. This approach is known also as calibration. The calibration experiments on synthetic option prices data established, that the cross-sections of options with multiple maturities are significantly more efficient for the calibration of AMSM model than the cross-section with the only one maturity. In particular, the calibration procedure based on the ASA optimization method provides the best results in the sense of accuracy and precision in the case of the AMSM1 model version, while the procedure based on the Levenberg-Marquardt algorithm has vast superiority over the one based on Adaptive Simulated Annealing in the case of the AMSM2 model version. The comparison of AMSM model parameters calibration results for two versions of AMSM model revealed that the AMSM1 model version is significantly less robust; it is more noisy and biased in general. Thus, the AMSM2 model variant and its calibration appeared to be more promising during the simulation stage for practical implementation with a real market data.

The equity unit-risk premium (EURP)[2] is considered an exogenous parameter in the calibration procedure, while the second and the third approaches jointly estimate both, the AMSM model parameters and EURP. The underlying asset log-returns data was used to enhance the estimation procedure. It is important to note, that an estimation of ERP is considered to be a tricky and controversial task in the literature. In order to solve this estimation problem, two likelihood functions were constructed. The construction of the first one is based on only historical log-returns data. It establishes the second AMSM model estimation approach via a maximization of this likelihood with respect to the model parameters (MLE method). The third suggested approach adds the second component incorporating real market options prices data to this likelihood function. As a result, the procedure of maximization of this two-component likelihood function establishes the third estimation approach.

During the simulation and estimation experiments with the second approach, it was revealed that the second estimation approach is adequate with an estimation of the AMSM

[1] In this case the couples of option prices are calculated with the AMSM model but using different seeds for the Monte Carlo method. The first price in any of these couple is calculated with the predefined parameters of the AMSM model, while the parameters necessary to calculate the second one are assumed to be unknown and be the subject of a calibration procedure.

[2] The equity risk premium is assumed to be the product of the volatility and the equity unit-risk premium in the AMSM model.

model parameters and equity unit-risk premium; the most difficult for estimation by this approach is the leverage parameter. The MLE based on the likelihood constructed over log-returns and option prices datasets has turned out to be a good trade-off in the sense of accuracy and precision of the estimates of the model parameters and ERP for the AMSM1 model. At the same time, the best quality of estimates on artificial data over all combinations of optimization methods and model versions was achieved for the MLE based on the likelihood constructed over log-returns only and AMSM2 model version, at least for estimation of the crucial terms of the model: leverage effect parameter and the equity unit-risk premium.

The implementation of all three estimation approaches was tested by the out-of-sample performance of options pricing using the same methodology and data in order to obtain comparable results. The difference is in the benchmark model used for the second approach. Namely, the Heston-Nandi closed-form option valuation solution has been used as a benchmark; as is the second approach, this solution is based on only underlying asset returns, which makes the comparison more valid. As a result, the superiority of evaluation based on the AMSM model was established. The results of the implementation of the other two approaches are less straightforward. The first approach (calibration on the real market option prices) turned out to be superior in the sense of out-of-sample performance of option pricing over the benchmark Ad-Hoc model, at least for long-run option pricing. This is logical in the sense of the theoretical properties of the AMSM model, in particular, its long memory, leverage effect and the multifrequency properties of the volatility component of the AMSM model. In general, the third estimation approach, despite being quite robust during the simulations, did not provide advantages over the first approach in the sense of out-of-sample performance and settings used, but it allows to estimate EURP jointly with the model parameters.

There are a few options for further research directions. During the simulations stage, a tight σ_0 and ρ connection was revealed, thus the model could incorporate this fact in order to decrease the number of parameters. This could make the model parameters estimation easier and more straightforward. An estimation of the equity and variance risk premium by sequential estimation approaches in Section 3.5.1 are another point of further research, but it still could be better to assume the risk premiums as endogenous parameters in order to increase robustness of the calibration and estimation procedures of the AMSM model.

A

Appendices

A.1. Proof of Theorem 1 known as Kolmogorov's theorem.

Proof. The formal definition of non-successive transition from any state i at time s to any other state j (see Figure 2.1), taking into account Markov property of $\mathbf{X}$, is given by

$$P_{ij}^X(t) = P\left(X_{s+t} = \frac{j}{N}\middle| X_s = \frac{i}{N}\right) = P\left(X_t = \frac{j}{N}\middle| X_0 = \frac{i}{N}\right).$$

If there is an intermediate moment of time $\tau < t$ during a transition from i to j, then due to the *Chapman-Kolmogorov equation*

$$P_{ij}^X(t) = P\left(X_t = j/N\middle| X_0 = i/N\right) = \sum_{k\in I} P_{ik}^X(\tau)\, P_{kj}^X(t-\tau). \tag{A.1}$$

The probability of two transitions during time interval $(\tau, t]$ is negligible for a sufficiently close τ and t, then the probability of not necessary successive transition $P_{ij}^X(t-\tau)$ and the probability $P_{ij}^Y(t-\tau)$ of successive transition within time not greater than $t-\tau$ (see Definition 2.5) are equal. Since, $t-\tau$ is small, then the Taylor series expansion ("T.exp.") of exponential function allows us to yield the following result

$$\begin{aligned}
P_{ij}^X(t-\tau) &= P_{ij}^Y(t-\tau) \\
&= \left[1-\exp(-q_i(t-\tau))\right] P_{ij}^e \\
&\overset{T.exp.}{=} q_i P_{ij}^e(t-\tau) + o(t-\tau), \\
&= q_{ij}(t-\tau) + o(t-\tau), \\
P_{ii}^X(t-\tau) &= 1 - \sum_{i\neq j} P_{ij}^X(t-\tau) \\
&= 1 - (t-\tau)\sum_{i\neq j} q_{ij} + o(t-\tau) \\
&= 1 - (t-\tau)q_i + o(t-\tau),
\end{aligned}$$

if to substitute these expressions into Chapman-Kolmogorov equation (A.1)

$$P_{ij}^X(t) = P_{ij}^X(\tau)\left(1-(t-\tau)q_j\right) + \sum_{k\neq j} P_{ik}^X(\tau)q_{kj}(t-\tau) + o(t-\tau), \tag{A.2}$$

after some algebra

$$\frac{P_{ij}^X(t) - P_{ij}^X(\tau)}{t-\tau} = \sum_{k\neq j} P_{ik}^X(\tau)q_{kj} - P_{ij}^X(\tau)q_j + o(t-\tau).$$

Then, taking the limit $\tau \to t$, the equation known as the *Kolmogorov forward equations* for continuous-time Markov processes is turned out

$$\frac{dP_{ij}^X(t)}{dt} = \sum_{k\neq j}\left(P_{ik}^X(t)q_{kj}\right) - P_{ij}^X(t)q_j.$$

This equation is known as the Master equation in the case of jump processes, while it is known as the Fokker-Plank equation in the case of diffusion processes.

There is also a matrix form of the forward Kolmogorov equations

$$\frac{dP^X(t)}{dt} = P^X(t)Q,$$

where Q is a matrix of intensity rates q_{ij}.

Note, if we assume that the time interval τ to be small enough and consists of not more than one transition, then the Taylor expansion can be used for an approximation of probabilities $P_{ij}^X(\tau)$ in the right hand side term of the Chapman-Kolmogorov expression (A.1), which gives us

$$P_{ij}^X(t) = \left(1-\tau q_i\right)P_{ij}^X(t-\tau) + \sum_{k\neq j} q_{ik}\tau P_{kj}^X(t-\tau) + o(\tau),$$

after some algebra

$$\frac{P_{ij}^X(t) - P_{ij}^X(t-\tau)}{\tau} = \sum_{k\neq j} q_{ik}P_{kj}^X(t-\tau) - q_i P_{ij}^X(t-\tau) + o(\tau).$$

Then taking the limit $\tau \to 0$, the equations known as the *Kolmogorov backward equations* for continuous-time Markov processes are derived, that is

$$\frac{dP_{ij}^X(t)}{dt} = \sum_{k\neq j}\left(q_{ik}P_{kj}^X(t)\right) - q_i P_{ij}^X(t),$$

or in the matrix form

$$\frac{dP^X(t)}{dt} = QP^X(t),$$

where Q is a matrix of intensity rates q_{ij}.

□

A.2. Proof of Lemma 1.

Proof.

$$\begin{aligned} f^Y_{i,j}(x) &\overset{def}{=} \frac{dP^Y_{ij}(x)}{dx} \\ &= P^e_{ij}\frac{dP^U_i(x)}{dx} \\ &= P^e_{ij}\frac{d}{dx}(1-\exp(-q_i x)) \\ &= P^e_{ij} q_i \exp(-q_i x). \end{aligned}$$

□

A.3. Proof of Theorem 2.

Proof. According to definitions of likelihood and probability density functions

$$L^c(\theta|x) \overset{def}{=} \prod_{k=0}^{M-1} f^Y_{X_{S_k}X_{S_{k+1}}}(U_{k+1}) \tag{A.3}$$

$$\overset{(2.20)}{=} \prod_{k=0}^{M-1} P^e_{X_{S_k}X_{S_{k+1}}}\left[q_{X_{S_k}}\exp(-q_{X_{S_k}}U_{k+1})\right] \tag{A.4}$$

$$= \left[\prod_{k=0}^{M-1} P^e_{X_{S_k}X_{S_{k+1}}} q_{X_{S_k}}\right]\left[\prod_{k=0}^{M-1}\exp(-q_{X_{S_k}}U_{k+1})\right], \tag{A.5}$$

where $S_0 < \ldots < S_k < \ldots < T$ are moments of time when transitions occur in the data sample $\mathbf{x}$, X_{S_k} is a state of $\mathbf{X}$ at time S_k, while $U_k = S_k - S_{k-1}$.

Further, it is more convenient to consider the two multipliers in the brackets (A.5) separately. The idea is to proceed from the products with respect to the process $\mathbf{X}$ successive states at transition times S_k to products based on counting of each kind of transitions $N_{ij}(T)$ and time spend in each state $R_i(T)$. The first multiplier in (A.5) can be expressed as

$$\prod_{k=0}^{M-1} P^e_{X_{S_k}X_{S_{k+1}}} q_{X_{S_k}} \overset{(2.7)}{=} \prod_{k=0}^{M-1} q_{X_{S_k}X_{S_{k+1}}} = \prod_{i=-N}^{N}\prod_{j\neq i} q_{ij}^{N_{ij}(T)},$$

while the second multiplier in (A.5) is transformed as

$$\begin{aligned}\prod_{k=0}^{M-1}\exp(-q_{X_{S_k}}U_{k+1}) &= \exp\left(-\sum_{k=0}^{M-1}q_{S_k}U_{k+1}\right)\\ &\overset{(2.19)}{=}\exp\left(-\sum_{i=-N}^{N}q_iR_i(T)\right)\\ &=\prod_{i=-N}^{N}\exp(-q_iR_i(T))\\ &\overset{(2.8)}{=}\prod_{i=-N}^{N}\exp\left(-R_i(T)\sum_{j\neq i}q_{ij}\right)\\ &=\prod_{i=-N}^{N}\prod_{j\neq i}\exp(-q_{ij}R_i(T)).\end{aligned}$$

Combining the former results

$$\begin{aligned}L^c(\theta|x) &= \left[\prod_{i=-N}^{N}\prod_{j\neq i}q_{ij}^{N_{ij}(T)}\right]\left[\prod_{i=-N}^{N}\prod_{j\neq i}\exp(-q_{ij}R_i(T))\right]\\ &=\prod_{i=-N}^{N}\prod_{j\neq i}q_{ij}^{N_{ij}(T)}\exp(-q_{i,j}R_i(T)).\end{aligned}$$

□

A.4. Proof of Corollary 3.

Proof. The proof[1] begins with the definition of likelihood function given in Theorem 2

$$\begin{aligned}l^c(\theta|x) &\overset{Thm.2}{=}\log\left(\prod_{i=-N}^{N}\prod_{j\neq i}q_{ij}^{N_{ij}}(\theta)\exp\left(-q_{ij}(\theta)R_i\right)\right)\\ &=\log\left(\prod_{i=-N}^{N}q_{ii+1}^{N_{ii+1}}(\theta)q_{ii-1}^{N_{ii-1}}(\theta)\exp\left(-[q_{ii+1}(\theta)+q_{ii-1}(\theta)]R_i\right)\right)\\ &=\sum_{i=-N}^{N}\left[N_{ii+1}\log(q_{ii+1}(\theta))+N_{ii-1}\log(q_{ii-1}(\theta))-[q_{ii+1}(\theta)+q_{ii-1}(\theta)]R_i\right].\\ &=\sum_{i=-N}^{N}\left[(N_{ii+1}+N_{ii-1})\log(\nu)+(N_{ii+1}-N_{ii-1})\left(\alpha_0+\alpha_1\frac{i}{N}\right)\right]\\ &-\sum_{i=-N}^{N}\left[\nu R_i\left(\exp\left(\alpha_0+\alpha_1\frac{i}{N}\right)+\exp\left(-\alpha_0-\alpha_1\frac{i}{N}\right)\right)\right].\end{aligned}$$

[1] The holding times $R_i(T)$ are reduced to R_i for simplicity of denotations in this proof.

Next, the partial derivatives with respect to all parameters $\theta = (\nu, \alpha_0, \alpha_1)$ are

$$\frac{\delta l^c(\theta|x)}{\delta \nu} = \sum_{i=-N}^{N} \left[(N_{ii+1} + N_{ii-1}) \frac{1}{\nu} - R_i \left(\exp\left(\alpha_0 + \alpha_1 \frac{i}{N}\right) + \exp\left(-\alpha_0 - \alpha_1 \frac{i}{N}\right) \right) \right] = 0,$$

$$\frac{\delta l^c(\theta|x)}{\delta \alpha_0} = \sum_{i=-N}^{N} \left[(N_{ii+1} - N_{ii-1}) - \nu R_i \left(\exp\left(\alpha_0 + \alpha_1 \frac{i}{N}\right) - \exp\left(-\alpha_0 - \alpha_1 \frac{i}{N}\right) \right) \right] = 0,$$

$$\frac{\delta l^c(\theta|x)}{\delta \alpha_1} = \sum_{i=-N}^{N} \frac{i}{N} \left[(N_{ii+1} - N_{ii-1}) - \nu R_i \left(\exp\left(\alpha_0 + \alpha_1 \frac{i}{N}\right) - \exp\left(-\alpha_0 - \alpha_1 \frac{i}{N}\right) \right) \right] = 0.$$

□

A.5. Proof of Theorem 5.

Proof. The idea of the proof is based on Propositions 3.6, 3.7 in the book of P. Guttorp [41] for the case of unconditional expectations.

Assume the process $\mathbf{X}$ was continuously observed until time t and we have complete data about its behavior, in particular, the number of its transitions $N_{ij}(t)$ from all combinations of states i and j and time periods it spent in each state $R_i(t)$.

It is possible to split the observation interval $[0, t]$ into subintervals such that no more than one transition occurs per each segment. More formally, let us divide $[0, t]$ into D subintervals $[(r-1)h, rh]$, $r = 1, \ldots, D$, where the number of transitions from i to j in segment r is denoted as $n_r(i, j)$, therefore $N_{ij}(t) = \sum_{r=1}^{D} n_r(i, j)$. Next, h can be chosen small enough to $n_r(i, j)$ would take two values (0 or 1) almost surely, in other words with the probability $1 - o(h)$. Then, let us consider the probability of $n_r(i, j)$ to be equal unity jointly the fact that the process $\mathbf{X}$ initial state $X_0 = k$ and the final observed state $X_t = l$. Therefore, the process $\mathbf{X}$ was in arbitrary state $a \in I$ in the beginning of the considered observation segment r, then it transited to some arbitrary state $b \in I$, thus

$$\begin{aligned} &P\left(n_r(i, j) = 1, X_0 = k, X_t = l\right) = \\ &= \sum_{a,b \in I} P\left(n_r(i, j) = 1, X_0 = k, X_{(r-1)h} = a, X_{rh} = b, X_t = l\right). \end{aligned} \tag{A.6}$$

The probability inside of the sum on the r.h.s. can be rearranged as a product of a few probabilities due to the Markov property of $\mathbf{X}$. The first one is the probability to transit (likely through intermediate states) from k to a within comparably long time $(r-1)h$. This probability is a solution of the Kolmogorov equation, namely the matrix exponential $P^X(\cdot)$ obtained in Theorem 1. The second one is a probability of transition during the *small* interval indicated by $n_r(i, j) = 1$ in conjunction with the fact $X_{(r-1)h} = a$ and $X_{rh} = b$. The final one is the probability of transition from b to l within *long* time $t - rh$ which is also an

element of matrix exponential $P^X(\cdot)$. Therefore,

$$P\left(n_r(i,j)=1, X_0=k, X_{(r-1)h}=a, X_{rh}=b, X_t=l\right) =$$
$$= P^X_{k,a}((r-1)h)P\left(n_r(i,j)=1, X_{(r-1)h}=a, X_{rh}=b\right)P^X_{b,l}(t-rh).$$

The value of the central probability on the r.h.s. of the expression above depends on which of two cases $a \neq i,\ b \neq j$ or $a=i,\ b=j$ occurred

$$P\left(n_r(i,j)=1, X_{(r-1)h}=a, X_{rh}=b\right) = \begin{cases} P^Y_{a,i}(h)P^Y_{i,j}(h)P^Y_{j,b}(h)+o(h), & \text{if } a\neq i,\ b\neq j, \\ P^Y_{a,b}(h)+o(h), & \text{if } a=i,\ b=j \end{cases} \tag{A.7}$$

In the first case, it means we need to calculate the probability of three transitions ($a \to i \to j \to b$) during the time interval chosen to have only one transition almost surely. It is possible to consider this probability as a product of separate probabilities[2] to have three successive transitions in time not greater than length of the interval h, denoted as $P^Y_{\cdot,\cdot}(\cdot)$ (the difference between P^X and P^Y discussed earlier and visualized by Figure 2.9). The calculations are similar to the proof of Theorem 1; it also uses Taylor expansion, denoted as "T.exp." in the calculations below, of exponential function. So, the following holds in the case $a \neq i,\ b \neq j$

$$\begin{aligned} &P\left(n_r(i,j)=1, X_{(r-1)h}=a, X_{rh}=b\right) = P^Y_{a,i}(h)P^Y_{i,j}(h)P^Y_{j,b}(h)+o(h) \\ &= \left[1-\exp(-q_a h)\right]P^e_{ai}\left[1-\exp(-q_i h)\right]P^e_{ij}\left[1-\exp(-q_j h)\right]P^e_{jb}+o(h) \\ &\overset{T.exp.}{=} \left[q_a P^e_{ai}h+o(h)\right]\left[q_i P^e_{ij}h+o(h)\right]\left[q_j P^e_{jb}h+o(h)\right]+o(h), \\ &= \left(q_{ai}h+o(h)\right)\left(q_{ij}h+o(h)\right)\left(q_{jb}h+o(h)\right)+o(h) \\ &= O(h^3)=o(h) \end{aligned}$$

In the second case ($a=i,\ b=j$), it is proven similarly

$$P\left(n_r(i,j)=1, X_{rh}=j, X_{(r-1)h}=i\right) = q_{ij}h+o(h).$$

It means that, in the sum (A.6) the unique term is not of order $o(h)$, namely the term with $a=i, b=j$. Therefore,

$$\begin{aligned} &P\left(n_r(i,j)=1, X_t=l, X_0=k\right) \\ &= P^X_{k,i}((r-1)h)P\left(n_r(i,j)=1, X_{rh}=j, X_{(r-1)h}=i\right)P^X_{j,l}(t-rh)+o(h) \\ &= P^X_{k,i}((r-1)h)q_{ij}hP^X_{j,l}(t-rh)+o(h). \end{aligned}$$

[2]This probability is given in Definition 4.

Going back to the main statement of the theorem, the following expectation should be calculated

$$\begin{aligned}
E_{\hat{\theta}_m}[N_{ij}(t)|X_t=l,X_0=k] &= \sum_{r=1}^{D} P\left(n_r(i,j)=1\middle|X_t=l,X_0=k\right)+o(h)\\
&= \sum_{r=1}^{D}\frac{P\left(n_r(i,j)=1,X_t=l,X_0=k\right)}{P(X_t=l,X_0=k)}+o(h)\\
&= \sum_{r=1}^{D}\frac{P\left(n_r(i,j)=1,X_t=l,X_0=k\right)}{P_{kl}^{X}\left(t;\hat{\theta}_m\right)}+o(h)
\end{aligned}$$

where $o(h)$ denotes negligible probability of other possible outcomes. Finally, we obtain

$$\begin{aligned}
E_{\hat{\theta}_m}[N_{ij}(t)|X_t=l,X_0=k] &= \sum_{r=1}^{D}\frac{P_{k,i}^{X}\left((r-1)h;\hat{\theta}_m\right)q_{ij}\left(\hat{\theta}_m\right)hP_{j,l}^{X}\left(t-rh;\hat{\theta}_m\right)}{P_{kl}^{X}\left(t;\hat{\theta}_m\right)}+o(h)\\
&= \frac{q_{ij}\left(\hat{\theta}_m\right)}{P_{kl}^{X}\left(t;\hat{\theta}_m\right)}\sum_{r=1}^{D}P_{k,i}^{X}\left((r-1)h;\hat{\theta}_m\right)hP_{j,l}^{X}\left(t-rh;\hat{\theta}_m\right)+o(h)\\
&\overset{h\to 0}{\to} \frac{q_{ij}\left(\hat{\theta}_m\right)}{P_{kl}^{X}\left(t;\hat{\theta}_m\right)}\int_0^t P_{ki}^{X}\left(s;\hat{\theta}_m\right)P_{jl}^{X}\left(t-s;\hat{\theta}_m\right)ds.
\end{aligned}$$

The calculation of conditional expectation of $R_i(t)=\int_0^t I(X_u=i)du$ is significantly simpler and based on the possibility to change the order of integration under certain conditions (Fubini's theorem). Further, the well-known fact that the mathematical expectation of indicator function is equal to the probability distribution function is used. Thus,

$$\begin{aligned}
E_{\hat{\theta}_m}\left[R_i(t)\middle|X_t=l,X_0=k\right] &= E_{\hat{\theta}_m}\left[\int_0^t I(X_u=i)du\middle|X_t=l,X_0=k\right]\\
&= \int_0^t E_{\hat{\theta}_m}\left[I(X_u=i)\middle|X_t=l,X_0=k\right]du\\
&= \int_0^t P_{\hat{\theta}_m}\left(X_u=i\middle|X_t=l,X_0=k\right)du\\
&= \int_0^t \frac{P_{\hat{\theta}_m}(X_u=i,X_t=l,X_0=k)}{P(X_t=l,X_0=k)}du\\
&= \int_0^t \frac{P_{\hat{\theta}_m}(X_u=i,X_t=l,X_0=k)}{P_{kl}^{X}(t)}du\\
&= \int_0^t \frac{P_{k,i}^{X}(u)P_{i,l}^{X}(t-u)}{P_{kl}^{X}(t)}du\\
&= \frac{1}{P_{kl}^{X}(t;\hat{\theta}_m)}\int_0^t P_{ki}^{X}(u;\hat{\theta}_m)P_{il}^{X}(t-u;\hat{\theta}_m)du,
\end{aligned}$$

where $I(cond)$ is an indicator function taking value 1 if $cond$ is true, otherwise 0.

Finally, due to the Markov property of process $\mathbf{X}$

$$\begin{aligned}
E_{\hat{\theta}_m}[N_{ij}(T)|y] &= \sum_{k,l\in I} c_{kl} E_{\hat{\theta}_m}[N_{ij}(\Delta t)|X_{\Delta t}=l, X_0=k] \\
&= \sum_{k,l\in I} \frac{c_{kl} q_{ij}(\hat{\theta}_m)}{P^X_{kl}(t;\hat{\theta}_m)} \int_0^t P^X_{ki}(s;\hat{\theta}_m) P^X_{jl}(t-s;\hat{\theta}_m) ds \\
E_{\hat{\theta}_m}[R_j(T)|y] &= E_{\hat{\theta}_m}[R_j(T)|X_0=y_0,\dots,X_{i\Delta t}=y_i,\dots,X_T=y_M] \\
&\overset{M.p.}{=} \sum_{i=1}^{M} E_{\hat{\theta}_m}[R_j(\Delta t)|X_{(i-1)\Delta t}=y_{i-1}, X_{i\Delta t}=y_i] \\
&\overset{M.p.}{=} \sum_{k,l\in I} c_{kl} E_{\hat{\theta}_m}[R_j(\Delta t)|X_{\Delta t}=l, X_0=k] \\
&= \sum_{k,l\in I} \frac{c_{kl}}{P^X_{kl}(t;\hat{\theta}_m)} \int_0^t P^X_{ki}(u;\hat{\theta}_m) P^X_{il}(t-u;\hat{\theta}_m) du
\end{aligned}$$

□

A.6. Proof of Theorem 6.

Proof. Recall, Q[3] is a tridiagonal matrix of intensity rates. For example, it is given for the case $N=2$ by

$$Q = \begin{bmatrix}
-q_{-2,-1} & q_{-2,-1} & 0 & 0 & 0 \\
q_{-1,-2} & -(q_{-1,-2}+q_{-1,0}) & q_{-1,0} & 0 & 0 \\
0 & q_{0,-1} & -(q_{0,-1}+q_{0,1}) & q_{0,1} & 0 \\
0 & 0 & q_{1,0} & -(q_{1,0}+q_{1,2}) & q_{1,2} \\
0 & 0 & 0 & q_{2,1} & -q_{2,1}
\end{bmatrix}$$

Let us begin with the proof of matrix exponential property that follows from its Definition 6

$$\begin{aligned}
tQ\exp(tQ) &= tQ\left(I + \frac{tQ}{1!} + \frac{t^2Q^2}{2!} + \frac{t^3Q^3}{3!} + \dots\right) \\
&= tQ + \frac{t^2QQ}{1!} + \frac{t^3QQ^2}{2!} + \frac{t^4QQ^3}{3!} + \dots \\
&= tQ + \frac{t^2Q^2}{1!} + \frac{t^3Q^3}{2!} + \frac{t^4Q^4}{3!} + \dots \\
&= tQ + \frac{t^2QQ}{1!} + \frac{t^3Q^2Q}{2!} + \frac{t^4Q^3Q}{3!} + \dots \\
&= \left(I + \frac{tQ}{1!} + \frac{t^2Q^2}{2!} + \frac{t^3Q^3}{3!} + \dots\right) tQ \\
&= \exp(tQ)\,tQ,
\end{aligned}$$

[3] Further, we will omit dependence Q on θ due to simplicity of writing.

therefore

$$tQP^X(t) = tQ\exp(tQ) = \exp(tQ)tQ = P^X(t)tQ.$$

Further, k-th columns of the l.h.s. and r.h.s. of the expression above[4] are expressed as

$$\left[Q\exp(tQ)\right]_{\bullet,k} = \begin{bmatrix} \sum_{i=N}^{N} q_{-N,i} P^X_{i,k}(t) \\ \cdots \\ \sum_{i=-N}^{N} q_{N,i} P^X_{i,k}(t) \end{bmatrix} = \begin{bmatrix} \sum_{i=k-1}^{k+1} q_{-N,i} P^X_{i,k}(t) \\ \cdots \\ \sum_{i=k-1}^{k+1} q_{N,i} P^X_{i,k}(t) \end{bmatrix} = QP^X_{\bullet,k}(t),$$

$$\left[\exp(tQ)Q\right]_{\bullet,k} = \begin{bmatrix} \sum_{i=N}^{N} P^X_{-N,i}(t) q_{i,k} \\ \cdots \\ \sum_{i=-N}^{N} P^X_{N,i}(t) q_{i,k} \end{bmatrix} = \begin{bmatrix} \sum_{i=k-1}^{k+1} P^X_{-N,i}(t) q_{i,k} \\ \cdots \\ \sum_{i=k-1}^{k+1} P^X_{N,i}(t) q_{i,k} \end{bmatrix} = \sum_{i=k-1}^{k+1} P_{\bullet,i}(t) q_{i,k},$$

where $k \geq -N+1$, $P^X_{\bullet,k}$ is k-th column of the transition probability matrix $P^X(t)$. As a result, we obtain

$$\sum_{i=k-1}^{k+1} P_{\bullet,i}(t) q_{i,k} = Q \cdot P^X_{\bullet,k}(t)$$

All the superdiagonal entries of Q are positive by definitions (see the expressions (2.11). We can rewrite it after some algebra with respect to the $(k-1)$-th column of $P^X(t)$ as

$$P^X_{\bullet,k-1}(t;\theta) = \frac{1}{q_{k-1,k}(\theta)} \left(Q(\theta) P^X_{\bullet,k}(t) - q_{k,k}(\theta) P^X_{\bullet,k}(t) - q_{k+1,k}(\theta) P^X_{\bullet,k+1}(t) \right),$$

where $k \in \{-N+1, \ldots, N\}$.

□

A.7. Affine-Jump Diffusion (AJD) processes in option pricing

Beneficial analytic properties of AJD allow explicit, or at least efficient, numerical calculations and explain popularity of the class of AJD processes in quantitative finance. Besides, AJD are able to model variety of underlying process properties, such as the mean-reversion, jumps, correlations, and stochastic volatility. The foundational work on the topic is by Duffie et al. (see [31]), while J.Kallsen et al. (see [55]) give many examples of affine stochastic processes in the context of semimartingales.

We begin with the definition of jump-diffusion processes according to [32].

Definition 16. *A Markov process $\boldsymbol{X} = \{X_t\}_{t\geq 0}$ on a state space $D \subset R^n$ defined on filtered probability space $(\Omega, \mathscr{F}, \{\mathscr{F}_t\}_{t\geq 0}, P)$ is called the jump-diffusion process if it is solving the stochastic differential equation*

$$dX_t = \mu(X_t)dt + \sigma(x_t)dW_t + dZ_t, \tag{A.8}$$

[4] Taking into account that $P^X(t;\theta) = exp(tQ(\theta))$ by Corollary 1 from Section 2.1.

where $\mu : D \to R^n$, $\sigma : D \to R^{n\times n}$, $\{W_t\}_{t\geq 0}$ is a $\mathscr{F}_t$-measurable Wiener process in R^n, $(Z_t)_{t\geq 0}$ is a right-continuous pure jump process in R^n with probability distribution of jumps, ν, and intensity $\{\lambda(X_s)\}_{s\in[0,t]}$, $\lambda : D \to [0,\infty)$.

The idea of the Affine Jump-Diffusion (AJD) processes is an assumption of affine structure of μ, $\sigma\sigma^T$ and λ on D. In this case, certain restrictions have to be imposed on it in order to obtain desirable properties of the AJD process. Let us adopt the restrictions from [32] necessary for successful option pricing with AJD processes.

Definition 17. *The jump-diffusion process $\boldsymbol{X} = \{X_t\}_{t\geq 0}$ defined above is called an affine jump-diffusion process, if μ, $\sigma\sigma^T$ and λ are determined by pairs of coefficients K_0, K_1, H_0, H_1, l_0, l_1 and ρ_0, ρ_1 as follows*

- $\mu(x) = K_0 + K_1 x$, *for* $K_0 \in R^n$, $K_1 \in R^{n\times n}$;
- $(\sigma(x)\sigma(x)^T)_{ij} = (H_0)_{ij} + (H_1)_{ij} \cdot x$, *for* $H_0 \in R^{n\times n}$, $H_1 \in R^{n\times n\times n}$;
- $\lambda(x) = l_0 + l_1 \cdot x$, *for* $l_0 \in R$, $l_1 \in R^n$.

Further, it is necessary to define the affine discount-rate function $R : D \to R$ as $R(x) = \rho_0 + \rho_1 \cdot x$ for $\rho_0 \in R$, $\rho_1 \in R^n$. Also the transformation $\theta(c) = \int_{R^n} \exp(c \cdot z) d\nu(z)$, $c \in C^n$ (n-dimensional complex numbers), where the integral is assumed to be well defined.

If the vector $\chi = (K, H, l, \theta, \rho)$ is well defined (see [32]), then it allows us to determine a transformation at $t \leq T$, by

$$\psi^{\chi}(u, X_t, t, T) = E^{\chi}\left[e^{-\int_t^T R(X_s)ds} e^{u\cdot X_t} \middle| \mathscr{F}_t\right], \tag{A.9}$$

where E^{ψ} is an expectation under assumption the distribution of X is determined by χ.

As it was shown in [32], the transformation ψ^{χ} can be presented in the following form

$$\psi^{\chi}(u, x, t, T) = e^{\alpha(t)+\beta(t)\cdot x}, \tag{A.10}$$

where $\alpha(t)$ and $\beta(t)$ are a solutions of the ODEs

$$(\alpha(t))' = \rho_0 - K_0\beta(t) - \frac{1}{2}\beta(t)^T H_0 \beta(t) - l_0(\theta(\beta(t)) - 1), \tag{A.11}$$

$$(\beta(t))' = \rho_1 - K_1^T\beta(t) - \frac{1}{2}\beta(t)^T H_1 \beta(t) - l_1(\theta(\beta(t)) - 1), \tag{A.12}$$

with boundary conditions

$$\alpha(T) = 0, \tag{A.13}$$

$$\beta(T) = u. \tag{A.14}$$

Thus, the transformation ψ^{χ} can be performed solving the ODEs above. These ODEs can be solved explicitly or numericaly in some cases.

The idea is to create such AJD models that allow us to solve these ODEs quickly and efficiently.

A.8. Proof of Theorem 7. Mean-reversion property in the case of AMSM1 model.

Proof. Consider the conditional expectation at the moment t of the future volatility σ_{t+n} of the AMSM1 model for $n \geq 2$.

$$E\left[\sigma_{t+n}\middle|\mathscr{F}_t\right] = E\left[\sigma_0 \prod_{k=1}^{\hat{k}} M_{k,t+n}^{\frac{1}{2}}\middle|\mathscr{F}_t\right] = \sigma_0 \prod_{k=1}^{\hat{k}} E\left[M_{k,t+n}^{\frac{1}{2}}|\mathscr{F}_t\right].$$

Let us calculate the more general case of the expectation inside of the product above and implement the chain rule of mathematical expectations to it

$$E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_t\right] = E\left[E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_{t+n-2}\right]\middle|\mathscr{F}_t\right]$$

$$M_{k,t+n} = \begin{cases} m_0, & \text{if } u_{k,t+n-1} \in \left[0, \gamma_k(1-\Phi(\rho\epsilon_{t+n-1}))\right), \\ 2-m_0, & \text{if } u_{k,t+n-1} \in \left[\gamma_k(1-\Phi(\rho\epsilon_{t+n-1})), \gamma_k\right), \\ M_{k,t-1}, & \text{if } u_{k,t+n-1} \in \left[\gamma_k, 1\right), \end{cases}$$

where $q > 0$, $u_{k,t+n-1}$ are independent standard uniform variables, then

$$\begin{aligned} E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_{t+n-2}\right] = {} & m_0^q P\left(u_{k,t+n-1} \in \left[0, \gamma_k(1-\Phi(\rho\epsilon_{t+n-1}))\right)\middle|\mathscr{F}_{t+n-2}\right) + \\ & (2-m_0)^q P\left(u_{k,t+n-1} \in \left[\gamma_k(1-\Phi(\rho\epsilon_{t+n-1})), \gamma_k\right)\middle|\mathscr{F}_{t+n-2}\right) + \\ & M_{k,t+n-1}^{q} P\left(u_{k,t+n-1} \in [\gamma_k, 1)\middle|\mathscr{F}_{t+n-2}\right). \end{aligned}$$

Due to ϵ_{t+n-1} and $u_{k,t+n-1}$ are $\mathscr{F}_{t+n-1}$-measurable and i.i.d. for any k,n and t the conditional probabilities can be substituted by unconditional ones. Also, the probability $P(x \in (a,b))$ of continuous standard uniform random variable x is equal to $b-a$ for $a,b \in [0,1]$. Therefore,

$$\begin{aligned} E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_{t+n-2}\right] = {} & m_0^q \gamma_k(1-\Phi(\rho\epsilon_{t+n-1})) + \\ & (2-m_0)^q\left(\gamma_k - \gamma_k(1-\Phi(\rho\epsilon_{t+n-1}))\right) + \\ & M_{k,t-1}^{q}(1-\gamma_k) \end{aligned}$$

Then,

$$\begin{aligned} E\left[E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_{t+n-2}\right]\middle|\mathscr{F}_t\right] = {} & \gamma_k\left(m_0^q\left(1-E\left[\Phi(\rho\epsilon_{t+n-1})\middle|\mathscr{F}_t\right]\right)\right) + \\ & \gamma_k\left((2-m_0)^q E\left[\Phi(\rho\epsilon_{t+n-1})\middle|\mathscr{F}_t\right]\right) + \\ & (1-\gamma_k)E\left[M_{k,t+n-1}^{q}\middle|\mathscr{F}_t\right], \end{aligned}$$

where the following expression may be denoted as

$$E_M^q = \gamma_k\left(m_0^q\left(1-E\left[\Phi(\rho\epsilon_{t+n-1})\middle|\mathscr{F}_t\right]\right)\right) + \gamma_k\left((2-m_0)^q E\left[\Phi(\rho\epsilon_{t+n-1})\middle|\mathscr{F}_t\right]\right). \tag{A.15}$$

Taking into account that $\{\epsilon_t\}$ is an i.i.d. sequence, the values of $\{\epsilon_t\}$ do not depend on past information. Thus, we can replace the conditional expectations in E_M^q in (A.15). Moreover, due to the stationarity of white noise the expression E_M^q does not depend on t. Finally,

$$E\left[M_{k,t+n}^q\middle|\mathscr{F}_t\right] = E\left[E\left[M_{k,t+n}^q\middle|\mathscr{F}_{t+n-2}\right]\middle|\mathscr{F}_t\right] = (1-\gamma_k)E\left[M_{k,t+n-1}^q\middle|\mathscr{F}_t\right] + \gamma_k E_M^q. \quad \text{(A.16)}$$

Note, this expression is defined for $n \geq 2$. Since $M_{k,t}^q$ is $\mathscr{F}_t$-measurable, it is necessary to substitute the conditional expectation by (A.16) only $n-1$ times, because after $n-1$ substitutions, the very last conditional expectation will be the same as for $n = 1$, namely

$$E\left[M_{k,t+1}^q\middle|\mathscr{F}_t\right] = M_{k,t+1}^q.$$

For example for $n = 3$

$$\begin{aligned} E\left[M_{k,t+3}^q\middle|\mathscr{F}_t\right] &= (1-\gamma_k)E\left[M_{k,t+2}^q\middle|\mathscr{F}_t\right] + \gamma_k E_M^q = \\ (1-\gamma_k)\left[(1-\gamma_k)E\left[M_{k,t+1}^q\middle|\mathscr{F}_t\right] + \gamma_k E_M^q\right] &+ \gamma_k E_M^q = \\ (1-\gamma_k)^2 M_{k,t+1}^q + (1-\gamma_k)\gamma_k E_M^q &+ \gamma_k E_M^q. \end{aligned}$$

Thus, we obtain after $n-1$ substitutions

$$E\left[M_{k,t+n}^q\middle|\mathscr{F}_t\right] = (1-\gamma_k)^{n-1} M_{k,t}^q + \gamma_k E_M^q \sum_{j=0}^{n-2} (1-\gamma_k)^j,\ n \geq 2. \quad \text{(A.17)}$$

The expression (A.17) can be – roughly speaking – interpreted as $(1-\gamma_k)^{n-1}$-share of today's shock incorporated in a future shock. If we let the n tend to infinity, then the first term in the r.h.s. tends to zero ($M_{k,t}^q$ is bounded). In the meantime, the sum in the second term of (A.17) is a sum of geometric progression, therefore

$$E\left[M_{k,t+n}^q\middle|\mathscr{F}_t\right] \to \gamma_k \frac{1}{1-(1-\gamma_k)} E_M^q = E_M^q, \text{ when } n \to \infty,$$

where EM^q is the reversion level of the sequence $\{M_{k,t}\}_{t=0}^{\infty}$. So, the underlying volatility process $\{\sigma_t\}_{t=0}^{\infty}$ of AMSM1 model has the following properties

$$E\left[\sigma_{t+n}\middle|\mathscr{F}_t\right] = \sigma_0 \prod_{k=1}^{\hat{k}} \left[(1-\gamma_k)^{n-1} M_{k,t}^{\frac{1}{2}} + \gamma_k E_M^{\frac{1}{2}} \sum_{j=0}^{n-2} (1-\gamma_k)^j\right], \quad \text{(A.18)}$$

$$E\left[\sigma_{t+n}\middle|\mathscr{F}_t\right] \to \sigma_0 E_M^{\hat{k}/2}, \text{ when } n \to \infty. \quad \text{(A.19)}$$

It means that an expected value of AMSM1 volatility goes to the mean reversion level of volatility (A.19), when the horizon of the expectation n goes to infinity.

□

A.9. Proof of Theorem 7. Mean-reversion property in the case of AMSM2 model.

Proof. The proof has the same milestones as the proof for the AMSM1 model. Thus, let us consider the conditional expectation of future volatility ($n \geq 2$)

$$E\left[\sigma_{t+n}\middle|\mathscr{F}_t\right] = E\left[\left(\rho\epsilon_{t+n-1} - \sqrt{\sigma_0}\right)^2 \prod_{k=1}^{\hat{k}} M_{k,t+n}^{\frac{1}{2}}\middle|\mathscr{F}_t\right] =$$
$$E\left[\left(\rho\epsilon_{t+n-1} - \sqrt{\sigma_0}\right)^2\middle|\mathscr{F}_t\right] \prod_{k=1}^{\hat{k}} E\left[M_{k,t+n}^{\frac{1}{2}}\middle|\mathscr{F}_t\right].$$

Since the sequence $\{\epsilon_t\}$ consists of independent and identically distributed variables, we can replace the conditional expectation by the unconditional one in the first expectation

$$E\left[\left(\rho\epsilon_{t+n-1} - \sqrt{\sigma_0}\right)^2\right] \prod_{k=1}^{\hat{k}} E\left[M_{k,t+n}^{\frac{1}{2}}\middle|\mathscr{F}_t\right] = \left(\rho^2 + \sigma_0\right) \prod_{k=1}^{\hat{k}} E\left[M_{k,t+n}^{\frac{1}{2}}\middle|\mathscr{F}_t\right].$$

Now implement the chain rule of expectations to the generalized version of the last one above

$$E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_t\right] = \tag{A.20}$$
$$E\left[E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_{t+n-2}\right]\middle|\mathscr{F}_t\right] = \tag{A.21}$$
$$E\left[\left((1-\gamma_k)M_{k,t+n-1}^{q} + \gamma_k\left(\frac{1}{2}m_0^q + \frac{1}{2}(2-m_0)^q\right)\right)\middle|\mathscr{F}_t\right] = \tag{A.22}$$
$$(1-\gamma_k)E\left[M_{k,t+n-1}^{q}\middle|\mathscr{F}_t\right] + \gamma_k E_M^q = \tag{A.23}$$
$$\alpha_k + \beta_k E\left[M_{k,t+n-1}^{q}\middle|\mathscr{F}_t\right], \tag{A.24}$$

where $\alpha_k = \gamma_k E_M^q$, $\beta_k = (1-\gamma_k) \in (0,1)$, $E_M^q = (m_0^q + (2-m_0)^q)/2$, $m_0 \in [1,2)$.

The last expression can be interpreted as an expected value of $M_{k,t+n}$ incorporating only β_k-share of expectation of the previous value $M_{k,t+n-1}$. If we recurrently implement it to itself $n-1$ times, we obtain

$$E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_t\right] = (1-\gamma_k)^{n-1} M_{k,t}^{q} + \gamma_k E_M^q \sum_{j=0}^{n-2} (1-\gamma_k)^j. \tag{A.25}$$

When the lag n tends to infinity, the first term in the r.h.s. of (A.25) tends to zero ($M_{k,t}^q$ is bounded), while the sum in the second term of (A.25) is a sum of geometric progression, therefore

$$E\left[M_{k,t+n}^{q}\middle|\mathscr{F}_t\right] \to \gamma_k \frac{1}{1-(1-\gamma_k)} E_M^q = E_M^q, \text{ when } n \to \infty$$

where E_M^q is a reversion level of the sequence $\{M_{k,t}\}_{t=0}^{\infty}$. So, the underlying volatility process $\{\sigma_t\}_{t=0}^{\infty}$ of the AMSM2 model has the following properties

$$E\left[\sigma_{t+n}\middle|\mathscr{F}_t\right] = \left(\rho^2+\sigma_0\right)\prod_{k=1}^{\hat{k}}\left[(1-\gamma_k)^{n-1}M_{k,t}^{\frac{1}{2}} + \gamma_k E_M^{\frac{1}{2}}\sum_{j=0}^{n-2}(1-\gamma_k)^j\right], \tag{A.26}$$

$$E\left[\sigma_{t+n}\middle|\mathscr{F}_t\right] \to \left(\rho^2+\sigma_0\right)E_M^{\hat{k}/2}, \text{ when } n\to\infty. \tag{A.27}$$

It means that the expected value of the AMSM2 model volatility goes to mean reversion level (A.27), when the horizon of expectation n goes to infinity. Note, that this level is an unconditional means of volatility, which was directly proven in Lemma 3.

□

A.10. Proof of Theorem 8. Leverage effect of AMSM1 model.

Proof. We need to define the sign of $\operatorname{cov}(\sigma_t,\epsilon_{t-1})$, which is given by

$$\operatorname{cov}(\sigma_t,\epsilon_{t-1}) = E[(\sigma_t - E\sigma_t)\epsilon_{t-1}] = E[\sigma_t\epsilon_{t-1}] - E\sigma_t E\epsilon_{t-1} =$$
$$= \sigma_0 E\left[\epsilon_{t-1}\prod_{k=1}^{\hat{k}}M_{k,t}^{\frac{1}{2}}\right].$$

Let us implement the chain rule of mathematical expectations, where $E_{t-1}[\cdot]$ denotes a conditional expectation with respect to the information $\mathscr{F}_{t-1}$

$$\sigma_0 E\left[\epsilon_{t-1}\prod_{k=1}^{\hat{k}}M_{k,t}^{\frac{1}{2}}\right] =$$
$$\sigma_0 E\left[E_{t-2}\left[\epsilon_{t-1}\prod_{k=1}^{\hat{k}}M_{k,t}^{\frac{1}{2}}\middle|\epsilon_{t-1}\right]\right] =$$
$$\sigma_0 E\left[\epsilon_{t-1}\prod_{k=1}^{\hat{k}}E_{t-2}\left[M_{k,t}^{\frac{1}{2}}\middle|\epsilon_{t-1}\right]\right].$$

Now, we will calculate conditional expectation inside

$$\begin{aligned}E_{t-2}\left[M_{k,t}^{\frac{1}{2}}\middle|\epsilon_{t-1}\right] =& m_0^{\frac{1}{2}}P\left(u_{k,t-1}\in[0,\gamma_k(1-\Phi(\rho\epsilon_{t-1}))\middle|\epsilon_{t-1}\wedge\mathscr{F}_{t-2}\right)+\\ &(2-m_0)^{\frac{1}{2}}P\left(u_{k,t-1}\in[\gamma_k(1-\Phi(\rho\epsilon_{t-1})),\gamma_k)\middle|\epsilon_{t-1}\wedge\mathscr{F}_{t-2}\right)+\\ &M_{k,t-1}^{\frac{1}{2}}P\left(u_{k,t-1}\in[\gamma_k,1)\middle|\epsilon_{t-1}\wedge\mathscr{F}_{t-2}\right)=\\ &\gamma_k\left(m_0^{\frac{1}{2}}\left(1-\Phi(\rho\epsilon_{t-1})\right)+(2-m_0)^{\frac{1}{2}}\Phi(\rho\epsilon_{t-1})\right)+(1-\gamma_k)M_{k,t-1}^{\frac{1}{2}},\end{aligned}$$

where $\Phi(\cdot)$ is a cumulative distribution function of standard normal variable, $u_{k,t-1}$ is standard uniform variable. Let us denote as

$$\hat{M}^{k/2}(\epsilon_{t-1}) = m_0^{\frac{k}{2}}\left(1-\Phi(\rho\epsilon_{t-1})\right)+(2-m_0)^{\frac{k}{2}}\Phi(\rho\epsilon_{t-1}) \tag{A.28}$$

Then, taking into account $M_{k,t-1}^{1/2}$ being $\mathscr{F}_{t-2}$-measurable depends[5] on ϵ_{t-2} and u_{t-2}

$$\operatorname{cov}(\sigma_t, \epsilon_{t-1}) = \sigma_0 E\left[\epsilon_{t-1} \prod_{k=1}^{\hat{k}} \left((1-\gamma_k) M_{k,t-1}^{1/2}(\epsilon_{t-2}, u_{t-2}) + \gamma_k \hat{M}^{1/2}(\epsilon_{t-1})\right)\right]. \tag{A.29}$$

In order to calculate the covariance for the general value $\hat{k}$ we would consider two partial cases, $\hat{k}=1$ and $\hat{k}=2$, then the general case.

The case of $k=1$.

$$\begin{aligned}
\operatorname{cov}(\sigma_t, \epsilon_{t-1}) = \\
\sigma_0 E\left[(1-\gamma_1) M_{1,t-1}^{\frac{1}{2}} \epsilon_{t-1} + \gamma_1 \bar{M}^{\frac{1}{2}} \epsilon_{t-1}\right] = \\
\sigma_0(1-\gamma_1) E\left[M_{1,t-1}^{\frac{1}{2}}\right] E[\epsilon_{t-1}] + \sigma_0 \gamma_1 E\left[\bar{M}^{\frac{1}{2}} \epsilon_{t-1}\right] = \\
c_1 E\left[\bar{M}^{\frac{1}{2}} \epsilon_{t-1}\right],
\end{aligned}$$

where $c_1 = \sigma_0 \gamma_1 > 0$.

The case of $k=2$.

$$\begin{aligned}
\operatorname{cov}(\sigma_t, \epsilon_{t-1}) = \sigma_0 E\left[\left((1-\gamma_1) M_{1,t-1}^{\frac{1}{2}} + \gamma_1 \bar{M}^{\frac{1}{2}}\right)\left((1-\gamma_2) M_{2,t-1}^{\frac{1}{2}} + \gamma_2 \bar{M}^{\frac{1}{2}}\right)\epsilon_{t-1}\right] = \\
\sigma_0 E\left[\left((1-\gamma_1)(1-\gamma_2) M_{1,t-1}^{\frac{1}{2}} M_{2,t-1}^{\frac{1}{2}} + (1-\gamma_1)\gamma_2 M_{1,t-1}^{\frac{1}{2}} \bar{M}^{\frac{1}{2}} + \right.\right. \\
\left.\left.(1-\gamma_2)\gamma_1 M_{2,t-1}^{\frac{1}{2}} \bar{M}^{\frac{1}{2}} + \gamma_1\gamma_2 (\bar{M}^{\frac{1}{2}})^2\right)\epsilon_{t-1}\right] = \\
\sum_{k=0}^{2} c_k E\left[\left(\bar{M}^{\frac{1}{2}}\right)^k \epsilon_{t-1}\right],
\end{aligned}$$

where

$$\begin{aligned}
c_0 &= \sigma_0(1-\gamma_1)(1-\gamma_2) E\left[M_{1,t-1}^{\frac{1}{2}}\right] E\left[M_{2,t-1}^{\frac{1}{2}}\right] E[\epsilon_{t-1}] = 0, \\
c_1 &= \sigma_0(1-\gamma_1)\gamma_2 E\left[M_{1,t-1}^{\frac{1}{2}}\right] + \sigma_0(1-\gamma_2)\gamma_1 E\left[M_{2,t-1}^{\frac{1}{2}}\right] > 0, \\
c_2 &= \gamma_1\gamma_2 > 0.
\end{aligned}$$

Case of $k=n$. If we consider the general case with an arbitrary $k=n$, we see the same structure – the sum of $E\left[\epsilon_{t-1}\bar{M}^{k/2}(\epsilon_{t-1})\right]$ with the positive[6] coefficients c_k

$$\operatorname{cov}(\sigma_t, \epsilon_{t-1}) = \sum_{k=1}^{n} c_k E\left[\left(\bar{M}^{\frac{1}{2}}\right)^k \epsilon_{t-1}\right],\ c_k > 0. \tag{A.30}$$

[5]Further, the brackets denoting dependence of $M_{k,t-1}^{1/2}$ on ϵ_{t-2} and u_{t-2} as well as $\bar{M}^{1/2}$ depends on ϵ_{t-1} is omitted due to simplicity of writing in this proof.

[6]The coefficient c_0 is equal to zero and omitted further in the proof.

So, in order to define the sign of $\operatorname{cov}(\sigma_t, \epsilon_{t-1})$, we should find out the sign of expectations $E\left[\epsilon_{t-1}\bar{M}^{k/2}(\epsilon_{t-1})\right]$, where $\bar{M}^{\frac{k}{2}}$ is defined by (A.28)

$$E\left[\left(\bar{M}^{1/2}\right)^k \epsilon_{t-1}\right] = E\left[\left(m_0^{\frac{1}{2}}\Phi(-\rho\epsilon_{t-1}) + (2-m_0)^{\frac{1}{2}}\Phi(\rho\epsilon_{t-1})\right)^k \epsilon_{t-1}\right].$$

By using Newton's binomial theorem and grouping terms by couples $(1,k)$, $(2,k-1)$ and so on (such couples have equal binomial coefficients), we obtain[7]

$$\sum_{k\geq n>m} b_m^k E\left[\left(m_0^{\frac{n}{2}}(2-m_0)^{\frac{m}{2}}\Phi(-\rho\epsilon_{t-1})^n\Phi(\rho\epsilon_{t-1})^m + \right. \tag{A.31}$$

$$\left. m_0^{\frac{m}{2}}(2-m_0)^{\frac{n}{2}}\Phi(-\rho\epsilon_{t-1})^m\Phi(\rho\epsilon_{t-1})^n\right)\epsilon_{t-1}\Big], \tag{A.32}$$

where b_m^k are binomial coefficients.

Now, we need to mention the following fact. The functions

$$\Phi(-x)^n\Phi(x)^m x,$$
$$\Phi(-x)^m\Phi(x)^n x$$

are symmetrical reflections of each other with respect to the origin. Also, the probability density function of standard normal variable (ϵ_{t-1}) is symmetric with respect to the vertical axis. As a result, the following expression holds

$$E(n,m) = E\left[\Phi(-\rho\epsilon_{t-1})^n\Phi(\rho\epsilon_{t-1})^m\epsilon_{t-1}\right] = -E\left[\Phi(-\rho\epsilon_{t-1})^m\Phi(\rho\epsilon_{t-1})^n\epsilon_{t-1}\right].$$

It allows us to rewrite the sum (A.31, A.32) in the form

$$E\left[\left(\bar{M}^{\frac{1}{2}}\right)^k\epsilon_{t-1}\right] = \sum_{k\geq n>m} b_m^k\left(m_0^{\frac{n}{2}}(2-m_0)^{\frac{m}{2}} - m_0^{\frac{m}{2}}(2-m_0)^{\frac{n}{2}}\right)E(n,m),$$

where

$$m_0^{\frac{n}{2}}(2-m_0)^{\frac{m}{2}} - m_0^{\frac{m}{2}}(2-m_0)^{\frac{n}{2}} > 0$$

for $n>m$, while $E(n,m)$ is negative for $n>m$, $\rho>0$. Thereby,

$$E\left[\left(\bar{M}^{\frac{1}{2}}\right)^k\epsilon_{t-1}\right] < 0,$$

for any $k \geq 1$.

Therefore, the main result of Theorem 8 is proven

$$\operatorname{cov}(\sigma_t, \epsilon_{t-1}) = \sum_{k=1}^{\hat{k}} c_k E\left[\left(\bar{M}^{\frac{1}{2}}\right)^k\epsilon_{t-1}\right] < 0,$$

for $\rho > 0$.

□

[7] This sum that has the additional (middle) term in the case of k is even, namely it is given by $m_0^{k/2+1}(2-m_0)^{k/2+1}E\left[\Phi(-\rho\epsilon_{t-1})^{k/2}\Phi(\rho\epsilon_{t-1})^{k/2}\epsilon_{t-1}\right]$, but this expectation is equal to zero for any k. As a result, this term is omitted in further calculations.

A.11. Proof of Theorem 8. Leverage effect of AMSM2 model.

Proof. To begin with the definition of covariance

$$\begin{aligned}\operatorname{cov}(\sigma_t,\epsilon_{t-1}) =\\ E\left[(\sigma_t - E(\sigma_t))(\epsilon_{t-1} - E(\epsilon_{t-1}))\right] =\\ E\left[\left(\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2 \sigma_t^M - E(\sigma_{t-1})\right)\epsilon_{t-1}\right] =\\ E\left[\left(\rho^2\epsilon_{t-1}^3 - 2\rho\sqrt{\sigma_0}\epsilon_{t-1}^2 + \sigma_0\epsilon_{t-1}\right)\sigma_t^M\right] =\end{aligned}$$

By using independence of u_{t-1} and ϵ_{t-1}

$$\left(\rho^2 E\left[\epsilon_{t-1}^3\right] - 2\rho\sqrt{\sigma_0}E\left[\epsilon_{t-1}^2\right] + \sigma_0 E[\epsilon_{t-1}]\right)E\left[\sigma_t^M\right] = -2\rho\sqrt{\sigma_0}E\left[\sigma_t^M\right] < 0,$$

where $\sigma_t^M = \left(\prod_{k=0}^{\hat{k}} M_{k,t}\right)^{1/2}$.

□

A.12. Proof of Lemma 3. Weak-stationarity of AMSM1/AMSM2 model.

We need to prove two facts about $\{\sigma_t\}_{t=0}^{\infty}$: that the first moment of it is a constant; and that the covariance does not vary with respect to a time.

So, let us begin with the first moment. We obtain for AMSM1 case

$$E[\sigma_t] = \sigma_0 E\left[\left(\prod_{k=0}^{\hat{k}} M_{k,t}\right)^{1/2}\right] = \sigma_0 E\left[\sigma_t^M\right],$$

By taking into account the independence of each $M_{k,t}$ and ϵ_{t-1} for AMSM2 case

$$E[\sigma_t] = E\left[\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2\left(\prod_{k=0}^{\hat{k}} M_{k,t}\right)^{1/2}\right] = E\left[\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2\right]E\left[\sigma_t^M\right],$$

where

$$E\left[\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^2\right] = \rho^2 E\left[\epsilon_{t-1}^2\right] - 2\rho\sqrt{\sigma_0}E[\epsilon_{t-1}] + \sigma_0 = \rho^2 + \sigma_0,$$

$$\left(\prod_{k=0}^{\hat{k}} M_{k,t}\right)^{1/2} \equiv \sigma_t^M.$$

In both cases we have

$$E\left[\sigma_t^M\right] = \prod_{k=0}^{\hat{k}} E\left[M_{k,t}^{1/2}\right], \tag{A.33}$$

$$E\left[M_{k,t}^{1/2}\right] = (1-\gamma_k)E\left[M_{k,t-1}^{1/2}\right] + \gamma_k E\left[M^{1/2}\right] \tag{A.34}$$

In order to have volatility process stationary, we assume that $E\left[M_{k,t}^{1/2}\right]$ is constant independent of t

$$E\left[M_{k,t}^{1/2}\right] = E\left[M_{k,t-j}^{1/2}\right] = \mu.$$

Then, the equation (A.34) is rearranged after some algebra

$$\mu = (1-\gamma_k)\mu + \gamma_k E\left[M^{1/2}\right],$$
$$\mu = E\left[M^{1/2}\right],$$

for each model. In other words, we have to assume that the unconditional expectation of each volatility component $M_{k,t}^{1/2}$ is equal to the expectation of binomial variable $M^{1/2}$. Thus, we obtain

$$E[\sigma_t] = \hat{\sigma} E[\sigma_t^M] = \hat{\sigma}\mu^{\hat{k}}, \tag{A.35}$$

where $\hat{\sigma}$ denotes the parameter σ_0 (in the case of AMSM1) and $(\rho^2+\sigma_0)$ (in the case of AMSM2).

Now, let us consider the covariance

$$\operatorname{cov}(\sigma_t, \sigma_{t+\tau}) = E\left[(\sigma_t - E[\sigma_t])(\sigma_{t+\tau} - E[\sigma_t])\right] =$$
$$E\left[\left(\sigma_t - \hat{\sigma}\mu^{\hat{k}}\right)\left(\sigma_{t+\tau} - \hat{\sigma}\mu^{\hat{k}}\right)\right] = \hat{\sigma}^2 E\left[\sigma_t^M \sigma_{t+\tau}^M\right] - \hat{\sigma}^2\mu^{2\hat{k}},$$

Then taking into account the independence of $M_{k,t}$ and $M_{m,t}$ $(k \neq m)$, we obtain

$$E\left[\sigma_t^M \sigma_{t+\tau}^M\right] = E\left[\prod_{k=1}^{\hat{k}} M_{k,t}^{1/2} M_{k,t+\tau}^{1/2}\right] = \prod_{k=1}^{\hat{k}} E\left[M_{k,t}^{1/2} M_{k,t+\tau}^{1/2}\right].$$

Let $\tau = 1$, then by using the chain rule of conditional expectations we can rewrite

$$E\left[M_{k,t}^{q/2} M_{k,t+1}^{q/2}\right] = E\left[M_{k,t}^{q/2} E\left(M_{k,t+1}^{q/2}\middle|\mathscr{F}_{t-1}\right)\right] =$$
$$E\left[M_{k,t}^{q/2}\left(M_{k,t}^{q/2}(1-\gamma_k) + M^{q/2}\gamma_k\right)\right] = (1-\gamma_k)E\left[M_{k,t}^{q}\right] + \gamma_k\left(E\left[M^{q/2}\right]\right)^2 =$$
$$\left(E\left[M^{q/2}\right]\right)^2\left((1-\gamma_k)\frac{E\left[M_{k,t}^{q}\right]}{\left(E\left[M^{q/2}\right]\right)^2} + \gamma_k\right) = \left(E\left[M^{q/2}\right]\right)^2\left[(1-\gamma_k)(1+a_q) + \gamma_k\right] =$$
$$\left(E\left[M^{q/2}\right]\right)^2\left(1 + a_q(1-\gamma_k)^1\right).$$

Thus, we could assume that the following expression holds

$$E\left[M_{k,t}^{q/2} M_{k,t+\tau}^{q/2}\right] = \left(E[M^{q/2}]\right)^2\left(1 + a_q(1-\gamma_k)^{\tau}\right), \tag{A.36}$$

where

$$a_q = \frac{E(M^q)}{E[M^{q/2}]^2} - 1.$$

The expression (A.36) holds for $\tau = m$. Let us prove that it also holds for $\tau = m+1$.

$$\begin{aligned} E\left[M_{k,t}^{q/2} M_{k,t+m+1}^{q/2}\right] = \\ E\left[M_{k,t}^{q/2} E\left(M_{k,t+m+1}^{q/2} \middle| \mathscr{F}_{t+m-1}\right)\right] = \\ E\left[M_{k,t}^{q/2}\left((1-\gamma_k) M_{k,t+m}^{q/2} + \gamma_k M^{q/2}\right)\right] = \\ (1-\gamma_k) E\left[M_{k,t}^{q/2} M_{k,t+m}^{q/2}\right] + \gamma_k \left(E\left[M^{q/2}\right]\right)^2 = \end{aligned}$$

Since, the expression (A.36) holds for $\tau = m$, then

$$\begin{aligned} \left(E\left[M^{q/2}\right]\right)^2 \left(1 + a_q (1-\gamma_k)^m\right)(1-\gamma_k) + \left(E\left[M^{q/2}\right]\right)^2 \gamma_k = \\ \left(E\left[M^{q/2}\right]\right)^2 \left[1 - \gamma_k + a_q (1-\gamma_k)^{m+1} + \gamma_k\right] = \\ \left(E\left[M^{q/2}\right]\right)^2 \left[1 + a_q (1-\gamma_k)^{m+1}\right]. \end{aligned}$$

Therefore, the expression holds for any natural positive τ.

The results above imply the final result for covariance function

$$\operatorname{cov}(\sigma_t, \sigma_{t+\tau}) = \hat{\sigma}^2 \mu^{2\hat{k}} \prod_{k=1}^{\hat{k}} \left(1 + a_1 (1-\gamma_k)^{\tau}\right) - \hat{\sigma}^2 \mu^{2\hat{k}}, \tag{A.37}$$

where $\tau > 0$ and

$$a_1 = \frac{E(M)}{E(M^{1/2})^2} - 1 = \frac{1}{\mu^2} - 1.$$

As we can see, the covariance does not depend on t, only on the lag τ. Therefore, the weak stationarity property holds for the volatility process of AMSM1/AMSM2 model with the assumption $E\left[M_{k,t}^{1/2}\right] = E\left[M^{1/2}\right] = \mu$.

A.13. Proof of Theorem 9. Long memory of AMSM2 model.

Proof. First of all, we will prove this theorem for the more general definition of the AMSM2 model's transition probabilities γ_k given in (3.15) rather than the simplified definition (3.16) aimed to mimic shape of (3.15) in case of $(\gamma_{\hat{k}}, b) = (0.95, 3)$ being more computationally easy. Namely, we assume, as well did Calvet & Fisher in their original work, that

$$\gamma_k = 1 - (1-\gamma_1)^{b^{k-1}},\ \gamma_1 \in (0,1), b \in (1,\infty). \tag{A.38}$$

Let us consider the autocorrelation function of the AMSM2 process, taking into account independence of $\{\epsilon_t^q\}$ and $\{\sigma_t^q\}$ sequences

$$\begin{aligned} \psi_q(n) = \operatorname{corr}\left(|r_t|^q, |r_{t+n}|^q\right) = \operatorname{corr}\left(|\epsilon_t|^q \sigma_t^q, |\epsilon_{t+n}|^q \sigma_{t+n}^q\right) = \\ \frac{E\left(|\epsilon_t|^q\right) E\left(|\epsilon_{t+n}|^q\right)\left(E\left(\sigma_t^q \sigma_{t+n}^q\right) - E\left(\sigma_t^q\right) E\left(\sigma_{t+n}^q\right)\right)}{E\left(|\epsilon_t|^{2q}\right) E\left(\sigma_t^{2q}\right)}. \end{aligned}$$

By using the stationarity of white noise

$$\psi_q(n) = c_q \frac{E\left(\sigma_t^q \sigma_{t+n}^q\right) - E\left(\sigma_t^q\right) E\left(\sigma_{t+n}^q\right)}{E\left(\sigma_t^{2q}\right)}, \tag{A.39}$$

where

$$c_q = \frac{E\left(|\epsilon_t|^q\right)^2}{E\left(\left|\epsilon_t^{2q}\right|\right)}.$$

In order to calculate the expression (A.39), we calculate each expectation separately and then collect the results. So,

$$E\sigma_t^q = E\left[\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^{2q}\right] \prod_{i=1}^{\hat{k}} E\left[M_{i,t}^{q/2}\right] = e_{2q} E\left[M^{q/2}\right]^{\hat{k}},$$

$$E\sigma_{t+n}^q = E\left[\left(\rho\epsilon_{t+n-1} - \sqrt{\sigma_0}\right)^{2q}\right] \prod_{i=1}^{\hat{k}} E[M_{i,t+n}^{q/2}] = e_{2q} E\left[M^{q/2}\right]^{\hat{k}},$$

$$E\sigma_t^{2q} = E\left[\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^{4q}\right] \prod_{i=1}^{\hat{k}} E\left[M_{i,t}^{q}\right] = e_{4q} E\left[M^{q}\right]^{\hat{k}},$$

$$E\left(\sigma_t^q \sigma_{t+n}^q\right) = E\left[\left(\rho\epsilon_{t-1} - \sqrt{\sigma_0}\right)^{2q}\right] E\left[\left(\rho\epsilon_{t+n-1} - \sqrt{\sigma_0}\right)^{2q}\right] \prod_{i=1}^{\hat{k}} E\left[M_{i,t}^{q/2} M_{i,t+n}^{q/2}\right],$$

where e_y is a deterministic variable that depends on y and also on parameters ρ and σ_0 due to stationarity and standard normal distribution of a standard white noise.

As a result, the autocorrelation function $\psi_q(n)$ and the quantity $K_q(n)$ from the proof of Calvet and Fisher are different for the AMSM2 model. It is given by

$$\psi_q(n) = \frac{\bar{c}_q}{(1+a_q)^{\hat{k}}} \left(\prod_{k=1}^{\hat{k}} \left(1 + a_q(1-\gamma_k)^n\right) - 1 \right),$$

$$K_q(n) = \frac{\bar{c}_q}{(1+a_q)^{\hat{k}}} \prod_{k=1}^{\hat{k}} \left(1 + a_q(1-\gamma_k)^n\right),$$

where

$$a_q = \frac{E\left(M^q\right)}{\left[E\left(M^{q/2}\right)\right]^2} - 1, \; \bar{c}_q = c_q \frac{e_{2q}^2}{e_{4q}}.$$

The rest of the proof is the same as in Calvet and Fisher's paper, but instead of c_q should be used $\bar{c}_q$.

□

A.14. Proof of Theorem 10.

Lemma 4. *If Y_t has a normal distribution conditionally on $\mathscr{F}_{t-1}$ with constant mean and variance under P, and*

$$S_{t-1} = E^P[\exp(-\rho + Y_t)S_t|\mathscr{F}_{t-1}], \tag{A.40}$$

$$\frac{dQ}{dP} = \exp\left((r-\rho)T + \sum_{t=1}^{T} Y_t\right) \text{ (Radon-Nikodym derivative)}, \tag{A.41}$$

then

1. *Q – is a probability measure, which is equivalent*[8] *to P;*
2. *Q and P satisfy LRNVR.*

Proof. The proof is generally based on Duan's results:

1. The proof is given for Lemma A.1 in Duan [28];
2. The proof in Lemma A.2 in Duan [28] is based on the assumption that volatility σ_t is $\mathscr{F}_{t-1}$-measurable, which is true for the GARCH model, as well as for the AMSM models. Thus, the proof is same.

□

The Lemma above defines the general form of Radon-Nikodym derivative necessary to measure transition from P to Q.

Proof. The conditions of the theorem lead to the normality of the logarithmic marginal rate of substitution.

(1) The coefficient of relative risk aversion is defined as

$$\lambda_1 = -C\frac{u''(C)}{u'(C)} = -\left(\frac{d\ln(C)}{dC}\right)^{-1}\frac{d\ln(u'(C))}{dC} \approx -\frac{\ln(u'(C_t)) - \ln(u'(C_{t-1}))}{\ln(C_t) - \ln(C_{t-1})}.$$

After some algebra

$$\ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right) \approx -\lambda_1 \ln\left(\frac{C_t}{C_{t-1}}\right)\Big|\mathscr{F}_{t-1} \overset{P}{\sim} \mathscr{N}(\mu,\sigma)$$

(2) The coefficient of absolute risk aversion λ_2, defined as

$$\lambda_2 = -\frac{u''(C)}{u'(C)} = -\frac{d\ln(u'(C))}{dC} \approx -\frac{\ln(u'(C_t) - \ln(u'(C_{t-1})))}{C_t - C_{t-1}}$$

Thus,

$$\ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right) \approx -\lambda_2(C_t - C_{t-1})|\mathscr{F}_{t-1} \overset{P}{\sim} \mathscr{N}(\mu,\sigma)$$

[8] Two measures are equivalent, if and only if they are absolutely continuous to each other.

(3) We have that $u(C_t) = aC_t + b$, therefore $u'(C_t) = a$. Hence,

$$\frac{u'(C_t)}{u'(C_{t-1})} = 1 \Rightarrow \ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right)\Big|\mathscr{F}_{t-1} \overset{P}{\sim} \mathscr{N}(0,0).$$

If we denote

$$\begin{aligned}
Y_t &= \ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right)\Big|\mathscr{F}_{t-1} \overset{P}{\sim} \mathscr{N}(\mu,\sigma^2),\\
\frac{dQ}{dp} &= \exp\left((r-\rho)T + \sum_{t=1}^{T}\ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right)\right) = \exp\left((r-\rho)T + \sum_{t=1}^{T} Y_t\right),\\
S_{t-1} &= E^P\left[\exp\left(-\rho + \ln\left(\frac{u'(C_t)}{u'(C_{t-1})}\right)\right)S_t\Big|\mathscr{F}_{t-1}\right] = E^P\left[\exp\left(-\rho + Y_t\right)S_t\Big|\mathscr{F}_{t-1}\right]
\end{aligned}$$

then all condition of Lemma 4 are fulfilled. Therefore, Q is LRNVR probability measure equivalent to P .

□

A.15. Proof of Theorem 11.

Proof. Since, $r_t = \ln(S_t/S_{t-1})$ are normally distributed conditionally on $\mathscr{F}_{t-1}$ w.r.t. Q according to Definition 11 of LRNVR (item 1), we could write it down as follows

$$\ln\frac{S_t}{S_{t-1}} = \mu_t^\star + \epsilon_t^\star\sigma_t^\star,$$

where $\epsilon_t^\star \overset{Q}{\sim} \mathscr{N}(0,1)$, $\mu_t^\star$ and $\sigma_t^\star$ are $\mathscr{F}_{t-1}$-measurable, therefore

$$\frac{S_t}{S_{t-1}} = \exp(\mu_t^\star + \epsilon_t^\star\sigma_t^\star),$$

by taking conditional expectation w.r.t. measure Q and using a definition of moment generating function of normal random variable, we obtain

$$E^Q\left[\frac{S_t}{S_{t-1}}\Big|\mathscr{F}_{t-1}\right] = E^Q[\exp(\mu_t^\star + \epsilon_t^\star\sigma_t^\star)|\mathscr{F}_{t-1}] \overset{MGF}{=} \exp\left(\mu_t^\star + \frac{(\sigma_t^\star)^2}{2}\right).$$

We also know from the Definition 11 of LRNVR (item 3) that

$$E^Q\left[\frac{S_t}{S_{t-1}}\Big|\mathscr{F}_{t-1}\right] = \exp(r).$$

Thus, we obtain

$$\mu_t^\star = r - \frac{1}{2}(\sigma_t^\star)^2.$$

Definition 11 of LRNVR (item 2) connects the variances under P and Q, namely

$$\sigma_t = \sigma_t(\epsilon_{t-1}^\star, \xi_{t-1}^\star)^2 = V^Q\left[\ln\frac{S_t}{S_{t-1}}\Big|\mathscr{F}_{t-1}\right] = V^P\left[\ln\frac{S_t}{S_{t-1}}\Big|\mathscr{F}_{t-1}\right] = \sigma_t(\epsilon_{t-1}, \xi_{t-1})^2.$$

Since under the measure P

$$r_t = \ln\frac{S_t}{S_{t-1}} = r + \lambda\sigma_t - \frac{1}{2}\sigma_t^2 + \epsilon_t\sigma_t,$$
$$\sigma_t = \sigma_t(\epsilon_{t-1}, \xi_{t-1}).$$

and in the meantime under the measure Q

$$r_t = \ln\frac{S_t}{S_{t-1}} = r - \frac{1}{2}(\sigma_t^\star)^2 + \epsilon_t^\star\sigma_t^\star,$$

hence

$$r + \lambda\sigma_t - \frac{1}{2}\sigma_t^2 + \epsilon_t\sigma_t = r - \frac{1}{2}\sigma_t^2 + \epsilon_t^\star\sigma_t$$
$$\epsilon_t = \epsilon_t^\star - \lambda.$$

Therefore, the distribution of $M_{k,t}$ given in the definition of the AMSM1 model (3.17) is changed under Q in the case of AMSM1 to the following one

$$M_{k,t}^\star = \begin{cases} m_0, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in \left[0, \gamma_k(1-\Phi(\rho(\epsilon_{t-1}^\star - \lambda)))\right), \\ 2 - m_0, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in \left[\gamma_k(1-\Phi(\rho(\epsilon_{t-1}^\star - \lambda))), \gamma_k\Phi(\rho(\epsilon_{t-1}^\star - \lambda))\right), \\ M_{k,t-1}, & \text{if } \Phi(\xi_{k,t-1}^\star - \nu) \in \left[\gamma_k, 1\right), \end{cases}$$

while the change of the measure leads to change in the volatility process definition in the case of the AMSM2 model, namely

$$\sigma_t^\star = \left(\rho(\epsilon_t^\star - \lambda) - \sigma_0\right)^2 \left(\prod_{i=k}^{\hat{k}} M_{k,t}^\star\right)^{\frac{1}{2}}.$$

In order to define the risk-neutral correction of the vector ξ_t under Q, let us consider the Moment Generation Function (MGF) of its components $\xi_{k,t}$

$$E^Q\left[\exp\left(c\xi_{k,t}\right)\middle|\mathscr{F}_{t-1}\right] = E^P\left[\exp\left(c\xi_{k,t} + (r-\rho) + Y_t\right)\middle|\mathscr{F}_{t-1}\right],$$

where the variables $\xi_{k,t}$ and $Y_t = \ln(MRS_t)$ from (3.38) are distributed normally under measure P conditionally on $\mathscr{F}_{t-1}$

$$\xi_{k,t}|\mathscr{F}_{t-1} \sim \mathscr{N}(0,1),$$
$$Y_t|\mathscr{F}_{t-1} \sim \mathscr{N}(\mu,\sigma).$$

Therefore, we can represent Y_t as the following combination

$$Y_t|\mathscr{F}_{t-1} \overset{D}{=} \alpha_t + \nu_t\xi_{k,t} + U_t,$$

where $U_t|\mathscr{F}_{t-1} \sim \mathscr{N}\left(0, E[U_t^2|\mathscr{F}_{t-1}]\right)$, $U_t \perp \xi_{k,t}$, then the MGF of $\xi_{k,t}$ is rearranged as

$$E^Q\left[\exp\left(c\xi_{k,t}\right)\Big|\mathscr{F}_{t-1}\right] = E^P\left[\exp\left(\alpha_t + (r-\rho) + U_t\right)|\mathscr{F}_{t-1}\right] E^P\left[\exp\left((\nu_t + c)\xi_{k,t}\right)|\mathscr{F}_{t-1}\right]$$

since

$$\begin{aligned}\alpha_t + (r-\rho) + U_t|\mathscr{F}_{t-1} \sim\ & \mathscr{N}\left(\alpha_t + r - \rho, E^P[U_t^2|\mathscr{F}_{t-1}]\right),\\ (\nu_t + c)\xi_{k,t}|\mathscr{F}_{t-1} \sim\ & \mathscr{N}\left(0, (\nu_t + c)^2\right),\end{aligned}$$

then by using knowledge of normal variable's MGF we obtain

$$E^Q\left[\exp\left(c\xi_{k,t}\right)\Big|\mathscr{F}_{t-1}\right] = \exp\left(\alpha_t + (r-\rho) + \frac{1}{2}E^P\left[U_t^2|\mathscr{F}_{t-1}\right]\right)\exp\left(\frac{\nu_t^2}{2} + \frac{c^2}{2} + c\nu_t\right).$$

Let $c = 0$, then

$$1 = \exp\left(\alpha_t + (r-\rho) + \frac{1}{2}E^P\left[U_t^2|\mathscr{F}_{t-1}\right] + \frac{\nu_t^2}{2}\right).$$

Hence we have

$$E^Q\left[\exp\left(c\xi_{k,t}\right)|\mathscr{F}_{t-1}\right] = \exp\left(\frac{c^2}{2} + c\nu_t\right),$$

and this implies $\xi_{k,t}|\mathscr{F}_{t-1} \sim \mathscr{N}(\nu_t, 1)$ under Q. We assume further for simplicity that ν_t is a constant. Therefore, we can define

$$\xi^\star_{k,t} = \xi_{k,t} + \nu.$$

Note, the independence of $\epsilon_t^\star$ and $\xi^\star_{k,t}$ is proved in a similar way as given in the appendix of [29].

□

A.16. Proof of Corollary 5.

Proof. Since

$$S_t = S_{t-1}\exp\left(r - \frac{1}{2}(\sigma_t^\star)^2 + \sigma_t^\star\epsilon_t^\star\right),$$

where $\epsilon_t^\star \overset{Q}{\sim} \mathscr{N}(0,1)$ i.i.d. Then

$$\begin{aligned}\hat{S}_t = S_t\exp\left(-r\,t\right)) =& S_{t-1}\exp\left(-r(t-1) - r\right)\exp\left(r - \frac{1}{2}(\sigma_t^\star)^2 + \sigma_t^\star\epsilon_t^\star\right) =\\ & \hat{S}_{t-1}\exp\left(-\frac{1}{2}(\sigma_t^\star)^2 + \sigma_t^\star\epsilon_t^\star\right).\end{aligned}$$

Continuing, we obtain the following general expression

$$\hat{S}_T = \hat{S}_t\exp\left(-\frac{1}{2}\sum_{s=t}^{T}(\sigma_s^\star)^2 + \sum_{s=t}^{T}\sigma_s^\star\epsilon_s^\star\right).$$

Let us take the conditional expectation of $\hat{S}_T$ given $\mathscr{F}_t$

$$
\begin{aligned}
&E^Q\left[\hat{S}_T\middle|\mathscr{F}_t\right] = \\
&\hat{S}_t E^Q\left[\exp\left(-\frac{1}{2}\sum_{s=t}^{T}\sigma_s^2+\sum_{s=t}^{T}\sigma_s\xi_s\right)\middle|\mathscr{F}_t\right] = \\
&\hat{S}_t E^Q\left[\exp\left(-\frac{1}{2}\sum_{s=t}^{T-1}\sigma_s^2+\sum_{s=t}^{T-1}\sigma_s\xi_s\right)E^Q\left[\exp\left(-\frac{1}{2}\sigma_T^2+\sigma_T\xi_T\right)\middle|\mathscr{F}_{T-1}\right]\middle|\mathscr{F}_t\right] \overset{MGF}{=} \\
&\hat{S}_t E^Q\left[\exp\left(-\frac{1}{2}\sum_{s=t}^{T-2}\sigma_s^2+\sum_{s=t}^{T-2}\sigma_s\xi_s-\frac{1}{2}\sigma_{T-1}^2+\sigma_{T-1}\xi_{T-1}\right)\middle|\mathscr{F}_t\right] = \\
&\hat{S}_t E^Q\left[\exp\left(-\frac{1}{2}\sum_{s=t}^{T-2}\sigma_s^2+\sum_{s=t}^{T-2}\sigma_s\xi_s\right)E^Q\left[\exp\left(-\frac{1}{2}\sigma_{T-1}^2+\sigma_{T-1}\xi_{T-1}\right)\right.\right. \\
&\left.\left.\middle|\mathscr{F}_{T-2}\right]\middle|\mathscr{F}_t\right] = \ldots = \hat{S}_t.
\end{aligned}
$$

Thus, we have proven that Q is a martingale measure.

□

B

Bibliography

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth Dover printing, tenth GPO printing edition, 1964.

[2] P. Abry and D. Veitch. Wavelet analysis of long-range-dependent traffic. *IEEE Transactions on Information Theory*, 44(1):2–15, Jan 1998. doi:10.1109/18.650984.

[3] P. J. Acklam. An algorithm for computing the inverse normal cumulative distribution function (unpublished). University of Oslo, Statistics Division, 2003.

[4] Inc. Amazon.com. Amazon Web Services, 2017. URL: http://www.aws.amazon.com/ec2/.

[5] S. Asmussen, M. Olsson, and O. Nerman. Fitting phase-type distributions via the EM algorithm. *Scandinavian Journal of Statistics*, 23(4):419–441, 1996.

[6] G. I. Belyavsky, N. V. Danilova, and S. S. Sushko. The ways of finding fair prices of contingent claims for the discrete and continious cases when parameters of (b,s)-market may change at stochastic point of time. *IZVESTIYA VUZOV. SEVERO-KAVKAZSKII REGION. NATURAL SCIENCE.*, 6:5–8, 2010. doi:10.24412/FfcERkBxJlk.

[7] F. Bennett. Sobol C++ library, 2017. URL: https://people.sc.fsu.edu/~jburkardt/cpp_src/sobol/sobol.html.

[8] J. Beran. *Statistics for long-memory processes. Monographs on statistics and applied probability (Vol. 61)*. New York: Chapman and Hall/CRC, 1994.

[9] J. Beran, Y. Feng, S. Ghosh, and R. Kulik. *Long-memory processes: Probabilistic Properties and Statistical Methods*. Berlin [u.a.] : Springer, 2013. doi:10.1007/978-3-642-35512-7.

[10] P. Billingsley. *Statistical Inference for Markov Processes*. The University of Chicago Press, Chicago, 1961.

[11] F. Black. Studies of stock price volatility changes. In *Proceedings of the 1976 Meetings of the American Statistical Association*, pages 171–181, 1976.

[12] M. Bladt and M. Sorensen. Statistical inference for discretely observed Markov jump processes. *Journal of the Royal Statistical Society. Series B. Methodological*, 39(12):395–410, 2005. doi:10.1111/j.1467-9868.2005.00508.x.

[13] S. A. Bochkanov. AlgLib – cross-platform numerical analysis and data processing library, 2017. URL: http://www.alglib.net.

[14] S. A. Bochkanov. BLEIC – Bound and Linear Equality/Inequality Constrained optimization, 2017. URL: http://www.alglib.net.

[15] T. Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31:307–327, 1986.

[16] T. Bollerslev. Glossary to ARCH (GARCH). *CREATES Research Paper 2008-49*, 2008. doi:http://dx.doi.org/10.2139/ssrn.1263250.

[17] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 06 1958. URL: http://dx.doi.org/10.1214/aoms/1177706645, doi:10.1214/aoms/1177706645.

[18] C. G. Broyden. The convergence of a class of double-rank minimization algorithms 1. General considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 03 1970. URL: https://dx.doi.org/10.1093/imamat/6.1.76, arXiv:http://oup.prod.sis.lan/imamat/article-pdf/6/1/76/2233756/6-1-76.pdf, doi:10.1093/imamat/6.1.76.

[19] L.E. Calvet and A. Fisher. Forecasting multifractal volatility. *Journal of Econometrics, Elsevier*, 105(1):27–58, 2001.

[20] L.E. Calvet and A.J. Fisher. How to forecast long-run volatility: Regime switching and the estimation of multifractal processes. *Journal of Financial Econometrics*, 2(1):49–83, 2004.

[21] A.A. Christie. The stochastic behavior of common stock variances: value, leverage and interest rate effects. *Journal of Financial Economics*, 10:407–432, 1982.

[22] P. Christoffersen, S. Heston, and K. Jacobs. Option anomalies and the pricing kernel. Working Papers 11-17, University of Pennsylvania, Wharton School, Weiss Center, June 2010. URL: https://ideas.repec.org/p/ecl/upafin/11-17.html.

[23] P. Christoffersen, K. Jacobs, and C. Ornthanalai. GARCH option valuation: Theory and evidence. CREATES Research Papers 2012-50, Department of Economics and Business Economics, Aarhus University, May 2012. URL: https://ideas.repec.org/p/aah/create/2012-50.html.

[24] Boost community. Boost – free peer-reviewed portable C++ source libraries, 2017. URL: https://www.boost.org.

[25] Nvidia Corporation. Nvidia CUDA – a parallel computing platform and programming model, 2017. URL: http://www.nvidia.com/cuda.

[26] A. Damodaran. Equity risk premiums (ERP): determinants, estimation and implications – a post-crisis update. *Financial Markets, Institutions & Instruments*, 18:289 – 370, 11 2009. doi:10.1111/j.1468-0416.2009.00151.x.

[27] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B. Methodological*, 39(6):1–38, 1977.

[28] J.-C. Duan. The GARCH option pricing model. *Mathematical Finance*, 5:13–32, 1995.

[29] J.-C. Duan, I. Popova, and P. Ritchken. Option pricing under regime switching. *Quantitative Finance*, 2(2):116–132, 2002. doi:10.1088/1469-7688/2/2/303.

[30] J.-C. Duan, P. Ritchken, and Z. Sun. Jump starting GARCH: Pricing and hedging options with jumps in returns and volatilities. *SSRN Electronic Journal*, 02 2006. doi:10.2139/ssrn.479483.

[31] D. Duffie, D. Filipović, and W. Schachermayer. Affine processes and applications in finance. *Annals of Applied Probability*, 13, 08 2003. doi:10.1214/aoap/1060202833.

[32] D. Duffie, J. Pan, and K. Singleton. Transform analysis and asset pricing for affine jump-diffusions. *Econometrica*, 68:1343–1376, 02 2000. doi:10.2139/ssrn.157733.

[33] B. Dumas, J. Fleming, and R. E. Whaley. Implied volatility functions: Empirical tests. *Journal of Finance*, 53(6):2059–2106, 1998. URL: https://EconPapers.repec.org/RePEc:bla:jfinan:v:53:y:1998:i:6:p:2059-2106.

[34] R. F. Engle and V. K. Ng. Measuring and testing the impact of news on volatility. *The Journal of Finance*, 1993.

[35] R.F. Engle. Autoregressive conditional heteroskedasticity with estimates of the variance of U.K. inflation. *Econometrica*, 50:987–1008, 1982.

[36] R.G. Gallager. *Discrete Stochastic Processes*. Springer, 1996. doi:10.1007/978-1-4615-2329-1.

[37] R.G. Gallager. *Stochastic processes theory for applications*. Springer, 2013.

[38] J. Geweke and S. Porter-Hudak. The estimation and application of long memory time series models. *Journal of Time Series Analysis*, 4:221, 1983.

[39] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer New York, 2003. doi:10.1007/978-0-387-21617-1.

[40] L. R. Glosten, R. Jagannathan, and D. E. Runkle. On the relation between the expected value and the volatility of the nominal excess return on stocks. *The Journal of Finance*, 48(5):1779–1801, 1993. URL: https://onlinelibrary.wiley.com/doi/

abs/10.1111/j.1540-6261.1993.tb05128.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1540-6261.1993.tb05128.x, doi:10.1111/j.1540-6261.1993.tb05128.x.

[41] P. Guttorp. *Stochastic modeling of scientific data*. Chapman and Hall, 1995.

[42] William H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007. URL: http://www.amazon.com/Numerical-Recipes-3rd-Scientific-Computing/dp/0521880688/ref=sr_1_1?ie=UTF8&s=books&qid=1280322496&sr=8-1.

[43] J. D. Hamilton. *Time series analysis*. Princeton Univ. Press, Princeton, NJ, 1994. URL: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+126800421&sourceid=fbw_bibsonomy.

[44] J.D. Hamilton. Regime-switching models. *Palgrave Dictionary of Economics*, 2005.

[45] J. Haslett and A.E. Raftery. Space-time modeling with long-memory dependence: Assessing Ireland's wind power resource (with discussion). *Applied Statistics*, 38:1–50, 1989.

[46] N. Hautsch and Y. Ou. Discrete-time stochastic volatility models and MCMC-based statistical inference. *SSRN Electronic Journal*, 10 2008. doi:10.2139/ssrn.1292494.

[47] S. Heston and S. Nandi. A closed-form GARCH option valuation model. *Review of Financial Studies*, 13:585–625, 02 2000. doi:10.2139/ssrn.210009.

[48] E. Hillebrand. *Mean reversion models of financial markets*. PhD thesis, Bremen University, Bremen, 2003. URL: https://suche.suub.uni-bremen.de/peid=B36703941.

[49] E. Hlawka. *The theory of uniform distribution*, volume 1. A B Academic, 1984.

[50] P. J. Huber. The behavior of maximum likelihood estimates under nonstandard conditions. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 221–233, Berkeley, Calif., 1967. University of California Press. URL: https://projecteuclid.org/euclid.bsmsp/1200512988.

[51] J. Hull and A. White. The pricing of options on assets with stochastic volatilities. *The Journal of Finance*, 42:281–300, 1987.

[52] H.E. Hurst. Long-term storage capacity of reservoirs. *Transactions of the American Society of Civil Engineers*, 116:770, 1951.

[53] Lester Ingber. Adaptive simulated annealing (ASA): Lessons learned, 2000. arXiv:cs/0001018.

[54] P. Jäckel. *Monte Carlo Methods in Finance*. Wiley, 2002.

[55] J. Kallsen, Y. Kabanov, R. Liptser, and J. Stoyanov. *A Didactic Note on Affine Stochastic Volatility Models*, pages 343–368. 04 2007. doi:10.1007/978-3-540-30788-4_18.

[56] J. Kallsen and M. Taqqu. Option pricing in ARCH-type models. *Mathematical Finance*, 8:13 – 26, 01 1998. doi:10.1111/1467-9965.00042.

[57] A. Klenke. *Probability Theory: A Comprehensive Course*. Universitext. Springer London, 2013. URL: https://books.google.ru/books?id=eTYMnQEACAAJ.

[58] K.A. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2:164–168, 1944.

[59] A. E. Leövey. *Multifractal Models : Estimation, Forecasting and Option Pricing*. PhD thesis, CAU, Kiel, 2015.

[60] R. Liu, T. Di Matteo, and T. Lux. True and apparent scaling: The proximity of the Markov-switching multifractal model to long-range dependence. *Physica A: Statistical Mechanics and its Applications*, 2007. doi:10.1016/j.physa.2007.04.085.

[61] R. Liu and T. Lux. Long memory in financial time series: Estimation of the bivariate multi-fractal model and its application for value-at-risk, 2005.

[62] R. Liu and T. Lux. Flexible and robust modelling of volatility comovements: a comparison of two multifractal models. Kiel Working Papers 1594, Kiel Institute for the World Economy (IfW), 2010. URL: https://ideas.repec.org/p/zbw/ifwkwp/1594.html.

[63] R. Liu and T. Lux. Non-homogeneous volatility correlations in the bivariate multifractal model. *European Journal of Finance*, 2014. doi:10.1080/1351847X.2014.897960.

[64] R. Liu and T. Lux. Generalized method of moment estimation of multivariate multifractal models. *Economic Modelling*, 2017. doi:10.1016/j.econmod.2016.11.010.

[65] A. W. Lo. Long-term memory in stock market prices. *Econometrica*, 59(5):1279–1313, 1991.

[66] T. Lux. Rational forecasts or social opinion dynamics? Identification of interaction effects in a business climate survey. *Journal of Economic Behavior & Organization*, 72(2):638–655, 2009. doi:10.1016/j.jebo.2009.07.003.

[67] B.B. Mandelbrot and J.R. Wallis. Robustness of the rescaled range R/S in the measurement of noncyclic long-run statistical dependence. *Water Resources Research*, 5:967, 1969.

[68] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998. doi:10.1145/272991.272995.

[69] G. McLachlan and T. Krishnan. *The EM algorithm and extensions*. John Wiley & Sons, New York, 1996.

[70] P. Metzner, E. Dittmer, T. Jahnke, and C. Schütte. Generator estimation of Markov jump processes. *Journal of Computational Physics*, 227(1):353–375, 2007. doi:10.1016/j.jcp.2007.07.032.

[71] P. Metzner, I. Horenko, and C. Schütte. Generator estimation of Markov jump processes based on incomplete observations non-equidistant in time. *Phys. Rev. E*, 76(06):066702, 2007. doi:10.1103/PhysRevE.76.066702.

[72] C. Moler and C. Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003. arXiv:https://doi.org/10.1137/S00361445024180, doi:10.1137/S00361445024180.

[73] B. Moro. The full Monte. *Risk Magazine*, 8(2):57–58, 1995.

[74] H. R. Neave. On using the Box-Muller transformation with multiplicative congruential pseudo-random number generators. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 22(1):92–97, 1973. URL: http://www.jstor.org/stable/2346308.

[75] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 01 1965. URL: https://dx.doi.org/10.1093/comjnl/7.4.308, arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/7/4/308/1013182/7-4-308.pdf, doi:10.1093/comjnl/7.4.308.

[76] D.B. Nelson. Conditional heteroskedasticity in asset returns: a new approach. *Econometrica*, 59:347–370, 1991.

[77] J. R. Norris. *Markov chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997. doi:10.1017/CBO9780511810633.

[78] C.-K. Peng, S. V. Buldyrev, S. Havlin, M. Simons, H. E. Stanley, and A. L. Goldberger. Mosaic organization of DNA nucleotides. *Phys. Rev. E*, 49:1685–1689, Feb 1994. URL: https://link.aps.org/doi/10.1103/PhysRevE.49.1685, doi:10.1103/PhysRevE.49.1685.

[79] R. Pozo. Template Numerical Toolkit – an interface for scientific computing in C++, 2004. URL: https://www.math.nist.gov/tnt.

[80] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019. URL: https://www.R-project.org/.

[81] P. Rao. *Maximum likelihood estimation for Markov processes*. Kluwer Academic Publishers, 1972. URL: https://link.springer.com/article/10.1007/BF02479763, doi:10.1007/BF02479763.

[82] V.A. Reisen. Estimation of the fractional difference parameter in the ARFIMA(p,d,q) model using the smoothed periodogram. *Journal of Time Series Analysis*, 15(1):335–350, 1994.

[83] C. Sammut and G. I. Webb, editors. *EM Clustering*, pages 311–311. Springer US, Boston, MA, 2010. doi:10.1007/978-0-387-30164-8_248.

[84] M. Segnon and T. Lux. Multifractal models in finance: Their origin, properties, and applications. Kiel Working Papers 1860, Kiel Institute for the World Economy (IfW), 2013. URL: https://ideas.repec.org/p/zbw/ifwkwp/1860.html.

[85] S. V. Semenovskaya, K. A. Khachaturyan, and A. G. Khachaturyan. Statistical mechanics approach to the structure determination of a crystal. *Acta Crystallographica Section A*, 41(3):268–273, May 1985. doi:10.1107/S0108767385000563.

[86] R. Seydel. *Tools for Computational Finance (3ed.),*. Springer, 2006.

[87] J. C. Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37:332–341, 01 1992. doi:10.1109/9.119632.

[88] W. J. Stewart. *Probability, Markov chains, queues, and simulation: The mathematical basis*. Princeton University Press, 2009.

[89] T. A. Straeter. *On the Extension of the Davidon-Broyden Class of Rank One, Quasi-Newton Minimization Methods to an Infinite Dimensional Hilbert Space with Application to Optimal Control Problems*. North Carolina State University at Raleigh, 1971.

[90] S. S. Sushko. The black-scholes models with deterministic and random time moment of the parameters changes. *IZVESTIYA VUZOV. SEVERO-KAVKAZSKII REGION. NATURAL SCIENCE.*, 4:21–24, 2010. doi:10.24412/FfcEfmEve40.

[91] S.S. Sushko. Pricing of european type options for levy and conditionally levy type models (dissertation). Master's thesis, Högskolan i Halmstad/Sektionen för Informationsvetenskap, Data- och Elektroteknik (IDE), 2008.

[92] C. S. Tapiero. *Risk and financial management*. Wiley, 2004.

[93] M. S. Taqqu and V. Teverovsky. On estimating the intensity of long-range dependence in finite and infinite variance time series. *A practical guide to heavy tails: statistical techniques and applications*, 177:218, 1998.

[94] M.S. Taqqu, V. Teverovsky, and W. Willinger. Estimators for long-range dependence: an empirical study. *Fractals*, 3:785–798, 1995.

[95] S. J. Taylor. *Modeling Financial Time Series*. Wiley, 1986.

[96] V. Teverovsky, M. Taqqu, and W. Willinger. A critical look at Lo's modified R/S statistic. *Journal of Statistical Planning and Inference*, 80:211–227, 08 1999. doi:10.1016/S0378-3758(98)00250-X.

[97] W. Weidlich. Physics and social science – the approach of synergetics. *Physics Reports*, 204(1):1–163, 1991. doi:10.1016/0370-1573(91)90024-G.

[98] W. Weidlich. *Sociodynamics – a systematic approach to mathematical modeling in the social sciences*. CRC Press, 2000.

[99] V. Zarnowitz, National Bureau of Economic Research, and National Bureau of Economic Research. Conference on Research in Business Cycles. *Business Cycles: Theory, History, Indicators, and Forecasting*. National Bureau of Economic Research Cambridge, Mass: National Bureau of Economic Research studies in business cycles. University of Chicago Press, 1992. URL: https://books.google.ru/books?id=ldv7TbwAE1kC.

[100] D. Zhuanxin, C.W.J. Granger, and R.F. Engle. A long memory property of stock market returns and a new model. *Journal of Empirical Finance*, 1(1):83–106, 1993.

C

Code listings

C.1. ABM model

The purpose of the program is to estimate the ABM model parameters. The code is based on the Object-Oriented Programming (OOP) principles. They allow to join a data with methods of data manipulation. In the presented code such an object is called *InputDataClass*. It contains the input data necessary for the model estimation (see comments in the code listings below). Further, *InputDataClass* contains two methods that are standard for C++: the constructor and destructor. The first one allocates memory and assigns the values read from *InputData* structure containing sample paths, the data extracted from the sample to the corresponding private variables of the object *InputDataClass*. In opposite, the second method frees the used memory. As a result, the complete set of matrices and vectors necessary for the ABM model estimation can be created, copied, transmitted and destroyed in one line of code.

C.1.1. Header file with main structures and global variables

This file is used to separate the variables and structures definitions from the subroutines code.

```
// Inclusion of standard  C++ header files
#include <time.h>
#include <windows.h> // "winbase.h"// (include
#include <iostream>
#include <vector>

// Inclusion of AlgLib algebraic computation library files
#include "src\AlgLib\ap.h"
#include "src\AlgLib\alglibinternal.h"
#include "src\AlgLib\alglibmisc.h"
#include "src\AlgLib\linalg.h"
```

```
#include "src\AlgLib\statistics.h"
#include "src\AlgLib\dataanalysis.h"
#include "src\AlgLib\specialfunctions.h"
#include "src\AlgLib\solvers.h"
#include "src\AlgLib\optimization.h"
#include "src\AlgLib\diffequations.h"
#include "src\AlgLib\fasttransforms.h"
#include "src\AlgLib\integration.h"
#include "src\AlgLib\interpolation.h"

//  Inclusion of TNT algebraic computaion library files
#include "src\TNT\tnt.h"
#include "src\TNT\jama_eig.h"
#include "src\TNT\jama_lu.h"

// Inclusion of Boost computaion library files
// necessary for  random number generation and o.d.e.
// solving
#include "boost\numeric\odeint.hpp"
#include "boost\random\lagged_fibonacci.hpp"
#include "boost\random\uniform_01.hpp"
#include "boost\random\variate_generator.hpp"
#include "boost\random\mersenne_twister.hpp"
#include "boost\math\tools\minima.hpp"

using namespace alglib;

// disable some irrelevant warnings
#if (AE_COMPILER==AE_MSVC)
#pragma warning(disable:4100)
#pragma warning(disable:4127)
#pragma warning(disable:4702)
#pragma warning(disable:4996)
#endif

// Some other standard headers
#include <string.h>
#include <iostream>
#include <string>
#include <fstream>
#include <conio.h>
#include <math.h>
#include <stdlib.h>

// GLOBALS
```

```
#define SDK_SUCCESS 0
#define SDK_FAILURE 1

// Defining file with  algorithms settings to read
std::ifstream settings;

// Data structure containing sample path with observations
// and corresponding objects
struct InputData
{
   int method;     // 0: Lc-MLE; 1: EM-algorithm; 2,3: Ld-MLE

   int ** c;       // matrix with number of transitions in y
   double ** N;    // matrix with number of transitions in y_ob
   double ** Q;    // intensity rates matrix
   double * R;     // vector of holding time in each state
   double * y;     // vector of continuos-time observed states
   double * rho;   // transition intervals in continuos-time sample path
   double * y_ob;  // discrete-time sample paths states

   int NN;         // number of agents
   int MM;         // limit of number of transitions in y
   int MM_max;     // real number of transitions in y
   double TT;      // time horizon of observed path
   int TT_d;       // number of discrete transitions
   double delta_tk;// discretization step of y_ob

   double ** P_d;        // transition probability matrix
   // vector of diagonal elements of eigen decomposition
   double ** D_lambda_d;
   double ** U_d;
   // matrix with eigen vectors of eigen decomposition
   double ** invU_d;     // inverse of U_d

   // frequncy of reinitialization of
   // transition probability calculation
   int Reinit;
};


class InputDataClass
{
private:

   double ** Q;         // intensity rates matrix
```

```
    int ** c_tmp;        // matrix with number of transitions in y
    double ** N_tmp;     // matrix with number of transitions in y_ob
    double * R_tmp;      // vector of holding time in each state
    double * y_tmp;      // vector of continuos-time observed states
    double * rho_tmp;    // transition intervals in cont.-time sample path
    double * y_ob_tmp;   // discrete-time sample paths states
    double * y_ob_tmp2;  // discrete-time sample paths states

    int MM_max; // real number of transitions in y
    int N;      // number of agents
    int M;      // limit of number of transitions in y

    double ** P_d;       // transition probability matrix
    // vector of diagonal elements of eigen decomposition
    double ** D_lambda_d;
    double ** U_d;
    // matrix with eigen vectors of eigen decomposition
    double ** invU_d;    // inverse of U_d

    InputData Data;

public:
    InputDataClass(InputData * Init);
    ~InputDataClass();
};

InputDataClass::InputDataClass(InputData * Init)
{

    int NN = (*Init).NN;
    int MM = (*Init).MM;

    //===== Memory Allocation =====
    Q = new double *[2 * NN + 1];
    for (int i = 0; i < 2 * NN + 1; i++)
        Q[i] = new double[2 * NN + 1];
    for (int i = 0; i <= 2 * NN; i++){
        for (int j = 0; j <= 2 * NN; j++)
        { Q[i][j] = 0; } }

    c_tmp = new int *[2 * NN + 1];
    for (int i = 0; i < 2 * NN + 1; i++)
        c_tmp[i] = new int[2 * NN + 1];

    N_tmp = new double *[2 * NN + 1];
```

```
for (int i = 0; i < 2 * NN + 1; i++)
    N_tmp[i] = new double[2 * NN + 1];

R_tmp = new double[2 * NN + 1];
y_tmp = new double[MM];
y_tmp[0] = 0; // INITIAL POINT !
rho_tmp = new double[MM];

y_ob_tmp = new double[MM];


//======== Mem Alloc for Ld-MLE ===========
P_d = new double *[2 * NN + 1];
for (int i = 0; i < 2 * NN + 1; i++)
    P_d[i] = new double[2 * NN + 1];

D_lambda_d = new double *[2 * NN + 1];
for (int i = 0; i < 2 * NN + 1; i++)
    D_lambda_d[i] = new double[2 * NN + 1];

U_d = new double *[2 * NN + 1];
for (int i = 0; i < 2 * NN + 1; i++)
    U_d[i] = new double[2 * NN + 1];

invU_d = new double *[2 * NN + 1];
for (int i = 0; i < 2 * NN + 1; i++)
    invU_d[i] = new double[2 * NN + 1];

// Assign InputData structure the pointers
// of allocated memory
(*Init).c = c_tmp;
(*Init).N = N_tmp;
(*Init).R = R_tmp;
(*Init).y = y_tmp;
(*Init).y_ob = y_ob_tmp;
(*Init).rho = rho_tmp;
(*Init).Q = Q;
(*Init).P_d = P_d;
(*Init).D_lambda_d = D_lambda_d;
(*Init).U_d = U_d;
(*Init).invU_d = invU_d;

// Assign whole InputData struct Init to
// Data private variable in the class
Data = (*Init);
```

```
}

InputDataClass::~InputDataClass()
{
    // Memory free
    delete[](R_tmp);
    delete[](y_tmp);
    delete[](rho_tmp);
    delete[](y_ob_tmp);
    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] c_tmp[i];
    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] N_tmp[i];

    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] Q[i];
    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] P_d[i];
    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] D_lambda_d[i];
    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] U_d[i];
    for (int i = 0; i<2 * Data.NN + 1; i++)
        delete[] invU_d[i];
};

struct Results{ double x[3]; };

// Global variables
int method;
int do_parallel; // trigger of parallel computations

int NN;
int MM;
int Reinit;
double TT;
double delta_tk;

// real parameters values
double nu_real;
double alpha0_real;
double alpha1_real;

// range of simulations
int MC_iterations1;
```

```
int MC_iterations2;

// Initial point of optimization method
int IP;

//  Numerical O.D.E. solver from AlgLib
// 1: runge_kutta4; 2: runge_kutta_fehlberg78;
// 3: bulirsch_stoer
int ode_stepper;

// Array of Prime number  can be used as seeds of
// random number generators
unsigned int  PrimesSampleArr[1000];

int    k;
double counter;

// stopping criterions
double epsg; // = 1.0e-7;
double epsf; // = 0; // 15 ?
double epsx; // = 0;
double diffstep; // = 1.0e-7;
double ode_step_size;

// Input data
real_1d_array y_ob;
real_2d_array xx;

// Boundaries of search regions of optimizers
real_1d_array bndl; // = "[1.1, -1.5, 0.0]";
real_1d_array bndu; // = "[5,    1.5, 5.0]";
real_1d_array scale; // = "[5,    1.5, 5.0]";

// trigger
int output;

// The type of container used to hold the state vector
typedef std::vector< double > state_type;

hqrndstate rnd_state, rnd_state2;


// Additional functions
void StartTimer(_int64 *pt1)
{
```

```
    QueryPerformanceCounter((LARGE_INTEGER*)pt1);
}

double StopTimer(_int64 t1)
{
    _int64 t2, ldFreq;

    QueryPerformanceCounter((LARGE_INTEGER*)&t2);
    QueryPerformanceFrequency((LARGE_INTEGER*)&ldFreq);
    return ((double)(t2 - t1) / (double)ldFreq);
}

std::string IntToStr(int x)
{
    std::stringstream r;
    r << x;
    return r.str();
}

std::string FloatToStr(float x)
{
    std::stringstream r;
    r << x;
    return r.str();
}

float StrToFloat(std::string s)
{
    return (float)::atof(s.c_str());
}
```

C.1.2. Main file with code

This file contains all the subroutines necessary to estimate the ABM model using any of three methods described in the main part of this dissertation, in particular, see Subsections 2.3.2, 2.4.2 and 2.5.2. In addition, there are subroutines for simulation of artificial continuous-time sample paths, discretization of them, parsing of a real data and the file with settings. Finally, the main function running all the code is also in this file.

```
// Inclusion of standard headers and ABM.hpp header file
#include "stdafx.h"
#include "ppl.h"

#include "ABM.hpp"

// This function computes intensity rate  w_u of  transition up (q_ii+1)
```

```
double w_u(double x, double nu, double alpha0, double alpha1)
{
   return (1 - x) * nu * exp( (alpha0 + alpha1*x));
   //return              nu * exp( (alpha0 + alpha1*x));
}

// This function computes intensity rate  w_u of  transition down (q_ii-1)
double w_d(double x, double nu, double alpha0, double alpha1)
{
   return (1 + x) * nu * exp( - (alpha0 + alpha1*x) );
   //return              nu * exp( - (alpha0 + alpha1*x) );
}

// This function computes transition probability of transition up (q_ii+1)
double pi_u(double x, double nu, double alpha0, double alpha1)
{
   if(x!=1.f){
      return w_u(x, nu, alpha0, alpha1)/
      ( w_u(x, nu, alpha0, alpha1) + w_d(x, nu, alpha0, alpha1) );}
   else{
      return 0;}
}

// This function computes transition probability of transition up (P_ii-1)
double pi_d(double x, double nu, double alpha0, double alpha1)
{
   if(x!=1.f){
      return w_d(x, nu, alpha0, alpha1)/
      ( w_u(x, nu, alpha0, alpha1) + w_d(x, nu, alpha0, alpha1) );}
   else{
      return 0;}
}

// This function computes log-likelihood function log(L^c(theta|x))
double L(
   double nu,  double alpha0, double alpha1,
   int NN, double** N, double* R)
{
   double sum = 0;
   for(int i=-NN;i<NN;i++)
   {
      sum += N[i][i+1] * log(w_u( (double)i/
         (double)NN, nu,alpha0,alpha1)) -
      R[i] * w_u( (double)i/
         (double)NN, nu,alpha0,alpha1);
```

```
    }
    for(int i=-NN+1;i<NN+1;i++)
    {
        sum += N[i][i-1] * log(w_d( (double)i/
            (double)NN, nu,alpha0,alpha1)) -
        R[i] * w_d( (double)i/
            (double)NN, nu,alpha0,alpha1);
    }
    return sum;
}

// This function computes function Phi_pq(t;Theta)
// from Corollary 4
void Matrix_Ksi(double t, double ** Ksi, double ** D_lambda, int NN)
{
    for(int i=0; i<=2 * NN; i++)
    {
        for(int j=0; j<=2 * NN; j++)
        {
            if(D_lambda[i][i]==D_lambda[j][j]){
                Ksi[i][j] = t * exp(D_lambda[i][i]);}
            else {
                Ksi[i][j] = (exp(t*D_lambda[i][i]) -
                exp(t*D_lambda[j][j]))/
                (D_lambda[i][i] - D_lambda[j][j]);
            }
        }
    }
    //return *Ksi_;
}

// This function computes matrix of intensity rates Q
void Matrix_Q(double ** Q, double nu, double alpha0, double alpha1, int NN)
{
    double w1 = 0, w2 = 0;
    for(int i=1; i<=2 * NN - 1; i++)
        {
            w1 = (double)w_d( (double)(-NN+i)/
            (double)NN, nu, alpha0, alpha1);
            w2 = (double)w_u( (double)(-NN+i)/
            (double)NN, nu, alpha0, alpha1);
            Q[i][i-1] =   w1;
            Q[i][i]   = - w2 - w1;
            Q[i][i+1] =   w2;
        }
```

```
    w1 = w_u(-1.0,nu,alpha0,alpha1);
    w2 = w_d( 1.0,nu,alpha0,alpha1);
    Q[0][0]               = - w1;
    Q[0][1]               =   w1;
    Q[2 * NN][2 * NN - 1] =   w2;
    Q[2 * NN][2 * NN]     = - w2;
}

// This function computes matrix exponential of diagonal matrix D
void Matrix_expD(double t, double** D_lambda, int NN)
{
    for(int i=0;i<=2 * NN; i++)
    {
        D_lambda[i][i] = t * exp(D_lambda[i][i]);
    }
}

using namespace TNT;
using namespace JAMA;

// Inversion function for TNT library from
// http://wiki.cs.princeton.edu/index.php/TNT
template<class T>
TNT::Array2D<T> invert(const TNT::Array2D<T> &M)
{
    assert(M.dim1() == M.dim2()); // Is M square matrix

    // solve for inverse with LU decomposition
    JAMA::LU<T> lu(M);

    // create identity matrix
    TNT::Array2D<T> id(M.dim1(), M.dim2(), (T)0);
    for (int i = 0; i < M.dim1(); i++) id[i][i] = 1;

     // solves A * A_inv = Identity
    return lu.solve(id);
}

// This function computes transition probability matrix P
// based on an eigen decomposition of matrix
// using subroutines from TNT library
void Matrix_P_eigen_decomposition_TNT(
    double t,double** P,double** Q,
    double** U, double** D_lambda,
```

```
    double** invU, int NN, int flag)
{
    // Memory allocation for matrices tempP and A
    double ** tempP = new double *[2 * NN + 1];
    Array2D< double > A(2*NN+1,2*NN+1, 0.0);

    // Initialize array A values with intensity rate q_ij
     for (int i=0; i <= 2*NN; i++)
       for (int j=0; j <= 2*NN; j++)
          A[i][j] = Q[i][j];

      // Create (2*NN+1)x(2*NN+1) arrays V, D
      Array2D<double> V(2*NN+1,2*NN+1);
      Array2D<double> D(2*NN+1,2*NN+1);

      // Create (2*NN+1) vector e
      Array1D<double> e(2*NN+1);

    // Eigen-decomposition of A using methods of TNT library
     Eigenvalue<double> EV(A);
     EV.getV(V); // Assign eigen vectors of A to V
     EV.getD(D); // Assign eigen number A to D

    // if flag is true, rewrite values of D to D_lambda array
    //    from matrix format of TNT to convenient one
    if(flag)
    {
       for(int i=0;i<= 2 * NN; i++)
       {

          D_lambda[i][i] = D[i][i];
          for(int j=0;j<= 2 * NN; j++)
          {
              // in U eigenvectors are rows
             U[j][i] = V[j][i];
          }
       }
    }

    double sum;
    // Multiplicate matrix V on matrix exponential of D
    for(int i=0;i<= 2 * NN; i++)
    {
       for(int j=0;j<= 2 * NN; j++)
       {
```

```
            tempP[i][j] = (double)V[i][j] *
            exp((double)t *  D[j][j]);
        }
    }

    // Find a lower triangular matrix L and an upper triangular matrix U
    // so that matrix lu is  matrix V (from JAMA package)
    JAMA::LU<double> lu(V);

    // create identity matrix id
    TNT::Array2D<double> id(V.dim1(), V.dim1(), (double)0);
    for (int i = 0; i < V.dim1(); i++) id[i][i] = 1;

     // solves for inverse matrix, V * V_inv = Identity
    D = lu.solve(id);

    // Rewrite obtained inverse matrix D of V to
    // matrix inv U of convenient data type
    if(flag)
    {
        for(int i=0;i<= 2 * NN; i++)
        {
            for(int j=0;j<= 2 * NN; j++)
            {
                invU[i][j] = D[i][j]; //vr[i][j];
            }
        }
    }

    // Calculate transition probability matrix P as U exp(tD) invU
    for(int i=0;i<= 2 * NN; i++)
    {

        for(int j=0;j<= 2 * NN; j++)
        {
            sum = 0;
            for(int k=0;k<= 2 * NN; k++)
            {
                sum += tempP[i][k] *  D[k][j];
            }
            P[i][j] = sum;
        }
    }

    for (int i = 0; i<2 * NN + 1; i++) delete[] tempP[i];
```

```
}

// This function computes transition probability matrix P
// based on an eigen decomposition of matrix
// using alglib library subroutines
void Matrix_P_eigen_decomposition(
   double t,double** P,double** Q,double** U, d
   ouble** D_lambda, double** invU, int NN, int flag)
{
   // Memory allocation and initialization
   double ** tempP = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++)
      tempP[i] = new double[2 * NN + 1];

   alglib::real_1d_array wr;
    alglib::real_1d_array wi;
    alglib::real_2d_array vl;
    alglib::real_2d_array vr;
   alglib::real_2d_array vr2;

   alglib::real_2d_array a;
   a.setlength(2 * NN + 1, 2 * NN + 1);
   for(int i=0;i<= 2 * NN; i++){
      for(int j=0;j<= 2 * NN; j++){
         a[i][j] = Q[i][j];
      }}

   // Eigen decomposition of intensity rates matrix a
   // using alglib library subroutines. The result is
   alglib::rmatrixevd(
      a,//real_2d_array a,
      2 * NN + 1,//ae_int_t n,
      1,//ae_int_t v needed,
      wr,  // eigenvalues
      wi,
      vl,
      vr);  // eigenvectors are columns

   // Rewrite eigen vectors and values from matrix
   // format of alglib to convenient one
   if(flag)
   {
      for(int i=0;i<= 2 * NN; i++)
      {
```

```
        D_lambda[i][i] = wr[i];;
        for(int j=0;j<= 2 * NN; j++)
        {
            // in U eigenvectors are rows
            U[j][i] = vr[j][i];
        }
    }
}

double sum;
// Multiplicate matrix V on matrix exponential of D
for(int i=0;i<= 2 * NN; i++)
{
    for(int j=0;j<= 2 * NN; j++)
    {
        tempP[i][j] = (double)vr[i][j] *
            exp((double)t * (double)wr[j]);
    }
}

// Inverse eigen vectors matrix vr aka U
ae_int_t info;
 matinvreport rep;
 rmatrixinverse(vr, info, rep);

// Rewrite eigen vectors and values from matrix
// format of alglib to convenient one
if(flag)
{
    for(int i=0;i<= 2 * NN; i++)
    {
        for(int j=0;j<= 2 * NN; j++)
        {
            invU[i][j] = vr[i][j];
        }
    }
}

// Calculate transition probability matrix P as U exp(tD) invU
for(int i=0;i<= 2 * NN; i++)
{
    for(int j=0;j<= 2 * NN; j++)
    {
        sum = 0;
        for(int k=0;k<= 2 * NN; k++)
```

```
            {
                double temp = vr[k][j];
                sum += tempP[i][k] * (double)vr[k][j];
            }
            P[i][j] = sum;
        }
    }

    for (int i = 0; i<2 * NN + 1; i++) delete[](tempP[i]);
}

// This function computes integral int_0^{delta t}(P_ki P_il)
// then divides it by P_kl from Theorem 5 and Corollary 4
void Matrix_Mk(
    double t, int k, int l, int i, double *Mk,
    double** P, double** Ksi, double** D_lambda,
    double** U, double** invU, int NN)
{
    double sum = 0;
    for(int j=0;j<=2 * NN;j++)
    {
        double sum2 = 0;
        for(int q=0;q<=2 * NN;q++)
        {
            sum2 += (double)U[i][q] *
            (double)invU[q][l] *
            (double)Ksi[j][q];
        }
        sum += (double)U[k][j] * (double)invU[j][i] * (double)sum2;
    }
    if (P[k][l]!=(double)0) {
        *Mk = 1/P[k][l] * sum;}else{ *Mk = (double)0;}
}

// This function computes expectation of R vector elements
// using the formula from Theorem 5
void Matrix_E_R(
    double t, double* E_R, int** c, double** P,
    double** Ksi, double** D_lambda, double** U,
    double** invU, int NN)
{
    double Mk = (double)0;

    for(int i=0;i<=2 * NN;i++)
    {
```

```
        double sum = 0;
        for(int k=0;k<=2 * NN;k++)
        {
            for(int l=0;l<=2 * NN;l++)
            {
                Matrix_Mk( t,k,l,i,&Mk,P,Ksi,
                    D_lambda,U,invU,NN);
                sum += (double)c[k][l] * Mk;
            }
        }
        E_R[i] = (double)sum;
    }
}

// This function computes integral int_0^{delta t}(P_ki P_il)
// then multiplies it  by c_kl/P_kl from Theorem 5
// and Corollary 4
void Matrix_fk(
    double t,int k,int l,int i,int j,double *fk,
    double** Q,double** P,double** Ksi,
    double** D_lambda,double** U,
    double** invU, int NN)
{
        if(i!=j)
        {
            double sum = 0;
            for(int pp=0;pp<=2 * NN;pp++)
            {
                double sum2 = 0;
                for(int qq=0;qq<=2 * NN;qq++)
                {
                    sum2 += (double)U[j][qq] *
                    (double)invU[qq][l] *
                    (double)Ksi[pp][qq];
                }
                sum += (double)U[k][pp] *
                (double)invU[pp][i] * sum2;
            }
            if (P[k][l] != (double)0) {
                *fk = (double)Q[i][j]/P[k][l] * sum;}
            else {*fk = (double)0;}
        }
        else
        {*fk /*[i][i]*/ = 0.L;}
}
```

```
// This function computes expectation of N matrix elements
// using the formula from Theorem 5
void Matrix_E_N(
   double t, double** E_N, int** c, double** Q,
   double** P, double** Ksi, double** D_lambda,
   double** U, double** invU, int NN)
{
   double fk = 0.L;
   for(int i=0;i<=2 * NN;i++)
   {for(int j=0;j<=2 * NN;j++)
   {
      double sum = 0;
      for(int k=0;k<=2 * NN;k++)
      {
         for(int l=0;l<=2 * NN;l++)
         {
            Matrix_fk(t,k,l,i,j,&fk,Q,P,Ksi,
               D_lambda,U,invU,NN);
            sum += (double)c[k][l] * fk/*[i][j]*/;
         }
      }
      E_N[i][j] = (double)sum;
   }}
}

// This function computes  expectation of log-likelihood L^c
// from the formula (1.33)
double c_Likelihood(
   double nu, double alpha_0, double alpha_1,
   double** E_N, double* E_R,  int NN)
{
   double sum = 0;

   double ** Q = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++)
      Q[i] = new double[2 * NN + 1];

   Matrix_Q( Q, nu, alpha_0, alpha_1, NN);

   for(int i=1;i<=2 * NN - 1;i++)
   {
      sum += log(Q[i][i-1]) * E_N[i][i-1] - Q[i][i-1] * E_R[i];
      sum += log(Q[i][i+1]) * E_N[i][i+1] - Q[i][i+1] * E_R[i];
   }
```

```
    sum += log(Q[0][1]) * E_N[0][1] - Q[0][1] * E_R[0];
    sum += log(Q[2 * NN][2 * NN - 1]) * E_N[2 * NN][2 * NN - 1] -
        Q[2 * NN][2 * NN - 1] * E_R[2 * NN];


    for (int i = 0; i<2 * NN + 1; i++) delete[](Q[i]);

    return sum;
}

// This function solves O.D.E.
using namespace alglib;
void ode_function_1_diff(
    const real_1d_array &y, double x,
    real_1d_array &dy, void *ptr)
{
    InputData data_tmp = *((InputData *)ptr);
    for(int i=1;i<2*NN;i++)
    {
        dy[i] = data_tmp.Q[i][i-1] * y[i-1] +
                    data_tmp.Q[i][i]   * y[i] +
                    data_tmp.Q[i][i+1] * y[i+1];
    }
    dy[0] = data_tmp.Q[0][0] * y[0] +
        data_tmp.Q[0][1] * y[1];
    dy[2 * NN] = data_tmp.Q[2 * NN][2 * NN - 1] * y[2 * NN - 1] +
        data_tmp.Q[2 * NN][2 * NN] * y[2 * NN];

}

using namespace std;
using namespace boost::numeric::odeint;

// Create a class necessary for application boost o.d.e. solver
// The class is describing Boundary Value Problem
// namely Forward Kolmogorov Equation (Theorem 1)
// for  i-th column of transition probability matrix
// https://en.wikipedia.org/wiki/Boundary_value_problem
class Forward_Kolmogorov_Equation
{
    struct InputData T1;

    public:
    Forward_Kolmogorov_Equation(struct InputData G) : T1(G) {}
```

```
    void operator() (
        const state_type &y, state_type &dydt, const double)
    {
        for (int i = 1; i<2 * T1.NN; i++)
        {
            dydt[i] =
                T1.Q[i][i - 1] *
                y[i - 1] +
                T1.Q[i][i] * y[i]+
                T1.Q[i][i + 1] *
                y[i + 1];
        }
        dydt[0] = T1.Q[0][0] * y[0]+ T1.Q[0][1] * y[1];
        dydt[2 * T1.NN] =
            T1.Q[2 * T1.NN][2 * T1.NN - 1] *
            y[2 * T1.NN - 1]+
            T1.Q[2 * T1.NN][2 * T1.NN] *
            y[2 * T1.NN];
    }
};

void write_out(const state_type &x, const double t)
{
    //std::out << t << '\t' << x[0] << endl;
}

// This function computes transition probability matrix P
// based on an eigen decomposition of matrix
// using alglib o.d.e. and method describe in
//  Section 1.6.2, Theorem 6
void Matrix_P_recursive_ode( InputData * data_tmp)
{
    using namespace std;
    using namespace boost::numeric::odeint;

    // Memory allocation
    double ** Q_ode = (*data_tmp).Q;
    double ** P_ode = (*data_tmp).P_d;
    int          NN = (*data_tmp).NN;
    double delta_tk = (*data_tmp).delta_tk;

    // Definition and initialization of variables
    real_1d_array y2;
     double eps2 = 0.000000001;  // stopping criterion
     double h = 0;
```

```
 odesolverstate s3;
 real_1d_array xtbl;
 real_2d_array ytbl;
 odesolverreport rep;
double init_step_size = delta_tk * 0.1;
state_type x(2 * NN + 1);
double sum1 , sum2;
double precision_cut = 0.0;

// O.D.E. based computation of the last column of
// probability matrix P^X(t; theta)

// Solve Forward Kolmogorov Equation for N-th
// column, equation (1.44)
for(int k=2 * NN - 1; k>=0; k--)
{
   // If it is the last column of transition probability
   // matrix P^X(t; theta) solve the o.d.e.
   if(k==2 * NN - 1)
   {
      for(int i=0;i<=2*NN;i++)
      {
         x[i] = 0.0; // start at x=1.0, p=0.0
      }
      x[2 * NN] = 1.0;

      // Define system of o.d.e. as system
      Forward_Kolmogorov_Equation system(*data_tmp);

      // See Table 1.6. Stepper Algorithms in
      // https://www.boost.org/doc/libs/1_70_0/l
      // ibs/numeric/odeint/doc/html/boost_numeric_odeint/
      // odeint_in_detail/steppers.html

      // Solve Forward Kolmogorov Equation
      // "system" by Runge-Kutta 4  algorithm
      if (ode_stepper == 1)
      {
         runge_kutta4 < state_type > rk4;
         size_t steps = integrate_const(
            rk4, system, x, 0.0,
            delta_tk, ode_step_size);
      }

      // By Fehlberg 78 algorithm
```

```
    if (ode_stepper == 2) {
       runge_kutta_fehlberg78 < state_type > rk78;
       size_t steps = integrate_const(
          rk78, system, x, 0.0,
          delta_tk, ode_step_size);
    }

    // By Bulirsch-Stoer algorithm
    if (ode_stepper == 3)
    {
       bulirsch_stoer < state_type > rk4;
       size_t steps = integrate_const(
          rk4, system, x, 0.0,
          delta_tk, ode_step_size);
    }

    // Write solution of o.d.e. as (k+1)-column,
    // otherwise force the transition probability
    // to be zero (if it is lower then
    // numerical precision)
    for(int i=0; i<2*NN + 1; i++)
    {
       if ((x[i] >= precision_cut) && (x[i] <= 1)) {
             P_ode[i][k + 1] = x[i];
       }
       else { P_ode[i][k + 1] = 0; }
    }
}

// Solve Forward Kolmogorov Equation for each
// (*data_tmp).Reinit-th column in order to prevent
// catastrophic cancelation (accumulation of errors)
if((k!=0) && (k % (*data_tmp).Reinit == 0))
{

    for(int i=0;i<=2*NN;i++)
    {
       x[i] = 0.0; // start at x=1.0, p=0.0
    }
    x[k+1] = 1.0;

    Forward_Kolmogorov_Equation system(*data_tmp);

    if (ode_stepper == 1)
    {
```

```
    runge_kutta4 < state_type > rk4;
    size_t steps = integrate_const(
      rk4, system, x, 0.0,
      delta_tk, ode_step_size);
}

if (ode_stepper == 2) {
    runge_kutta_fehlberg78 < state_type > rkf78;
    size_t steps = integrate_const(
      rkf78, system, x, 0.0,
      delta_tk, ode_step_size);
}

for (int i = 0; i<2 * NN + 1; i++)
{
    if ((x[i] >= precision_cut) && (x[i] <= 1)) {
      P_ode[i][k + 1] = x[i];
    }
    else { P_ode[i][k + 1] = 0; }
}

for(int i=0;i<=2*NN;i++)
{
    x[i] = 0.0;
}
x[k+2] = 1.0;

if (ode_stepper == 1)
{
    runge_kutta4 < state_type > rk4;
    size_t steps = integrate_const(
      rk4, system, x, 0.0,
      delta_tk, ode_step_size);
}

if (ode_stepper == 2) {
    runge_kutta_fehlberg78 < state_type > rkf78;
    integrate_const(
      rkf78, system, x, 0.0,
      delta_tk, ode_step_size);
}

for (int i = 0; i<2 * NN + 1; i++)
{
    //P_ode[i][k + 1] = ytbl[1][i];
```

```
        if ((x[i] >= precision_cut) && (x[i] <= 1)) {
           P_ode[i][k + 2] = x[i]; //
        }
        else { P_ode[i][k + 2] = 0; }
     }

}


//  If it is not the last and not one of each
// (*data_tmp).Reinit columns calculated by
// o.d.e. solving, then calculate using
// iterative formula from Theorem 6
sum1 = 0, sum2 = 0;
for(int j=0; j<2*NN + 1; j++)
{
    // Neither the first nor the last element
    // of k-th column
    if((j!=0)&&(j!=2 * NN)){
       sum1 =
       (Q_ode[j][j-1]    /
       Q_ode[k][k+1]) *
       P_ode[j-1][k+1]+
       ((Q_ode[j][j] - Q_ode[k+1][k+1])/
       Q_ode[k][k+1]) *
       P_ode[j][k+1]+
        (Q_ode[j][j+1] /
        Q_ode[k][k+1]) *
        P_ode[j+1][k+1]; }

    // If the first element of k-th column
    if (j==0){ sum1 =
        ((Q_ode[j][j] - Q_ode[k+1][k+1])/
       Q_ode[k][k+1]) * P_ode[j][k+1]+
        (Q_ode[j][j+1] / Q_ode[k][k+1]) *
        P_ode[j+1][k+1];

    // If the last element of k-th column
    if (j==2 * NN){ sum1 =
        (Q_ode[j][j-1]    /Q_ode[k][k+1]) *
        P_ode[j-1][k+1] +
       ((Q_ode[j][j] - Q_ode[k+1][k+1])/
       Q_ode[k][k+1]) * P_ode[j][k+1];}

    // Computational trick excluding too
```

```
            // small numbers from any further computations
            if (k == 2 * NN - 1)
            {
               if ((sum1 >= precision_cut) && (sum1 <= 1)){
                  P_ode[j][k] = sum1; }
               else { P_ode[j][k] = 0; };
            }
            else {
                  sum2 = (Q_ode[k + 2][k + 1] /
                          Q_ode[k][k + 1]) *
                          P_ode[j][k + 2];
               if ( (sum1 - sum2 >= precision_cut) &&
                     (sum1 - sum2 <= 1)) {
                  P_ode[j][k] = sum1 - sum2; }
               else
               {
                  P_ode[j][k] = 0;
               };
            }
         }


      }

   }

// Computation of discrete likelihood defined by the Theorem 3
// from real data sample
double d_Likelihood(
   // Structure with initial data
   InputData * data_tmp,
   // model parameters
   double nu, double alpha_0, double alpha_1)
{
   // Initialize transition intensity matrix Q and
   // transition probability P by zeros
   for (int i = 0; i <= 2 * (*data_tmp).NN; i++) {
      for (int j = 0; j <= 2 * (*data_tmp).NN; j++) {
         (*data_tmp).Q[i][j]   = 0.0;
         (*data_tmp).P_d[i][j] = 0.0;
      } }

   // Calculate Q elements for given parameters nu, alpha_0, alpha_1
   // and the model with  (*data_tmp).NN agents. Store them in
   // matrix (*data_tmp).Q of data structure data_tmp
```

```
Matrix_Q( (*data_tmp).Q, nu, alpha_0, alpha_1, (*data_tmp).NN);

// Log matrix Q to matrix_Q.ini if output is 1
if ((output==1)) {
   ofstream d_sample("matrix_Q.ini");
   for (int i = 0; i < 2*(*data_tmp).NN + 1; i++){
      for (int j = 0; j < 2 * (*data_tmp).NN + 1; j++){
         d_sample << (*data_tmp).Q[i][j] << ", ";}
      d_sample << endl;
   }
   d_sample.close();
}

// If method field of data_tmp is not 3 (not lower Hessenberg
// matrix approach), then use eigen-decomposition,
// otherwise use recursive formula and o.d.e. solver
// from Section 1.6
if ((*data_tmp).method != 3) {
   Matrix_P_eigen_decomposition(
      (*data_tmp).delta_tk,
      (*data_tmp).P_d,
      (*data_tmp).Q,
      (*data_tmp).U_d,
      (*data_tmp).D_lambda_d,
      (*data_tmp).invU_d,
      (*data_tmp).NN,
      1); }
else{
   Matrix_P_recursive_ode(data_tmp); }

// Log matrix P to matrix_P.ini if output is 1
if ((output==1)) {
   ofstream d_sample("matrix_P.ini");
   for (int i = 0; i < 2 * (*data_tmp).NN + 1; i++) {
      for (int j = 0; j < 2 * (*data_tmp).NN + 1; j++) {
         d_sample << (*data_tmp).P_d[i][j] << ", ";
      }
      d_sample << endl;
   }
   d_sample.close();
}

// Computation of discrete likelihood defined by the Theorem 3
double sum = 0;
for(int i=0;i<= (*data_tmp).TT_d - 1;i++)
```

```
    {
        // Rounding real index value stored in y_ob[i]
        // to pick certain state
        int state1 =
            (*data_tmp).NN +
                (int)boost::math::round(
                    (*data_tmp).NN *
                    (*data_tmp).y_ob[i]);
        int state2 =
            (*data_tmp).NN +
                (int)boost::math::round(
                    (*data_tmp).NN *
                    (*data_tmp).y_ob[i + 1]);
        double value =  (*data_tmp).P_d[ state1 ][ state2 ];

        if( ( value>0 ) &&
            ( value<=1) ) {
            sum += log(value);
        }
    }

    // Print calculated likelihood value in order to observe convergence
    // of an optimization (maximization) procedure
    if (((output == 1) || (output == 2)) && (do_parallel != 1))
    {
        cout << "nu_d = " << nu << " alpha_0_d = " << alpha_0 <<
        " alpha_1_d = " << alpha_1
        << " Disc. Lik. = " << sum << endl;
    }


    /**likelihood_val =*/ //return sum/((1 + 2 * NN) * (1 + 2 * NN));
    return sum/( (*data_tmp).TT_d + 1);
}

using namespace alglib;

// Another version of continuous-time likelihood computation
// with other types of input data
void fcn_c_lik(const real_1d_array &x, double &func, void *ptr)
{
    InputData data_tmp = *((InputData *)ptr);
    func = -c_Likelihood(
        x[0], x[1], x[2], data_tmp.N, data_tmp.R, data_tmp.NN);
}
```

```
// Another version of discrete-time likelihood computation
// with other types of input data
void fcn_d_lik(const real_1d_array &x, double &func, void *ptr)
{
   InputData data_tmp = *((InputData *)ptr);
   func = -d_Likelihood( &data_tmp, x[0], x[1], x[2] );
}

// One more version of continuous-time likelihood computation
// with other types of input data
void fcn_d_lik2(const real_1d_array &x, real_1d_array &fi, void *ptr)
{
   InputData data_tmp = *((InputData *)ptr);
    fi[0] = -d_Likelihood( &data_tmp, x[0],x[1], x[2] );
}

// This function estimates input data stored in data_tmp and
// returning the optimal parameter values as nu_EM and etc
// Optimizer initial point is stored in xx vector
void EM_algorithm(
   InputData * data_tmp, int ipoint, double *nu_EM,
   double *alpha_0_EM,double *alpha_1_EM)
{
   // Memory Allocation
   InputData data_tmp2 = *data_tmp;

   double ** Ksi = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++) Ksi[i] = new double [2 * NN + 1];

   double ** D_lambda = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++)
      D_lambda[i] = new double [2 * NN + 1];

   double ** U = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++) U[i] = new double [2 * NN + 1];

   double ** invU = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++) invU[i] = new double [2 * NN + 1];

   double ** P = new double *[2 * NN + 1];
   for (int i = 0; i<2 * NN + 1; i++) P[i] = new double [2 * NN + 1];

   double * E_R = new double [2 * NN + 1];
```

```
double ** E_N = new double *[2 * NN + 1];
for (int i = 0; i<2 * NN + 1; i++) E_N[i] = new double [2 * NN + 1];

double ** Q = new double *[2 * NN + 1];
for (int i = 0; i<2 * NN + 1; i++) Q[i] = new double[2 * NN + 1];
for (int i = 0; i <= 2 * NN; i++) {
   for (int j = 0; j <= 2 * NN; j++) { Q[i][j] = 0; } }


// Initialization
double nu_m = 1.0, alpha_0_m = 0.0, alpha_1_m = 1.0;
double lv1 = d_Likelihood( data_tmp, nu_m, alpha_0_m, alpha_1_m );
double lv3 = 0;
int counter = 0;

// Estimation until abs(lv1 - lv3)>=0.0000001)&
// (iterations counter <=1000)
while((abs(lv1 - lv3)>=0.0000001)&&(counter<=1000))
{
   // Calculate necessary matrices
   Matrix_Q( Q, nu_m, alpha_0_m, alpha_1_m, NN);
   Matrix_P_eigen_decomposition(
      (*data_tmp).delta_tk,
      P,
      Q,
      U,
      D_lambda,
      invU,
      NN,
      1 );
   Matrix_Ksi( (*data_tmp).delta_tk, Ksi, D_lambda, NN);
   Matrix_E_R(
      (*data_tmp).delta_tk, E_R, (*data_tmp).c,
      P, Ksi, D_lambda, U, invU, NN );
   Matrix_E_N(
      (*data_tmp).delta_tk, E_N, (*data_tmp).c,
      Q, P, Ksi, D_lambda, U, invU, NN );

   // Assign initial point stored in xx vector by ipoint number
   real_1d_array x;
   x.setcontent(3, xx[ipoint]);

   // Initialization of optimizer
   minbleicstate state;
   minbleicreport rep;
```

```
    minbleiccreatef(          x, diffstep, state );
    // Set boundaries of search region
    minbleicsetbc(            state, bndl, bndu );
    // Define stopping criterions
    minbleicsetinnercond(     state, epsg, epsf, epsx );

    // Assign calculated expectations of matrix N and vector R
    data_tmp2.N = E_N;
    data_tmp2.R = E_R;

    // Begin optimization, namely maximization of
    // likelihood fcn_c_lik with  parameters and
    // input data stored in data_tmp2 structure
    alglib::minbleicoptimize(
       state, fcn_c_lik, NULL, (void*)&data_tmp2 );
    minbleicresults(          state, x, rep );

    // Write the result of optimization (vector x) to variables:
    nu_m = x[0], alpha_0_m = x[1], alpha_1_m = x[2];

    // Store current likelihood value
    lv1 = lv3;
    // Calculate discrete likelihood L^d(y; theta_i) for new
    // parameters nu_m,  alpha_0_m, alpha_1_m
    lv3 = d_Likelihood( data_tmp, nu_m, alpha_0_m, alpha_1_m );

    // Counter of iterations
    counter+=1;

}

// Save results
*nu_EM = nu_m;
*alpha_0_EM = alpha_0_m;
*alpha_1_EM = alpha_1_m;

// Clean Up
for (int i = 0; i<2 * NN + 1; i++) delete[](D_lambda[i]);
for (int i = 0; i<2 * NN + 1; i++) delete[](Ksi[i]);
for (int i = 0; i<2 * NN + 1; i++) delete[](U[i]);
for (int i = 0; i<2 * NN + 1; i++) delete[](invU[i]);
for (int i = 0; i<2 * NN + 1; i++) delete[](P[i]);
for (int i = 0; i<2 * NN + 1; i++) delete[](E_N[i]);
for (int i = 0; i<2 * NN + 1; i++) delete[](Q[i]);
delete[](E_R);
```

```
}

// Wrapper for MLE estimation subroutines
Results MLE_wrapper(InputData * data_tmp, int ipoint)
{
   // Assign initial point
   real_1d_array x;
   x.setcontent(3, xx[ipoint]);

   // Initialize optimizer
   minbleicstate state2;
   minbleicreport rep2;
   minbleiccreatef(3, x, diffstep, state2);
   minbleicsetscale(state2, scale);
   minbleicsetbc(state2, bndl, bndu);
   minbleicsetinnercond(state2, epsg, epsf, epsx);

   // If filed method of data_tmp structure is 0, then
   // maximize continuous-time likelihood fcn_c_lik
   // otherwise discrete fcn_d_lik
   if ((*data_tmp).method == 0) {
      alglib::minbleicoptimize(state2, fcn_c_lik, NULL, data_tmp);
   }
   else {
      alglib::minbleicoptimize(state2, fcn_d_lik, NULL, data_tmp);
   }

   minbleicresults(state2, x, rep2);

   // Return the results as a structure Res
   Results Res;
   Res.x[0] = x[0];
   Res.x[1] = x[1];
   Res.x[2] = x[2];

   return(Res);
}

// This function writing all the data in structure data_tmp
// in the text files
void output_fcn(InputData * data_tmp)
{
   ofstream c_sample("matrix_c_sample.ini");
   for (int i = 0; i <= (*data_tmp).MM_max; i++){
```

```
        c_sample << (*data_tmp).y[i] << ", "; }
    c_sample.close();

    ofstream rho("matrix_rho.ini");
    for (int i = 0; i <= (*data_tmp).MM_max; i++) {
       rho << (*data_tmp).rho[i] << ", ";}
    rho.close();

    ofstream R("matrix_R.ini");
    for (int i = 0; i < 2* (*data_tmp).NN+1; i++) {
        R << (*data_tmp).R[i] << ", ";}
    R.close();

    ofstream N("matrix_N.ini");
    for (int i = 0; i < 2 * (*data_tmp).NN + 1; i++) {
       for (int j = 0; j < 2 * (*data_tmp).NN + 1; j++) {
          N << (*data_tmp).N[i][j] << ", ";}
       N << endl;
    }
    N.close();

    ofstream d_sample("matrix_d_sample.ini");
    for (int i = 0; i <= (*data_tmp).TT_d; i++) {
       d_sample << (*data_tmp).y_ob[i] << ", ";
    }
    d_sample.close();

    ofstream c("matrix_c.ini");
    for (int i = 0; i < 2 * NN + 1; i++) {
       for (int j = 0; j < 2 * NN + 1; j++) {
          c << (*data_tmp).c[i][j] << ", ";
       }
       c << endl;
    }
    c.close();
}

// Simulate complete-data (continuous-time) sample path
int GenContinuousSample(
    int k, InputData * data_tmp, double nu_real,
    double alpha0_real, double alpha1_real)
{
    // Memory allocation
    double* R   = (*data_tmp).R;
    double** N  = (*data_tmp).N;
```

```
double* rho = (*data_tmp).rho;
double* y   = (*data_tmp).y;
double TT   = (*data_tmp).TT;
int MM      = (*data_tmp).MM;
int NN      = (*data_tmp).NN;
double * Ud  = new double [2*MM];
double * Ud2 = new double [MM];

// Initialization of R, N and rho with zeros
for(int j=0;j<=2*NN;j++) R[j]=0;
for(int i=0;i<=2*NN;i++) {
   for(int j=0;j<=2*NN;j++){
      N[i][j] = 0;
   }
}
for(int j=0;j<=MM-1;j++) rho[j]=0;

// Initialization of two independent uniform
// random number  generators rng1 and rng2
boost::random::mt19937 base_rng;
boost::random::uniform_01<> u01;

boost::random::mt19937 base_rng_2;
boost::random::uniform_01<> u01_2;

// Set different seeds from array PrimesSampleArr
// with prime numbers
base_rng.seed(   PrimesSampleArr[2 * (k - 1)]     );
base_rng_2.seed( PrimesSampleArr[2 * (k - 1) + 1] );

boost::random::variate_generator<boost::mt19937&,
   boost::uniform_01<> >
   rng(  base_rng, u01);
boost::random::variate_generator<boost::mt19937&,
   boost::uniform_01<> >
   rng2( base_rng_2, u01_2);

// Fill uniform random variables vectors Ud and Ud2
// with values generated by rng1 and rng2
for (int j = 0; j <= MM - 1; j++) {
   Ud[j]  = rng();
   //outfileRNG<<(int)randNum[j]<<"\n";
   Ud2[j] = rng2();
   if (Ud2[j] == 0) { Ud2[j] = rng2(); }
}
```

```
// Simulate continuous-time sample path as described in Section 1.2.5
double t = 0; int m = 0;
while(t<=TT)
{
   double p1=pi_u(
      (double)y[m],
      nu_real,
      alpha0_real,
      alpha1_real );
   // Generate new state
   if((0.0 <= Ud[m]) && (Ud[m] <= p1))
   {
      if( (int)boost::math::round(NN * y[m]) != NN )  {
         y[m+1]= y[m] + (double)1 / (double)NN;}
      else {
         y[m+1]= y[m] - (double)1 / (double)NN;}
   }
   if((p1 < Ud[m]) && (Ud[m] <= 1.0))
   {
      if( (int)boost::math::round(NN * y[m]) !=-NN )  {
         y[m+1]= y[m] - (double)1 / (double)NN;}
      else {
         y[m+1]= y[m] + (double)1 / (double)NN;}
   }

   // Generate interval between transitions
   rho[m+1] = -log(1-Ud2[m]) /
      ( w_u(
         (double)y[m],
         nu_real,
         alpha0_real,
         alpha1_real) +
        w_d(
           (double)y[m],
           nu_real,
           alpha0_real,
           alpha1_real) );
   t       += rho[m+1];
  m++;
}

// Cut the sample to have end point at TT
rho[m] = TT - t + rho[m];
y[m]   = y[ m - 1];
```

```
    (*data_tmp).MM_max = m;

    double qq = 0; double s = 0; int p = 0,q = 0;

    // Fill vector of holding time R and
    // matrix N with number of transitions
    for(int j=0;j<=m-2;j++)
    {
       p    = NN   + (int)boost::math::round( NN * y[j] );
       R[p] = R[p] + rho[j+1];

       q    = NN   + (int)boost::math::round( NN * y[j+1] );
       N[p][q] = N[p][q] + 1;
    }

    delete[](Ud );
    delete[](Ud2);

    return 0;
}

// This function return number of transitions in continuous-time
// sample until time t through vector of intervals between transitions rho
int y_k(double t,double* rho, int MM)
{
    double s = 0; int i = 0;
    while((s<t) && (i<=MM-1))
    {
       s = s + rho[i];
       i = i + 1;
    }
     return i-1;
}

// This function write discrete-time sample y_ob from continuous-time y
// to corresponding fields of the structure data_tmp with input data
int GenDiscreteSample( InputData * data_tmp)
{
    int MM = (*data_tmp).MM_max;
    double tk = 0;
    (*data_tmp).y_ob[0] = (*data_tmp).y[0];

    // Write state of the process y in discretization time stamps tk
    // to y_ob[]
    int t = 1;
```

```
    while(tk<= (*data_tmp).TT)
    {
        tk = tk + (*data_tmp).delta_tk;
        int k = y_k(tk, (*data_tmp).rho, MM);
        (*data_tmp).y_ob[t] = (*data_tmp).y[k];
        t++;
    }
    t--; t--;
    (*data_tmp).TT_d = t; // real vector size of y_ob

    // Initialize matrix c with zeros
    for(int i=0;i<=2 * (*data_tmp).NN;i++) {
        for(int j=0;j<=2 * (*data_tmp).NN;j++) {
            (*data_tmp).c[i][j]=(int)0; };}

    // Fill c with quantities of transitions from i to j in y_ob
    for(int i=1;i<=t;i++)
    {
        int i1 = NN +
            (int)boost::math::round((*data_tmp).NN *
            (*data_tmp).y_ob[i-1]);
        int i2 = NN +
            (int)boost::math::round((*data_tmp).NN *
            (*data_tmp).y_ob[i]);
        (*data_tmp).c[i1][i2] = (*data_tmp).c[i1][i2] + (int)1;
    }

    // Log all generated data from data_tmp to hard drive
    if (output==2) { output_fcn(data_tmp); }

    return(0);
}

// Parser of real data sample y_ob
int ParseRealSample(InputData * data_tmp, real_1d_array y_ob)
{
    // Assign number of observations to TT_d
    (*data_tmp).TT_d = y_ob.length() - 1;

    // Fill y_ob of structure data_tmp
    for (int i = 0; i <= (*data_tmp).TT_d; i++)
    {
        (*data_tmp).y_ob[i] = y_ob[i];
    }
```

```
    // Initialization of c with zeros
    for (int i = 0; i <= 2 * (*data_tmp).NN; i++) {
        for (int j = 0; j <= 2 * (*data_tmp).NN; j++) {
            (*data_tmp).c[i][j] = (int)0; }; }

    // Fill c with quantities of transitions from i to j in y_ob
    for (int i = 1; i <= (*data_tmp).TT_d; i++)
    {
        int i1 = NN +
            (int)boost::math::round((*data_tmp).NN *
            (*data_tmp).y_ob[i - 1]);
        int i2 = NN +
            (int)boost::math::round((*data_tmp).NN *
            (*data_tmp).y_ob[i]);
        (*data_tmp).c[i1][i2] = (*data_tmp).c[i1][i2] + (int)1;
    }

    // Log sample paths y_ob and matrix c to hard drive
    if (output==1) {
        ofstream d_sample("matrix_d_sample.ini");
        for (int i = 0; i <= (*data_tmp).TT_d; i++) {
            d_sample << (*data_tmp).y_ob[i] << ", ";
        }
        d_sample.close();

        ofstream c("matrix_c.ini");
        for (int i = 0; i < 2 * NN + 1; i++) {
            for (int j = 0; j < 2 * NN + 1; j++) {
                c << (*data_tmp).c[i][j] << ", ";
            }
            c << endl;
        }
    }

    return(0);
}

// This function calculates statistical metrics RMSE, FSSE, median, mean
// of estimated parameters. Note, this function used for fast accessing
// quality of estimates.
// The results presented in the main text were analyzed with R.
void StandardErrorsMed(
    double ** par, int IP, int size, double real_par_value,
    double *par_av, double *FSSE, double *RMSE, double *med)
{
```

```
   *par_av = 0; *RMSE = 0; *med = 0; *FSSE = 0;
   for(int kk=1;kk<=size;kk++)
   {
      *par_av += par[kk-1][IP];
      *RMSE +=(par[kk-1][IP] - real_par_value )*
         (par[kk-1][IP] - real_par_value);
   }
   *RMSE = sqrt(*RMSE/size);

   *par_av /=size;

   // Sample Error of Mean based on biased (but corrected) s
   // ample st. dev. or just sample st.dev.
   for(int kk=1;kk<=size;kk++)
   {
      *FSSE +=(par[kk-1][IP] - *par_av )*
         (par[kk-1][IP] - *par_av);
   }

   *FSSE = sqrt(*FSSE/(size-1));

   for(int i=0;i<= size - 1; i++)
   {
      for(int j=0;j<= size - 2; j++)
      {
         if(par[j][IP]<par[j+1][IP])
         {
            double temp = par[j][IP];
            par[j][IP] = par[j+1][IP];
            par[j+1][IP] = temp;
         }
      }
   }
   *med = par[size/(int)2][IP];

}

// This function calculates median
void Par_Median(double ** par, int IP, int iter)
{
   double par2[20];
   for(int i=0;i<=IP-1; i++) par2[i] = par[iter][i];
   for(int i=0;i<=IP-1; i++)
   {
      for(int j=0;j<= IP - 2; j++)
```

```
        {
            if(par2[j]<par2[j+1])
            {
                double temp = par2[j];
                par2[j] = par2[j+1];
                par2[j+1] = temp;
            }
        }
    }
    par[iter][IP] = (par2[IP/(int)2 - 1] + par2[IP/(int)2])/2;
}

// Parser of settings file with model and estimation algorithms parameters
int ParseSettings(ofstream &simulations, ifstream &settings)
{
    // Check existence of settings file
    try { settings.open("Settings.ini"); }
    catch (ios_base::failure e) {
        cout << "No Settings.ini file found! Exception
            opening/reading/closing file!\\\\n";
        getch();
        return 0;
    }

    // Read settings file line by line
    string line;
    while (getline(settings, line))
    {
        istringstream is_line(line);
        string key;
        if (getline(is_line, key, ' '))
        {
            string value;
            if (getline(is_line, value, '=')) {
                getline(is_line, value);
                // Estimation method (EM, MLE)
                if (key == "method") method =
                    (int)StrToFloat(value);
                // Parallel estimation for each initial point
                if (key == "do_parallel") do_parallel =
                    (int)StrToFloat(value);

                // Number of agents
                if (key == "N")
                    NN = (int)StrToFloat(value);
```

```
// Sample size
if (key == "M")
  MM = (int)StrToFloat(value);
// Time horizon
if (key == "T")
  TT = StrToFloat(value);
// Switcher of reinitialization described
// in Section 1.6.3
if (key == "Reinit")
  Reinit = (int)StrToFloat(value);
// Discretization step
if (key == "delta_tk")
  delta_tk = StrToFloat(value);

// Parameters values for simulation
if (key == "nu_real")
  nu_real     = StrToFloat(value);
if (key == "alpha0_real")
  alpha0_real = StrToFloat(value);
if (key == "alpha1_real")
  alpha1_real = StrToFloat(value);

// Discrete-time sample
if (key == "y_real") {
  y_ob        = value.c_str(); }

// Initial point, boundaries and scale
// of search region
if (key == "xx")    {
  xx    = value.c_str(); }
if (key == "ub")    {
  bndu  = value.c_str(); }
if (key == "lb")    {
  bndl  = value.c_str(); }
if (key == "scale") {
  scale = value.c_str(); }

// Stopping criterions
if (key == "epsg")
  epsg = StrToFloat(value);
if (key == "epsf")
  epsf = StrToFloat(value);
if (key == "epsx")
  epsx = StrToFloat(value);
```

```
            // Optimizer step size
            if (key == "diffstep")
               diffstep = StrToFloat(value);

            // O.D.E. stepper size of step
            if (key == "ode_step_size")
               ode_step_size = StrToFloat(value);
            // O.D.E. stepper algorithm (rk4 and etc)
            if (key == "ode_stepper")
               ode_stepper = (int)StrToFloat(value);

            // Logging on/off
            if (key == "output")
               output = (int)StrToFloat(value);
         }
      }
   }
   settings.close();


   try { settings.open("Settings.ini"); }
   catch (ios_base::failure e) {
      cout << "No Real Data file found! Exception
      of opening/reading/closing file!\\\\n";
      getch();
      return 0;
   }

   // Write the settings to the file  simulations
   // with the estimations results
   while (getline(settings, line))
   {
      simulations << line << endl;
   }
   settings.close();

   return 1;
}

// Write the array of likelihood function values in discrete points
// on 2D rectangle  for plotting
void PlotLik(InputData data_tmp)
{
   double xv = 0.01;
   double yv = 0.01;
```

```
double dx = 1.6;
double dy = 0.02;

double xlim = 120;
double ylim = 2;

ofstream plot3D("plot3D.ini");

while (xv <= xlim)
{
   yv = 0.01;
   while (yv <= ylim)
   {
   // if (do_parallel) {
   //concurrency::parallel_for(0, 200,
   //        [&](int k)
   //    {
      double val = d_Likelihood(&data_tmp, xv, 0, yv);
      plot3D << val << ", ";
      yv = yv + dy;
   }
   //);
   plot3D << endl;
   xv = xv + dx;
   cout << xv << endl;
}
plot3D << endl << endl;


try { settings.open("Settings.ini"); }
catch (ios_base::failure e) {
   cout << "No Settings.ini file found! Exception
   opening/reading/closing file!\\\\n";
   getch();
// return 0;
}

string line;
while (getline(settings, line))
{
   plot3D << line << endl;
}
settings.close();
```

```
    plot3D.close();

}

// ==============Main Function==============
int
main(int argc, char * argv[])
{

    // Enter  range of iterations (seeds) from the keyboard
    cout << "MC iterations from #:                    ";
    cin >> MC_iterations1;
    cout << "MC iterations to   #:                    ";
    cin >> MC_iterations2;

     time_t t = time(0);    // get time now
    struct tm * now = localtime(&t);

    string tt3 = IntToStr(MC_iterations1);
    string tt4 = IntToStr(MC_iterations2);

    // Create file with estimation experiment
    // results in folder Simulations
    // with a name containing Date and Time
    string tt = "Simulations\\Seed=(" + tt3 + "," + tt4 + ")_"
       + IntToStr(now->tm_year + 1900) + '-'
       + IntToStr(now->tm_mon + 1) + '-'
       + IntToStr(now->tm_mday) + " "
       + IntToStr(now->tm_hour) + "-"
       + IntToStr(now->tm_min) + "-"
       + IntToStr(now->tm_sec) + ".txt";

    ofstream simulations(tt);

    // Parse Settings.ini
    ParseSettings(simulations, settings);

    IP = xx.rows(); // Number of initial points

    //=======Mem Alloc for Errors Metrics==========
    double FSSE_nu_c = 0,  FSSE_alpha_0_c  = 0, FSSE_alpha_1_c  = 0;
    double FSSE_nu_d = 0,  FSSE_alpha_0_d  = 0, FSSE_alpha_1_d  = 0;
    double FSSE_nu_EM = 0, FSSE_alpha_0_EM = 0, FSSE_alpha_1_EM = 0;

    double RMSE_nu_c = 0,  RMSE_alpha_0_c  = 0, RMSE_alpha_1_c  = 0;
```

```
double RMSE_nu_d = 0,  RMSE_alpha_0_d  = 0, RMSE_alpha_1_d  = 0;
double RMSE_nu_EM = 0, RMSE_alpha_0_EM = 0, RMSE_alpha_1_EM = 0;

double med_nu_c = 0,  med_alpha_0_c  = 0, med_alpha_1_c  = 0;
double med_nu_d = 0,  med_alpha_0_d  = 0, med_alpha_1_d  = 0;
double med_nu_EM = 0, med_alpha_0_EM = 0, med_alpha_1_EM = 0;

// Number of iterations
int       itRange = MC_iterations2 - MC_iterations1 + 1;

double ** nu_c = new double *[itRange];
for (int i = 0; i<itRange; i++) nu_c[i] = new double [IP];
double ** nu_d = new double *[itRange];
for (int i = 0; i<itRange; i++) nu_d[i] = new double [IP];
double ** nu_EM = new double *[itRange];
for (int i = 0; i<itRange; i++) nu_EM[i] = new double [IP];

double ** alpha_0_c = new double *[itRange];
for (int i = 0; i<itRange; i++) alpha_0_c[i] = new double [IP];
double ** alpha_0_d = new double *[itRange];
for (int i = 0; i<itRange; i++) alpha_0_d[i] = new double [IP];
double ** alpha_0_EM = new double *[itRange];
for (int i = 0; i<itRange; i++) alpha_0_EM[i] = new double [IP];

double ** alpha_1_c = new double *[itRange];
for (int i = 0; i<itRange; i++) alpha_1_c[i] = new double [IP];
double ** alpha_1_d = new double *[itRange];
for (int i = 0; i<itRange; i++) alpha_1_d[i] = new double [IP];
double ** alpha_1_EM = new double *[itRange];
for (int i = 0; i<itRange; i++) alpha_1_EM[i] = new double [IP];

// Timers' arrays
double ** timerLc = new double *[itRange];
for (int i = 0; i<itRange; i++) timerLc[i] = new double[IP];
for (int i = 0; i < itRange; i++)
   for (int j = 0; j < IP; j++)  timerLc[i][j] = (_int64)0;


double ** timer = new double *[itRange];
for (int i = 0; i<itRange; i++) timer[i] = new double[IP];
for (int i = 0; i < itRange; i++)
   for (int j = 0; j < IP; j++)  timer[i][j] = (_int64)0;

// Seed Initialization
ifstream PrimesSample;
```

```
// Read file with primes numbers (seeds)
try { PrimesSample.open("PrimesSample.txt"); }
catch (ios_base::failure e) {
   cout << "No Seed Data file found! Exception
   opening/reading/closing file!\\\\n";
   getch();
   return 0;
}
char* buffer = new char[1024];
for (int i = 0; i < 200; i++) { PrimesSample >> PrimesSampleArr[i]; }


double nu_c_av = 0,  alpha_0_c_av = 0,  alpha_1_c_av = 0;
double nu_d_av = 0,  alpha_0_d_av = 0,  alpha_1_d_av = 0;
double nu_EM_av = 0, alpha_0_EM_av = 0, alpha_1_EM_av = 0;

counter = 0;

// Allocate k x 3 array of estimates
int k = xx.rows();
double ** estimates = new double *[k];
for (int i = 0; i<k; i++) estimates[i] = new double[3];

// ===== Real-data estimation =====
// If real parameters values are unknown (real data)
if ((nu_real == 0.0) &&
   (alpha0_real == 0.0)
   && (alpha1_real == 0.0)) {
   if ((y_ob.length() > 0))
   {
      // Progress bar filed with | for each initial point
      cout << endl;
      for (k = 1; k <= (int)xx.rows(); k++) {
         cout << "|"; }
      cout << endl;

      if (do_parallel) {
         // Estimate data sample starting with each
         // initial point separately in parallel
         concurrency::parallel_for(0, k - 1,
            [&](int ipoint)
            {
            // Fill real_data and real_data_cl
            // structures with settings
```

```
// from settings file
InputData real_data;
real_data.method = method;
real_data.NN = NN;
real_data.MM = MM;
real_data.Reinit = Reinit;
real_data.delta_tk = delta_tk;
real_data.k = 0;

InputDataClass real_data_cl(
   &real_data);

// Parse real sample and
// write to real_data
ParseRealSample(&real_data, y_ob);

// Log likelihood function surface
if ((output == 3) && (ipoint == 0)) {
   PlotLik(real_data); }

// structure with results
Results Res;

// If method is not 1 (EM) do MLE
// otherwise EM algorithm
if (method != 1) {
   Res =
      MLE_wrapper(
         &real_data,
         ipoint); }
else {
   double temp_nu_EM = 0,
   temp_alpha_0_EM = 0,
   temp_alpha_1_EM = 0,
   lv3 = 0;

   EM_algorithm(
      &real_data, ipoint,
      &temp_nu_EM,
      &temp_alpha_0_EM,
      &temp_alpha_1_EM);
   Res.x[0] = temp_nu_EM;
   Res.x[1] = temp_alpha_0_EM;
   Res.x[2] = temp_alpha_1_EM;
}
```

```
        // Write the results of
        // estimates to array
        estimates[ipoint][0] = Res.x[0];
        estimates[ipoint][1] = Res.x[1];
        estimates[ipoint][2] = Res.x[2];

        // Add to progress bar |
        cout << "|";
    }
    );
}
// If not in parallel
else {
    for (
        int ipoint = 0;
        ipoint < xx.rows();
        ipoint++)
    {
        // Fill real_data and real_data_cl
        // structure with settings
        // from settings file
        InputData real_data;
        real_data.method = method;
        real_data.NN = NN;
        real_data.MM = MM;
        real_data.Reinit = Reinit;
        real_data.delta_tk = delta_tk;
        real_data.k = 0;

        InputDataClass real_data_cl(
            &real_data);

        // Parse real sample and
        // write to real_data
        ParseRealSample(&real_data, y_ob);

        // Log likelihood function surface
        if ((output == 3) && (ipoint == 0)) {
            PlotLik(real_data); }

        // structure with results
        Results Res;

        // If method is not 1 (EM) do MLE
```

```
            // otherwise EM algorithm
            if (method != 1) {
               Res =
               MLE_wrapper(
                  &real_data,
                  ipoint); }
            else {
               double temp_nu_EM = 0,
               temp_alpha_0_EM = 0,
               temp_alpha_1_EM = 0, lv3 = 0;

               EM_algorithm(
                  &real_data, ipoint,
                  &temp_nu_EM,
                  &temp_alpha_0_EM,
                  &temp_alpha_1_EM);
               Res.x[0] = temp_nu_EM;
               Res.x[1] = temp_alpha_0_EM;
               Res.x[2] = temp_alpha_1_EM;
            }

            estimates[ipoint][0] = Res.x[0];
            estimates[ipoint][1] = Res.x[1];
            estimates[ipoint][2] = Res.x[2];

            cout << "|";
         }
      }

      // Print out and log the estimates
      cout << endl;
      for (int ipoint = 0; ipoint < xx.rows(); ipoint++)
      {
         cout << "nu_real = " <<
         estimates[ipoint][0] <<
         " alpha_0_real = " <<
         estimates[ipoint][1] <<
         " alpha_1_real = " <<
         estimates[ipoint][2] << endl;
         simulations << "0" << "," <<
         estimates[ipoint][0] <<
          "," << estimates[ipoint][1] << "," <<
          estimates[ipoint][2] << "," << "," << endl;
      }
   }
```

```
    nu_real      = estimates[0][0];
    alpha0_real = estimates[0][1];
    alpha1_real = estimates[0][2];
}

// Progress bar
cout << endl;
for (k = MC_iterations1; k <= MC_iterations2; k++)
{
    cout << "|";
}
cout << endl;

// Conduct Monte Carlo simulation experiment with
// MC_iterations2 - MC_iterations2 + 1 replications
concurrency::parallel_for(
    MC_iterations1, MC_iterations2 + 1, [&](int k)
{

    // Fill simulated_data and sim_data_cl structure
    // with settings from settings file
    InputData simulated_data;

    simulated_data.method = method;
    simulated_data.NN = NN;
    simulated_data.MM = MM;
    simulated_data.TT = TT;
    simulated_data.Reinit = Reinit;
    simulated_data.delta_tk = delta_tk; // TT/MM
    simulated_data.k = k;

    InputDataClass sim_data_cl(&simulated_data);


    //===== Artificial Data Simulation =====
    // Continuous-time sample
    GenContinuousSample(
        k,
        &simulated_data,
        nu_real,
        alpha0_real,
        alpha1_real);
    // Discretization
    GenDiscreteSample(&simulated_data);
```

```
    //=== Complete-data case estimation (MLE) ===
    if (method == 0){
       for (int ipoint = 0; ipoint < xx.rows(); ipoint++)
       {
          _int64 timeLc = (_int64)0;
          StartTimer(&timeLc);

          Results Res = MLE_wrapper(
             &simulated_data, ipoint);

          nu_c[k - MC_iterations1][ipoint]      =
             Res.x[0];
          alpha_0_c[k - MC_iterations1][ipoint] =
             Res.x[1];
          alpha_1_c[k - MC_iterations1][ipoint] =
             Res.x[2];

          timerLc[k - MC_iterations1][ipoint] =
             StopTimer(timeLc);
       }

}

    //===== Incomplete-data case estimation =====
    //== EM-algorithm ==
    if (method == 1) {
       for (int ipoint = 0; ipoint < xx.rows(); ipoint++)
       {
          // Initial point and
          // other parameters initialization
          real_1d_array x;
          x.setcontent(3, xx[ipoint]);

          _int64 time = (_int64)0;
          StartTimer(&time);

          double temp_nu_EM = 0,
          temp_alpha_0_EM = 0,
          temp_alpha_1_EM = 0,
          lv3 = 0;

          // Estimation
```

```
            EM_algorithm(
                &simulated_data, ipoint,
                &temp_nu_EM,
                &temp_alpha_0_EM,
                &temp_alpha_1_EM);

            // Write the results and timing in arrays
            timer[      k - MC_iterations1][ipoint] =
                StopTimer(time);
            nu_EM[      k - MC_iterations1][ipoint] =
                temp_nu_EM;
            alpha_0_EM[k - MC_iterations1][ipoint] =
                temp_alpha_0_EM;
            alpha_1_EM[k - MC_iterations1][ipoint] =
                temp_alpha_1_EM;
        }

    }

    //== MLE ==
    if ((method == 2) || (method == 3)) {

        for (int ipoint = 0; ipoint < xx.rows(); ipoint++)
        {
            // Timer start
            _int64 time = (_int64)0;
            StartTimer(&time);

            Results Res = MLE_wrapper(
                &simulated_data, ipoint);

            // Write results and timing in arrays
            timer[     k - MC_iterations1][ipoint] =
                StopTimer(time);
            nu_d[      k - MC_iterations1][ipoint] =
                Res.x[0];
            alpha_0_d[k - MC_iterations1][ipoint] =
                Res.x[1];
            alpha_1_d[k - MC_iterations1][ipoint] =
                Res.x[2];

        }

    }
```

```
    cout << "|";
}
);

// Print out the results of estimation
cout<<endl<< endl <<"NN = "<<NN<<" nu = "<<
nu_real<<" alpha_0 = "<<alpha0_real<<" alpha_1 = "<<
alpha1_real<< endl;

// Statistical metrics of estimation experiment
for (int ipoint = 0; ipoint < xx.rows(); ipoint++)
{
    // Write initial point to log file simulations
    simulations << endl << endl << "IP = [" <<
    xx[ipoint][0] << ", " << xx[ipoint][1] << ", " <<
    xx[ipoint][2] << "]" << endl << endl;

    // Log the results of continuous-time sample estimation
    if (method == 0) {

        for (k = MC_iterations1; k <= MC_iterations2; k++)
        {
            simulations << k << "," <<
            nu_c[k - MC_iterations1][ipoint] <<
             "," <<
             alpha_0_c[k - MC_iterations1][ipoint] <<
             "," <<
             alpha_1_c[k - MC_iterations1][ipoint] <<
             "," <<
             method << "," <<
              timerLc[k - MC_iterations1][ipoint] <<
              endl;
        }
        simulations << endl;

        // Calculate error metrics
        StandardErrorsMed(
            nu_c, ipoint, itRange, nu_real,
            &nu_c_av, &FSSE_nu_c, &RMSE_nu_c, &med_nu_c);
        StandardErrorsMed(
            alpha_0_c, ipoint, itRange, alpha0_real,
            &alpha_0_c_av, &FSSE_alpha_0_c,
            &RMSE_alpha_0_c, &med_alpha_0_c);
        StandardErrorsMed(
            alpha_1_c, ipoint, itRange, alpha1_real,
```

```
        &alpha_1_c_av, &FSSE_alpha_1_c,
        &RMSE_alpha_1_c, &med_alpha_1_c);

    // Print error metrics on screen
    simulations << "nu_c  = " <<
    nu_c_av << " alpha_0_c  = " <<
     alpha_0_c_av << " alpha_1_c  = " << alpha_1_c_av <<
      " Cont. Lik. = " << endl;
    simulations << "FSSE_nu_c    = " << FSSE_nu_c <<
    " FSSE_alpha_0_c    = " << FSSE_alpha_0_c <<
    " FSSE_alpha_1_c    = " << FSSE_alpha_1_c << endl;
    simulations << "RMSE_nu_c    = " << RMSE_nu_c <<
    " RMSE_alpha_0_c    = " << RMSE_alpha_0_c <<
    " RMSE_alpha_1_c    = " << RMSE_alpha_1_c << endl;
    simulations << "Median_nu_c  = " << med_nu_c <<
    " Median_alpha_0_c  = " << med_alpha_0_c <<
    " Median_alpha_1_c  = " <<
    med_alpha_1_c << endl << endl;
}

// Log the results of discrete-time sample estimation by EM
if (method == 1) {
    for (k = MC_iterations1; k <= MC_iterations2; k++)
    {
        simulations << k << "," <<
        nu_EM[k - MC_iterations1][ipoint] <<
        ","<<
        alpha_0_EM[k - MC_iterations1][ipoint] <<
        ","<<
        alpha_1_EM[k - MC_iterations1][ipoint] <<
        "," << method << "," <<
        PrimesSampleArr[2 * (k - 1)] <<
        "," << PrimesSampleArr[2 * (k - 1) + 1] <<
        "," << timer[k - MC_iterations1][ipoint] <<
        endl;
    }

    // Calculate error metrics
    StandardErrorsMed(
        nu_EM,      ipoint, itRange, nu_real,
        &nu_EM_av, &FSSE_nu_EM,
        &RMSE_nu_EM, &med_nu_EM);
    StandardErrorsMed(
        alpha_0_EM, ipoint, itRange, alpha0_real,
        &alpha_0_EM_av, &FSSE_alpha_0_EM,
```

```
        &RMSE_alpha_0_EM, &med_alpha_0_EM);
    StandardErrorsMed(
        alpha_1_EM, ipoint, itRange, alpha1_real,
        &alpha_1_EM_av, &FSSE_alpha_1_EM,
        &RMSE_alpha_1_EM, &med_alpha_1_EM);

    // Print error metrics on screen
    simulations << endl << "nu_EM = " << nu_EM_av <<
    " alpha_0_EM = " << alpha_0_EM_av <<
    " alpha_1_EM = " <<
    alpha_1_EM_av << " Disc. Lik. = 0" << endl;
    simulations << "SD_nu_EM  = " << FSSE_nu_EM <<
    " SD_alpha_0_EM  = " << FSSE_alpha_0_EM <<
    " SD_alpha_1_EM  = " << FSSE_alpha_1_EM << endl;
    simulations << "RMSE_nu_EM  = " << RMSE_nu_EM <<
    " RMSE_alpha_0_EM  = " << RMSE_alpha_0_EM <<
    " RMSE_alpha_1_EM  = " << RMSE_alpha_1_EM << endl;
    simulations << "Median_nu_EM  = " << med_nu_EM <<
     " Median_alpha_0_EM  = " << med_alpha_0_EM <<
      " Median_alpha_1_EM  = " << med_alpha_1_EM << endl;
}

// Log the results of discrete-time sample estimation by MLE
if ((method == 2) || (method == 3)) {
    for (k = MC_iterations1; k <= MC_iterations2; k++)
    {
        simulations << k << "," <<
        nu_d[k - MC_iterations1][ipoint] << ","
        << alpha_0_d[k - MC_iterations1][ipoint] <<
        "," <<
         alpha_1_d[k - MC_iterations1][ipoint] <<
         "," << method <<
          "," << PrimesSampleArr[2 * (k - 1)] <<
          "," <<
           PrimesSampleArr[2 * (k - 1) + 1] <<
           "," <<
            timer[k - MC_iterations1][ipoint] <<
           endl;
    }
    simulations << endl;

    // Calculate error metrics
    StandardErrorsMed(
        nu_d,       ipoint, itRange, nu_real,
        &nu_d_av, &FSSE_nu_d, &RMSE_nu_d, &med_nu_d);
```

```
        StandardErrorsMed(
            alpha_0_d, ipoint, itRange,
            alpha0_real,
            &alpha_0_d_av,
            &FSSE_alpha_0_d,
            &RMSE_alpha_0_d,
            &med_alpha_0_d);
        StandardErrorsMed(
            alpha_1_d, ipoint, itRange, alpha1_real,
            &alpha_1_d_av,
            &FSSE_alpha_1_d,
            &RMSE_alpha_1_d,
            &med_alpha_1_d);

        // Print error metrics on screen
        simulations << "nu_d  = " << nu_d_av <<
        " alpha_0_d  = " <<
         alpha_0_d_av << " alpha_1_d  = " << alpha_1_d_av <<
         " Disc. Lik. = " << endl;
        simulations << "SD_nu_d  = " << FSSE_nu_d <<
        " SD_alpha_0_d  = " << FSSE_alpha_0_d <<
         " SD_alpha_1_d  = " << FSSE_alpha_1_d << endl;
        simulations << "RMSE_nu_d  = " << RMSE_nu_d <<
         " RMSE_alpha_0_d  = " << RMSE_alpha_0_d <<
          " RMSE_alpha_1_d  = " << RMSE_alpha_1_d << endl;
        simulations << "Median_nu_d  = " << med_nu_d <<
         " Median_alpha_0_d  = " << med_alpha_0_d <<
         " Median_alpha_1_d  = " << med_alpha_1_d <<
         endl << endl;

    }
}

simulations.close();
settings.close();

// Clean up
for (int i = 0; i < itRange; i++) delete[] nu_c[i];
for (int i = 0; i < itRange; i++) delete[] nu_d[i];
for (int i = 0; i < itRange; i++) delete[] nu_EM[i];
for (int i = 0; i < itRange; i++) delete[] alpha_0_c[i];
for (int i = 0; i < itRange; i++) delete[] alpha_0_d[i];
for (int i = 0; i < itRange; i++) delete[] alpha_0_EM[i];
for (int i = 0; i < itRange; i++) delete[] alpha_1_c[i];
for (int i = 0; i < itRange; i++) delete[] alpha_1_d[i];
```

```
    for (int i = 0; i < itRange; i++) delete[] alpha_1_EM[i];
    for (int i = 0; i < itRange; i++) delete[] timer[i];
    for (int i = 0; i < itRange; i++) delete[] timerLc[i];


    getch();

     return SDK_SUCCESS;
}
```

C.2. AMSM model

The code for the AMSM model consists of three file: the header file with all the definitions, the main file with the code of subroutines and the file with the code of OpenCL kernel running on GPU.

The code of the OpenCL kernel for the Monte Carlo simulation of AMSM sample paths is based on toy example code of the Black-Scholes option pricing based on the Monte Carlo method provided in AMD APP SDK 2.7[1]. The code below adopts a few denotations, the uniform random number generator function *generateRand* and two accessory functions *lshift128* and *rshift128* from the AMD APP SDK. Also, this code uses the subroutine of computation of the inverse of cumulative distribution function of a normal random variable invented by Boris Moro [73] and the subroutine with the same purpose from Numerical Recipes [42]. The main function of the kernel is *calPriceVega*. In this function new values of 8 sample paths are generated iteratively and simultaneously. Another crucial function is *calSigma* which is dedicated to calculation of a new volatility state. In the function *calOutputs* are calculated payoff values. The other functions in this file are supplementary and aimed to compute the next pseudo-random or quasi-random number, or to transform them to Gaussian numbers.

C.2.1. OpenCL kernel for parallel computations

```
// 0 - MSM
// 1 - AMSM1
// 2 - AMSM2

// The kernel gets external parameters as
// data structure MonteCarloAttrib
 typedef struct _MonteCalroAttrib
{
    float4 strikePrice;
    // Switcher between QRNG and PRNG
    int4   RNG;

    // AMSM model parameters
```

[1] AMD APP SDK is outdated on 2019 and substituted with GPUOpen initiative, see gpuopen.com.

```
    float4 lambda;
    float4 nu;
     float4 b;
     float4 m0;
     float4 rho;
     int4    k;
    float4 gkk;

     // Contract parameters
     float4 interest;
     float4 initPrice;
     float4 sigma0;
     int4    model;
} MonteCarloAttrib;


/**
 * @brief Left shift
 * @param input input to be shifted
 * @param shift shifting count
 * @param output result after shifting input
 */
void
lshift128(uint4 input, uint shift, uint4* output)
{
    unsigned int invshift = 32u - shift;

    uint4 temp;
    temp.x = input.x << shift;
    temp.y = (input.y << shift) | (input.x >> invshift);
    temp.z = (input.z << shift) | (input.y >> invshift);
    temp.w = (input.w << shift) | (input.z >> invshift);

    *output = temp;
}

/**
 * @brief Right shift
 * @param input input to be shifted
 * @param shift shifting count
 * @param output result after shifting input
 */

void
rshift128(uint4 input, uint shift, uint4* output)
```

```
{
    unsigned int invshift = 32u - shift;

    uint4 temp;

    temp.w = input.w >> shift;
    temp.z = (input.z >> shift) | (input.w << invshift);
    temp.y = (input.y >> shift) | (input.z << invshift);
    temp.x = (input.x >> shift) | (input.y << invshift);

    *output = temp;
}

/**
 * @brief Generates gaussian random numbers by using
 *        Mersenne Twister algo and box muller transformation
 * @param seedArray  seed
 * @param gaussianRand1 gaussian random number generated
 * @param gaussianRand2 gaussian random number generated
 * @param nextRand  generated seed for next usage
 */
void generateRand(uint4 seed,
                  float4 *UniformRand1,
                  float4 *UniformRand2,
                  uint4 *nextRand)
{

    uint mulFactor = 4;
    uint4 temp[8];

    uint4 state1 = seed;
    uint4 state2 = (uint4)(0);
    uint4 state3 = (uint4)(0);
    uint4 state4 = (uint4)(0);
    uint4 state5 = (uint4)(0);

    uint stateMask = 1812433253u;
    uint thirty = 30u;
    uint4 mask4 = (uint4)(stateMask);
    uint4 thirty4 = (uint4)(thirty);
    uint4 one4 = (uint4)(1u);
    uint4 two4 = (uint4)(2u);
    uint4 three4 = (uint4)(3u);
    uint4 four4 = (uint4)(4u);
```

```
uint4 r1 = (uint4)(0);
uint4 r2 = (uint4)(0);

uint4 a = (uint4)(0);
uint4 b = (uint4)(0);

uint4 e = (uint4)(0);
uint4 f = (uint4)(0);

unsigned int thirteen  = 13u;
unsigned int fifteen = 15u;
unsigned int shift = 8u * 3u;

unsigned int mask11 = 0xfdff37ffu;
unsigned int mask12 = 0xef7f3f7du;
unsigned int mask13 = 0xff777b7du;
unsigned int mask14 = 0x7ff7fb2fu;


const float one = 1.f;
const float intMax = 4294967296.f;


// float4 temp1;
// float4 temp2;

//Initializing states.
state2 = mask4 * (state1 ^ (state1 >> thirty4)) + one4;
state3 = mask4 * (state2 ^ (state2 >> thirty4)) + two4;
state4 = mask4 * (state3 ^ (state3 >> thirty4)) + three4;
state5 = mask4 * (state4 ^ (state4 >> thirty4)) + four4;

uint i = 0;
for(i = 0; i < mulFactor; ++i)
{
    switch(i)
    {
        case 0:
            r1 = state4;
            r2 = state5;
            a = state1;
            b = state3;
            break;
        case 1:
            r1 = r2;
```

```
                r2 = temp[0];
                a = state2;
                b = state4;
                break;
            case 2:
                r1 = r2;
                r2 = temp[1];
                a = state3;
                b = state5;
                break;
            case 3:
                r1 = r2;
                r2 = temp[2];
                a = state4;
                b = state1;
                break;
            default:
                break;

        }

        lshift128(a, shift, &e);
        rshift128(r1, shift, &f);

        uint4 temp2;

        temp2.x = a.x ^ e.x ^ ((b.x >> thirteen) & mask11) ^
            f.x ^ (r2.x << fifteen);
        temp2.y = a.y ^ e.y ^ ((b.y >> thirteen) & mask12) ^
            f.y ^ (r2.y << fifteen);
        temp2.z = a.z ^ e.z ^ ((b.z >> thirteen) & mask13) ^
            f.z ^ (r2.z << fifteen);
        temp2.w = a.w ^ e.w ^ ((b.w >> thirteen) & mask14) ^
            f.w ^ (r2.w << fifteen);

        temp[i] = temp2;

    }

    *UniformRand1 = convert_float4(temp[0]) * one / intMax;
    *UniformRand2 = convert_float4(temp[1]) * one / intMax;
    *nextRand = temp[2]; //*****************

}
```

```
/* This function returns the inverse of cumulative
normal distribution function.
Reference: The Full Monte, by Boris Moro,
Union Bank of Switzerland. RISK 1995(2) */
float cndev(float u)
{
   float a[4]={
        2.50662823884f,
        -18.61500062529f,
        41.39119773534f,
        -25.44106049637f};
   float b[4]={ -8.47351093090f,
        23.08336743743f,
        -21.06224101826f,
        3.13082909833f};
   float c[9]={
        0.3374754822726147f,
        0.9761690190917186f,
        0.1607979714918209f,
        0.0276438810333863f,
        0.0038405729373609f,
        0.0003951896511919f,
        0.0000321767881768f,
        0.0000002888167364f,
        0.0000003960315187f};
   float x, r;
   x     = u-0.5f;
   if (fabs(x)<0.42f)
   {
      r = x*x;
      r = x*(((a[3]*r+a[2])*r+a[1])*r+a[0])/
            ((((b[3]*r+b[2])*r+b[1])*r+b[0])*r+1.f);
      return(r);
   }

   r = u;
   if(x>0.f) r=1.f-u;
   r = native_log(-native_log(r));
   r = c[0]+r*(c[1]+r*(c[2]+r*(c[3]+r*(c[4]+
        r*(c[5]+r*(c[6]+r*(c[7]+r*c[8])))))));
   if(x<0.f) r=-r;
   return(r);

}
```

```
void generateRandGaussian(
    uint4 seed,
    float4 *gaussianRand1,
    float4 *gaussianRand2,
    uint4 *nextRand)
{

     uint4 tempRand = seed;

     float4 temp1 = (float4)0.0f;
     float4 temp2 = (float4)0.0f;
     //float4 temp3 = (float4)0.0f;
     //float4 temp4 = (float4)0.0f;


     float4 r;
     float4 s;
     float4 var_phi;
     const float PI = 3.14159265358979f;
     const float two = 2.0f;


     generateRand(tempRand, &temp1, &temp2, nextRand);

     // ======Applying Box Mullar Transformations=====
     r = sqrt( (-two) * log(temp1) );
     var_phi  = two * PI * temp2;
     *gaussianRand1 = r * native_cos(var_phi);  //    ***********
     *gaussianRand2 = r * native_sin(var_phi);
     //*nextRand = temp[2];

     // The methods of transformation which are not in use
     // are commented out
     //=====Applying Box Mullar Transformations (Polar version)=====
     /*s = temp1*temp1+temp2*temp2;
    var_phi  = -2 * log(s)/s;
     *gaussianRand1 = temp1 * sqrt(var_phi);
     *gaussianRand2 = temp2 * sqrt(var_phi);*/
     // *nextRand = temp[2];

     //=======Moro Inverse Formula=========
    /*
    temp3.x = cndev(temp1.x);
    temp3.y = cndev(temp1.y);
```

```
    temp3.z = cndev(temp1.z);
    temp3.w = cndev(temp1.w);

    temp4.x = cndev(temp2.x);
    temp4.y = cndev(temp2.y);
    temp4.z = cndev(temp2.z);
    temp4.w = cndev(temp2.w);

     *gaussianRand1 = temp3;
     *gaussianRand2 = temp4; */

}

// Approximation of Normal CDF
float phi(float x)  // Normal CDF
{
    // constants
    float a1 =  0.254829592f;
    float a2 = -0.284496736f;
    float a3 =  1.421413741f;
    float a4 = -1.453152027f;
    float a5 =  1.061405429f;
    float p  =  0.3275911f;

    // Save the sign of x
    float sign = 1.f;
    if (x < 0.f)
    sign = -1.f;
    x = fabs(x)/sqrt(2.0f);

    // A&S formula 7.1.26
    float t = 1.0f/(1.0f + p*x);
    float y = 1.0f - (((((a5*t + a4)*t) + a3)*t +
         a2)*t + a1)*t*exp(-x*x);

    return 0.5f*(1.0f + sign*y);
}

// Numerical recipes 6.14 inverse
// normal CDF approximation
float inv_phi(float p) {
    float x, err, t, pp;
    float mu = 0.f, sig = 1.f;

    float p_ = p *2.f;
```

```
    if (p_ >= 2.0f) return -100.f;
    if (p_ <= 0.0f) return 100.f;
    pp = (p_ < 1.0f) ? p_ : 2.f - p_;
    t = sqrt(-2.f*log(pp / 2.f));
    x = -0.70711f*((2.30753f + t*0.27061f) /
         (1.f + t*(0.99229f + t*0.04481f)) - t);
    for (int j = 0; j<2; j++) {
       err = 2.f - 2.f*phi(sqrt(2.f)*x) - pp;
       x += err / (1.12837916709551257f*exp(-sqrt(x)) - x*err);
    }
    float tmp = (p_ < 1.0f ? x : -x);
    return -1.41421356237309505f*sig*tmp + mu;
}

// Normal CDF float4 type version
float4 phi4(float4 x)
{
    float4 temp;
    temp.x = phi(x.x);
    temp.y = phi(x.y);
    temp.z = phi(x.z);
    temp.w = phi(x.w);

return temp;
}

// Gaussian quasi-random number generator fcn
void generateQRandGaussian(
    float4 temp1,
    float4 temp2,
    float4 *gaussianRand1,
    float4 *gaussianRand2
)
{

    float4 r;
    float4 s;
    float4 var_phi;
    const float PI = 3.14159265358979f;
    const float two = 2.0f;

    // ====Applying Box Mullar Transformations====
    /*r = sqrt( (-two) * native_log(temp1));
    var_phi = two * PI * temp2;
    *gaussianRand1 = r * native_cos(var_phi);
```

```
    *gaussianRand2 = r * native_sin(var_phi);*/
    //*nextRand = temp[2];

    //==Applying Box Mullar Transformations (Polar version)==
    /*s = temp1*temp1+temp2*temp2;
    var_phi  = -2 * log(s)/s;
    *gaussianRand1 = temp1 * sqrt(var_phi);
    *gaussianRand2 = temp2 * sqrt(var_phi);*/
    //  *nextRand = temp[2];

    //====Moro Inverse Formula=====

    //float4 temp3, temp4;
    //temp3.x = cndev(temp1.x);
    //temp3.y = cndev(temp1.y);
    //temp3.z = cndev(temp1.z);
    //temp3.w = cndev(temp1.w);

    //temp4.x = cndev(temp2.x);
    //temp4.y = cndev(temp2.y);
    //temp4.z = cndev(temp2.z);
    //temp4.w = cndev(temp2.w);

    //*gaussianRand1 = temp3;
    //*gaussianRand2 = temp4;

    //==== A&S Inverse Formula ====
    float4 temp3, temp4;
    temp3.x = inv_phi(temp1.x);
    temp3.y = inv_phi(temp1.y);
    temp3.z = inv_phi(temp1.z);
    temp3.w = inv_phi(temp1.w);

    temp4.x = inv_phi(temp2.x);
    temp4.y = inv_phi(temp2.y);
    temp4.z = inv_phi(temp2.z);
    temp4.w = inv_phi(temp2.w);

    *gaussianRand1 = temp3;
    *gaussianRand2 = temp4;

}

// Quadratic spline for smoothing payoff fcn
// (not used in current version)
```

```
float QuadraticSpline(float x)
{
   return (0.5f * x * x + 0.5f * x + 0.125f);
};

// Payoff calculation for given Strike,
float4 calOutputs(
               float4 strike,
                float4 price//,
                /*float4 trajPrice2,*/
                //float4* pathDeriv1,
                /*float4* pathDeriv2*/
                )
{
    //float4 tempDiff1 = 0.f;
    //tempDiff1.x = trajPrice1.x - strikePrice.x;
    //tempDiff1.y = trajPrice1.y - strikePrice.y;
    //tempDiff1.z = trajPrice1.z - strikePrice.z;
    //tempDiff1.w = trajPrice1.w - strikePrice.w;
    float4 tempDiff1 = 0.f;
   tempDiff1 = price - strike;
    //float  barrier = 0.f;

   // The lines necessary for smoothing payoff fcn
   // are commented out in current version
  /*if((tempDiff1.x<-barrier) || (tempDiff1.x>barrier))
  {*/
    if(tempDiff1.x < 0.f) { tempDiff1.x = 0.f; }
/*  }
  else
  { tempDiff1.x = QuadraticSpline(tempDiff1.x);};

  if((tempDiff1.y<-barrier) || (tempDiff1.y>barrier))
  {*/
    if(tempDiff1.y < 0.f) { tempDiff1.y = 0.f; }
/*  }
  else
  { tempDiff1.y = QuadraticSpline(tempDiff1.y); };


  if((tempDiff1.z<-barrier) || (tempDiff1.z>barrier))
  {  */
    if(tempDiff1.z < 0.f) { tempDiff1.z = 0.f; }
 /* }
  else
```

```
  { tempDiff1.z = QuadraticSpline(tempDiff1.z);};

  if((tempDiff1.w<-barrier) || (tempDiff1.w>barrier))
  {  */
    if(tempDiff1.w < 0.f) { tempDiff1.w = 0.f; }
 /* }
  else
  { tempDiff1.w = QuadraticSpline(tempDiff1.w);};
*/

/*
  if((tempDiff2.x<-barrier) || (tempDiff2.x>barrier))
  {
    if(tempDiff2.x < 0.0f)
    { tempDiff2.x = 0.0f;} }
  else
  { tempDiff2.x = QuadraticSpline(tempDiff2.x); };

  if((tempDiff2.y<-barrier) || (tempDiff2.y>barrier))
  {
    if(tempDiff2.y < 0.0f)
    { tempDiff2.y = 0.0f;} }
  else
  { tempDiff2.y = QuadraticSpline(tempDiff2.y); };

  if((tempDiff2.z<-barrier) || (tempDiff2.z>barrier))
  {
    if(tempDiff2.z < 0.0f)
    { tempDiff2.z = 0.0f;}}
  else
  { tempDiff2.z = QuadraticSpline(tempDiff2.z); };

  if((tempDiff2.w<-barrier) || (tempDiff2.w>barrier))
  {
    if(tempDiff2.w < 0.0f)
    { tempDiff2.w = 0.0f; } }
  else
  { tempDiff2.w = QuadraticSpline(tempDiff2.w); };
    */
     //**********************

   //pathDeriv1 = &tempDiff1;
   //pathDeriv2 = &tempDiff2;

  return tempDiff1;
```

```
}

// This approximation is used instead of normal CDF
// in AMSM1 model as more numerically efficient analogue
float cuted_sin(float var_x)
{
 float res_value;
 //const float PI = 3.141592653589793f;
 float barrier = 2.042035225f;
 if ((var_x<=barrier)&&(var_x>=-barrier))
 {
   res_value = ((native_sin(var_x/1.3f)+1.f)/2.f);
 };
 if (var_x<-barrier)
{
   res_value = 0.f;
};
 if (var_x> barrier)
{
   res_value = 1.f;
};

 return res_value;
 //return phi(var_x);
}

// This function calculates new volatility state
// k-th frequency for 4 sample paths simultaneously
float4 calSigma(
   float4 sigma0, // model parameter
   float  gk, // fixed model parameter
   float4 m0, // model parameter
   int4   Model, // (A)MSM(1|2)
   int    i, // trigger of initial state
   float4 Randf1, // uniform r.v.
   float4 Randf2, // uniform r.v.
   float4 *next, // next state
   float4 *prev, // previous state
   float4 rho, // model parameter
   float4 prevRandGaus, // previous eps_i
   float4 lambda, // model parameter
   float4 TProb // initial transition prob.
)
{
   float4 sigma    = sigma0;
```

```
float4 tempNext = *next;
float4 tempPrev = *prev;

// First sample paths ====
if(Randf1.x <= gk)
{
   //if(i>1){
         // AMSM1 (eq.2.42)
      if( Model.x <= 1 ) {
          if( Randf2.x <= 1.f -
                 cuted_sin( rho.x *(prevRandGaus.x - lambda.x))) {
              tempNext.x = m0.x; }
            else {
                tempNext.x = 2.f-m0.x; } }
         // AMSM2 (eq.2.13,2.14,2.44 for nu=0)
      if( Model.x == 2 ) {
          if( Randf2.x <= 0.5f ) {
              tempNext.x = m0.x; }
           else {
              tempNext.x = 2.f-m0.x; } }
   //     }
   //else{ if( Randf2.x <= TProb.x ) {
     //     tempNext.x = m0.x; } else { tempNext.x = 2.f-m0.x; } }
   // !!!!!!!!!!!!!!! TProb !!!!!!!!!!!!!!!!!!!!!!!
 }
 else {
     tempNext.x = tempPrev.x; }

 // Rewrite previous and next volatility component state
tempPrev.x = tempNext.x;
sigma.x       = sigma0.x * sqrt(tempNext.x);

 // Second sample path ====
if(Randf1.y <= gk)
{
   //if(i>1){
         // AMSM1 (eq.2.42)
      if( Model.x <= 1 ) {
          if( Randf2.y <= 1.f -
             cuted_sin( rho.y *(prevRandGaus.y - lambda.y))) {
              tempNext.y = m0.y; }
           else {
              tempNext.y = 2.f-m0.y; } }
         // AMSM2 (eq.2.13,2.14,2.44 for nu=0)
         if( Model.x == 2 ) {
```

```
            if( Randf2.y <= 0.5f ){
                tempNext.y = m0.y; }
             else {
                tempNext.y = 2.f-m0.y; } }
    //     }
    //else{ if( Randf2.y <= TProb.y ) {
      //     tempNext.y = m0.y; } else { tempNext.y = 2.f-m0.y; } }
    // !!!!!!!!!!!!!!! TProb !!!!!!!!!!!!!!!!!!!!!!!
}
else{
     tempNext.y = tempPrev.y; }

 // Rewrite previous and next volatility component state
tempPrev.y =                  tempNext.y;
sigma.y    = sigma0.y * sqrt(tempNext.y);

 // Third sample path ====
if(Randf1.z <= gk)
{
    //if(i>1){
          // AMSM1 (eq.2.42)
       if( Model.x <= 1 ) {
            if( Randf2.z <= 1.f -
               cuted_sin( rho.z *(prevRandGaus.z - lambda.z))) {
                   tempNext.z = m0.z; }
             else {
                   tempNext.z = 2.f-m0.z; } }
          // AMSM2 (eq.2.13,2.14,2.44 for nu=0)
       if( Model.x == 2 ) {
            if( Randf2.z <= 0.5f ) {
                tempNext.z = m0.z; }
             else {
                tempNext.z = 2.f-m0.z; } }
    //     }
    //else{ if( Randf2.z <= TProb.z ){
      //     tempNext.z = m0.z; } else { tempNext.z = 2.f-m0.z; } }
    // !!!!!!!!!!!!!!! TProb !!!!!!!!!!!!!!!!!!!!!!!
}
else {
     tempNext.z = tempPrev.z; }

 // Rewrite previous and next volatility component state
tempPrev.z =                  tempNext.z;
sigma.z    = sigma0.z * sqrt(tempNext.z);
```

```
    // Fourth sample path ====
   if(Randf1.w <= gk)
   {
      //if(i>1){
            // AMSM1 (eq.2.42)
         if( Model.x <= 1 ) {
             if( Randf2.w <= 1.f -
                cuted_sin( rho.w *(prevRandGaus.w - lambda.w))) {
                    tempNext.w = m0.w; }
              else {
                    tempNext.w = 2.f-m0.w; } }
            // AMSM2 (eq.2.13,2.14,2.44 for nu=0)
         if( Model.x == 2 ) {
             if( Randf2.w <= 0.5f ) {
                  tempNext.w = m0.w; }
              else {
                  tempNext.w = 2.f-m0.w; } }
      //     }
      // else{ if( Randf2.w <= TProb.w ) {
        //     tempNext.w = m0.w; } else { tempNext.w = 2.f-m0.w; } }
      // !!!!!!!!!!!!!!! TProb !!!!!!!!!!!!!!!!!!!!!!
   }
   else {
       tempNext.w = tempPrev.w; }

    // Rewrite previous and next volatility component state
   tempPrev.w =                  tempNext.w;
   sigma.w    = sigma0.w * sqrt(tempNext.w);

   *next = tempNext;
   *prev = tempPrev;

   return sigma;
}

float4 absolute(float4 value)
{
   if (value.x>= 0.f) {return value;}
   else {return -value;}
}

/** ==== Main function ====
 * @brief   Calculates the  price for all trajectories for
    given random numbers
 * @param   attrib  structure of inputs for simulation
```

```
 * @param    width    width of random array
 * @param    priceSamples array of calculated price samples
 * @param    pathDeriv array calculated path derivatives
 */
__kernel
void
calPriceVega(
    // external structure with model and other parameters
    MonteCarloAttrib attrib,
    int noOfSum, // sample path length
    int width,
    __global uint4  *randArray,
   __global float4 *QrandArray,
    __global float4 *priceSamples,
    __global float4 *pathDeriv )
{
      //width_sobol   = width;
      //noOfSum_sobol = noOfSum;

        // Initialization of variables
        // float4 is a vector of 4 float values
        float4 b = attrib.b;
        float4 m0 = attrib.m0;
        int4   k = attrib.k;
       float4 gkk = attrib.gkk;
        float4 lambda  = attrib.lambda;
      float4 nu = attrib.nu;
        float4 sigma0 = attrib.sigma0;
        float4 rho = attrib.rho;
        float4 interest = attrib.interest;
        // int4 is a vector of 4 float values
      int4   Model = attrib.model;
      int4   RNG = attrib.RNG;
        //Model          = (int)Model4.x;

        float4 strikePrice    = attrib.strikePrice;
        float4 initPrice      = attrib.initPrice;

        // In order to get access to different elements
        // of external arrays global id are obtained
        size_t xPos           = get_global_id(0);
        size_t yPos           = get_global_id(1);

        float4 temp           = (float4)0.0f;
```

```
  //float4 price1          = (float4)0.0f;
  //float4 price2          = (float4)0.0f;
  float4 pathDeriv1    = (float4)0.0f;
  float4 pathDeriv2    = (float4)0.0f;
  float4 pathDeriv1_neg = (float4)0.0f;
  float4 pathDeriv2_neg = (float4)0.0f;
  float4 PayOffBS1      = (float4)0.0f;
  float4 PayOffBS2      = (float4)0.0f;

  float4 trajPrice1     = initPrice;
  float4 trajPrice2     = initPrice;
  float4 trajPrice1_neg = initPrice;
  float4 trajPrice2_neg = initPrice;
  float4 trajPrice1_BS  = initPrice;
  float4 trajPrice2_BS  = initPrice;

  float4 finalRandf1    = temp;
  float4 finalRandf2    = temp;
  float4 finalRandf3    = temp;
  float4 finalRandf4    = temp;

  float4 finalRandf1gaus = temp;
  float4 finalRandf2gaus = temp;
float4 finalRandf3gaus = temp;
  float4 finalRandf4gaus = temp;
float4 finalRandf5gaus = temp;
  float4 finalRandf6gaus = temp;

  float4 prevfinalRandf1gaus = temp;
  float4 prevfinalRandf2gaus = temp;

  float4 tempNext = (float4)1.0f;
float4 tempPrev = (float4)1.0f;

  uint4  nextRand = randArray[yPos * width + xPos];

 int i;
  int j;

// Array with initial transition probabilities
  // (this feature is not used in final version)
  float4 TP[10];
/*
  TP[0]=(float4)0.2789f;
TP[1]=(float4)0.4202f;
```

```
TP[2]=(float4)0.5156f;
TP[3]=(float4)0.5387f;
TP[4]=(float4)0.5024f;
TP[5]=(float4)0.4605f;*/
TP[0]=0.5f;
TP[1]=0.5f;
TP[2]=0.5f;
TP[3]=0.5f;
TP[4]=0.5f;
TP[5]=0.5f;TP[6]=TP[7]=TP[8]=TP[9]=0.5f;

  float4 Mt[10],nextMt[10],Mt2[10],nextMt2[10];
  float4 Mt_neg[10],nextMt_neg[10],Mt2_neg[10],nextMt2_neg[10];
  for(int i=0;i<10;i++)
   {
    Mt[i] = 1.f;  nextMt[i] = 1.f;
       Mt2[i] = 1.f; nextMt2[i] = 1.f;
    Mt_neg[i] = 1.f; nextMt_neg[i] = 1.f;
       Mt2_neg[i] = 1.f; nextMt2_neg[i] = 1.f;
   };

  int   kmax   = k.x;
  float parb   = b.x;
  float pargkk = gkk.x;
  float gk[10];

  // Calculation of vector of gk
  // (A)MSM(1)
if(Model.x <= 1){
     gk[kmax-1] = gkk.x;
    gk[0] = 1.f - exp( log(1.f - pargkk) *
          exp(log(parb) * (1.f-kmax)) );
   for(int i=1; i<kmax - 1; i++){
          // eq.2.15
          gk[i] = 1.f - exp( log(1.f - gk[0])  *
              exp(log(parb) * (float)i));}; }
else {
  // AMSM2
   for(int i=1; i<kmax;      i++){
          // eq.2.16
          gk[i-1]    = exp((i-kmax) * log(2.f)); };
       gk[kmax-1] = gkk.x; }

  float4 sigma1,sigma2,sigma1_neg,sigma2_neg;
```

```
  uint4 tempRand = nextRand;
if (RNG.x == 1) // If PseudoRandom Numbers are used
  {
        // Generate iteratively the next Gaussian R.N.
    generateRandGaussian(
          tempRand,
          &prevfinalRandf1gaus,
          &prevfinalRandf2gaus,
          &nextRand);
  } else // If QuasiRandom Numbers are used
  {
    //QRNvector = 0;
    //finalRandf1 = QrandArray[(yPos *
        //width + xPos)*
        // (2 * k.x + 1) * 2 * max_maturity +
        // QRNvector * max_maturity + i];
    //QRNvector = QRNvector + 1;
    //finalRandf2 = QrandArray[(yPos *
        //width + xPos)*
        // (2 * k.x + 1) * 2 * max_maturity +
        // QRNvector * max_maturity + i];
    //generateQRandGaussian(
        //     finalRandf1,
        //     finalRandf2,
        //     &finalRandf1gaus,
        //     &finalRandf2gaus);

    prevfinalRandf1gaus = (float4)0.f;
    prevfinalRandf2gaus = (float4)0.f;
  }


  //Run the Monte Carlo simulation of sample
  // path of (Num_Sum - 1) length
int max_maturity = 10; int QRNvector;
  for(i = 1; i < noOfSum; i++)
  {

   if (RNG.x == 1) //PRNG
   {
    tempRand = nextRand;
        // Generate 8 epsilon_t from eq.2.1
        // for 8 sample paths simultaneously
    generateRandGaussian(
          tempRand,
```

```
                &finalRandf1gaus,
                &finalRandf2gaus,
                &nextRand);
    } else // QRNG
    {
        QRNvector = 0;

            // Read uniform 4 QRN from the external
            // array QrandArray in non-GPU memory
            // from position xPos, yPos
        finalRandf1 = QrandArray[(yPos *
                width + xPos)*
                (2 * k.x + 1) * 2 * max_maturity +
                QRNvector * max_maturity + i];

            QRNvector = QRNvector + 1;

            // Read another 4 uniform QRN
        finalRandf2 = QrandArray[(yPos *
                width + xPos)*
                (2 * k.x + 1) * 2 * max_maturity +
                QRNvector * max_maturity + i];

            // Transform 8 QRN in 8 epsilon_t from eq.2.1
            // for 8 sample paths simultaneously
            generateQRandGaussian(
                finalRandf1,
                finalRandf2,
                &finalRandf1gaus,
                &finalRandf2gaus);
    }

// ====Computation of Sigma1 & Sigma2====
// Sigma1 is a vector 4 sigma states for 4
// samples paths, Sigma2 contains another
// 4 sigmas for another 4 paths

    // AMSM1 or MSM model
    if(Model.x <= 1){
        sigma1      = sigma0;
        sigma2      = sigma0;
            // Antithetic variables (not used in final version)
        //sigma1_neg = sigma0;
        //sigma2_neg = sigma0;
    }
```

```
else {
    // AMSM2 model
    sigma1 =
            ( rho* (prevfinalRandf1gaus - lambda) - sqrt(sigma0))*
            ( rho* (prevfinalRandf1gaus - lambda) - sqrt(sigma0));
    sigma2 =
            ( rho* (prevfinalRandf2gaus - lambda) - sqrt(sigma0))*
            ( rho* (prevfinalRandf2gaus - lambda) - sqrt(sigma0));
        // Antithetic variables (not used in final version)
    //sigma1_neg =
        //  ( rho* (prevfinalRandf1gaus - lambda) - sqrt(sigma0))*
        //  ( rho* (prevfinalRandf1gaus - lambda) - sqrt(sigma0));
    //sigma2_neg =
        //   ( rho* (prevfinalRandf2gaus - lambda) - sqrt(sigma0))*
        //( rho* (prevfinalRandf2gaus - lambda) - sqrt(sigma0));
}

// Mkt frequencies calculation
    // and multiplication
    for(j = 0; j <=k.x-1; j++)
    {
    if (RNG.x == 1){ // PRNG
        //tempRand = nextRand;
        //generateRand(
                //tempRand, &finalRandf1,
                //&finalRandf2, &nextRand);
            // QRNvector = 2,3; 6,7; ...

        tempNext        = nextMt[j];
        tempPrev        = Mt[j];

        tempRand        = nextRand;
        generateRandGaussian(
                tempRand,
                &finalRandf3gaus,
                &finalRandf4gaus,
                &nextRand);
            // QRNvector = 0,1; t = i

            // Calculate vector sigma1
        sigma1          = calSigma(
                sigma1, gk[j], m0, Model, i,
                phi4(finalRandf3gaus - nu),
                phi4(finalRandf4gaus - nu),
                &tempNext, &tempPrev,
```

```
                rho, prevfinalRandf1gaus,
                lambda,TP[j]);

        } else {          // QRNG
              // Counter of numbers read
              // from QrandArray
            QRNvector = QRNvector + 1;
              // Read next 4 QRN
            finalRandf1 =
                QrandArray[
                    (yPos * width + xPos)*(2 * k.x + 1) *
                    2 * max_maturity +
                    QRNvector * max_maturity + i];

            QRNvector = QRNvector + 1;
              // Read another 4 QRN
            finalRandf2 =
                QrandArray[
                    (yPos * width + xPos)*(2 * k.x + 1) *
                    2 * max_maturity +
                QRNvector * max_maturity + i];

              // Rewrite next and previous M_t
              // vector states
            tempNext        = nextMt[j];
            tempPrev        = Mt[j];

              // Transform uniform 8 QRN to 8
              // Gaussian QRN
            generateQRandGaussian(
                finalRandf1,
                finalRandf2,
                &finalRandf3gaus,
                &finalRandf4gaus);

              // Calculation of the next volatility state
              // for 4 sample paths
              sigma1 = calSigma(
                sigma1,      gk[j], m0, Model, i,
                phi4(finalRandf3gaus - nu),
                phi4(finalRandf4gaus - nu),
                &tempNext, &tempPrev,
                rho,
                prevfinalRandf1gaus,
                lambda,
```

```
            TP[j]);
    }

     // Rewrite next and previous M_t
    nextMt[j]       = tempNext;
  Mt[j]           = tempPrev;


// Mkt for another 4 paths
if (RNG.x == 1) //PRNG
{
    //tempRand = nextRand;
    //generateRand(
            //tempRand, &finalRandf3,
            //&finalRandf4, &nextRand);
        // QRNvector = 4,5; 8,9; ...

    tempNext        = nextMt2[j];
    tempPrev        = Mt2[j];

    tempRand = nextRand;
    generateRandGaussian(
            tempRand,
            &finalRandf5gaus,
            &finalRandf6gaus,
            &nextRand);
        // QRNvector = 0,1; t = i

        // Calculate vector sigma2
    sigma2 = calSigma(
            sigma2, gk[j], m0,  Model, i,
            phi4(finalRandf5gaus - nu),
            phi4(finalRandf6gaus - nu),
            &tempNext, &tempPrev,
            rho,
            prevfinalRandf2gaus,
            lambda,
            TP[j]);

 } else // QRNG
{
    QRNvector = QRNvector + 1;
        // Read next 4 QRN
    finalRandf3 = QrandArray[
          (yPos * width + xPos)*(2 * k.x + 1) *
```

```
                2 * max_maturity +
                QRNvector * max_maturity + i];

            QRNvector = QRNvector + 1;
            // Read another 4 QRN
        finalRandf4 = QrandArray[
                (yPos * width + xPos)*(2 * k.x + 1) *
                2 * max_maturity +
                QRNvector * max_maturity + i];

        tempNext        = nextMt2[j];
        tempPrev        = Mt2[j];

            // Transform uniform 8 QRN to 8
            // Gaussian QRN
        generateQRandGaussian(
                finalRandf3,
                finalRandf4,
                &finalRandf5gaus,
                &finalRandf6gaus);

            // Calculate vector sigma2
        sigma2 = calSigma(
                sigma2, gk[j], m0, Model, i,
                phi4(finalRandf5gaus - nu),
                phi4(finalRandf6gaus - nu),
                &tempNext, &tempPrev,
                rho, prevfinalRandf2gaus,
                lambda,TP[j]);
    }

      nextMt2[j]      = tempNext;
    Mt2[j]          = tempPrev;




// Computation of Sigma1_neg & Sigma2_neg
// ====== (Antithetic Variates) ======
// This block is not used in current version

    //tempRand = nextRand;
    //generateRand(
        //tempRand, &finalRandf1,
```

```
                //&finalRandf2, &nextRand);
    /*
        tempNext        = nextMt_neg[j];
        tempPrev        = Mt_neg[j];
        sigma1_neg      = calSigma(
                sigma1_neg, gk[j], m0, Model, i,
                1.f-finalRandf1 , 1.f-finalRandf2 ,
                &tempNext,&tempPrev,
                rho,-prevfinalRandf1gaus,
                lambda,TP[j]);
          nextMt[j]       = tempNext;
        Mt[j]           = tempPrev;

            //tempRand = nextRand;
            //generateRand(
                //tempRand,
                //&finalRandf1,
                //&finalRandf2,
                //&nextRand);

        tempNext = nextMt2_neg[j];
        tempPrev = Mt2_neg[j];
        sigma2_neg = calSigma(
                sigma2_neg, gk[j], m0, Model, i,
                1.f-finalRandf3 ,
                1.f-finalRandf4 ,
                &tempNext,
                &tempPrev,rho,
                -prevfinalRandf2gaus,
                lambda,TP[j]);
          nextMt2_neg[j] = tempNext;
        Mt2_neg[j]      = tempPrev;
    */
        }


        //Calculate the trajectory price and
        // sum price for all trajectories
        trajPrice1 = trajPrice1*
            exp( interest - 0.5f * sigma1 * sigma1
            + finalRandf1gaus * sigma1 );
        // For Antithetic variates
        //trajPrice1_neg = trajPrice1_neg *
            //exp( interest - 0.5f * sigma1_neg *
            // sigma1_neg-
```

```
            // finalRandf1gaus * sigma1_neg );

        trajPrice2 = trajPrice2 * exp(
            interest -
            0.5f * sigma2 * sigma2 +
            finalRandf2gaus * sigma2 );
        // For Antithetic variates
        //trajPrice2_neg = trajPrice2_neg *
            //exp( interest -
            //0.5f * sigma2_neg * sigma2_neg -
            // finalRandf2gaus * sigma2_neg );

        trajPrice1_BS  =
            trajPrice1_BS * exp( interest -
            0.5f * sigma0 * sigma0 +
            finalRandf1gaus * sigma0 );
        // Control variate
        trajPrice2_BS  =
            trajPrice2_BS * exp( interest -
            0.5f * sigma0 * sigma0
            + finalRandf2gaus * sigma0 );

        prevfinalRandf1gaus = finalRandf1gaus;
        prevfinalRandf2gaus = finalRandf2gaus;

    }

    // For Antithetic variables ===
//trajPrice1_neg = trajPrice1;
//trajPrice2_neg = trajPrice2;
    /*calOutputs(
        strikePrice,
        trajPrice1,
        trajPrice2,
        pathDeriv1,
        pathDeriv2);
    calOutputs(
        strikePrice,
        trajPrice1_neg,
        trajPrice2_neg,
        pathDeriv1_neg,
        pathDeriv2_neg);
calOutputs(
        strikePrice,
        trajPrice1_BS ,
```

```
        trajPrice2_BS ,
        PayOffBS1,
        PayOffBS2); */

    // Payoff (A)MSM model case
  pathDeriv1 = calOutputs( strikePrice, trajPrice1);
  pathDeriv2 = calOutputs( strikePrice, trajPrice2);
    // For Antithetic variables
  //pathDeriv1_neg = calOutputs(
    //   strikePrice,
    //    trajPrice1_neg);
  //pathDeriv2_neg = calOutputs(
    //    strikePrice,
    //    trajPrice2_neg);

    // Payoff Black-Scholes case
  PayOffBS1 = calOutputs(
        strikePrice,
        trajPrice1_BS);
  PayOffBS2 = calOutputs(
        strikePrice,
        trajPrice2_BS);

    // Write 8 AMSM Payoffs into external non-GPU based array
  priceSamples[(yPos * width + xPos) * 2] = pathDeriv1;
    // For Antithetic case ( pathDeriv1 + pathDeriv1_neg )/2.f;
  priceSamples[(yPos * width + xPos) * 2 + 1] = pathDeriv2;
    // For Antithetic case ( pathDeriv2 + pathDeriv2_neg )/2.f;

    // Write 8 B.-S. Payoffs into external non-GPU based array
  pathDeriv[(yPos * width + xPos) * 2] = PayOffBS1;
  pathDeriv[(yPos * width + xPos) * 2 + 1] = PayOffBS2;

}
```

C.2.2. Header file with main structures

This file has very simple structure and it is used to define the variables and main class *MonteCarloAMSM* containing all the settings of the AMSM model, Monte Carlo simulation and methods of optimization. In addition, this file includes headers of external open source libraries used, such as: the code of Sobol quasi-random generator by John Burkardt[2], AlgLib library by Sergey Bochkanov[3], Boost library [4], Adaptive Simulated Annealing (ASA) code by

[2] The Department of Scientific Computing, Florida State University, `people.sc.fsu.edu/~jburkardt/`

[3] `www.alglib.net`

[4] `www.boost.org`

Lester Ingber[5] and OpenCL SDK headers.

```
/**********************************************************************
Copyright 2014 Advanced Micro Devices, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or
without modification, are permitted provided that the following
conditions are met:

Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the
distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
 THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
 ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY
 WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 OF THE POSSIBILITY OF SUCH DAMAGE.
 ********************************************************************/

#ifndef MonteCarloAMSM_H_
#define MonteCarloAMSM_H_
#define NOMINMAX // cppOpt

// global_work_size
// global memory buffer is a matrix X*Y*4 (float4)
// global memory buffer * 4* 2(each kernel generates 4*2 paths) =
// number of paths
#define GLOBAL_MEMORY_SIZE_X 64   // width
//#define GLOBAL_MEMORY_SIZE_Y 32   // height
```

[5] www.ingber.com

```
int GLOBAL_MEMORY_SIZE_Y = 32;

// local_work_size
#define GROUP_SIZE 64 //128
// OpenCL settings
#define GPU_FORCE_64BIT_PTR 0
#define GPU_USE_SYNC_OBJECTS 1
#define GPU_MAX_ALLOC_PERCENT 100
#define GPU_SINGLE_ALLOC_PERCENT 100
#define GPU_MAX_HEAP_SIZE 100

#define __CL_ENABLE_EXCEPTIONS

#ifndef Pi
#define Pi 3.14159265358979323846264 3
#endif

// standard C++ headers
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <conio.h> // for getch()
#include <fstream>
//#include <iostream>
//#include <iomanip>
//#include <sstream>

// OpenCL headers
#include <SDKCommon.hpp>
#include <SDKApplication.hpp>
#include <SDKFile.hpp>

#include <time.h>

// AlgLib headers and settings//

// disable some irrelevant warnings
#if (AE_COMPILER==AE_MSVC)
#pragma warning(disable:4100)
#pragma warning(disable:4127)
#pragma warning(disable:4702)
#pragma warning(disable:4996)
#endif
```

```
#include "src\alglibmisc.h"
#include "src\alglibinternal.h"
#include "src\ap.h"
#include "src\stdafx.h"
#include "src\data.h"
#include "src\linalg.h"
#include "src\statistics.h"
#include "src\dataanalysis.h"
#include "src\specialfunctions.h"
#include "src\solvers.h"
#include "src\optimization.h"
#include "src\diffequations.h"
#include "src\fasttransforms.h"
#include "src\integration.h"
#include "src\interpolation.h"

using namespace alglib;

/**
 * MonteCarloAMSM
 * Class implements OpenCL Monte Carlo Simulation sample
 for AMSM Option pricing
 * Derived from SDKSample base class
 */


// BOOST math library settings
#include "Trash\boost\random\lagged_fibonacci.hpp"
#include "Trash\boost\random\uniform_int_distribution.hpp"
#include "Trash\boost\random\mersenne_twister.hpp"
#include "Trash\boost\random\normal_distribution.hpp"
#include "Trash\boost\random.hpp"

// ASA math library settings
#include "ASA\asa.h"
#include "ASA\asa_usr.h"
#include "ASA\asa_usr_asa.h"

// Sobol' QRNs header file
#include "src\sobol.hpp"

// AMSM model parameters
double      m0_real, sigma_real, rho_real, lambda_real, nu_real;
double      m0_tmp, sigma_tmp, rho_tmp, lambda_tmp, nu_tmp;
int         model;              //MSM, AMSM1, AMSM2
```

```
int             khat;                   // Fixed Param
//float         lambda;         // Option Param
//float         nu;
float       initPrice;      // Option Param
float       interest = 0.f; // Option Param (default)

// Simulation parameters
unsigned int  maxCalculations;
int             path_length;
int             data_gen;
int             RNG;
int             ControlVariates;
int             max_maturity;
int             b_gkk_est;

// Optimization procedures parameters
real_1d_array IPo, IPo2, s, s2, LoBoundary, UpBoundary,
    LoBoundary2, UpBoundary2, y, x;
double          IPoint[5], LBoundary[5], UBoundary[5];
int             Par_number; // number of estimated parameters
double *        r;
double **       DataPointArr;
int             number_of_C = 0;
double          diffstep, diffstep2; // step size
int             estimation  = 1;
int             performance = 1;
int             silent = 0;

ae_int_t        acctype; // acceleration of minimization fcn
int             likMixed;
float objective_fcn;

// Timers
_int64 Timer = (_int64)0;
int price_calc_counter;

// Some minor parameters
int             printout, logging = 1;
//extern          std::ofstream log_file;
const char*     directory;
double          BestRSS;
float       optimum[5];
int             NumOfPoints1 = 0, NumOfPoints2 = 0;

unsigned int  seed_of_simulation_of_seed = 2984140826;
```

```
unsigned int  seed = 0;

unsigned int  PrimesSampleArr[1000];
int           SeedNumber = 0;

int           counter1;

double        objective_func(
    double b, double m0, double gkk, double rho,
    double sigma, double lambda_, double nu_,
    int model, int from, int to);
double        objective_func2(
    double b, double m0, double gkk, double rho,
    double sigma, double lambda_, double nu_,
    int model, int from, int to);

std::string   Method;
std::string   ObjFcn;
std::string   Method2;
std::string   ObjFcn2;
std::string   Memory;

int           Stage;
int           lambda_external;

std::ifstream settings;

// Class containing Monte Carlo simulation settings
  class MonteCarloAMSM : public SDKSample
   {

    cl_int steps; /**< Steps for AMSM Monte Carlo simulation */
    cl_float initPrice;                 /**< Initial price */
    cl_float strikePrice;               /**< Strike price */
    cl_float interest;                  /**< Interest rate */
    cl_float maturity;                  /**< maturity */

    cl_float lambda;                    /**< risk-premium */
   cl_float nu;          /**< risk-premium */
    cl_float b;                         /**< parameter b */
    cl_float m0;                        /**< parameter m0 */
    cl_float m1;                        /**< parameter m0 */
    cl_int   k;                         /**< number of frequencies */
   cl_float gkk;                       /**< switch probability  */
    cl_float *gk;   /**< array of switch probabilities  */
```

```
cl_float sigma0;                /**< initial sigma0 value */
 cl_float rho; /**< correlation between Returns and Switch */
//cl_int a;

 cl_int noOfSum;          /**< Number of exercise points */
 cl_int noOfTraj;           /**< Number of samples */

  /**< time taken to setup OpenCL resources
  and building kernel */
 cl_double setupTime;
 /**< time taken to run kernel and read result back */
 cl_double kernelTime;

 cl_float *sigma;                    /**< Array of sigma values */
 cl_float *price;                    /**< Array of price values */
 //cl_float *vega;                     /**< Array of vega values */

 cl_float *refPrice;          /**< Array of reference price values */

 cl_uint *randNum;        /**< Array of random numbers */

cl_float *QrandNum;     /**< Array of Quasi-random numbers */

 /**< Array of price values for given samples */
 cl_float *priceVals;
 /**< Array of price derivative values for given samples */
 cl_float *priceDeriv;

 cl_context context;    /**< CL context */
 cl_device_id *devices; /**< CL device list */

 cl_mem priceBuf;           /**< CL memory buffer for sigma */
 cl_mem priceDerivBuf;      /**< CL memory buffer for price */
 cl_mem randBuf;      /**< CL memory buffer for random number */
cl_mem QrandBuf; /**< CL memory buffer for random number */

 cl_command_queue commandQueue; /**< CL command queue */
 cl_program program;                 /**< CL program  */
 cl_kernel kernel;                   /**< CL kernel */

 cl_int width;
 cl_int height;

 size_t blockSizeX;                  /**< Group-size in x-direction */
 size_t blockSizeY;                  /**< Group-size in y-direction */
```

```
    int iterations; /**< Number of iterations for kernel execution */

    bool dUseInPersistent;
    bool dUseOutAllocHostPtr;
    bool disableMapping;
    bool disableAsync;

    // Required only when anync enabled
    cl_mem priceBufAsync;      /**< CL memory buffer for sigma */
    cl_mem priceDerivBufAsync; /**< CL memory buffer for price */
    /**< CL memory buffer for random number */
    cl_mem randBufAsync;

    // Required only when anync and mapping enabled
     /**< Array of price values for given samples */
    cl_float *priceValsAsync;
    /**< Array of price derivative values for given samples */
    cl_float *priceDerivAsync;
    /**< SDKDeviceInfo object instance */
    streamsdk::SDKDeviceInfo deviceInfo;
    /**< KernelWorkGroupInfo Object instance */
    streamsdk::KernelWorkGroupInfo kernelInfo;

public:
    /**
     * Constructor
     * Initialize member variables
     * @param name name of sample (string)
     */
    MonteCarloAMSM(std::string name)
        : SDKSample(name)
    {
        steps = 1;
        initPrice = 50.f;
        strikePrice = 55.f;
        interest = 0.00018f;
        maturity = 1.f;

      lambda = 0.51f;
      nu = 0.f;
        b = 3.f;
        m0 = 1.4f;
        m1 = 2.f - m0;
        k = 6;
```

```
    gkk = 0.95f;
      gk = NULL;
    sigma0 = 0.01f;
    rho = 0.3f;

      setupTime = 0;
      kernelTime = 0;

      blockSizeX = GROUP_SIZE; //!!!!!!!!!!!!!!!!!!!!
      blockSizeY = 1;
      noOfSum = 300;
      //GLOBAL_MEMORY_SIZE_X*GLOBAL_MEMORY_SIZE_Y =
      noOfTraj = 256;
      width = GLOBAL_MEMORY_SIZE_X;

      sigma = NULL;
      price = NULL;
      refPrice = NULL;
      randNum = NULL;
    QrandNum = NULL;
      priceVals = NULL;
      priceDeriv = NULL;
      devices = NULL;
      iterations = 1;
      dUseInPersistent = true;
      dUseOutAllocHostPtr = true;
      disableMapping = true;
      disableAsync = true;

      priceValsAsync = NULL;
      priceDerivAsync = NULL;
}

/**
 * Constructor
 * Initialize member variables
 * @param name name of sample (const char*)
 */
MonteCarloAMSM(const char* name)
    : SDKSample(name)
{
    steps = 1;
    initPrice = 50.f;
    strikePrice = 55.f;
    interest = 0.00018f;
```

```
        maturity = 1.f;

    lambda = 0.51f;
    nu = 0.f;
        b = 3.f;
        m0 = 1.4f;
        m1 = 2.f - m0;
        k = 6;
    gkk = 0.95f;
        gk = NULL;
    sigma0 = 0.01f;
    rho = 0.3f;

        setupTime = 0;
        kernelTime = 0;

        blockSizeX = GROUP_SIZE;
        blockSizeY = 1;
        noOfSum = 300;
        noOfTraj = 256;
        width = GLOBAL_MEMORY_SIZE_X;

        sigma = NULL;
        price = NULL;
//          vega = NULL;
        refPrice = NULL;
//          refVega = NULL;
        randNum = NULL;
    QrandNum = NULL;
        priceVals = NULL;
        priceDeriv = NULL;
        devices = NULL;
        iterations = 1;
        dUseInPersistent = true;
        dUseOutAllocHostPtr = true;
        disableMapping = true;
        disableAsync = true;

        priceValsAsync = NULL;
        priceDerivAsync = NULL;
    }

    /**
     * Destructor
     */
```

```
    ~MonteCarloAMSM()
    {

        FREE(sigma);
        FREE(price);
//              FREE(vega);
        FREE(refPrice);
//              FREE(refVega);

      FREE(gk);


        if(randNum)
        {
            #ifdef _WIN32
                ALIGNED_FREE(randNum);
            #else
                FREE(randNum);
            #endif
            randNum = NULL;
        }

      FREE(QrandNum);

        FREE(priceVals);
        FREE(priceDeriv);
        FREE(devices);
    }


    /**
     * Allocate and initialize host memory with appropriate values
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
    int setupMonteCarloAMSM();

    /**
     * Override from SDKSample, Generate binary image of given kernel
     * and exit application
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
    int genBinaryImage();

    /**
     * OpenCL related initializations.
```

```
     * Set up Context, Device list, Command Queue, Memory buffers
     * Build CL kernel program executable
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
    int setupCL();

    /**
     * Set values for kernels' arguments, enqueue calls to the kernels
     * on to the command queue, wait till end of kernel execution.
     * Get kernel start and end time if timing is enabled
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
  //  int runCLKernels();

// delegate: Callback(int str);

   float runCLKernels2(
            float gkk,          // Model Param
            float b,            // Model Param
            float rho,          // Model Param
            float sigma0,       // Model Param
            float m0,           // Model Param
            int   k,            // Fixed Param
            int   model,
            float lambda,       // Option Param
            float nu,           // Option Param
            float initPrice,    // Option Param
            float strikePrice,  // Option Param
            float interest,     // Option Param
            int noOfSum,        // Sym Param
            int width,          // Sym Param
            int height,
            float *Price
      );


    /**
     * Override from SDKSample. Print sample stats.
     */
    //void printStats();

    /**
     * Override from SDKSample. Initialize
     * command line parser, add custom options
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
```

```
     */
    int initialize();

    /**
     * Override from SDKSample, adjust width and height
     * of execution domain, perform all sample setup
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
    int setup();

    /**
     * Override from SDKSample
     * Run OpenCL Bitonic Sort
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
    int run();

    /**
     * Override from SDKSample
    * Cleanup memory allocations
    * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */

  //  int runCalibration();

    int cleanup();

    /**
     * Override from SDKSample
     * Verify against reference implementation
     * @return SDL_SUCCEE on success and SDK_FAILURE on failure
     */
    int verifyResults();

    private:

};

#endif
```

C.2.3. Main file with code

This file contains all the methods and algorithms necessary for the Monte Carlo simulations and calibration of the AMSM model based on various optimization methods. In the first part of the file are defined general subroutines such as the subroutine calculating the

error metrics, normal CDF and etc. The next part is dedicated to the subroutines providing the GPU based computation infrastructure, such as: memory allocation, initialization, writing and reading from buffers, launching computations. The third block of subroutines supports the optimization (minimization) functionality. In this block the objective functions and other necessary objects for Levenberg-Marquardt, Simulated Annealing, SPSA and other optimizers are defined. Finally, the main subroutine includes the parser of settings and input data, logging functionality, error metrics calculation, launcher of the calibration and estimation subroutines based on various optimizers.

```
#include "MonteCarloAMSM.hpp"

#include <math.h>
#include <malloc.h>
#include <thread>
#include <future>

// Technical Functions
#include <windows.h>

// Returns the path to executable file of this program
string getexepath()
{
   char result[MAX_PATH];
   string result2 = string(result, GetModuleFileName(
      NULL, result, MAX_PATH));

   int pos = result2.find("MonteCarloAMSM.exe");
   result2 = result2.substr(0, pos);

   return result2;
}

// The function starts a timer
void StartTimer( _int64 *pt1 )
{
   QueryPerformanceCounter( (LARGE_INTEGER*)pt1 );
}

// The function stopes a timer
double StopTimer( _int64 t1 )
{
   _int64 t2, ldFreq;

   QueryPerformanceCounter( (LARGE_INTEGER*)&t2 );
   QueryPerformanceFrequency( (LARGE_INTEGER*)&ldFreq );
   return ((double)( t2 - t1 ) / (double)ldFreq);
```

```
}

// The function for quadratic spline
float QuadraticSpline(float x)
{
   return (0.5f * x * x + 0.5f * x + 0.125f);
};

// The function converts an integer number to a string
std::string IntToStr(int x)
{
   std::stringstream r;
   r << x;
   return r.str();
}

// The function converts a float number to a string
std::string FloatToStr(float x)
{
   std::stringstream r;
   r << x;
   return r.str();
}

// The function converts a string to a float number
float StrToFloat(std::string s)
{
   return (float)::atof(s.c_str());
}

// The function computes approximation of Normal CDF
double CND(double X)
{
   double L, K, w;

   double const a1 = 0.31938153, a2 = -0.356563782;
   double const a3 = 1.781477937;
   double const a4 = -1.821255978, a5 = 1.330274429;

   L = fabs(X);
   K = 1.0 / (1.0 + 0.2316419 * L);
   w = 1.0 - 1.0 / sqrt(2 * Pi) * exp(-L *L / 2) *
      (a1 * K + a2 * K *K + a3 * pow(K, 3) +
      a4 * pow(K, 4) + a5 * pow(K, 5));
```

```
    if (X < 0) { w = 1.0 - w; }
    return w;
}

// The function computes another approximation of Normal CDF
double phi(double x)
{
    // constants
    double a1 = 0.254829592f;
    double a2 = -0.284496736f;
    double a3 = 1.421413741f;
    double a4 = -1.453152027f;
    double a5 = 1.061405429f;
    double p = 0.3275911f;

    // Save the sign of x
    int sign = 1;
    if (x < 0) sign = -1;
    x = fabs(x) / sqrt(2.0);

    // A&S formula 7.1.26
    double t = 1.0 / (1.0 + p*x);
    double y = 1.0 - (((((a5*t + a4)*t) + a3)*t +
        a2)*t + a1)*t*exp(-x*x);

    return 0.5*(1.0 + sign*y);
}

// The function computes normal density fcn
double dnorm(double x, double mean, double sd)
{
    return(1.0 / sqrt(2 * (double)Pi*pow(sd, 2)) *
        exp(-pow((x - mean) / sd, 2) / 2));
}

// The function computes Black Scholes European call option Vega
double d_j(
    int j, double S, double K, double rf,
    double sigma, double T)
{
    double d_1 = (log(S / K) + (rf + pow(sigma, 2) / 2)*T) /
        (sigma * sqrt(T));
    if (j == 1) { return(d_1); }
    else { return(d_1 - sigma * sqrt(T)); }
}
```

```
// The function computes Black and Scholes (1973) stock option formula
double BlackScholes(
   char CallPutFlag, double S, double X,
   double T, double r, double v)
{
   double d1, d2;

   d1 = (log(S / X) + (r + v*v / 2)*T) / (v*sqrt(T));
   d2 = d1 - v*sqrt(T);

   if (CallPutFlag == 'c')
      return S *CND(d1) - X * exp(-r*T)*CND(d2);
   else if (CallPutFlag == 'p')
      return X * exp(-r * T) * CND(-d2) - S * CND(-d1);
   else return 0;
}

// The function computes B-S call option vega
double BS_Call_Option_Vega(
   double S, double K, double r, double sigma, double T)
{
   return S * dnorm(d_j(1, S, K, r, sigma, T), 0, 1) * sqrt(T);
}

// The function computes error metrics: average, FSSE, RMSE, median
void StandardErrorsMed(
   double ** par, int IP, int size, double real_par_value,
   double *par_av, double *FSSE, double *RMSE, double *med)
{
   double * par_tmp = new double[size];
   for (int i = 0; i<size; i++) par_tmp[i] = par[i][IP];


   *par_av = 0; *RMSE = 0; *med = 0; *FSSE = 0;
   for (int kk = 1; kk <= size; kk++)
   {
      *par_av += par_tmp[kk - 1];
      *RMSE += (par_tmp[kk - 1] -
         real_par_value)*(par_tmp[kk - 1] -
         real_par_value);
   }
   *RMSE = sqrt(*RMSE / size); // RSMD

   *par_av /= size;
```

```
    // Sample Error of Mean based on biased (but corrected)
    // sample st. dev. or just sample st.dev.
    for (int kk = 1; kk <= size; kk++)
    {//if((nu_c[kk-1] >= 1.01)||(nu_c[kk-1] <= 0.99))
     //   {
        *FSSE += (par_tmp[kk - 1] - *par_av)*
            (par_tmp[kk - 1] - *par_av);
        // }
    }

    // Standard Error
    *FSSE = sqrt(*FSSE / (size - 1));

    // Median
    for (int i = 0; i <= size - 1; i++)
    {
        for (int j = 0; j <= size - 2; j++)
        {
            if (par_tmp[j]<par_tmp[j + 1])
            {
                double temp = par_tmp[j];
                par_tmp[j] = par_tmp[j + 1];
                par_tmp[j + 1] = temp;
            }
        }
    }
    *med = par_tmp[size / (int)2];

    delete[] par_tmp;
}

// Root-mean squared error
double RMSE( double * y, double * y_m, int size)
{
    double RMSE = 0;
    for (int kk = 1; kk <= size; kk++)
    {
        RMSE += (y[kk - 1] - y_m[kk - 1])*
            (y[kk - 1] - y_m[kk - 1]);
    }
    RMSE = sqrt(RMSE / size);
    return(RMSE);
}
```

```
// Median
void Par_Median(double ** par, int IP, int iter)
{
   double par2[20];
   for (int i = 0; i <= IP - 1; i++) par2[i] = par[iter][i];
   for (int i = 0; i <= IP - 1; i++)
   {
      for (int j = 0; j <= IP - 2; j++)
      {
         if (par2[j]<par2[j + 1])
         {
            double temp = par2[j];
            par2[j] = par2[j + 1];
            par2[j + 1] = temp;
         }
      }
   }
   par[iter][IP] = (par2[IP / (int)2 - 1] + par2[IP / (int)2]) / 2;
}

// The function logs objective fcn values on 2D grid
void Plot3D()
{
   // Create file Plot3D.txt for logs
   ofstream plot3D(
      getexepath().append(
         "\\Simulations\\Plot3D.txt").c_str());
   float yy, xx = 0.f, y_bar = 0.5f, x_bar = 0.5f,
      x_step = 0.05f, y_step = 0.05f, real_val = 0.f;
   double rss = 0;
   int count = 0;

   // Write to file
   yy = 0.f;
   plot3D << ", ";
   while (yy <= y_bar) {
      plot3D << yy << ", ";
      yy = yy + y_step;
   }
   plot3D << "\n";

   while (xx <= x_bar)
   {
      // Separators of values are commas
      plot3D << xx << ", ";
```

```
        yy = 0.f;
        while (yy <= y_bar)
        {
            rss = objective_func2(
                (float)3, (float)1.4, (float)0.95,
                0.05f, 0.02f, (float)yy, (float)xx,
                model, NumOfPoints1, NumOfPoints2);
            plot3D << rss << ", ";
            count = count + 1;
            cout << count << " xx = " <<
                xx << " yy = " << yy << "\n";
            yy = yy + y_step;
            //plotY<<yy<<" ";
        };
    // New line
    plot3D << "\n";
    //plotX<<xx<<"\n";
    xx = xx + x_step;
    };
    plot3D.close();
    //plotY.close();
    //plotX.close();

    getch();
}


// ==== Parallel (GPU/CPU) computations ====
// Data structure with model parameters
typedef struct _MonteCalroAttrib
{
     cl_float4 strikePrice;
    cl_int4   RNG;

    cl_float4 lambda;
    cl_float4 nu;
     cl_float4 b;
     cl_float4 m0;
     cl_float4 rho;
     cl_int4   k;
    cl_float4 gkk;
     cl_float4 interest;

     cl_float4 initPrice;
     cl_float4 sigma0;
```

```
    cl_int4    model;
} MonteCarloAttrib;


// OpenCL data structure with model parameters
MonteCarloAMSM clMonteCarloAMSM(
    "OpenCL Monte Carlo simulation
    for AMSM Option Pricing");

// Initialize MonteCarloAMSM with AMSM model,
// Monte Carlo and OpenCL parameters
int
MonteCarloAMSM::initialize()
{
    // Call base class Initialize to get default configuration
    CHECK_ERROR(
        this->SDKSample::initialize(),
        SDK_SUCCESS,
        "OpenCL resource initialization failed");

    // Allocate memory for options list
    const int optionsCount = 16;
    streamsdk::Option *optionList =
        new streamsdk::Option[optionsCount];
    CHECK_ALLOCATION(
        optionList,
        " Allocate memory failed (optionList)\\\\n");

    // AMSM model parameters
    optionList[0]._sVersion = "c";
    optionList[0]._lVersion = "steps";
    optionList[0]._description =
        "Steps of Monte Carlo simulation";
    optionList[0]._type = streamsdk::CA_ARG_INT;
    optionList[0]._value = &steps;

    optionList[1]._sVersion = "P";
    optionList[1]._lVersion = "initPrice";
    optionList[1]._description = "Initial price(Default value 50)";
    optionList[1]._type = streamsdk::CA_ARG_FLOAT; //STRING;
    optionList[1]._value = &initPrice;

    optionList[2]._sVersion = "s";
    optionList[2]._lVersion = "strikePrice";
    optionList[2]._description = "Strike price (Default value 55)";
```

```
optionList[2]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[2]._value = &strikePrice;

optionList[3]._sVersion = "r";
optionList[3]._lVersion = "interest";
optionList[3]._description = "interest rate
   (Default value 0.00018)";
optionList[3]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[3]._value = &interest;

optionList[4]._sVersion = "m";
optionList[4]._lVersion = "maturity";
optionList[4]._description = "Maturity (Default value 1)";
optionList[4]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[4]._value = &maturity;

optionList[5]._sVersion = "lambda";
optionList[5]._lVersion = "risk-premium";
optionList[5]._description = "Risk-premium (Default value 0.1)";
optionList[5]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[5]._value = &lambda;

optionList[6]._sVersion = "b";
optionList[6]._lVersion = "param b";
optionList[6]._description = "Param b (Default value 3)";
optionList[6]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[6]._value = &b;

optionList[7]._sVersion = "m0";
optionList[7]._lVersion = "param m0";
optionList[7]._description = "Param m0 (Default value 1.4)";
optionList[7]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[7]._value = &m0;

optionList[8]._sVersion = "k";
optionList[8]._lVersion = "Number of frequencies";
optionList[8]._description =
   "Number of frequencies (Default value 6)";
optionList[8]._type = streamsdk::CA_ARG_INT; //STRING;
optionList[8]._value = &k;

optionList[9]._sVersion = "gamma_k";
optionList[9]._lVersion = "probability of switch";
optionList[9]._description =
   "Probability of the most frequent
```

```
    switch (Default value 0.95)";
optionList[9]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[9]._value = &gkk;

optionList[10]._sVersion = "sigma_0";
optionList[10]._lVersion = "Initial volatility";
optionList[10]._description =
    "Initial volatility (Default value 0.01)";
optionList[10]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[10]._value = &sigma0;

optionList[11]._sVersion = "rho";
optionList[11]._lVersion =
    "Correlation between Gaussian and Switch";
optionList[11]._description =
    "Correlation between Gaussian and
    Switch (Default value 1)";
optionList[11]._type = streamsdk::CA_ARG_FLOAT; //STRING;
optionList[11]._value = &rho;

// Monte Carlo simulations parameters
optionList[12]._sVersion = "noOfSum";
optionList[12]._lVersion = "number of points in path";
optionList[12]._description = "number of points
    (Default value 50)";
optionList[12]._type = streamsdk::CA_ARG_INT; //STRING;
optionList[12]._value = &noOfSum;

optionList[13]._sVersion = "noOfTraj";
optionList[13]._lVersion = "sqrt(number of paths)";
optionList[13]._description =
    "sqrt(number of paths) (Default value 256)";
optionList[13]._type = streamsdk::CA_ARG_INT; //STRING;
optionList[13]._value = &noOfTraj;

// OpenCL options
optionList[14]._sVersion = "width";
optionList[14]._lVersion = "width";
optionList[14]._description = "width (Default value 128)";
optionList[14]._type = streamsdk::CA_ARG_INT; //STRING;
optionList[14]._value = &width;

optionList[15]._sVersion = "blockSizeX";
optionList[15]._lVersion = "blockSizeX";
optionList[15]._description = "blockSizeX (Default value 128)";
```

```
optionList[15]._type = streamsdk::CA_ARG_INT; //STRING;
optionList[15]._value = &blockSizeX;


for (cl_int i = 0; i < optionsCount; ++i)
   sampleArgs->AddOption(&optionList[i]);

delete[] optionList;

streamsdk::Option* iteration_option =
   new streamsdk::Option;
CHECK_ALLOCATION(
   iteration_option,
   "Failed to allocate memory (iteration_option)\\\\n");

iteration_option->_sVersion = "i";
iteration_option->_lVersion = "iterations";
iteration_option->_description =
   "Number of iterations to execute kernel";
iteration_option->_type = streamsdk::CA_ARG_INT;
iteration_option->_value = &iterations;

sampleArgs->AddOption(iteration_option);

delete iteration_option;

// Technical options defining OpenCL regimes
streamsdk::Option* inPersistent_option =
   new streamsdk::Option;
CHECK_ALLOCATION(
   inPersistent_option,
   "Failed to allocate memory (inPersistent_option)\\\\n");

inPersistent_option->_sVersion = "";
inPersistent_option->_lVersion = "dInPersistent";
inPersistent_option->_description =
   "Disables the Persistent memory for input buffers";
inPersistent_option->_type = streamsdk::CA_NO_ARGUMENT;
inPersistent_option->_value = &dUseInPersistent;

sampleArgs->AddOption(inPersistent_option);

delete inPersistent_option;

streamsdk::Option* outAllocHostPtr_option =
```

```
    new streamsdk::Option;
CHECK_ALLOCATION(
    outAllocHostPtr_option,
    "Failed to allocate memory (outAllocHostPtr_option)\\\\n");

outAllocHostPtr_option->_sVersion = "";
outAllocHostPtr_option->_lVersion = "dOutAllocHostPtr";
outAllocHostPtr_option->_description =
    "Disables the Alloc host ptr for output buffers";
outAllocHostPtr_option->_type = streamsdk::CA_NO_ARGUMENT;
outAllocHostPtr_option->_value = &dUseOutAllocHostPtr;

sampleArgs->AddOption(outAllocHostPtr_option);

delete outAllocHostPtr_option;

streamsdk::Option* disableMapping_option =
    new streamsdk::Option;
CHECK_ALLOCATION(
    disableMapping_option,
    "Failed to allocate memory (disableMapping_option)\\\\n");

disableMapping_option->_sVersion = "";
disableMapping_option->_lVersion = "dMapping";
disableMapping_option->_description =
    "Disables mapping/unmapping and uses read/write buffers.";
disableMapping_option->_type = streamsdk::CA_NO_ARGUMENT;
disableMapping_option->_value = &disableMapping;

sampleArgs->AddOption(disableMapping_option);

delete disableMapping_option;

streamsdk::Option* disableAsync_option =
    new streamsdk::Option;
CHECK_ALLOCATION(
    disableAsync_option,
    "Failed to allocate memory (disableAsync_option)\\\\n");

disableAsync_option->_sVersion = "";
disableAsync_option->_lVersion = "dAsync";
disableAsync_option->_description = "Disables Asynchronous.";
disableAsync_option->_type = streamsdk::CA_NO_ARGUMENT;
disableAsync_option->_value = &disableAsync;
```

```
    sampleArgs->AddOption(disableAsync_option);

    delete disableAsync_option;

    return SDK_SUCCESS;
}

// Few technical OpenCL subroutines
int MonteCarloAMSM::setup()
{
    if (setupMonteCarloAMSM() != SDK_SUCCESS)
        return SDK_FAILURE;

    if (setupCL() != SDK_SUCCESS)
        return SDK_FAILURE;

    return SDK_SUCCESS;
}

int MonteCarloAMSM::run()
{
    return SDK_SUCCESS;
}


int MonteCarloAMSM::verifyResults()
{
    return SDK_SUCCESS;
}

// Allocated memory release
int MonteCarloAMSM::cleanup()
{
    // Releases OpenCL resources (Context, Memory etc.)
    cl_int status;

    // Clean bufers
    status = clReleaseMemObject(priceBuf);
    CHECK_OPENCL_ERROR(
        status,
        "clReleaseMemObject(priceBuf) failed.");

    status = clReleaseMemObject(priceDerivBuf);
    CHECK_OPENCL_ERROR(
        status,
```

```
    "clReleaseMemObject(priceDerivBuf) failed.");

status = clReleaseMemObject(randBuf);
CHECK_OPENCL_ERROR(
    status,
    "clReleaseMemObject(randBuf) failed.");

status = clReleaseMemObject(QrandBuf);
CHECK_OPENCL_ERROR(
    status,
    "clReleaseMemObject(QrandBuf) failed.");

// For Async OpenCL option (not used)
if (!disableAsync)
{
    status = clReleaseMemObject(priceBufAsync);
    CHECK_OPENCL_ERROR(
        status,
        "clReleaseMemObject(priceBufAsync) failed.");

    status = clReleaseMemObject(priceDerivBufAsync);
    CHECK_OPENCL_ERROR(
        status,
        "clReleaseMemObject(priceDerivBufAsync) failed.");

    status = clReleaseMemObject(randBufAsync);
    CHECK_OPENCL_ERROR(
        status,
        "clReleaseMemObject(randBufAsync) failed.");
}

// Clean of OpenCL objects (kernel and etc)
status = clReleaseKernel(kernel);
CHECK_OPENCL_ERROR(
    status,
    "clReleaseKernel(kernel) failed.");

status = clReleaseProgram(program);
CHECK_OPENCL_ERROR(
    status,
    "clReleaseProgram(program) failed.");

status = clReleaseCommandQueue(commandQueue);
CHECK_OPENCL_ERROR(
    status,
```

```
        "clReleaseCommandQueue(readKernel) failed.");

    status = clReleaseContext(context);
    CHECK_OPENCL_ERROR(
        status,
        "clReleaseContext(context) failed.");

    // Clean some other buffers
    FREE(sigma);
    FREE(price);
    FREE(refPrice);
    FREE(priceVals);
    FREE(priceDeriv);
    FREE(QrandNum);

    if (!disableAsync && disableMapping)
    {

        FREE(priceValsAsync);
        FREE(priceDerivAsync);

    }
    FREE(devices);
    return SDK_SUCCESS;
}

// Create binary file from OpenCL kernel code
int
MonteCarloAMSM::genBinaryImage()
{
    streamsdk::bifData binaryData;
    binaryData.kernelName = std::string(
        "MonteCarloAMSM_Kernels.cl");
    binaryData.flagsStr = std::string("");
    if (isComplierFlagsSpecified())
        binaryData.flagsFileName = std::string(flags.c_str());

    binaryData.binaryName = std::string(dumpBinary.c_str());
    int status = sampleCommon->generateBinaryImage(binaryData);
    return status;
}

// Memory allocation
int
MonteCarloAMSM::setupMonteCarloAMSM()
```

```
{

    // Allocate and init memory used by host
    price = (cl_float*) malloc(steps * sizeof(cl_float));
    CHECK_ALLOCATION(
      price,
      "Failed to allocate host memory. (price)");
    memset((void*)price, 0, steps * sizeof(cl_float));

    refPrice = (cl_float*) malloc(steps * sizeof(cl_float));
    CHECK_ALLOCATION(
      refPrice,
      "Failed to allocate host memory. (refPrice)");
    memset((void*)refPrice, 0, steps * sizeof(cl_float));

    // Set samples and exercise points
    height = GLOBAL_MEMORY_SIZE_Y;

#if defined (_WIN32)
    randNum = (cl_uint*)_aligned_malloc(
      width * height * sizeof(cl_uint4),
      16);  //!
#else
    randNum = (cl_uint*)memalign(
      16,
      width * height * sizeof(cl_uint4));
#endif

    CHECK_ALLOCATION(
      randNum,
      "Failed to allocate host memory. (randNum)");  //!

   // Buffer array QrandNum contains all quasi-random numbers
   // for all paths and for each transition
   // width * height is number of kernel in parallel
   //  2 * 4 paths in each kernel
   // (khat * 2 + 1) is a number of r.n.
   // necessary for each transition
   // max_maturity is length of path (number of transitions)
   int size = width * height * 2 *
      sizeof(cl_float4) * (khat * 2 + 1) * max_maturity;
   QrandNum = (cl_float*)malloc(
      width * height * 2 * sizeof(cl_float4) *
      (khat * 2 + 1) * max_maturity);
   CHECK_ALLOCATION(
```

```
        QrandNum,
        "Failed to allocate host memory. (QrandNum)");
    memset(
        (void*)QrandNum,
        0,
        width * height * 2 * sizeof(cl_float4) *
        (khat * 2 + 1) * max_maturity);

    // Buffer array for each path value at maturity T
    priceVals = (cl_float*)malloc(
        width * height * 2 * sizeof(cl_float4));
     CHECK_ALLOCATION(
        priceVals,
        "Failed to allocate host memory. (priceVals)");
     memset(
        (void*)priceVals,
        0,
        width * height * 2 * sizeof(cl_float4));

    // Buffer array for each derivative value at maturity T
     priceDeriv = (cl_float*)malloc(
        width * height * 2 * sizeof(cl_float4));
     CHECK_ALLOCATION(
        priceDeriv,
        "Failed to allocate host memory. (priceDeriv)");
     memset(
        (void*)priceDeriv,
        0,
        width * height * 2 * sizeof(cl_float4));

     return SDK_SUCCESS;
}

char DevInf[1024];
// The function setups and connects to OpenCL kernel
int
MonteCarloAMSM::setupCL(void)
{
     cl_int status = 0;

     cl_device_type dType;

    // Check device type (CPU or GPU)
     if(deviceType.compare("cpu") == 0)
     {
```

```
        dType = CL_DEVICE_TYPE_CPU;
    }
    else //deviceType = "gpu"
    {
        dType = CL_DEVICE_TYPE_GPU;
      if(isThereGPU() == false)
        {
            std::cout <<
"GPU not found. Falling back to CPU device" <<
            std::endl;
            dType = CL_DEVICE_TYPE_CPU;
        }
    }

    // Get platform
    cl_platform_id platform = NULL;
    int retValue = sampleCommon->
      getPlatform(
          platform,
          platformId,
          isPlatformEnabled());
    CHECK_ERROR(
      retValue,
      SDK_SUCCESS,
      "sampleCommon::getPlatform() failed");

    // Display available devices.
    retValue = sampleCommon->
      displayDevices(platform, dType);
    CHECK_ERROR(
      retValue,
      SDK_SUCCESS,
      "sampleCommon::displayDevices() failed");


    // If we could find our platform, use it.
   // Otherwise use just available platform.
    cl_context_properties cps[3] =
    {
        CL_CONTEXT_PLATFORM,
        (cl_context_properties)platform,
        0
    };

    context = clCreateContextFromType(
```

```
    cps,
    dType,
      NULL,
      NULL,
      &status);
  CHECK_OPENCL_ERROR(
    status,
    "clCreateContextFromType() failed.");

  status = sampleCommon->
    getDevices(
       context,
       &devices,
       deviceId,
       isDeviceIdEnabled());
  CHECK_ERROR(
    status,
    SDK_SUCCESS,
    "sampleCommon::getDevices() failed");

  //Set device info of given cl_device_id
  retValue = deviceInfo.setDeviceInfo(
    devices[deviceId]);
  CHECK_ERROR(
    retValue,
    SDK_SUCCESS,
    "SDKDeviceInfo::setDeviceInfo() failed");

  commandQueue = clCreateCommandQueue(
    context,
    devices[deviceId],
    0,
    &status);
  CHECK_OPENCL_ERROR(
    status,
    "clCreateCommandQueue(commandQueue) failed.");


unsigned int size = sizeof(DevInf);
retValue = clGetDeviceInfo( devices[deviceId],
    CL_DEVICE_NAME,
    size,
    DevInf,
    NULL);
```

```
if (Memory.find("GPU") != std::string::npos) {
    // create Normal Buffer,
    // if presistent memory is not in use
    randBuf = clCreateBuffer(context,
        CL_MEM_COPY_HOST_PTR,
        sizeof(cl_uint4) * width  * height,
        randNum,
        &status);
    CHECK_OPENCL_ERROR(
        status,
        "clCreateBuffer(randBuf) failed.");


    // create buffer QrandBuf  with QRNs
    // for GPU in host memory
    QrandBuf = clCreateBuffer(context,
        CL_MEM_COPY_HOST_PTR,
        // 2 * (max_k * 2 + 1) Uniform RNs
        sizeof(cl_float4) * width * height * 2 *
        (khat * 2 + 1)  * max_maturity,
        QrandNum,
        &status);
}
else {
    // create buffer for PRNs,
    // if persistent memory is not in use
    randBuf = clCreateBuffer(context,
        CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
        sizeof(cl_uint4) * width  * height,
        randNum,
        &status);
    CHECK_OPENCL_ERROR(
        status,
        "clCreateBuffer(randBuf) failed.");


    // create buffer QrandBuf for QRNs
    QrandBuf = clCreateBuffer(context,
        CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
         // 2 * (max_k * 2 + 1) Uniform RNs
        sizeof(cl_float4) * width * height * 2 *
        (khat * 2 + 1)  * max_maturity,
        QrandNum,
        &status);
    CHECK_OPENCL_ERROR(
```

```
        status,
        "clCreateBuffer(QrandBuf) failed.");
}


    // create buffer on GPU for priceBuf
    priceBuf = clCreateBuffer(
     context,
     CL_MEM_WRITE_ONLY,
     sizeof(cl_float4) * width * height * 2,
     NULL,
     &status);
    CHECK_OPENCL_ERROR(
     status,
     "clCreateBuffer(priceBuf) failed.");

    // create Buffer for priceDerivBuf
    priceDerivBuf = clCreateBuffer(
     context,
     CL_MEM_WRITE_ONLY,
     sizeof(cl_float4) * width * height * 2,
     NULL,
     &status);
    CHECK_OPENCL_ERROR(
     status,
     "clCreateBuffer(priceDerivBuf) failed.");


// Create a CL program using the kernel source
streamsdk::buildProgramData buildData;
buildData.kernelName = std::string(
  "MonteCarloAMSM_Kernels.cl");
buildData.devices = devices;
buildData.deviceId = deviceId;
buildData.flagsStr = std::string("");
if(isLoadBinaryEnabled())
    buildData.binaryName = std::string(loadBinary.c_str());

if(isComplierFlagsSpecified())
    buildData.flagsFileName = std::string(flags.c_str());

retValue = sampleCommon->buildOpenCLProgram(
  program,
  context,
  buildData);
```

```
CHECK_ERROR(
  retValue,
  SDK_SUCCESS,
  "sampleCommon::buildOpenCLProgram() failed");

// get a kernel object handler for a kernel with the given name
kernel = clCreateKernel(
  program,
  "calPriceVega",
  &status);
CHECK_OPENCL_ERROR(status, "clCreateKernel(calPriceVega) failed.");

// Check group-size against what is returned by kernel
status = kernelInfo.setKernelWorkGroupInfo(
  kernel,
  devices[deviceId]);
CHECK_OPENCL_ERROR(
  status,
  "kernelInfo.setKernelWorkGroupInfo failed");


std::cout << "Max Group Size supported on the kernel : " <<
  kernelInfo.kernelWorkGroupSize << std::endl;
if((blockSizeX * blockSizeY) > kernelInfo.kernelWorkGroupSize)
{
    if(!quiet)
    {
        std::cout << "Out of Resources!" << std::endl;
        std::cout << "Group Size specified : "
                  << blockSizeX * blockSizeY << std::endl;

        std::cout << "Falling back to " <<
     kernelInfo.kernelWorkGroupSize << std::endl;
    }

    // Three possible cases
    if(blockSizeX > kernelInfo.kernelWorkGroupSize)
    {
        blockSizeX = kernelInfo.kernelWorkGroupSize;
        blockSizeY = 1;
    }
}

// Fill the buffer randNum with pseudo-random numbers
seed = PrimesSampleArr[SeedNumber - 1];
```

```
// 2834947879; //4294967291; ////2984140826;
//1 586 349 558;  //4294967291; //3715061396;

 // Pseudo Random Number Generator from boost library,
// namely, lagged_fibonacci44497
boost::random::lagged_fibonacci44497 gen;
boost::random::uniform_int_distribution<> ud(1, 45000000);
boost::variate_generator<
   boost::random::lagged_fibonacci44497&,
   boost::random::uniform_int_distribution<>>
      randUniform(gen, ud);

gen.seed( seed );
std::cout << std::endl << "seed = "<< seed <<
std::endl << std::endl;

long buffer=0; //std::ofstream outfileRNG("RNG.txt");
// Warm up
for(int j = 0; j < 1000; j++){
   buffer = randUniform(); //ud(gen);
     //outfileRNG<<(int)randNum[j]<<"\\\\n";
   };
for(int j = 0; j < (width * height * 4); j++){
   buffer = randUniform();
   randNum[j] = (cl_uint)buffer; // (cl_uint)randUniform();
   //ud(gen);
   //outfileRNG<<(int)randNum[j]<<std::endl;
   };
//outfileRNG.close();




// Fill buffer QrandNum with quasi-random numbers
int seed_sobol = seed;
float * rnm1 = new float[max_maturity];
float * rnm2 = new float[max_maturity];
float * rnm3 = new float[max_maturity];
float * rnm4 = new float[max_maturity];

// Sobol quasi-random numbers generator
for (int j = 0; j < 1000; j++) {
   i4_sobol(max_maturity, &seed_sobol, rnm1);
   i4_sobol(max_maturity, &seed_sobol, rnm2);
   i4_sobol(max_maturity, &seed_sobol, rnm3);
```

```
        i4_sobol(max_maturity, &seed_sobol, rnm4);
    }

    for (int i = 0; i < width * height * (khat * 2 + 1) * 2 ; i++)
    {
        //seed = PrimesSampleArr[i + 1];
        i4_sobol(max_maturity, &seed_sobol, rnm1);
        i4_sobol(max_maturity, &seed_sobol, rnm2);
        i4_sobol(max_maturity, &seed_sobol, rnm3);
        i4_sobol(max_maturity, &seed_sobol, rnm4);

        // Write generated QRNs to buffer QrandNum
        for (int j = 0; j < max_maturity; j++) {
            QrandNum[i * 4 * max_maturity + j*4 + 0] = rnm1[j];
            //(float)(i * 4 * max_maturity + j * 4 + 0);//
            QrandNum[i * 4 * max_maturity + j*4 + 1] = rnm2[j];
            //(float)(i * 4 * max_maturity + j * 4 + 1);//
            QrandNum[i * 4 * max_maturity + j*4 + 2] = rnm3[j];
            //(float)(i * 4 * max_maturity + j * 4 + 2);//
            QrandNum[i * 4 * max_maturity + j*4 + 3] = rnm4[j];
            //(float)(i * 4 * max_maturity + j * 4 + 3);//
        }

    }

     return SDK_SUCCESS;
}



// Main Computation Function where all the magic of
// parallel computing on GPU happens!
float
MonteCarloAMSM::runCLKernels2(
    float gkk,          // Model Param
    float b,        // Model Param
    float rho,          // Model Param
    float sigma0,       // Model Param
    float m0,           // Model Param
    int   k,            // Fixed Param
    int   model,        // Model AMSM
    float lambda,       // Option Param (risk premium)
    float nu,           // Option Param (risk premium)
    float initPrice,    // Option Param
    float strikePrice,  // Option Param
```

```
    float interest,     // Option Param
    int noOfSum,        // Sym Param
    int width,          // Sym Param
    int height,         // Sym Param
    float *OPrice
)
{
    cl_int status;
    cl_int eventStatus = CL_QUEUED;
    // Assigning of  width and height
    size_t globalThreads[2] = {(size_t)width, (size_t)height};
    // Assigning of blockSizeX
    size_t localThreads[2]  = {blockSizeX, blockSizeY};

    // Declare attributes structure
    MonteCarloAttrib attributes;

    if(localThreads[0] > deviceInfo.maxWorkItemSizes[0] ||
       localThreads[1] > deviceInfo.maxWorkItemSizes[1] ||
       (localThreads[0] * localThreads[1]) > deviceInfo.maxWorkGroupSize)
    {
        std::cout << "Unsupported: Device does not support requested"
                  << ":number of work items.";
        return SDK_FAILURE;
    }

    // Assign noOfSum, width, randBuf argument to the kernel
    status = clSetKernelArg(
       kernel, 1, sizeof(cl_int), (void*)&noOfSum);
    CHECK_OPENCL_ERROR(status, "clSetKernelArg(noOfSum) failed.");

    // width - i.e number of columns in the array
    status = clSetKernelArg(
       kernel, 2, sizeof(cl_uint), (void*)&width);
    CHECK_OPENCL_ERROR(status, "clSetKernelArg(width) failed.");

    status = clSetKernelArg(
       kernel, 3, sizeof(cl_mem), (void*)&randBuf);
    CHECK_OPENCL_ERROR(status, "clSetKernelArg(randBuf) failed.");

    // Write host buffer to device buffer
    status = clEnqueueWriteBuffer(
       commandQueue, randBuf, CL_TRUE, 0,
       width * height * sizeof(int) * 4, randNum, 0, NULL, NULL);
       CHECK_OPENCL_ERROR(
```

```
        status,
        "clEnqueueWriteBuffer failed
        to write randNum array!");

// Assign QrandBuf argument to the kernel
status = clSetKernelArg(
    kernel, 4,
    sizeof(cl_mem), (void*)&QrandBuf);
    CHECK_OPENCL_ERROR(
        status,
        "clSetKernelArg(QrandBuf) failed.");

// Write host buffer to device buffer
status = clEnqueueWriteBuffer(
    commandQueue, QrandBuf, CL_TRUE, 0,
     width * height * sizeof(float) * 4 * 2 * (khat * 2 + 1)  *
     max_maturity, QrandNum, 0, NULL, NULL);
CHECK_OPENCL_ERROR(
    status,
    "clEnqueueWriteBuffer failed to write QrandNum array!");

status = clSetKernelArg(
    kernel, 5, sizeof(cl_mem), (void*)&priceBuf);
CHECK_OPENCL_ERROR(
        status,
        "clSetKernelArg(priceBuf) failed.");

status = clSetKernelArg(
    kernel, 6, sizeof(cl_mem), (void*)&priceDerivBuf);
 CHECK_OPENCL_ERROR(
        status,
        "clSetKernelArg(priceDerivBuf) failed.");

float timeStep = maturity / (noOfSum - 1);

cl_event events[1];

// Assign AMSM model parameters to attributes structure
const cl_float4 lambdaF4 = {lambda, lambda, lambda, lambda};
attributes.lambda = lambdaF4;

const cl_float4 nuF4 = { nu, nu, nu, nu };
attributes.nu = nuF4;

const cl_float4 bF4 = {b, b, b, b};
```

```
attributes.b = bF4;

const cl_float4 m0F4 = {m0, m0, m0, m0};
attributes.m0 = m0F4;

const cl_int4 kF4 = {k, k, k, k};
attributes.k = kF4;

const cl_float4 gkkF4 = {gkk, gkk, gkk, gkk};
attributes.gkk = gkkF4;

const cl_float4 sigma0F4 = {sigma0, sigma0, sigma0, sigma0};
attributes.sigma0 = sigma0F4;

const cl_float4 rhoF4 = {rho,rho,rho,rho};
attributes.rho = rhoF4;

const cl_float4 initPriceF4 =
   {initPrice, initPrice, initPrice, initPrice};
attributes.initPrice = initPriceF4;

const cl_float4 strikePriceF4 =
   {strikePrice, strikePrice, strikePrice, strikePrice};
attributes.strikePrice       = strikePriceF4;

const cl_float4 interestF4 = {interest, interest, interest, interest};
attributes.interest        = interestF4;

const cl_int4   modelF4 = {model, model, model, model};
attributes.model         = modelF4;

const cl_int4   RNGF4   = { RNG, RNG, RNG, RNG };
attributes.RNG       = RNGF4;

// Set attributes structure to the kernel
status = clSetKernelArg(
   kernel, 0, sizeof(attributes), (void*)&attributes);

CHECK_OPENCL_ERROR(status, "clSetKernelArg(attributes) failed.");

// Enqueue a kernel run call.
status = clEnqueueNDRangeKernel(
   commandQueue,
   kernel,
   2,
```

```
    NULL,
    globalThreads, /* global_work_size */
    localThreads, /* local_work_size */
    0,
    NULL,
    &events[0]);

CHECK_OPENCL_ERROR(status, "clEnqueueNDRangeKernel() failed.");

status = clFlush(commandQueue);
CHECK_OPENCL_ERROR(status, "clFlush() failed.");

// wait for the kernel call to finish execution
status = sampleCommon->waitForEventAndRelease(&events[0]);
CHECK_ERROR(status,0, "WaitForEventAndRelease(events[0]) Failed");

// Enqueue the results to application pointer
status = clEnqueueReadBuffer(
    commandQueue,
    priceBuf,
    CL_TRUE,
    0,
    width * height * 2 * sizeof(cl_float4),
    priceVals,
    0,
    NULL,
    &events[0]);

CHECK_OPENCL_ERROR(
    status,
    "clEnqueueReadBuffer(priceBuf) failed.");

// Wait for finishing execution  of all kernels to read buffer
status = sampleCommon->waitForEventAndRelease(&events[0]);
CHECK_OPENCL_ERROR(
    status,
    "clWaitForEventsAndRelease(events[0]) failed.");

 // Enqueue the results to application pointer
status = clEnqueueReadBuffer(
    commandQueue,
    priceDerivBuf,
    CL_TRUE,
    0,
    width * height * 2 * sizeof(cl_float4),
```

```
    priceDeriv,
    0,
    NULL,
    &events[0]);
CHECK_OPENCL_ERROR(
        status,
        "clEnqueueReadBuffer(priceDerivBuf) failed.");

// Wait for the read buffer to finish execution
status = sampleCommon->waitForEventAndRelease(&events[0]);
CHECK_OPENCL_ERROR(
    status,
    "clWaitForEventsAndRelease(events[0]) failed.");

//float TPrice = 0;
int kk=0; float count=0.f;

float ControlledVariate = 0.f;
float ratio = 1.f, sumU = 0.f, sumD = 0.f, control=0.f,
option_price_float=0.f, tempDiff = 0.f, barrier = 0.f;

price[kk]=0.f;float sum = 0.f;
float control2 = 0.f; float sum2 = 0.f;
for(int i = 0; i < width * 8 * height; i++)
{
    // Control variates
    control            = control + priceDeriv[i];

    //option_price += (double)priceVals[i];
    option_price_float = option_price_float + priceVals[i];
    /*if (logging) {
        log_file << priceVals[i] << "\n";}
        option_price_float += priceVals[i];*/

    count+=1.f;
}

// Averaging of payoffs to get math.expectation for AMSM
option_price_float /= count;
// Averaging of payoffs to get math.expectation
// for BS control variate
control            /= count;
//option_price /=count;

// Fixing of values due to control variate
```

```
    if (ControlVariates == 1) {
        for (int i = 0; i < width * 8 * height; i++)
        {
            if (
                (priceVals[i] >= 0) &&
                (priceVals[i] <= 10000000)){
                    sumU +=
                        (priceDeriv[i] -
                        control)*
                        (priceVals[i] -
                        option_price_float);
                    sumD +=
                        (priceDeriv[i] - control)*
                        (priceDeriv[i] - control);
                };
            };

            if (sumD != 0.f) {
                ratio = sumU / sumD;
            }
            else { ratio = 0; }

            float BS_price = (float)BlackScholes(
                'c', initPrice, strikePrice,
                noOfSum, interest, sigma0);
            option_price_float = (
                exp(-interest * (float)noOfSum ) *
                option_price_float - ratio *
                (exp(-interest * (float)noOfSum ) *
                control - /*initPrice*/BS_price));
            }
    // Return option price value
    *OPrice = option_price_float;
    return option_price_float;
}

// The function reads option prices data for calibration from RealData.txt
// and write it to the array DataPointArr
void DataPointRead(double ** DataPointArr, int n)
{
    ifstream infile;
    char* buffer = new char[1000000];

    // Try to read RealData.txt
    std::string app_path = getexepath();
```

```
app_path.append("\\RealData.txt");

try { infile.open(app_path.c_str()); }
catch (ios_base::failure e) {
    cout << "No Real Data file found! Exception
        opening/reading/closing file!\\\\n";
    getch();
    //return 0;
}

// Skip first two lines
infile.getline(buffer, 1000000);
infile.getline(buffer, 1000000);

// Maturity read from vector [,,,]
infile.getline(buffer, 1000000);
string s = string(buffer);
replace(s.begin(), s.end(), '[', ' ');
replace(s.begin(), s.end(), ']', ' ');
stringstream ss(s);
vector<string> vect;

while (ss.good()){
    string substr;
    getline(ss, substr, ',');
    vect.push_back(substr);    }

// Write all maturities to the first column of DataPointArr
for (int i = 0; i < (int)vect.size(); i++)   {
    DataPointArr[i][0] = StrToFloat(vect.at(i)); }

// Strike prices read
infile.getline(buffer, 1000000);
s = string(buffer);
replace(s.begin(), s.end(), '[', ' ');
replace(s.begin(), s.end(), ']', ' ');
stringstream ss2(s);
vector<string> vect2;

while (ss2.good()) {
    string substr;
    getline(ss2, substr, ',');
    vect2.push_back(substr);
}
```

```
    // Write all strike prices to the second column of DataPointArr
    for (int i = 0; i < n; i++) {
        DataPointArr[i][1] = StrToFloat(vect2.at(i));
    }

    // Interest rate read
    infile.getline(buffer, 1000000);
    s = string(buffer);
    replace(s.begin(), s.end(), '[', ' ');
    replace(s.begin(), s.end(), ']', ' ');
    stringstream ss3(s);
    vector<string> vect3;

    while (ss3.good()) {
        string substr;
        getline(ss3, substr, ',');
        vect3.push_back(substr);
    }

    // Write all int.rates to the third column of DataPointArr
    for (int i = 0; i < n; i++) {
        DataPointArr[i][2] = StrToFloat(vect3.at(i));
    }

    infile.close();
    delete[] buffer;
}

// Toy example array with option prices data for calibration
void DataPoint(int var1, int *noOfSum, float *strike, float *interest)
{
    if (data_gen == 0) {
        // Read & Write Data
        *noOfSum  = (int)DataPointArr[var1 - 1][0];
        *strike   = (float)DataPointArr[var1 - 1][1];
        *interest = (float)DataPointArr[var1 - 1][2];
    }
    else {
        switch ((int)var1)
        {
        case 1: {*noOfSum = 30; *strike = 40.f; } break;
        case 2: {*noOfSum = 30; *strike = 42.f; } break;
        case 3: {*noOfSum = 30; *strike = 44.f; } break;
        case 4: {*noOfSum = 30; *strike = 46.f; } break;
        case 5: {*noOfSum = 30; *strike = 48.f; } break;
```

```
case 6: {*noOfSum = 30; *strike = 50.f; } break;
case 7: {*noOfSum = 30; *strike = 52.f; } break;
case 8: {*noOfSum = 30; *strike = 54.f; } break;
case 9: {*noOfSum = 30; *strike = 56.f; } break;
case 10: {*noOfSum = 30; *strike = 58.f; } break;

case 11: {*noOfSum = 60; *strike = 40.f; } break;
case 12: {*noOfSum = 60; *strike = 42.f; } break;
case 13: {*noOfSum = 60; *strike = 44.f; } break;
case 14: {*noOfSum = 60; *strike = 46.f; } break;
case 15: {*noOfSum = 60; *strike = 48.f; } break;
case 16: {*noOfSum = 60; *strike = 50.f; } break;
case 17: {*noOfSum = 60; *strike = 52.f; } break;
case 18: {*noOfSum = 60; *strike = 54.f; } break;
case 19: {*noOfSum = 60; *strike = 56.f; } break;
case 20: {*noOfSum = 60; *strike = 58.f; } break;

case 21: {*noOfSum = 90; *strike = 40.f; } break;
case 22: {*noOfSum = 90; *strike = 42.f; } break;
case 23: {*noOfSum = 90; *strike = 44.f; } break;
case 24: {*noOfSum = 90; *strike = 46.f; } break;
case 25: {*noOfSum = 90; *strike = 48.f; } break;
case 26: {*noOfSum = 90; *strike = 50.f; } break;
case 27: {*noOfSum = 90; *strike = 52.f; } break;
case 28: {*noOfSum = 90; *strike = 54.f; } break;
case 29: {*noOfSum = 90; *strike = 56.f; } break;
case 30: {*noOfSum = 90; *strike = 58.f; } break;

case 31: {*noOfSum = 120; *strike = 40.f; } break;
case 32: {*noOfSum = 120; *strike = 42.f; } break;
case 33: {*noOfSum = 120; *strike = 44.f; } break;
case 34: {*noOfSum = 120; *strike = 46.f; } break;
case 35: {*noOfSum = 120; *strike = 48.f; } break;
case 36: {*noOfSum = 120; *strike = 50.f; } break;
case 37: {*noOfSum = 120; *strike = 52.f; } break;
case 38: {*noOfSum = 120; *strike = 54.f; } break;
case 39: {*noOfSum = 120; *strike = 56.f; } break;
case 40: {*noOfSum = 120; *strike = 58.f; } break;

case 41: {*noOfSum = 240; *strike = 40.f; } break;
case 42: {*noOfSum = 240; *strike = 42.f; } break;
case 43: {*noOfSum = 240; *strike = 44.f; } break;
case 44: {*noOfSum = 240; *strike = 46.f; } break;
case 45: {*noOfSum = 240; *strike = 48.f; } break;
case 46: {*noOfSum = 240; *strike = 50.f; } break;
```

```
        case 47: {*noOfSum = 240; *strike = 52.f; } break;
        case 48: {*noOfSum = 240; *strike = 54.f; } break;
        case 49: {*noOfSum = 240; *strike = 56.f; } break;
        case 50: {*noOfSum = 240; *strike = 58.f; } break;

        case 51: {*noOfSum = 360; *strike = 40.f; } break;
        case 52: {*noOfSum = 360; *strike = 42.f; } break;
        case 53: {*noOfSum = 360; *strike = 44.f; } break;
        case 54: {*noOfSum = 360; *strike = 46.f; } break;
        case 55: {*noOfSum = 360; *strike = 48.f; } break;
        case 56: {*noOfSum = 360; *strike = 50.f; } break;
        case 57: {*noOfSum = 360; *strike = 52.f; } break;
        case 58: {*noOfSum = 360; *strike = 54.f; } break;
        case 59: {*noOfSum = 360; *strike = 56.f; } break;
        case 60: {*noOfSum = 360; *strike = 58.f; } break;

        case 61: {*noOfSum = 720; *strike = 40.f; } break;
        case 62: {*noOfSum = 720; *strike = 42.f; } break;
        case 63: {*noOfSum = 720; *strike = 44.f; } break;
        case 64: {*noOfSum = 720; *strike = 46.f; } break;
        case 65: {*noOfSum = 720; *strike = 48.f; } break;
        case 66: {*noOfSum = 720; *strike = 50.f; } break;
        case 67: {*noOfSum = 720; *strike = 52.f; } break;
        case 68: {*noOfSum = 720; *strike = 54.f; } break;
        case 69: {*noOfSum = 720; *strike = 56.f; } break;
        case 70: {*noOfSum = 720; *strike = 58.f; } break;
        };
    }
};

// Wrapper to clMonteCarloAMSM.runCLKernels2, which runs it
// on option price data from var1-th row of DataPoint
double function_temp(
    float p1, float p2, float p3, float p4, float p5,
    float p6, float p7, int model, float var1)
{
    float strike      = 0.f;
    // float strikePrice = 65.f; // Option Param
    int   noOfSum     = 0;           // Sym Param
    float tmp_interest    = interest;

    int   width       = GLOBAL_MEMORY_SIZE_X; // Sym Param
    int   height      = GLOBAL_MEMORY_SIZE_Y; // Sym Param
    float temp;
```

```
    price_calc_counter = price_calc_counter + 1;

   DataPoint( (int)var1, &noOfSum, &strike, &tmp_interest);

    return (double)clMonteCarloAMSM.runCLKernels2(
     p3,//gkk,        // Model Param
     p1,//b,          // Model Param
     p4,// rho        // Model Param
     p5,//sigma0,     // Model Param
     p2,//m0          // Model Param
     khat,            // Fixed Param
     model,           //
     p6, //lambda,          // Option Param
     p7, //nu,           // Option Param
     initPrice,       // Option Param
     strike,//strikePrice, // Option Param
     tmp_interest,        // Option Param
     noOfSum,//noOfSum,             // Sym Param
     width,           // Sym Param
     height,          // Sym Param
     &temp);
// return exp(-p1*pow(var1, 2));

   std::cout<<" temp = " << (double)temp << std::endl;
}

// Wrapper to objective_func, which allows to calibrate
// various parameters by BFGS optimizer
void function1_func(const real_1d_array &xx, double &func, void *ptr)
{
    double ss=0;

    if (Par_number == 3) {
       ss = objective_func(
          // Calibrate m_0, sigma_0, rho
          3.0, xx[0]/*1.4*/, 0.95/*xx[0]*/,
          /*2.0*/xx[2], /*0.02*/xx[1],
          lambda_real, nu_real,
          model, NumOfPoints1, NumOfPoints2);
    }
    if (Par_number == 4) {
       if (!lambda_external) {
          // Calibrate m_0, sigma_0, rho, lambda
          ss = objective_func(
             3.0, xx[0]/*1.4*/, 0.95/*xx[0]*/,
```

```
            /*2.0*/xx[2], /*0.02*/xx[1],
            xx[3], nu_real,
            model,
            NumOfPoints1, NumOfPoints2);
      } else {
         // Calibrate m_0, sigma_0, rho, nu
         ss = objective_func(
            3.0, xx[0]/*1.4*/, 0.95/*xx[0]*/,
            /*2.0*/xx[2], /*0.02*/xx[1],
            lambda_real, xx[3],
            model,
            NumOfPoints1, NumOfPoints2);
      }
   }
   // Calibrate m_0, sigma_0, rho, lambda, nu
   if (Par_number == 5) {
      if (!b_gkk_est) {
         ss = objective_func(
            3.0, xx[0]/*1.4*/, 0.95/*xx[0]*/,
            /*2.0*/xx[2], /*0.02*/xx[1],
            xx[3], xx[4],
            model,
            NumOfPoints1, NumOfPoints2);
      }
      else {
         // Calibrate m_0, sigma_0, rho, b, gamma_k
         ss = objective_func(
            x[3], xx[0]/*1.4*/, x[4]/*xx[0]*/,
            /*2.0*/xx[2], /*0.02*/xx[1],
            lambda_real, nu_real,
            model,
            NumOfPoints1, NumOfPoints2); }
   }

   func = ss;
}

// Wrapper to objective_func, which allows to calibrate
// various parameters by BFGS optimizer
void function1_func2(const real_1d_array &xx, double &func, void *ptr)
{
   double ss = 0;

   if (Par_number == 1) {
      if (lambda_external) {
```

```
            // Calibrate lambda
            ss = objective_func2(
                3.0, m0_tmp, 0.95, rho_tmp, sigma_tmp,
                xx[0], nu_real,
                model,
                NumOfPoints1, NumOfPoints2);
        }
        else {
            // Calibrate nu
            ss = objective_func2(
                3.0, m0_tmp, 0.95, rho_tmp, sigma_tmp,
                lambda_tmp, xx[0],
                model,
                NumOfPoints1, NumOfPoints2);
        }

    }
    // Calibrate lambda, nu
    if (Par_number == 2) {
        ss = objective_func2(
            3.0, m0_tmp, 0.95, rho_tmp, sigma_tmp,
            xx[0], xx[1],
            model,
            NumOfPoints1, NumOfPoints2);
    }

    func = ss;
}

// Wrapper to objective_func, which allows to calibrate
// various parameters by Levenberg–Marquardt optimizer
void function1_fvec(const real_1d_array &x, real_1d_array &fi, void *ptr)
{
    float rss=0.f;double y_MC=0.f;

    // If method is LM and objective fcn is WRSS
    if (
        (Method.find("LM") != std::string::npos) &&
        (ObjFcn.find("WRSS") != std::string::npos)) {
        for (int i = NumOfPoints1; i <= NumOfPoints2; i++)
        {
            // Calibrate m_0, sigma_0, rho
            if (Par_number == 3) {
                y_MC = objective_func(
                    3, x[0], 0.95, x[2], x[1],
```

```
            lambda_real, nu_real, model, i, i);
    }
    if (Par_number == 4) {
        if (!lambda_external) {
            // Calibrate m_0, sigma_0,
            // rho, lambda
            y_MC = objective_func(3, x[0], 0.95,
            x[2], x[1],
            x[3], nu_real,
            model,
            i, i);
        } else {
            // Calibrate m_0, sigma_0,
            // rho, nu
            y_MC = objective_func(3, x[0], 0.95,
            x[2], x[1],
            lambda_real, x[3],
            model,
            i, i);
        }
    }
    if (Par_number == 5) {
        if (!b_gkk_est) {
            // Calibrate m_0, sigma_0, rho,
            // lambda, nu
            y_MC = objective_func(
                3, x[0], 0.95,
                x[2], x[1],
                x[3], x[4], model, i, i);
        }
        else {
            // Calibrate m_0, sigma_0, rho,
            // b, gamma_k
            y_MC = objective_func(
                x[3], x[0], x[4],
                x[2], x[1],
                lambda_real, nu_real,
                model,
                i, i);
        }
    }

    fi[i - NumOfPoints1] =
        (y_MC - y[i - 1]) / (y[i - 1]);
```

```
            rss = rss +
                (float)fi[i - NumOfPoints1] *
                (float)fi[i - NumOfPoints1];
        }
    }

    // If method is LM and objective fcn is not WRSS
    else
    {
        // Calibrate m_0, sigma_0, rho
        if (Par_number == 3) {
            fi[0] = 15000 + objective_func(
                3, x[0], 0.95, x[2], x[1],
                lambda_real, nu_real, model, -1, -1);
        }
        if (Par_number == 4) {
            if (!lambda_external) {
                // Calibrate m_0, sigma_0, rho, lambda
                fi[0] = 15000 + objective_func(
                    3, x[0], 0.95, x[2], x[1],
                    x[3], nu_real,
                    model,
                    -1, -1);
            }
            else {
                // Calibrate m_0, sigma_0, rho, nu
                fi[0] = 15000 + objective_func(
                    3, x[0], 0.95, x[2], x[1],
                    lambda_real, x[3],
                    model,
                    -1, -1);
            }
        }
        if (Par_number == 5) {
            if (!b_gkk_est) {
                // Calibrate m_0, sigma_0, rho,
                // lambda, nu
                fi[0] = 15000 + objective_func(
                    3, x[0], 0.95, x[2], x[1],
                    x[3], x[4],
                    model,
                    -1, -1);
            }
            else {
                // Calibrate m_0, sigma_0, rho, b,
```

```
            // gamma_k
            fi[0] = 15000 + objective_func(
               x[3], x[0], x[4], x[2], x[1],
               lambda_real, nu_real,
               model,
               -1, -1); }
      }
      rss    = (float)fi[0];
   }

   // Objective fcn evaluations counter
   counter1 = counter1 + 1;

   // Print on screen values of parameters
   // during current iteration
   if (printout) {
      if (Par_number == 3) {
         if ((counter1) % 8 == 0) {
            std::cout << " m0 = " << x[0] <<
            " sigma = " <<
            x[1] << " rho = " << x[2] <<
            " lambda = " <<
            lambda_real << " nu = " <<
            nu_real << " RSS = " <<
             rss << " k = " << counter1 <<
             " Time = " <<
             StopTimer(Timer) <<
             "\n"; StartTimer(&Timer); }
      }
      if (Par_number == 4) {
         if (!lambda_external) {
            if ((counter1) % 10 == 0) {
               std::cout << " m0 = " <<
               x[0] <<
               " sigma = " <<
               x[1] << " rho = " <<
               x[2] <<
               " lambda = " <<
               x[3] << " nu = " <<
               nu_real <<
               " RSS = " <<
               rss << " k = " <<
               counter1 <<
               " Time = " <<
               StopTimer(Timer) <<
```

```
                "\n"; StartTimer(&Timer); }
            }
            else {
                if ((counter1) % 10 == 0) {
                    std::cout <<
                    " m0 = " << x[0] <<
                    " sigma = " << x[1] <<
                    " rho = " << x[2] <<
                    " lambda = " << lambda_real <<
                    " nu = " << x[3] <<
                    " RSS = " << rss <<
                    " k = " << counter1 <<
                    " Time = " <<
                    StopTimer(Timer) <<
                    "\n"; StartTimer(&Timer); }
            }
        }
        if (Par_number == 5) {
            if ((counter1) % 12 == 0) {
                std::cout <<
                " m0 = " << x[0] <<
                " sigma = " << x[1] <<
                " rho = " << x[2] <<
                " lambda = " << x[3] <<
                " nu = " << x[4] <<
                " RSS = " << rss <<
                " k = " << counter1 <<
                " Time = " <<
                StopTimer(Timer) <<
                "\n"; StartTimer(&Timer); }
        }
    }

}

// Wrapper to objective_func, which allows to calibrate
// various parameters by Levenberg–Marquardt optimizer
void function1_fvec2(
    const real_1d_array &x, real_1d_array &fi, void *ptr)
{

    float rss = 0.f; double y_MC = 0.f;

    // If method is Levenberg–Marquardt
    if ((Method2.find("LM") != std::string::npos)) {
```

```
    for (int i = NumOfPoints1; i <= NumOfPoints2; i++)
    {
        if (Par_number == 1) {
            if (lambda_external) {
                // Calibrate lambda
                y_MC = objective_func2(
                    3, m0_tmp, 0.95,
                    rho_tmp, sigma_tmp,
                    x[0], nu_real,
                    model, i, i);
            }
            else {
                // Calibrate nu
                y_MC = objective_func2(
                    3, m0_tmp, 0.95,
                    rho_tmp, sigma_tmp,
                    lambda_tmp, x[0],
                    model, i, i);
            }
        }
        if (Par_number == 2) {
            // Calibrate lambda, nu
            y_MC = objective_func2(
                3, m0_tmp, 0.95,
                rho_tmp, sigma_tmp,
                x[0], x[1],
                model, i, i);
        }

        // Calculate weighted residual
        fi[i - NumOfPoints1] = (y_MC - y[i - 1]) /
            (y[i - 1]);

        // Calculate sum of squared residuals
        rss = rss +
            (float)fi[i - NumOfPoints1] *
            (float)fi[i - NumOfPoints1];
    }
}

// If method is not Levenberg-Marquardt
else
{
    if (Par_number == 1) {
        if (lambda_external) {
```

```
            // Calibrate lambda
            fi[0] = 15000 + objective_func2(
               3, m0_tmp, 0.95, rho_tmp,
               sigma_tmp,
               x[0], nu_real, model, -1, -1);
         }
         else {
            // Calibrate nu
            fi[0] = 15000 + objective_func2(
               3, m0_tmp, 0.95, rho_tmp,
               sigma_tmp,
               lambda_tmp, x[0],
               model, -1, -1);
         }
      }
      if (Par_number == 2) {
         // Calibrate lambda, nu
         fi[0] = 15000 + objective_func2(
            3, m0_tmp, 0.95, rho_tmp,
            sigma_tmp,
            x[0], x[1],
            model, -1, -1);
      }

      rss = (float)fi[0];
   }

   // Objective fcn evaluations counter
   counter1 = counter1 + 1;

   // Print on screen values of parameters
   // during current iteration
   if (printout) {
      if (Par_number == 1) {
         if (lambda_external) {
            if ((counter1) % 4 == 0) {
               std::cout <<
               " lambda = " << x[0] <<
               " nu = " << nu_real <<
               " RSS = " <<
               rss << " k = " <<
               counter1 << " Time = "<<
               StopTimer(Timer) <<
               "\n"; StartTimer(&Timer); }
         }
```

```
            else {
                if ((counter1) % 4 == 0) {
                    std::cout << " lambda = " <<
                    lambda_tmp <<
                    " nu = " << x[0] <<
                    " RSS = " << rss <<
                    " k = " << counter1 <<
                    " Time = " <<
                    StopTimer(Timer) <<
                    "\n"; StartTimer(&Timer); }
            }
        }
        if (Par_number == 2) {
            if ((counter1) % 6 == 0) {
                std::cout << " lambda = " <<
                x[0] << " nu = " <<
                x[1] << " RSS = " <<
                rss << " k = " <<
                counter1 <<
                " Time = " <<
                StopTimer(Timer) <<
                "\n"; StartTimer(&Timer); }
        }
    }
}

using namespace std;
// The function generates artificial option prices data
// then write it to RealDataSimulated.txt together with
// the settings from Settings.ini
real_1d_array AMSM_option_data(
    float b, float m0, float gkk, float rho, float sigma,
    float lambda_, float nu_, int model,
    int from, int to, ifstream &settings)
{
    float y_MC = 0.f;
    std::string s = "[";
    std::ofstream outfileData(
        getexepath().append(
            "\\Simulations\\RealDataSimulated.txt").c_str(),
            ios::out | ios::app);

    // Calculate prices from toy-example option data
    for (int i = 1; i <= 70; i++)
    {
```

```
        if ((i >= from) && (i <= to)) {
            y_MC = (float)function_temp(
                b, m0, gkk, rho, sigma,
                lambda_, nu_, model, (float)i);
        }
        else {
            y_MC = 0;
        }

        if (i != 70) { s = s + FloatToStr(y_MC) + ","; }
        else { s = s + FloatToStr(y_MC); };
    };
    s = s + "]\n";

    // Write vector [,,,] to the file
    outfileData << s << endl <<endl;

    // Write settings either line by line
    try { settings.open("Settings.ini"); }
    catch (ios_base::failure e) {
        cout << "No Real Data file found!
        Exception opening/reading/closing file!\\\\n";
        getch();
//      return 0;
    }

    string line;
    while (getline(settings, line))
    {
        outfileData << line << endl;
    }
    outfileData.close();

    real_1d_array buffer = s.c_str();
    return(buffer);
}


#include <iostream>
#include <cmath>

// Include cppOPT header
#include "C:\Users\stepa\OneDrive\Dissertation\AMSM\
CPP\MonteCarloAMSM\cppOpt\inc\cppOpt.h"
//#include "cppOpt\inc\OptBoundaries.h"
```

```
// The function necessary for Simulated Annealing optimizer
using namespace cppOpt;

// Toy example objective fcn
template <typename T>
class MySolver : public OptSolverBase<T>
{
public:
   //define your own calculation
   void calculate(OptCalculation<T> &optCalculation) const
   {
      //defined x^2 as function to be optimized
      optCalculation.result = pow(
      optCalculation.get_parameter("X"), 2);
   }
};

// Toy example objective fcn
template <typename T>
class MySolver2 : public OptSolverBase<T>
{
public:
   void calculate(OptCalculation<T> &optCalculation) const
   {
      double rss = pow(
         optCalculation.get_parameter("lambda") -
         0.4f, 2) +
         pow(optCalculation.get_parameter("nu") -
         0.02f, 2);

      optCalculation.result = (float)rss;
   }
};

// The following functions are based on
// cppOpt header-only numerical library
// containing 4 optimization algorithms:
// Simulated Annealing, Threshold Accepting,
// Great Deluge, EVolutionary.
// The code and manual are available on
// https://github.com/I3ck/cppOpt
using namespace cppOpt;
```

```
// Toy example objective fcn
template <typename T>
class MySolverTest : public OptSolverBase<T>
{
public:
   //define your own calculation
   void calculate(OptCalculation<T> &optCalculation) const
   {
      //defined x^2 as function to be optimized
      optCalculation.result = pow(
         optCalculation.get_parameter("lambda") -
         0.4f, 2) +
         pow(optCalculation.get_parameter("nu") -
         0.02f, 2);
   }
};

using namespace std;

// The function runs Threshold Accepting local search method
void optTA(
   float* optimum, unsigned int maxCalculations,
   int seed, real_1d_array UpBoundary,
   real_1d_array LoBoundary)
{
   // Assign boundary conditions
   OptBoundaries<float> optBoundaries;
   optBoundaries.add_boundary(
      (float)LoBoundary[0], (float)UpBoundary[0],
      "m0");
   optBoundaries.add_boundary(
      (float)LoBoundary[1], (float)UpBoundary[1],
      "sigma");
   optBoundaries.add_boundary(
      (float)LoBoundary[2], (float)UpBoundary[2],
      "rho");

   //instantiate your calculator
   MySolver<float> mySolver;

   //number of calculations
  //  unsigned int maxCalculations = 150;

   //we want to find the minimum
   OptTarget optTarget = MINIMIZE;
```

```
    //how fast the simulated annealing algorithm slows down
    //http://en.wikipedia.org/wiki/Simulated_annealing
    float coolingFactor = 0.95f;//0.95f;

    //the chance in the beginning to follow bad solutions
    float startChance = 0.25f; // 0.25f;

    //the starting threshold
    //should be somewhere close to the difference of
    //BEST_VALUE - WORST_VALUE
    //so 25 - 0 => 25 => 20 in this case
    //http://comisef.wikidot.com/concept:thresholdaccepting
    float threshold = 0.01f;

    //how much the threshold changes each iteration
    //should be similar to the cooling factor
    //http://comisef.wikidot.com/concept:thresholdaccepting
    float thresholdFactor = 3.92f;

    OptThresholdAccepting<float> optTA(optBoundaries,
        maxCalculations,
        &mySolver,
        optTarget,
        0.0, //only required if approaching / diverging
        coolingFactor,
        threshold,
        thresholdFactor);

    //let's go
    OptBase<float>::run_optimisations(1, seed);

    //print result
    OptCalculation<float> best = optTA.get_best_calculation();

    //cout << best.to_string_header() << endl;
    //cout << best.to_string_values() << endl;

    optimum[0] = best.get_parameter("m0");
    optimum[1] = best.get_parameter("sigma");
    optimum[2] = best.get_parameter("rho");
}

// The function runs Great Deluge optimization method
void optGD(
```

```
    float* optimum, unsigned int maxCalculations,
    int seed, real_1d_array UpBoundary,
    real_1d_array LoBoundary)
{
    OptBoundaries<float> optBoundaries;
    optBoundaries.add_boundary(
        (float)LoBoundary[0], (float)UpBoundary[0],
        "m0");
    optBoundaries.add_boundary(
        (float)LoBoundary[1], (float)UpBoundary[1],
        "sigma");
    optBoundaries.add_boundary(
        (float)LoBoundary[2], (float)UpBoundary[2],
        "rho");

    //instantiate your calculator
    MySolver<float> mySolver;

    //number of calculations
//  unsigned int maxCalculations = 150;

    //we want to find the minimum
    OptTarget optTarget = MINIMIZE;

    //how fast the simulated annealing algorithm slows down
    //http://en.wikipedia.org/wiki/Simulated_annealing
    float coolingFactor = 0.95f;//0.95f;

    //the chance in the beginning to follow bad solutions
    float startChance = 0.25f; // 0.25f;

    //the initial water level
    //http://en.wikipedia.org/wiki/Great_Deluge_algorithm
    float waterLevel = 25.0f;

    //how much rain is added to the water level per
    // iteration with x^2 from -5 to +5 the max value
    // is 25, while the min and wanted value is 0 with
    // 300 calculations, the water level should
    // be pretty close to the optimum of 0, 25/300 =>
    // 0.083333 [rain should AT LEAST be that much] =>
    // make it 0.15
    //http://en.wikipedia.org/wiki/Great_Deluge_algorithm
    float rain = 0.15f;
```

```
    // The main function
    OptGreatDeluge<float> optGD(optBoundaries,
        maxCalculations,
        &mySolver,
        optTarget,
        0.0, //only required if approaching / diverging
        coolingFactor,
        waterLevel,
        rain);

    //let's go
    OptBase<float>::run_optimisations(1, seed);

    //print result
    OptCalculation<float> best = optGD.get_best_calculation();

    //cout << best.to_string_header() << endl;
    //cout << best.to_string_values() << endl;

    optimum[0] = best.get_parameter("m0");
    optimum[1] = best.get_parameter("sigma");
    optimum[2] = best.get_parameter("rho");
}

// The function runs Evolutionary optimization method
void optEV(
    float* optimum, unsigned int maxCalculations,
    int seed, real_1d_array UpBoundary,
    real_1d_array LoBoundary)
{
    OptBoundaries<double> optBoundaries;
    optBoundaries.add_boundary(
        (float)LoBoundary[0], (float)UpBoundary[0],
        "m0");
    optBoundaries.add_boundary(
        (float)LoBoundary[1], (float)UpBoundary[1],
        "sigma");
    optBoundaries.add_boundary(
        (float)LoBoundary[2], (float)UpBoundary[2],
        "rho");

    //instantiate your calculator
    MySolver<double> mySolver;

    //we want to find the minimum
```

```
OptTarget optTarget = MINIMIZE;

//how fast the simulated annealing algorithm slows down
//http://en.wikipedia.org/wiki/Simulated_annealing
double coolingFactor = 0.95;//0.95f;

//how many individuals should be spawned in the beginning
//use an even, positiv number
//https://en.wikipedia.org/wiki/Evolutionary_algorithm
unsigned int nIndividualsStart = 500;

//how many individuals shall be selected each generation
//this should also be an even number
//https://en.wikipedia.org/wiki/Evolutionary_algorithm
unsigned int nIndividualsSelection = 20;

//how many children each parent pair should spawn
//use a number > 2
////https://en.wikipedia.org/wiki/Evolutionary_algorithm
unsigned int nIndividualsOffspring = 3;

//how much the offspring should be mutated
//or moved from the center of the parents
//https://en.wikipedia.org/wiki/Evolutionary_algorithm
double mutation = 0.1;

OptEvolutionary<double> optEV(optBoundaries,
    maxCalculations,
    &mySolver,
    optTarget,
    0.0, //only required if approaching / diverging
    coolingFactor,
    nIndividualsStart,
    nIndividualsSelection,
    nIndividualsOffspring,
    mutation);

//let's go
OptBase<double>::run_optimisations(1, seed);

//print result
OptCalculation<double> best = optEV.get_best_calculation();

//cout << best.to_string_header() << endl;
//cout << best.to_string_values() << endl;
```

```
    optimum[0] = (float)best.get_parameter("m0");
    optimum[1] = (float)best.get_parameter("sigma");
    optimum[2] = (float)best.get_parameter("rho");
}

// The version with another boundaries and settings
void optEV2(
    float* optimum, unsigned int maxCalculations,
    int seed, real_1d_array UpBoundary,
    real_1d_array LoBoundary)
{
    OptBoundaries<double> optBoundaries;
    optBoundaries.add_boundary(-5.0, 5.0, "X");

    //instantiate your calculator
    MySolver<double> mySolver;

    //number of calculations
    //unsigned int maxCalculations = 300;

    //we want to find the minimum
    OptTarget optTarget = MINIMIZE;

    //how fast the evolutionary algorithm slows down
    //https://en.wikipedia.org/wiki/Evolutionary_algorithm
    double coolingFactor = 0.95;

    //how many individuals should be spawned in the beginning
    //use an even, positive number
    //https://en.wikipedia.org/wiki/Evolutionary_algorithm
    unsigned int nIndividiualsStart = 50;

    //how many individuals shall be selected each generation
    //this should also be an even number
    //https://en.wikipedia.org/wiki/Evolutionary_algorithm
    unsigned int nIndividualsSelection = 10;

    //how many children each parent pair should spawn
    //use a number > 2
    ////https://en.wikipedia.org/wiki/Evolutionary_algorithm
    unsigned int nIndividualsOffspring = 3;

    //how much the offspring should be mutated
    //or moved from the center of the parents
```

```
    //https://en.wikipedia.org/wiki/Evolutionary_algorithm
    double mutation = 0.1;

    //create your optimizer
    //using the evolutionary algorithm
    OptEvolutionary<double> opt(optBoundaries,
        maxCalculations,
        &mySolver,
        optTarget,
        0.0, //only required if approaching / diverging
        coolingFactor,
        nIndividiualsStart,
        nIndividualsSelection,
        nIndividualsOffspring,
        mutation);

    //enable logging
    //boundaries object required to know
    //the parameters names for the header
    OptBase<double>::enable_logging(
        "example_evolutionary_x_square.log", optBoundaries);

    //let's go
    OptBase<double>::run_optimisations();

    //print result
    OptCalculation<double> best = opt.get_best_calculation();
    if (printout) {
        cout << best.to_string_header() << endl;
        cout << best.to_string_values() << endl;
    }

}

// The version for calibration of risk-premiums
// lambda and nu
void optEV3(
    float* optimum, unsigned int maxCalculations,
    int seed, real_1d_array UpBoundary,
    real_1d_array LoBoundary)
{

    OptBoundaries<double> optBoundaries;
    optBoundaries.add_boundary(
        (float)UpBoundary[0],
```

```
    (float)LoBoundary[0],
    "lambda");
optBoundaries.add_boundary(
    (float)UpBoundary[1],
    (float)LoBoundary[1],
    "nu");

//instantiate your calculator
MySolver2<double> mySolver;

//we want to find the minimum
OptTarget optTarget = MINIMIZE;

//how fast the simulated annealing algorithm slows down
//http://en.wikipedia.org/wiki/Simulated_annealing
double coolingFactor = 0.95;//0.95f;

/how many individuals should be spawned in the beginning
//use an even, positive number
//https://en.wikipedia.org/wiki/Evolutionary_algorithm
unsigned int nIndividualsStart = 500;

//how many individuals shall be selected each generation
//this should also be an even number
//https://en.wikipedia.org/wiki/Evolutionary_algorithm
unsigned int nIndividualsSelection = 20;

//how many children each parent pair should spawn
//use a number > 2
////https://en.wikipedia.org/wiki/Evolutionary_algorithm
unsigned int nIndividualsOffspring = 3;

//how much the offspring should be mutated
//or moved from the center of the parents
//https://en.wikipedia.org/wiki/Evolutionary_algorithm
double mutation = 0.1;

OptEvolutionary<double> optEV(optBoundaries,
    maxCalculations,
    &mySolver,
    optTarget,
    0.0, //only required if approaching / diverging
    coolingFactor,
    nIndividualsStart,
    nIndividualsSelection,
```

```
        nIndividualsOffspring,
        mutation);

    //let's go
    OptBase<double>::run_optimisations(1, seed);

    //print result
    OptCalculation<double> best = optEV.get_best_calculation();

    //cout << best.to_string_header() << endl;
    //cout << best.to_string_values() << endl;

    optimum[0] = (float)best.get_parameter("lambda");
    optimum[1] = (float)best.get_parameter("nu");

}

// Simultaneous perturbation stochastic approximation (SPSA)
// optimization algorithm
using namespace std;

// The function calculates the norm of matrix
float Norm_m(float ** a, int n)
{
    float sum = 0.f;
    for (int i = 0; i <= n - 1; i++)
        for (int j = 0; j <= n - 1; j++)
        {
            sum = sum + a[i][j] * a[i][j];
        }
    return sqrt(sum);
}

// The function calculates  the norm of vector
float Norm_v(float * a, int n)
{
    float sum = 0.f;
    for (int j = 0; j <= n - 1; j++)
    {
        sum = sum + a[j] * a[j];
    }
    return sqrt(sum);
}

// The function calculates the product of
```

```
// vector and matrix
void Product_1(
   float * invB_m, float ** invB,
   float * m, float * m_, int d)
{
   for (int i = 0; i <= d - 1; i++)
   {
      float sum = 0.f;
      for (int j = 0; j <= d - 1; j++)
         sum = sum +
         invB[i][j] * (m[j] - m_[j]);
      invB_m[i] = sum;
   }
}

// The function calculates the product of
// matrix and matrix
void Product_2(
   float ** invB_x, float ** invB, float ** x,
   float * m_, int d, int mu)
{
   for (int k = 0; k<mu; k++)
      for (int i = 0; i<d; i++)
      {
         float sum = 0.f;
         for (int j = 0; j<d; j++)
            sum = sum + invB[i][j] * (x[k][j] - m_[j]);
         invB_x[k][i] = sum;
      }
}

// The function calculates the product of
// matrix and diagonal matrix
void Product_3(
   float ** B, float ** ortB, float ** Diag, int d)
{
   for (int i = 0; i <= d - 1; i++)
      for (int k = 0; k <= d - 1; k++)
      {
         float sum = 0.f;
         for (int j = 0; j <= d - 1; j++)
            sum = sum + ortB[i][j] * Diag[j][k];
         B[i][k] = sum;
      }
}
```

```
// The function calculates the product of
// matrix and vector
void Product_m_v(
   float ** A, float * b, float * res, int d)
{
   for (int i = 0; i<d; i++)
   {
      float sum = 0.f;
      for (int j = 0; j<d; j++)
         sum = sum + A[i][j] * b[j];
      res[i] = sum;
   }
}

// The function calculates the inverse of matrix
void Inverse_matrix(
   float ** B, float ** invB, int d)
{
   // Initialization of A
   alglib::real_2d_array a;
   a.setlength(d, d);
   for (int i = 0;i< d; i++)
   {
      for (int j = 0;j< d; j++)
      {
         a[i][j] = B[i][j];
      }
   }

   // Computation of Inverse matrix
   ae_int_t info;
   matinvreport rep;
   rmatrixinverse(a, info, rep);

   // Filling of matrix invB
   for (int i = 0;i< d; i++)
   {
      for (int j = 0;j< d; j++)
      {
         invB[i][j] = (float)a[i][j];
         //sum += invU[i][j];
      }
   }
}
```

```
// The function calculates the eigen
// decomposition of matrix
void Eigen_decomposition(
   float ** C, float ** ortB, float ** D, int d)
{
   alglib::real_1d_array wr;
   alglib::real_1d_array wi;
   alglib::real_2d_array vl;
   alglib::real_2d_array vr;
   alglib::real_2d_array vr2;

   // Initialization of A
   alglib::real_2d_array a;
   a.setlength(d, d);
   for (int i = 0;i< d; i++)
   {
      for (int j = 0;j< d; j++)
      {
         a[i][j] = (double)C[i][j];
      }
   }

   // Eigen-decomposition
   alglib::rmatrixevd(
      a,//real_2d_array a,
      d,//ae_int_t n,
      1,//ae_int_t v needed,
      wr,
      wi,
      vl,
      vr);  // eigenvectors are columns

   // Filling of matrix of eigen-vectors
   for (int i = 0;i< d; i++)
   {
      for (int j = 0;j< d; j++)
      {
         // in U eigenvectors are rows
         ortB[j][i] = (float)vr[j][i];
         D[j][i] = 0.f;
      }
      D[i][i] = sqrtf((float)wr[i]);
   }
}
```

```
void AdaptiveEncoding(
   float ** B, float ** x, float ** C, float * p,
   float * m, float * w, int mu, int d, int * initialize)
{
   float ** invB = new float *[d];
   for (int i = 0; i<d; i++) invB[i] = new float[d];

   float * invB_m = new float[d];

   float ** invB_x = new float *[mu]; // 2 * d
   for (int i = 0; i<mu; i++) invB_x[i] = new float[d];

   float ** D = new float *[d];
   for (int i = 0; i<d; i++) D[i] = new float[d];

   float * m_ = new float[d];

   float ** z = new float *[d];
   for (int i = 0; i<d; i++) z[i] = new float[d];

   float ** Cmu = new float *[d];
   for (int i = 0; i<d; i++) Cmu[i] = new float[d];

   float c_p = 1.f / sqrtf((float)d);
   float c_1 = 0.5f / (float)d;
   float c_mu = 0.5f / (float)d;

   if (*initialize)
   {
      for (int i = 0; i<mu; i++) { w[i] = 1.f / mu; }
      for (int i = 0; i<d; i++) { p[i] = 0.f; }

      for (int i = 0; i <= d - 1; i++)
      {
         for (int j = 0; j <= d - 1; j++)
         {
            if (i != j) { C[i][j] = 0.f; }
            else { C[i][j] = 1.f; }
            if (i != j) { B[i][j] = 0.f; }
            else { B[i][j] = 1.f; };
         }
      }
```

```
    // for(int i=0; i<mu; i++)
    //     for(int j=0; j<d; j++)
    //         m[j] = m[j] + x[i][j] * w[i];

    *initialize = 0;
    //return 0;
}

for (int i = 0; i <= d - 1; i++)
{
    for (int j = 0; j <= d - 1; j++)
    {
        if (i != j) { Cmu[i][j] = 0.f; }
        else { Cmu[i][j] = 0.f; }
        //if(i!=j){B[i][j] = 0.f;}else{B[i][j] = 1.f;}
    }
}


for (int i = 0; i <= d - 1; i++)
    m_[i] = m[i];

for (int j = 0; j<d; j++) m[j] = 0.f;
for (int i = 0; i<mu; i++)
{
    for (int j = 0; j <= d - 1; j++)
        m[j] = m[j] + x[i][j] * w[i];
}

// inversion of B
Inverse_matrix(B, invB, d);

Product_1(invB_m, invB, m, m_, d);

float norm_1 = Norm_v(invB_m, d);

// Matrix Z
for (int j = 0; j <= d - 1; j++)
    if ((m[j] - m_[j]) != 0.f)
        z[j][0] =
            (sqrtf((float)d) *
            (m[j] - m_[j])) / norm_1;
    else
        z[j][0] = -1.f;
```

```
Product_2(invB_x, invB, x, m_, d, mu);
for (int i = 1; i <= mu - 1; i++)
{
   float norm_2 = 0.f;
   for (int j = 0; j<d; j++)
      norm_2 =
         norm_2 +
         (invB_x[i][j] * invB_x[i][j]);
   norm_2 = sqrt(norm_2);

   for (int j = 0; j<d; j++)
      if ((x[i][j] - m_[j]) != 0.f)
         z[j][i] =
            (sqrtf((float)d) *
            (x[i][j] - m_[j])) /
            norm_2;
      else
         z[j][i] = -1.f;
}

// Eigen-direction
for (int i = 0; i <= d - 1; i++)
   p[i] = (1.f - c_p) * p[i] +
      sqrtf(c_p * (2.f - c_p)) *
      z[i][0];

// Partial covariance matrix
float sum = 0.f;
for (int i = 0; i<mu; i++)
{
   //zz[j][k] = z[j] * z[k];
   for (int j = 0; j<d; j++)
      for (int k = 0; k<d; k++)
         Cmu[j][k] =
            Cmu[j][k] +
            w[i] * z[j][i] * z[k][i];
}

// New covariance matrix
for (int i = 0; i <= d - 1; i++)
   for (int j = 0; j <= d - 1; j++)
   {
      C[i][j] =
         (1.f - c_1 - c_mu) *
         C[i][j] +
```

```
                c_1 * p[i] * p[j] +
                c_mu * Cmu[i][j];
        }

    // Eigen decomposition
    Eigen_decomposition(C, invB, D, d);

    // New B
    Product_3(B, invB, D, d);
    for (int i = 0; i <= d - 1; i++) m[i] = m_[i];

    for (int i = 0; i<d; i++) delete[] invB[i];
    delete[](invB_m);
    for (int i = 0; i<mu; i++)delete[] invB_x[i];

    for (int i = 0; i<d; i++) delete[] D[i];
    delete[](m_);
    for (int i = 0; i<d; i++) delete[] z[i];
    for (int i = 0; i<d; i++) delete[] Cmu[i];

}

void ACiD(int d, int mu, float ** bound)
{

   float k_succ = 1.2f, k_unsucc = 0.5f;

   float ** B =     new float *[d];
   for (int i = 0; i<d; i++) B[i] = new float[d];

   float ** C =     new float *[d];
   for (int i = 0; i<d; i++) C[i] = new float[d];

   float ** x =     new float *[1000];
   for (int i = 0; i<d; i++) x[i] = new float[d];

   float *  p =     new float[d];
   float *  m =     new float[d];
   float *  w =     new float[10];

   float *  sigma = new float[d];
   float *  x_    = new float[d];


   //srand( (unsigned)time( NULL ) );
```

```
for (int i = 0;i<d; i++)
{
   /*float min = x[i][0], max = x[i][0];
   for(int j=0;j<mu; j++)
   {
   if(x[i][j]>max) max = x[i][j];
   if(x[i][j]>max) max = x[i][j];
   }*/
   m[i] = bound[i][0] + (float)rand() /
      (RAND_MAX + 1) * (bound[i][1] - bound[i][0]);

   sigma[i] = (bound[i][1] - bound[i][0]) / 4.f;
}

float f_best = (float)objective_func(
   /*3.f*/m[0], m[1]/*1.4f*/, m[2]/*0.95f*/,
   2/*m[1]*/, 0.02,
   lambda_real, nu_real,
   model,
   NumOfPoints1, NumOfPoints2);

if (printout) {
   cout << "gkk = " << m[0] <<
   " rho = " << m[1] << " gkk = " <<
   m[2] << " rss = " << f_best << endl;
}

int counter = 2;

for (int i = 0; i <= d - 1; i++)
{
   for (int j = 0; j <= d - 1; j++)
   {
      if (i != j) { C[i][j] = 0.f; }
      else { C[i][j] = 1.f; }
      if (i != j) { B[i][j] = 0.f; }
      else { B[i][j] = 1.f; };
   }
}

float f_1 = 0.f, f_2 = 0.f;
float * prod =  new float[d];

int initialize = 1;
```

```
float ** x_a =  new float *[2 * d];
for (int i = 0; i<2 * d; i++)
   x_a[i] = new float[d];

float * x_1 =   new float[d];
float * x_2 =   new float[d];

float * f_a =   new float[2 * d];
//for (int i = 0; i<d; i++)
   f_a[i] = new float [d];

float * buf_x = new float[d];

int i_x = 0;
while (f_best >= 0.000001)
{
   if (i_x == d) i_x = 0;

   //x_a[0][2 * i_x + 1] = 0.f;

   for (int i = 0; i<d;i++) x_[i] = 0.f;

   x_[i_x] = -sigma[i_x];
   Product_m_v(B, x_, prod, d);
   for (int j = 0; j<d; j++)
      x_1[j] = m[j] + prod[j];
   float f_1 = (float)objective_func(
      /*3.f*/x_1[0], x_1[1]/*1.4f*/, x_1[2]/*0.95f*/,
      2/*m[1]*/, 0.02,
      lambda_real, nu_real,
      model,
      NumOfPoints1, NumOfPoints2);

   if (printout) {
      cout << "[" << x_1[0] << ", " <<
      x_1[1] << ", " << x_1[2] <<
      "] rss = " << f_1 << " prod[" <<
      i_x << "] = " << prod[i_x] <<
      " k = " << counter << endl;
      counter = counter + 1;
   }

   x_[i_x] = sigma[i_x];
   Product_m_v(B, x_, prod, d);
   for (int j = 0; j<d; j++)
```

```
    x_2[j] = m[j] + prod[j];
float f_2 = (float)objective_func(
    /*3.f*/x_2[0], x_2[1]/*1.4f*/, x_2[2]/*0.95f*/,
    2/*m[1]*/, 0.02,
    lambda_real, nu_real,
    model,
    NumOfPoints1, NumOfPoints2);

if (printout) {
    cout << "[" << x_2[0] << ", " <<
    x_2[1] << ", " << x_2[2] <<
    "] rss = " << f_2 << " prod[" <<
    i_x << "] = " << prod[i_x] <<
    " k = " << counter << endl;
    counter = counter + 1;
}

int succ = 0;

if (f_1<f_best)
{
    f_best = f_1;
    for (int i = 0; i<d; i++) m[i] = x_1[i];

    if (printout) {
        cout << "gkk = " << m[0] <<
        " rho = " << m[1] <<
        " gkk = " << m[2] <<
        " rss = " << f_1 << endl;
    }
    succ = 1;
}

if (f_2<f_best)
{
    f_best = f_2;
    for (int i = 0; i<d; i++) m[i] = x_2[i];
    if (printout) {
        cout << "gkk = " << m[0] <<
        " rho = " << m[1] <<
        " gkk = " << m[2] <<
        " rss = " << f_2 << endl;
    }
    succ = 1;
}
```

```
if (succ == 1){
   sigma[i_x] = k_succ   * sigma[i_x];
else
{
   sigma[i_x] = k_unsucc * sigma[i_x];
   if (prod[i_x] <= 0.0001)
   {
      initialize = 1;
      sigma[i_x] =
         (bound[i_x][1] -
         bound[i_x][0]) / 4.f;
      if (printout) {
         cout <<
         "Reinitialization!" <<
         endl;
      }
   }
};

for (int j = 0; j<d; j++)
{
   x_a[2 * i_x][j] = x_1[j];
   x_a[2 * i_x + 1][j] = x_2[j];
}
f_a[2 * i_x] = f_1;
f_a[2 * i_x + 1] = f_2;

if (i_x == d - 1)
{
   for (int i = 0; i<2 * d; i++)
      for (int j = 0; j<2 * d - 1; j++)
         if (f_a[j] > f_a[j + 1])
         {
            float buf_f = f_a[j];
            for (int k = 0; k<d; k++)
               buf_x[k] = x_a[j][k];

            f_a[j] = f_a[j + 1];
            for (int k = 0; k<d; k++)
               x_a[j][k] = x_a[j + 1][k];

            f_a[j + 1] = buf_f;
            for (int k = 0; k<d; k++)
               x_a[j + 1][k] = buf_x[k];
```

```
            };

        AdaptiveEncoding(
            B, x_a, C, p, m, w, mu, d, &initialize);
      }
      i_x = i_x + 1;
   }

   for (int i = 0; i<d; i++) delete[] B[i];
   for (int i = 0; i<d; i++) delete[] C[i];
   for (int i = 0; i<d; i++) delete[] x[i];

   delete[](p);
   delete[](m);
   delete[](w);
   delete[](sigma);
   delete[](x_);
   delete[](prod);
   for (int i = 0; i<2 * d; i++) delete[] x_a[i];
   delete[](f_a);
   delete[](x_1);
   delete[](x_2);
   delete[](buf_x);
   delete[](f_a);
}

// The main function
void SPSA(
   float a, float c, float chat, float A,
   float alpha, float gamma,
   float * theta, int n, int d)
{
   float a_k = 0.f, c_k = 0.f, chat_k = 0.f,
   yplus = 0.f, yminus = 0.f, yplus1 = 0.f,
   yminus1 = 0.f, ycenter1 = 0.f;

   float * thetaplus    = new float[d];
   float * thetaminus   = new float[d];
   float * thetaplus1   = new float[d];
   float * thetaminus1  = new float[d];
   float * thetacenter1 = new float[d];

   float * delta        = new float[d];
   float * deltahat     = new float[d];
   float * deltahat2    = new float[d];
```

```
float * ghat          = new float[d];
float * ghat_k        = new float[d];
float * ghat_k1       = new float[d];
float * ghat_plus     = new float[d];
float * ghat_minus    = new float[d];

int k_lag = 5;
float ** ghat_k_lag  = new float *[d];
for (int i = 0; i <= k_lag; i++)
   ghat_k_lag[i] = new float[k_lag];

float ** H       = new float *[d];
for (int i = 0; i<d; i++)
   H[i]        = new float[d];
float ** HT      = new float *[d];
for (int i = 0; i<d; i++)
   HT[i]       = new float[d];
float ** Hbar_k  = new float *[d];
for (int i = 0; i<d; i++)
   Hbar_k[i]  = new float[d];
float ** Hbar_k1 = new float *[d];
for (int i = 0; i<d; i++)
   Hbar_k1[i] = new float[d];
float ** invH    = new float *[d];
for (int i = 0; i<d; i++)
   invH[i]    = new float[d];

// Optimization procedure
for (int k = 0;k <= n;k++)
{
   a_k = a / ((float)pow(A + (float)k + 1.f, alpha));
   c_k = c / ((float)pow((float)k + 1.f, gamma));//}
   chat_k = chat / ((float)pow((float)k + 1.f, gamma));//}

                          // Objective fcn in 4 points
   for (int p = 0;p<d;p++)
   {
      delta[p]     = 2.f * round((float)rand() /
         ((float)RAND_MAX + 1.f)) - 1.f;
      deltahat[p]  = 2.f * round((float)rand() /
         ((float)RAND_MAX + 1.f)) - 1.f;
      deltahat2[p] = 2.f * round((float)rand() /
         ((float)RAND_MAX + 1.f)) - 1.f;
```

```
    thetaplus[p]  = theta[p] + c_k * delta[p];
    thetaminus[p] = theta[p] - c_k * delta[p];
    thetaplus1[p] = theta[p] + c_k * delta[p] +
       chat_k * deltahat[p];
    thetaminus1[p] = theta[p] - c_k * delta[p] +
       chat_k * deltahat[p];
}

if (k != 0) cout << " ";
yminus  = (float)objective_func(3
    /*thetaminus[0]*/, (double)thetaminus[0]/*1.4f*/,
    /*x_2[2]*//*thetaminus[2]*/  0.95,
    0.05/*m[1]*/,
    /*0.02f*/(double)thetaminus[1],
    lambda_real, nu_real,
    model, NumOfPoints1, NumOfPoints2);

if (k != 0) cout << "*";
yplus   = (float)objective_func(
    3/* thetaplus[0]*/,
    (double)thetaplus[0]/*1.4f*/,
    /*x_2[2]*//*thetaplus[2]*/    0.95,
    0.05,
    /*0.02f*/(double)thetaplus[1] ,
    lambda_real, nu_real,
    model,
    NumOfPoints1, NumOfPoints2);
if (k != 0) cout << "*";
yminus1 = (float)objective_func(
    3/*thetaminus1[0]*/,
    (double)thetaminus1[0]/*1.4f*/,
    /*x_2[2]*//*thetaminus1[2]*/0.95,
    0.05/*thetaminus1[3]/*2.f/*m[1]*/,
    (double)thetaminus1[1],
    lambda_real, nu_real,
    model,
    NumOfPoints1, NumOfPoints2);

if (k != 0) cout << "*";
yplus1  = (float)objective_func(
    3/* thetaplus1[0]*/,
    (double)thetaplus1[0]/*1.4f*/,
    /*x_2[2]*//*thetaplus1[2]*/  0.95,
    0.05/*thetaplus1[3]/*2.f/*m[1]*/,
    (double)thetaplus1[1],
```

```
    lambda_real, nu_real,
    model,
    NumOfPoints1, NumOfPoints2);
if (k != 0) cout << "*";

// Derivatives approximations

// Averaging of few SA gradients
int k_average = 2;
if (yminus <= 0.1f) { k_average = 3; }
if (yminus <= 0.01f) { k_average = 4; }
if (yminus <= 0.001f) { k_average = 5; }

for (int p = 0;p<d;p++)
    ghat[p] = (yplus - yminus) /
    (2.f * c_k * delta[p]);

for (int ka = 2; ka <= k_average; ka++)
{
    for (int p = 0;p<d;p++)
    {
        delta[p] = 2.f * round((float)rand() /
            ((float)RAND_MAX + 1.f)) - 1.f;
        thetaplus[p] = theta[p] + c_k * delta[p];
        thetaminus[p] = theta[p] - c_k * delta[p];
    }
    yminus = (float)objective_func(
        3/*thetaminus[0]*/,
        (double)thetaminus[0]/*1.4f*/,
        /*x_2[2]*//*thetaminus[2]*/0.95,
        0.05/*m[1]*/,
        /*0.02f*/(double)thetaminus[1],
        lambda_real, nu_real,
        model,
        NumOfPoints1, NumOfPoints2);
    if (k != 0) cout << "*";
    yplus  = (float)objective_func(
        3/* thetaplus[0]*/,
        (double)thetaplus[0]/*1.4f*/,
        /*x_2[2]*//*thetaplus[2]*/  0.95,
        0.05,
        /*0.02f*/(double)thetaplus[1],
        lambda_real, nu_real,
        model,
        NumOfPoints1, NumOfPoints2);
```

```
    if (k != 0) cout << "*";
    for (int p = 0;p<d;p++)
      ghat[p] = ghat[p] + (yplus - yminus)
      / (2.f * c_k * delta[p]);
  }

  for (int p = 0;p<d;p++)
  {
    ghat_plus[p] = (yplus1 - yplus) /
      (chat_k * deltahat[p]);
    ghat_minus[p] = (yminus1 - yminus) /
      (chat_k * deltahat[p]);
    ghat[p] = ghat[p] / (float)k_average;
  }

  // Initialization of Hessian approximation
  if (k == 0) {
    for (int p1 = 0;p1<d;p1++) {
      for (int p2 = 0;p2<d;p2++) {
        H[p1][p2] = (ghat_plus[p1] - ghat_minus[p1]) /
          (2.f * c_k * delta[p2]
        HT[p2][p1] = H[p1][p2];
      }
    }
    for (int p1 = 0;p1<d;p1++) {
      for (int p2 = 0;p2<d;p2++) {
        Hbar_k1[p1][p2] = (H[p1][p2] + HT[p1][p2]) / 2.f;
      }
    }
  }
  else {
    // Hessian approximation
    for (int p1 = 0;p1<d;p1++) {
      for (int p2 = 0;p2<d;p2++) {
        H[p1][p2] = (ghat_plus[p1] - ghat_minus[p1]) /
          (2.f * c_k * delta[p2] /* chat_k * deltahat[p2]*/);
        HT[p2][p1] = H[p1][p2];
      }
    }
    for (int p1 = 0;p1<d;p1++) {
      for (int p2 = 0;p2<d;p2++) {
        Hbar_k[p1][p2] = ((float)k /
          ((float)k + 1.f)) * Hbar_k1[p1][p2] +
          (1.f / ((float)k + 1.f)) *
          /*(*/H[p1][p2] /*+ HT[p1][p2])/2.f*/;
```

```
            Hbar_k1[p1][p2] = Hbar_k[p1][p2];
         }
      }

      // Inverse of Hessian
      Inverse_matrix(Hbar_k, invH, d);

      // New iteration
      for (int p1 = 0;p1<d;p1++)
      {
         float s = 0.f;
         for (int p2 = 0;p2<d;p2++)
         {
            s = s + invH[p1][p2] * ghat[p2];
         }
         //if ghat[0]<=1.f {ghat[0] = 1.01f}

         theta[p1] = theta[p1] - a_k * s /* ghat[p1]*/;
         if (theta[0] <= 1.f) { theta[0] = 1.01f; }
         if (theta[0] >= 2.f) { theta[0] = 1.99f; }
         if (theta[1] <= 0.0001f) { theta[1] = 0.0001f; }
         if (theta[1] >= 3.0001f) { theta[1] = 3.0001f; }
         if (theta[2] <= 0.f) { theta[2] = 0.01f; }
         if (theta[2] >= 1.f) { theta[2] = 0.99f; }
         //if (theta[3]<=0.f) {theta[3] = 0.0001f;}
      }

      if (printout) {
         cout << " |" << k << "| m_0 = " <<
         theta[0] << " sigma = " << theta[1] <<
         " RSS = " << yminus << " || [" <<
         ghat[0] << ", " << ghat[1] <<
         "]" << endl;
      }
   }
}

delete[](thetaplus);
delete[](thetaminus);
delete[](thetaplus1);
delete[](thetaminus1);
delete[](thetacenter1);

delete[](delta);
delete[](deltahat);
```

```
   delete[](deltahat2);

   delete[](ghat);
   delete[](ghat_k);
   delete[](ghat_k1);
   delete[](ghat_plus);
   delete[](ghat_minus);

   for (int i = 0; i <= k_lag; i++) delete[]
      ghat_k_lag[i];

   for (int i = 0; i<d; i++) delete[] H[i];
   for (int i = 0; i<d; i++) delete[] HT[i];
   for (int i = 0; i<d; i++) delete[] Hbar_k[i];
   for (int i = 0; i<d; i++) delete[] Hbar_k1[i];
   for (int i = 0; i<d; i++) delete[] invH[i];

}

// === Calculation of Likelihood functions L^R and L^O ===
// see Section 2.6.1

// The function returns 1 if two numbers are not
// significantly different, otherwise 0
int Indicator(
   double a_, double b_)
{
   if ( abs(a_ - b_)<0.000001 )
   {
      return(1);
   } else { return(0); }
}

// The function calculates vector of transition probabilities gkk
void vGamma(
   double gkk_, double b_, int model_,
   int kmax_, double * gk)
{

   if (model_ >= 2) {
      for (int k = 0; k < kmax_-1; k++) {
         gk[k] = pow(2, k + 1 - kmax_);}
      gk[kmax_-1] = gkk_;
   }
```

```
    // MSM / AMSM1
    if (model_ <= 1) {
       gk[0] = 1 - pow( 1 - gkk_, pow( b_, 1 - (double)kmax_));
       for (int k = 1; k < kmax_-1; k++) {
          gk[k] = 1 - pow(1 - gk[0], pow( b_, (double)(k + 1 - 1) ));
       gk[kmax_ - 1] = gkk_;
       }  //             then all gk are also == 0 !
    };
}

// The function calculates the probability m^j_k -  m_0
// see the equations (2.62)-(2.64)
double PM(
    double mjk_, double m0_, double rho_,
    double lambda_, int model_, double r_t1_,
    double rf_, double sigma_t1_)
{
    // AMSM2
    if (model_ != 1){ return(0.5); }
    else {
    // AMSM1
       if ( abs(mjk_ - m0_)<(double)0.000001 ) {
          return(
             1 - phi(rho_ * (r_t1_ - rf_ -
             lambda_*sigma_t1_ +
             pow(sigma_t1_, 2)/2) /
             sigma_t1_)); }
       else {
          return(
             phi(rho_ * (r_t1_ - rf_ -
             lambda_* sigma_t1_ +
             pow(sigma_t1_, 2)/2) /
             sigma_t1_)); };
    };
}

// Matrix of possible states of Mkt for kmax up to 5
void mM2_(
    double m0, int kmax_, double ** M,
    string razm, int place)
{
    //int d = (int)pow(2, kmax_);

    if (kmax_ == 1) {
      M[0][0] = m0;
```

```
    M[1][0] = 2 - m0;
  }

  if (kmax_ == 2) {
    M[0][0] = m0;       M[0][1] = m0;
    M[1][0] = m0;       M[1][1] = 2 - m0;
    M[2][0] = 2 - m0;   M[2][1] = m0;
    M[3][0] = 2 - m0;   M[3][1] = 2 - m0;
  }

  if (kmax_ == 3) {
    M[0][0]=m0;     M[0][1]=m0;
     M[0][2]=m0;
    M[1][0]=m0;     M[1][1]=m0;
     M[1][2]=2 - m0;
    M[2][0]=m0;     M[2][1]=2 - m0;
     M[2][2]=m0;
    M[3][0]=2 - m0;M[3][1]=m0;
     M[3][2]=m0;
    M[4][0]=m0;     M[4][1]=2 - m0;
     M[4][2]=2 - m0;
    M[5][0]=2 - m0;M[5][1]=m0;
     M[5][2]=2 - m0;
    M[6][0]=2 - m0;M[6][1]=2 - m0;
     M[6][2]=m0;
     M[7][0]=2 - m0;M[7][1]=2 - m0;
        M[7][2]=2 - m0;
  }

  if (kmax_ == 4) {
     M[0][0] = m0;      M[0][1] = m0;
     M[0][2] = m0;      M[0][3] = m0;
     M[1][0] = m0;      M[1][1] = m0;
     M[1][2] = 2 - m0; M[1][3] = m0;
     M[2][0] = m0;      M[2][1] = 2 - m0;
     M[2][2] = m0;      M[2][3] = m0;
     M[3][0] = 2 - m0; M[3][1] = m0;
     M[3][2] = m0;      M[3][3] = m0;
     M[4][0] = m0;      M[4][1] = 2 - m0;
     M[4][2] = 2 - m0; M[4][3] = m0;
     M[5][0] = 2 - m0; M[5][1] = m0;
     M[5][2] = 2 - m0; M[5][3] = m0;
     M[6][0] = 2 - m0; M[6][1] = 2 - m0;
     M[6][2] = m0;      M[6][3] = m0;
     M[7][0] = 2 - m0; M[7][1] = 2 - m0;
```

```
    M[7][2] = 2 - m0; M[7][3] = m0;

    M[8][0] = m0;       M[8][1] = m0;
    M[8][2] = m0;       M[8][3] = 2-m0;
    M[9][0] = m0;       M[9][1] = m0;
    M[9][2] = 2 - m0;   M[9][3] = 2-m0;
    M[10][0] = m0;      M[10][1] = 2 - m0;
    M[10][2] = m0;      M[10][3] = 2-m0;
    M[11][0] = 2 - m0; M[11][1] = m0;
    M[11][2] = m0;      M[11][3] = 2-m0;
    M[12][0] = m0;      M[12][1] = 2 - m0;
    M[12][2] = 2 - m0; M[12][3] = 2-m0;
    M[13][0] = 2 - m0; M[13][1] = m0;
    M[13][2] = 2 - m0; M[13][3] = 2-m0;
    M[14][0] = 2 - m0; M[14][1] = 2 - m0;
    M[14][2] = m0;      M[14][3] = 2-m0;
    M[15][0] = 2 - m0; M[15][1] = 2 - m0;
    M[15][2] = 2 - m0; M[15][3] = 2-m0;
  }

  if (kmax_ == 5) {
    M[0][0] = m0;      M[0][1] = m0;
    M[0][2] = m0;      M[0][3] = m0;
    M[0][4] = m0;
    M[1][0] = m0;      M[1][1] = m0;
    M[1][2] = 2 - m0; M[1][3] = m0;
    M[1][4] = m0;
    M[2][0] = m0;      M[2][1] = 2 - m0;
    M[2][2] = m0;      M[2][3] = m0;
    M[2][4] = m0;
    M[3][0] = 2 - m0; M[3][1] = m0;
    M[3][2] = m0;      M[3][3] = m0;
    M[3][4] = m0;
    M[4][0] = m0;      M[4][1] = 2 - m0;
    M[4][2] = 2 - m0; M[4][3] = m0;
    M[4][4] = m0;
    M[5][0] = 2 - m0; M[5][1] = m0;
    M[5][2] = 2 - m0; M[5][3] = m0;
    M[5][4] = m0;
    M[6][0] = 2 - m0; M[6][1] = 2 - m0;
    M[6][2] = m0;      M[6][3] = m0;
    M[6][4] = m0;
    M[7][0] = 2 - m0; M[7][1] = 2 - m0;
    M[7][2] = 2 - m0; M[7][3] = m0;
    M[7][4] = m0;
```

```
M[8][0] = m0;       M[8][1] = m0;
M[8][2] = m0;       M[8][3] = 2 - m0;
M[8][4] = m0;
M[9][0] = m0;       M[9][1] = m0;
M[9][2] = 2 - m0;   M[9][3] = 2 - m0;
M[9][4] = m0;
M[10][0] = m0;      M[10][1] = 2 - m0;
M[10][2] = m0;      M[10][3] = 2 - m0;
M[10][4] = m0;
M[11][0] = 2 - m0;  M[11][1] = m0;
M[11][2] = m0;      M[11][3] = 2 - m0;
M[11][4] = m0;
M[12][0] = m0;      M[12][1] = 2 - m0;
M[12][2] = 2 - m0;  M[12][3] = 2 - m0;
M[12][4] = m0;
M[13][0] = 2 - m0;  M[13][1] = m0;
M[13][2] = 2 - m0;  M[13][3] = 2 - m0;
M[13][4] = m0;
M[14][0] = 2 - m0;  M[14][1] = 2 - m0;
M[14][2] = m0;      M[14][3] = 2 - m0;
M[14][4] = m0;
M[15][0] = 2 - m0;  M[15][1] = 2 - m0;
M[15][2] = 2 - m0;  M[15][3] = 2 - m0;
M[15][4] = m0;

M[16][0] = m0;      M[16][1] = m0;
M[16][2] = m0;      M[16][3] = m0;
M[16][4] = 2 - m0;
M[17][0] = m0;      M[17][1] = m0;
M[17][2] = 2 - m0;  M[17][3] = m0;
M[17][4] = 2 - m0;
M[18][0] = m0;      M[18][1] = 2 - m0;
M[18][2] = m0;      M[18][3] = m0;
M[18][4] = 2 - m0;
M[19][0] = 2 - m0;  M[19][1] = m0;
M[19][2] = m0;      M[19][3] = m0;
M[19][4] = 2 - m0;
M[20][0] = m0;      M[20][1] = 2 - m0;
M[20][2] = 2 - m0;  M[20][3] = m0;
M[20][4] = 2 - m0;
M[21][0] = 2 - m0;  M[21][1] = m0;
M[21][2] = 2 - m0;  M[21][3] = m0;
M[21][4] = 2 - m0;
M[22][0] = 2 - m0;  M[22][1] = 2 - m0;
```

```
      M[22][2] = m0;      M[22][3] = m0;
      M[22][4] = 2 - m0;
      M[23][0] = 2 - m0; M[23][1] = 2 - m0;
      M[23][2] = 2 - m0; M[23][3] = m0;
      M[23][4] = 2 - m0;

      M[24][0] = m0;      M[24][1] = m0;
      M[24][2] = m0;      M[24][3] = 2 - m0;
      M[24][4] = 2 - m0;
      M[25][0] = m0;      M[25][1] = m0;
      M[25][2] = 2 - m0; M[25][3] = 2 - m0;
      M[25][4] = 2 - m0;
      M[26][0] = m0;      M[26][1] = 2 - m0;
      M[26][2] = m0;      M[26][3] = 2 - m0;
      M[26][4] = 2 - m0;
      M[27][0] = 2 - m0; M[27][1] = m0;
      M[27][2] = m0;      M[27][3] = 2 - m0;
      M[27][4] = 2 - m0;
      M[28][0] = m0;      M[28][1] = 2 - m0;
      M[28][2] = 2 - m0; M[28][3] = 2 - m0;
      M[28][4] = 2 - m0;
      M[29][0] = 2 - m0; M[29][1] = m0;
      M[29][2] = 2 - m0; M[29][3] = 2 - m0;
      M[29][4] = 2 - m0;
      M[30][0] = 2 - m0; M[30][1] = 2 - m0;
      M[30][2] = m0;      M[30][3] = 2 - m0;
      M[30][4] = 2 - m0;
      M[31][0] = 2 - m0; M[31][1] = 2 - m0;
      M[31][2] = 2 - m0; M[31][3] = 2 - m0;
      M[31][4] = 2 - m0;
   }
// return(M);
}

// This recursive function returns the matrix of possible states
// of Mkt for any kmax
// Allocation with repeats
int count__;
void mM2(float m0, int k, float ** mMi2)
{
   int d  = (int)pow(2, k);
   int d1 = (int)pow(2, k - 1);
   float ** mMi1 = new float *[d1];
   for (int i = 0; i<d1; i++) mMi1[i] = new float[k-1];
```

```
    if (k > 1) {
        mM2(m0, k - 1, mMi1);

        for (int i = 0; i < d1; i++) {
            for (int j = 0; j < k-1; j++) {
                mMi2[i][j] = mMi1[i][j];
                mMi2[d1 + i][j] = mMi1[i][j];
            }
            mMi2[i][k - 1] = m0;
            mMi2[d1 + i][k - 1] = 2 - m0;
        }
    }
    else {
        mMi2[0][0] = m0;
        mMi2[1][0] = 2-m0;
    }

    for (int i = 0; i<d1; i++)
        delete[] mMi1[i];
}

// The function calculates the transition probabilities of
// {Mt} from the state m^i to the state m^j,
// see Equation (2.61)
void mP(
    double m0_, double rho_, double lambda_,
    int model_, int kmax_, double r_t1_, double rf_,
    double * sigma_t1_, float ** m_,
    double * gamma_, double ** P)
{
    //m = mM(m0_, kmax_)
    int d = (int)pow(2, kmax_);
    //gamma = vGamma(0.95, 3, model_, kmax_)

    for (int i = 0; i < d; i++) {
        for (int j = 0; j < d; j++) {
            P[i][j] = 1;
            for (int k = 0; k < kmax_; k++) {
                P[i][j] =
                    P[i][j] *
                    ((double)Indicator(
                        (double)m_[i][k],
                        (double)m_[j][k]) *
                    (1 - gamma_[k]) +
                    gamma_[k] *
```

```
                PM(
                    (double)m_[j][k],
                    m0_, rho_, lambda_,
                    model_,
                    r_t1_, rf_,
                    sigma_t1_[i]));
            }
        }
    };
}

// The function calculates the returns density function omega^{ij}_t
// see Equations (2.52-2.60)
void mOmega(
    double m0_, double sigma0_, double rho_, double lambda_,
    int model_, int kmax_, double * r, int t, double rf_,
    double * sigma_t1_, float ** m_, double ** omega)
{
    //Number of states of Mt
    int d = (int)pow(2, kmax_);

    //ofstream outOmega("Omega.txt");

    // Omega values for each possible transition from i to j
    for (int i = 0; i < d; i++) {
        for (int j = 0; j < d; j++) {

            // The product of all m_k components
            double prod = 1;
            for (int l = 0; l < kmax_; l++) {
                prod = prod * (double)m_[j][l]; }

            double sigma_t_ = 0.0;
            if (model_ != 2) {
                // AMSM1 (eq.2.60)
                sigma_t_ = sqrt(prod)*sigma0_;
            } else {
                // AMSM2 (eq.2.58)
                sigma_t_ =
                    sqrt(prod)*pow(
                        rho_*   (r[t - 1] - rf_ -
                        lambda_*sigma_t1_[i] +
                        pow(sigma_t1_[i], 2)/2) /
                        sigma_t1_[i] -
                        sqrt(sigma0_), 2);
```

```
            }
            // Eq.2.56
            omega[i][j] = dnorm(
                r[t],
                rf_ + lambda_*sigma_t_ - pow(sigma_t_,2)/2,
                sigma_t_);
        }
    }
}

// The function calculates log-likelihood function L^R
// see Equations (2.71)-(2.72)
double AMSMlikRet(
    double m0__, double sigma0__, double rho__, double lambda__,
    int model__, int kmax__, double * r__, double rf__, int tmax)
{
    // Memory allocation and variables initialization
    int       d     = (int)pow(2, kmax__);

    // Vector of transition probabilities of volatility
    // frequency components g_k
    double * gamma = new double[kmax__];
    vGamma(0.95, 3, model__, kmax__, gamma);

    // Matrix of all states of volatility
    // frequency components of Mkt
    float ** m    = new float *[d];
    for (int i = 0; i<d; i++) m[i] = new float[kmax__];

    mM2((float)m0__, kmax__, m);

    double * sigma_t1 = new double[d];
    for (int i = 1;i <= d;i++) {
        sigma_t1[i - 1] = sigma0__; };
    double * sigma_t  = new double[d];
    for (int i = 1;i <= d;i++) {
        sigma_t[ i - 1] = sigma0__; };

    d = (int)pow(2, kmax__);
    double ** PI      = new double *[tmax + 1];
    for (int i = 0; i<tmax + 1; i++){
        PI[i] = new double[d];
    }

    double ** P     = new double *[d];
```

```
for (int i = 0; i<d; i++) P[i]     = new double[d];

double ** Omega = new double *[d];
for (int i = 0; i<d; i++) Omega[i] = new double[d];

// Initialization of PI^i_t (see eq.(2.65))
for (int j = 0; j < d; j++) {
   double prod__ = 1;
   for (int k = 0; k < kmax__; k++) {
      prod__ = prod__ * PM(
         (double)m[j][k], m0__, rho__, lambda__,
         model__, rf__, rf__, sigma0__);}
   PI[0][j]  = prod__;
};


double lik    = -1000000001;
int first_lik = true;
// Logging
//ofstream outDen("Den.txt");
//ofstream outPI("PI.txt");

for (int t = 1; t < tmax; t++) {
   // Sigma_t vector calculation
   for (int i = 0; i < d;i++) {
      double prod = 1;
      for (int l = 0; l < kmax__; l++) {
         prod = prod * (double)m[i][l]; }
      if (model__ != 2) {
         sigma_t[i] = sqrt(prod)*sigma0__; }
      else {
         sigma_t[i] = sqrt(prod)*pow(
            rho__ * (r__[t - 1] - rf__ -
            lambda__*sigma_t1[i] +
            pow(sigma_t1[i], 2)/2) /
            sigma_t1[i] - sqrt(sigma0__), 2); }
   }

   // Omega_t and P_t matrices calculation
   mOmega(
      m0__, sigma0__, rho__, lambda__, model__, kmax__,
      r__, t, rf__, sigma_t1, m,
      Omega);
   mP(
      m0__, rho__, lambda__, model__, kmax__,
```

```
   r__[t - 1], rf__, sigma_t1, m,
   gamma, P);

// PI_t formula numerators and denominator calculation
double Den = 0;
for (int j = 0; j < d; j++) {
   PI[t][j] = 0;
   for (int i = 0; i < d; i++) {
      PI[t][j] = PI[t][j] + PI[t-1][i] * P[i][j] * Omega[i][j]; }
   Den = Den + PI[t][j];
}
//outDen << Den << endl;
if (Den != Den) { Den = 0; };

// PI_t calculation
for (int j = 0; j < d; j++) {
   if (Den != 0) { PI[t][j] = PI[t][j] / Den; }
   else {          PI[t][j] = PI[0][j]; }
   //outPI << PI[t][j] << ",";
}
//outPI << endl;


// Likelihood calculation
//if (lik >= 0.000000001) { lik = lik + log(Den); }
if (Den > 0) {
   if (first_lik) {
      lik = log(Den);
      first_lik = false;
   }
   else {
      lik = lik + log(Den);
   }
}

// Re-initialization if all PI_t^j are zeros
boolean flag = true;
for (int j = 0;j < d; j++) {
   if (PI[t][j] != 0 ) { flag = false; } }
if (flag) { for (int j = 0;j < d; j++) {
   PI[t][j] = PI[0][j]; }; }


for (int j = 0;j < d; j++) {
   sigma_t1[j] = sigma_t[j];     }
```

```
    }
    //outDen.close();
    //outPI.close();

     // Cleaning
     delete[]( gamma );
    for (int i = 0; i<d; i++)
       delete[] m[i];
    delete[]( sigma_t1 );
    delete[]( sigma_t );
    for (int i = 0; i<tmax + 1; i++)
       delete[] PI[i];
    for (int i = 0; i<d; i++)
       delete[] P[i];
    for (int i = 0; i<d; i++)
       delete[] Omega[i];

    return(-lik);
}

// The function calculates likelihood L^O,
// see Equations (2.75-2.78), weights are option prices
double AMSMlikOpt(
    double m0,   double sigma0,   double rho,
    double lambda_, double nu_,  int model)
{
    double rss = 0.f; double y_MC = 0; double fi = 0;

    for (int i = NumOfPoints1; i <= NumOfPoints2; i++)
    {
       // Option price
       y_MC = function_temp(
          (float)3.0, (float)m0, (float)0.95, (float)rho,
          (float)sigma0, (float)lambda_, (float)nu_,
          model, (float)i);
       // WRSS
       fi = (y_MC - y[i - 1]) / y[i - 1];
       rss  = rss + fi * fi;
    }

    double lik =
       - (NumOfPoints2 - NumOfPoints1 + 1)/2 *
       ( log( rss/(NumOfPoints2 - NumOfPoints1 + 1) ) + 1 );
```

```
    return(-lik);
}

// The function calculates likelihood L^O,
// see Equations (2.75-2.78), weights are Black-Scholes Vegas
double AMSMlikOptBSV(
    double m0, double sigma0, double rho,
    double lambda_, double nu_, int model)
{
    double rss = 0.f; double y_MC = 0; double fi = 0;
    float strike; int T; float interest;
    double BSV;

    for (int i = NumOfPoints1; i <= NumOfPoints2; i++)
    {
        y_MC = function_temp(
            (float)3.0, (float)m0, (float)0.95, (float)rho,
            (float)sigma0, (float)lambda_, (float)nu_, model, (float)i);

        DataPoint(i, &T, &strike, &interest);
        BSV = BS_Call_Option_Vega(
            (double)initPrice, (double)strike, interest, sigma0, (double)T);
        fi = (y_MC - y[i - 1]) / BSV;
        rss = rss + fi * fi;
    }

    double lik =
        -(NumOfPoints2 - NumOfPoints1 + 1) / 2 *
        (log(rss / (NumOfPoints2 - NumOfPoints1 + 1)) + 1);

    return(-lik);
}

// AMSM paths simulation, the sequences of uniform
// and Gaussian random numbers are generated outside
void AMSM(
    int NumberOfObservations, double x0, double rf,
    double m0, double sigma0, double rho, double lambda,
    double b, double gkk, int model, int kmax,
    double * urv, double * nrv, double * x)
{
    // Initialization
    int n_ = NumberOfObservations + 1;
     double * sigma = new double[n_];
    double * gk    = new double[kmax];
```

```
vGamma(0.95, 3, model, kmax, gk);

x[0]          = x0;
sigma[0]      = sigma0;

// Switching probabilities
double * Next = new double[kmax];
for (int i = 0;i < kmax;i++) {
   Next[i] = 1; };
double * Prev = new double[kmax];
for (int i = 0;i < kmax;i++) {
   Prev[i] = 1; };

//ofstream sample("sample.csv");// Logging

for (int j = 0; j < NumberOfObservations-1; j++) {
   // MSM or AMSM1
   if (model <= 1) {
      sigma[j + 1] = sigma0; }
   // AMSM2
   if (model == 2) {
      sigma[j + 1] =
         pow(rho * (nrv[j] /*- lambda*/) -
         sqrt(sigma0), 2); }
   // AMSM3
   //if (model == 3) {
      //sigma[j + 1] =
         //abs(rho * (nrv[j] - lambda) -
         // sqrt(sigma[0])); }
   //if (model == 4) {
      //sigma[j + 1] =
         //alpha * (mu - sigma[j]) +
         //pow(rho * (nrv[j] - lambda) -
         //sqrt(sigma[0])), 2); }

   // Calculate sqrt of product of M_i,t
   for (int i = 0; i < kmax; i++) {
      if (urv[j * 2 * kmax + i] <= gk[i])
      {
         // MSM or AMSM2
         if (model != 1) {
            if (urv[j * 2 * kmax + i + kmax] <= 0.5){
               Next[i] = m0; }
            else {
               Next[i] = 2 - m0; } }
```

```
            // AMSM1
            if (model == 1) {
               if (urv[j * 2 * kmax + i + kmax] <= 1 - phi(
                     rho * (nrv[j] /*- lambda*/))) {
                  Next[i] = m0; }
               else {
                  Next[i] = 2 - m0; } }
         }
         else { Next[i] = Prev[i]; }

         Prev[i] = Next[i];
         sigma[j + 1] = sigma[j + 1] * sqrt(Next[i]);
      }
      // Calculate return
      x[j + 1] = x[j] * exp(
         rf + lambda*sigma[j + 1] -
         sigma[j + 1]*sigma[j + 1]/2.0 +
         nrv[j + 1] * sigma[j + 1]);

      //sample << x[j] << endl;
   }
   //sample.close();

   // Memory clean up
   delete[](sigma);
   delete[](gk);
   delete[](Prev);
   delete[](Next);
}

// Wrapper to the function AMSM,
// which generates uniform and Gaussian sequences
void AMSM_logreturns(
   int n_, double m0_, double sigma0_, double rho_,
   double lambda_, int model_, int kmax_,
   double initPrice, double rf_, double * df_)
{
   int n__ = n_ + 1;
   double * x_amsm = new double[n__];
   int n___ = (n_ + 2) * 2 * kmax_;
    double * urv_ = new double[n___];
   double * nrv_ = new double[n_+1];

   ////////////////////////////
   // Random Sample Generation
```

```
boost::random::mt19937 gen;
boost::random::mt19937 gen2; // seed it once

//seed = PrimesSampleArr[SeedNumber - 1];
// 2834947879; //4294967291; ////2984140826;
//1 586 349 558;  //4294967291; //3715061396;

// Use the seed from the array PrimesSampleArr
// in order to provide different sample paths
// for Monte Carlo experiments
gen.seed(  PrimesSampleArr[SeedNumber - 1] );
gen2.seed( PrimesSampleArr[SeedNumber] );

boost::random::uniform_int_distribution<> ud(1, 45000000);
boost::random::normal_distribution<double> nd(0.0, 1.0);

// Initialize two Mersenne-Twister generators
// the first for uniform sequence, another for Gaussian
boost::variate_generator<
   boost::random::mt19937&,
   boost::random::uniform_int_distribution<>>
      randUniform( gen, ud );
boost::variate_generator<
   boost::random::mt19937&,
   boost::normal_distribution<double>>
      randNormal(  gen2, nd );

for (int j = 0; j < n_+1; j++){
   nrv_[j] = randNormal(); }

for (int j = 0; j < (n_ + 1) * 2 * kmax_ + 1; j++) {
   urv_[j] = (double)randUniform() / 45000000; }

AMSM(
   n_+1, initPrice, rf_, m0_, sigma0_, rho_, lambda_, 3, 0.95,
   model_, kmax_, urv_, nrv_, x_amsm);

// Write the simulated sample path to the file
std::ofstream outfileData(
   getexepath().append(
      "\\Simulations\\RealDataSimulated.txt").c_str());
string s = "[";
for (int j = 0; j < n_; j++) {
   df_[j] = log( x_amsm[j + 1] / x_amsm[j] );
```

```
        if (j != n_-1) { s = s + FloatToStr((float)df_[j]) + ","; }
        else { s = s + FloatToStr((float)df_[j]); };
    }
    s = s + "]\n";

    outfileData << s << endl << endl;
    outfileData.close();

    // Memory clean up
    delete[](x_amsm);
    delete[](urv_);
    delete[](nrv_);
}

// Generates artificial prices of options cross section,
// based on an assumption the process is AMSM process
//  for the array defined in the function DataPoint,
// write them in the file
double Cross_section_prices(
    double b, double m0, double gkk,
    double rho, double sigma,
    double lambda_, double nu_,
    int model, int from, int to)
{
    // Initialization of variables
    int Maturity = 0;
    float Strike = 0, Interest = 0;
    double y_MC = 0, rss = 0, Time = 0;
    _int64 t1 = (_int64)0;
    _int64 TotalTimer = (_int64)0;

    // Start timer
    StartTimer(&TotalTimer);

    // Open the file for write
    std::ofstream outfileData(
        getexepath().append(
            "\\Simulations\\Cross_section_prices.txt").c_str());

    // Simulate option prices and write to the file
    for (int i = from; i <= to; i++)
    {
        StartTimer(&t1);
```

```
        // Calculate the option price
        y_MC = (double)function_temp(
            (float)b, (float)m0, (float)gkk, (float)rho,
            (float)sigma, (float)lambda_, (float)nu_,
            model, (float)i);

        Time = StopTimer(t1);
        // WRSS
        rss = rss + (y_MC - y[i - 1]) * (y_MC - y[i - 1]) /
            (y[i - 1] * y[i - 1]);

        // Read Maturity, Strike, Interest
        DataPoint(i, &Maturity, &Strike, &Interest);

        // Write them to the file
        outfileData<< i  << "," << y_MC << "," << y[i-1] <<
        "," << Maturity << "," << Strike << "," << Interest <<
        "," << Time << endl;
    };

    outfileData << rss << "," << "" << "," << "" << "," <<
    "" << "," << StopTimer(TotalTimer) << endl;

    outfileData.close();

    return(0);
}

// Generates artificial prices of options cross section,
// based on an assumptiom the proces is AMSM process
//  for the array defined in the function DataPoint,
double Cross_section_prices2(
    double b, double m0, double gkk,
    double rho, double sigma,
    double lambda_, double nu_,
    int model, int from, int to,
    ofstream &simulations )
{
    // Initialization of variables
    int Maturity = 0;
    float Strike = 0, Interest = 0;
    double y_MC = 0, rss = 0, Time = 0;
    _int64 t1 = (_int64)0;
    _int64 TotalTimer = (_int64)0;
```

```
    // Start timer
    StartTimer(&TotalTimer);

    // Simulate option prices and write to the file
    for (int i = from; i <= to; i++)
    {
        StartTimer(&t1);

        y_MC = (double)function_temp(
            (float)b, (float)m0, (float)gkk, (float)rho,
            (float)sigma, (float)lambda_, (float)nu_,
            model, (float)i);

        Time = StopTimer(t1);
        rss = rss + (y_MC - y[i - 1]) * (y_MC - y[i - 1]) /
            (y[i - 1] * y[i - 1]);

        // Read Maturity, Strike, Interest
        DataPoint(i, &Maturity, &Strike, &Interest);

        simulations << i << "," << y_MC << "," << y[i - 1] <<
        "," << Maturity << "," << Strike << "," << Interest <<
        "," << Time << endl;
    };

    // Write the rss and time consumption in experiment log
    simulations << rss << "," << "" << "," << "" << "," <<
    "" << "," << StopTimer(TotalTimer) << endl;

    return(0);
}

// The wrapper to various objective functions and estimated
// model parameters
//std::ofstream outfileData(getexepath().append(
    //"\\Simulations\\RSS.txt").c_str());
double objective_func(
    double b, double m0, double gkk, double rho, double sigma,
    double lambda_, double nu_, int model, int from, int to)
{
    // Initialization
    double ss = 0; double rss = 0, y_MC = 0;

    // Boundaries of parameters values
    if ((m0       <= LoBoundary[0]) || (m0       >= UpBoundary[0])) {
```

```
    rss = 200000;  return(rss); }
if ((sigma   <= LoBoundary[1]) || (sigma   >= UpBoundary[1])) {
    rss = 200000;  return(rss);}
if ((rho     <  LoBoundary[2]) || (rho     >= UpBoundary[2])) {
    rss = 200000;  return(rss);}
if (Par_number >= 4) {
    if (!lambda_external) {
        if ((lambda_ < LoBoundary[3]) || (lambda_ >= UpBoundary[3])) {
            rss = 200000;  return(rss); }
    } else {
        if ((nu_     < LoBoundary[3]) || (nu_     >= UpBoundary[3])) {
            rss = 200000;  return(rss); } }
}
if (Par_number >= 5) {
    if ((nu_     < LoBoundary[4]) || (nu_     >= UpBoundary[4])) {
        rss = 200000;  return(rss); }
}

//  ==Objective function choice ==
// L^R likelihood based on log-returns
if (ObjFcn.find("likRet") != std::string::npos) {
    rss = AMSMlikRet(
        m0, sigma, rho, lambda_,
        model, khat, r, (double)interest, path_length);
};

// L^O likelihood based on option prices
if ((ObjFcn.find("likOpt") != std::string::npos) &&
    (ObjFcn.find("likOptBSV") == std::string::npos))  {
    rss = AMSMlikOpt(    m0, sigma, rho, lambda_, nu_, model);
}

// L^O likelihood based on option prices with
// Black-Scholes Vegas as weights
if (ObjFcn.find("likOptBSV") != std::string::npos) {
    rss = AMSMlikOptBSV( m0, sigma, rho, lambda_, nu_, model);
}

// L^M likelihood based on log-returns and option prices
if ((ObjFcn.find("likMixed") != std::string::npos) &&
(ObjFcn.find("likMixedBSV") == std::string::npos)) {
    double rss1 = (double)0;

    // Run calculation of L^R  likelihood in parallel
    std::future<double> ret = std::async(
```

```
        AMSMlikRet, m0, sigma, rho, lambda_,
        model, khat, r, interest, path_length);

    // Calculation of L^O likelihood
    double rss2 = AMSMlikOpt(
        m0, sigma, rho, lambda_, nu_, model);
    rss1 = ret.get();

    if (likMixed == 1) {
        rss =
            (path_length + NumOfPoints2 - NumOfPoints1 + 1) /
            (2 * path_length)                    * rss1 +
            (path_length + NumOfPoints2 - NumOfPoints1 + 1) /
            (2 * (NumOfPoints2 - NumOfPoints1 + 1)) * rss2;
    }
    if (likMixed == 2) {
        rss = rss1 / path_length + rss2 /
            (NumOfPoints2 - NumOfPoints1 + 1);
    }
    if (likMixed == 3) {
        rss = rss1 + rss2;
    }
    cout << "rss1 " << rss1 << "  rss2 " <<
    rss2 <<"  rss " << rss << endl;
}

// L^M likelihood based on log-returns and option prices with
// Black-Scholes Vegas as weights
if (ObjFcn.find("likMixedBSV") != std::string::npos) {
    // Calculate L^R likelihood
    double rss1 = AMSMlikRet(
        m0, sigma, rho, lambda_,
        model, khat, r, (double)interest, path_length);
    // Calculate L^O likelihood
    double rss2 = AMSMlikOptBSV(
        m0, sigma, rho, lambda_, nu_,
        model);

    // Various methods to construct mixed likelihood
    if (likMixed == 1) {
        rss =
            (path_length + NumOfPoints2 - NumOfPoints1 + 1) /
            (2 * path_length) * rss1 +
            (path_length + NumOfPoints2 - NumOfPoints1 + 1) /
            (2 * (NumOfPoints2 - NumOfPoints1 + 1)) * rss2;
```

```
    }
    if (likMixed == 2) {
       rss =
          rss1 / path_length +
          rss2 / (NumOfPoints2 - NumOfPoints1 + 1);
    }
    if (likMixed == 3) {
       rss = rss1 + rss2;
    }
 }

 // Weighted Sum of Squared Residuals as objective fcn
 if (ObjFcn.find("WRSS") != std::string::npos) {
    if (!(Method.find("LM") != std::string::npos)) {
       // Calculate whole cross-section basket of option prices and
       // sum up squared residuals
       for (int i = from; i <= to; i++)
       {
          y_MC = (double)function_temp(
             (float)b, (float)m0, (float)gkk, (float)rho, (float)sigma,
             (float)lambda_, (float)nu_, model, (float)i);
          if (y_MC == 0.0) y_MC = 20000;
          rss =
             rss +
             (y_MC - y[i - 1]) * (y_MC - y[i - 1]) /
             (y[i - 1] * y[i - 1]);
       };
    } else {
       // Calculate the only option with the number "from"
       // in the basket for Levenberg-Marquardt optimizer
          rss = (double)function_temp(
             (float)b, (float)m0, (float)gkk, (float)rho, (float)sigma,
             (float)lambda_, (float)nu_, model, (float)from);
    }
 }

 // Root Mean Square Error as objective fcn
 if (ObjFcn.find("RMSE") != std::string::npos) {
    if (!(Method.find("LM") != std::string::npos)) {
       double * y_MC_arr = new double[to - from + 1];
       double * y_arr    = new double[to - from + 1];
       // Calculate whole cross-section baasket of option prices and
       // sum up squared residuals
       for (int i = from; i <= to; i++)
       {
```

```
            y_MC_arr[i - from] = (double)function_temp(
                (float)b, (float)m0, (float)gkk, (float)rho, (float)sigma,
                (float)lambda_, (float)nu_, model, (float)i);
            if (y_MC == 0.0) y_MC = 20000;
            y_arr[i - from] = y[i - 1];
        };
        rss = RMSE(y_arr, y_MC_arr, to - from + 1);

        delete[] y_MC_arr;
        delete[] y_arr;
    }
    else {
        // Calulate the only option with the number "from"
        // in the basket for Levenberg-Marquardt optimizer
        rss = (double)function_temp(
            (float)b, (float)m0, (float)gkk, (float)rho, (float)sigma,
            (float)lambda_, (float)nu_, model, (float)from);
    }
}

// Print on screen
if (Method.find("LM") != std::string::npos) {
    //counter1 = counter1 + 1;
    //if ((counter1) % 8 == 0) {
        //counter1 = counter1 + 1;
        //std::cout << " m0 = " << m0 << " sigma = " <<
        //sigma << " rho = " << rho << " model = " <<
        // model << " RSS = " << rss << " k = " <<
        // counter1 << " Time = " << StopTimer(Timer) << endl;
    //}
}
else {
    if (Method.find("BFGS") != std::string::npos) {
        counter1 = counter1 + 1;

        if (printout) {
            if (Par_number == 3) {
                if ((counter1 - 1) % 13 == 0) {
                    std::cout << " m0 = " << m0 << " sigma = " << sigma <<
                    " rho = " << rho << " lambda = " << lambda_ <<
                    " nu = " << nu_ << " model = " << model <<
                    " RSS = " << rss << " k = " << counter1 <<
                    " Time = " << StopTimer(Timer) << endl;
                };
            }
```

```
            if (Par_number == 4) {
               if ((counter1 - 1) % 17 == 0) {
                  std::cout << " m0 = " << m0 << " sigma = " << sigma <<
                  " rho = " << rho << " lambda = " << lambda_ <<
                  " nu = " << nu_ << " model = " << model <<
                  " RSS = " << rss << " k = " << counter1 <<
                  " Time = " << StopTimer(Timer) << endl;
               };
            }
            if (Par_number == 5) {
               if ((counter1 - 1) % 21 == 0) {
                  std::cout << " m0 = " << m0 << " sigma = " << sigma <<
                  " rho = " << rho << " lambda = " << lambda_ <<
                  " nu = " << nu_ << " model = " << model <<
                  " RSS = " << rss << " k = " << counter1 <<
                  " Time = " << StopTimer(Timer) << endl;
               };
            }
         }
      }
      else {
             counter1 = counter1 + 1;
            if (printout) {
               std::cout << " m0 = " << m0 << " sigma = " << sigma <<
               " b = " << b << " gkk = " << gkk << " rho = " << rho <<
               " lambda = " << lambda_ << " nu = " << nu_ <<
               " model = " << model << " RSS = " << rss <<
               " k = " << counter1 << " Time = " <<
               StopTimer(Timer) << endl;
            }
      }
   }

   if (!_isnan(rss)) { return rss; }
   else {              return 200000; }
}

// The wrapper to various objective functions and estimated
// risk-premiums
double objective_func2(
   double b, double m0, double gkk, double rho, double sigma,
   double lambda_, double nu_, int model, int from, int to)
{
   // Initialization
   double ss = 0; double rss = 0, y_MC = 0;
```

```
// Boundaries
if (Par_number == 1) {
   if (lambda_external) {
      if ((lambda_  < LoBoundary2[0]) || (lambda_ > UpBoundary2[0])) {
         rss = 100000;  return(rss); }
   }
   else {
      if ((nu_      < LoBoundary2[0]) || (nu_     > UpBoundary2[0])) {
         rss = 100000;  return(rss); }
   }
}
if (Par_number >= 2) {
      if ((lambda_  < LoBoundary2[0]) || (lambda_ >  UpBoundary2[0])) {
         rss = 100000;  return(rss); }
      if (( nu_     < LoBoundary2[1]) || (nu_     >  UpBoundary2[1])) {
         rss = 100000;  return(rss); }
}

// L^O likelihood based on option prices as objective function
if (ObjFcn2.find("likOpt") != std::string::npos) {
   rss = AMSMlikOpt(m0, sigma, rho, lambda_, nu_, model);
}

// L^O likelihood based on option prices as objective function
// with Black-SCholes Vegas as weights
if (ObjFcn2.find("LikOptBSV") != std::string::npos) {
   rss = AMSMlikOptBSV(m0, sigma, rho, lambda_, nu_, model);
}

// Weighted Sum of Squared Residuals as objective fcn
if (ObjFcn2.find("WRSS") != std::string::npos) {
   if (!(Method2.find("LM") != std::string::npos)) {
      // Calculate whole cross-section basket of option prices and
      // sum up squared residuals
      for (int i = from; i <= to; i++)
      {
         y_MC = (double)function_temp(
            (float)b, (float)m0, (float)gkk, (float)rho, (float)sigma,
            (float)lambda_, (float)nu_, model, (float)i);
         rss = rss +
            (y_MC - y[i - 1]) * (y_MC - y[i - 1]) / (y[i - 1] * y[i - 1]);
      };
   }
   else {
```

```
        // Calculate the only option with the number "from"
        // in the basket for Levenberg–Marquardt optimizer
        rss = (double)function_temp(
           (float)b, (float)m0, (float)gkk, (float)rho, (float)sigma,
           (float)lambda_, (float)nu_, model, (float)from);

     }
  }

  // Print on screen
  if (Method2.find("LM") != std::string::npos) {
     //counter1 = counter1 + 1;
     //if ((counter1) % 8 == 0) {
     //counter1 = counter1 + 1;
     //std::cout << " m0 = " << m0 << " sigma = " <<
     // sigma << " rho = " << rho << " model = " <<
     // model << " RSS = " << rss << " k = " <<
     // counter1 << " Time = " << StopTimer(Timer) << endl;
     //}
  }
  else {
     if (Method2.find("BFGS") != std::string::npos) {
        counter1 = counter1 + 1;

        if (printout) {
           if (Par_number == 1) {
              if ((counter1 - 1) % 5 == 0) {
                 std::cout << " lambda = " << lambda_ <<
                 " nu = " << nu_ << " model = " << model <<
                 " RSS = " << rss << " k = " << counter1 <<
                 " Time = " << StopTimer(Timer) << endl;
              };
           }
           if (Par_number == 2) {
              if ((counter1 - 1) % 9 == 0) {
                 std::cout << " lambda = " << lambda_ <<
                 " nu = " << nu_ << " model = " << model <<
                 " RSS = " << rss << " k = " << counter1 <<
                 " Time = " << StopTimer(Timer) << endl;
              };
           }
        }
     }
     else {
          counter1 = counter1 + 1;
```

```
            if (printout) {
               std::cout << " lambda = " << lambda_ <<
               " nu = " << nu_ << " model = " << model <<
               " RSS = " << rss << " k = " << counter1 <<
               " Time = " << StopTimer(Timer) << endl;
            }
      }
   }

   if (!_isnan(rss)) { return rss; }
   else { return 100000; }
}

// The function parses the file with settings
int ParseSettings( ofstream &simulations, ifstream &settings)
{
   try { settings.open(getexepath().append("\\Settings.ini").c_str()); }
   catch (ios_base::failure e) {
      cout << "No Settings.ini file found! Exception
      opening/reading/closing file!\\\\n";
      getch();
      return 0;
   }

   string line;
   while (getline(settings, line))
   {
      istringstream is_line(line);
      string key;
      if (getline(is_line, key, ' '))
      {
         string value;
         if (getline(is_line, value, '=')) {
            getline(is_line, value);
            // Model parameters
            if ((key == "m0_real") && (!silent))
               m0_real    = StrToFloat(value);
            if ((key == "sigma_real") && (!silent))
               sigma_real = StrToFloat(value);
            if ((key == "rho_real") && (!silent))
               rho_real   = StrToFloat(value);
            if ((key == "model") && (!silent))
               model      = (int)StrToFloat(value);
            if (key == "khat")
               khat       = (int)StrToFloat(value);
```

```
if ((key == "lambda") && (!silent))
   lambda_real= StrToFloat(value);
if ((key == "nu") && (!silent))
   nu_real  = StrToFloat(value);
if ((key == "initPrice") && (!silent))
   initPrice  = StrToFloat(value);
if ((key == "interest") && (interest == 0.f))
   interest   = StrToFloat(value);
// Computation parameters
if (key == "maxCalculations")
   maxCalculations = (int)StrToFloat(value);
if ((key == "path_length") && (!silent))
   path_length     = (int)StrToFloat(value);
if (key == "data_gen")
   data_gen        = (int)StrToFloat(value);
if (key == "ControlVariates")
   ControlVariates = (int)StrToFloat(value);
// Optimization parameters
if (key == "initPoint")  {
   IPo          = value.c_str(); }
if (key == "scale")      {
   s         = value.c_str(); }
if (key == "loBoundary") {
   LoBoundary  = value.c_str(); }
if (key == "upBoundary") {
   UpBoundary  = value.c_str(); }
if (key == "initPoint2") {
   IPo2        = value.c_str(); }
if (key == "diffstep")   {
   diffstep    = (double)StrToFloat(value); }
if (key == "scale2") {
   s2          = value.c_str(); }
if (key == "loBoundary2") {
   LoBoundary2 = value.c_str(); }
if (key == "upBoundary2") {
   UpBoundary2 = value.c_str(); }
if (key == "diffstep2") {
   diffstep2 = (double)StrToFloat(value); }
if (key == "lambda_external")
   lambda_external = (int)StrToFloat(value);
if (key == "acctype")
   acctype     = (int)StrToFloat(value);
if (key == "likMixed")
   likMixed = (int)StrToFloat(value);
if (key == "printout")
```

```
            printout = (int)StrToFloat(value);
         if (key == "Memory")
            Memory = value;
         if (key == "b_gkk_est")
            b_gkk_est = (int)StrToFloat(value);
      }
   }
}
settings.close();

// Open settings file
try { settings.open(getexepath().append("Settings.ini")); }
catch (ios_base::failure e) {
   cout << "No Real Data file found! Exception
   opening/reading/closing file!\\\\n";
   getch();
   return 0;
}

// Add few settings to log with simulation
// experiment results first
if (silent) {
   simulations <<
      "initPrice = " << initPrice <<endl<<
      "path_length = " << path_length << endl<<
      "model = " << model << endl <<
      "RNG ="<<RNG<< endl <<
      "interest = " << interest << endl;
}

// Add the rest settings
while (getline(settings, line))
{
   if (silent) {
      if ((((line.find("initPrice") == string::npos) &&
      (line.find("path_length") == string::npos)) &&
      (line.find("model") == string::npos)) &&
      (line.find("interest") == string::npos)) {
         simulations << line << endl;
      }
   }
   else {
      simulations << line << endl;
   }
}
```

```
    settings.close();
    return 1;
}

using namespace std;

// ==== The main function ====
int
main(int argc, char * argv[])
{
    // Initialization
    int IP = 1;
    int MC_iterations1 = 1, MC_iterations2 = 2;
    double FSSE_m0 = 0.0, FSSE_sigma = 0.0, FSSE_rho = 0.0,
    FSSE_lambda = 0.0, FSSE_nu = 0.0;
    double RMSE_m0 = 0.0, RMSE_sigma = 0.0, RMSE_rho = 0.0,
    RMSE_lambda = 0.0, RMSE_nu = 0.0;
    double med_m0 = 0.0, med_sigma = 0.0, med_rho = 0.0,
    med_lambda = 0.0, med_nu = 0.0;
    double m0_av = 0.0, sigma_av = 0.0, rho_av = 0.0,
    lambda_av = 0.0, nu_av = 0.0;
    price_calc_counter = 0;

    // Seed Initialization
    ifstream PrimesSample;
    string app_path = getexepath();
    directory = app_path.c_str();
    app_path.append("\\PrimesSample.txt");

    try { PrimesSample.open(app_path.c_str()); }
    catch (ios_base::failure e) {
        cout << "No Seed Data file found! Exception
        opening/reading/closing file!\\\\n";
        getch();
        return 0;
    }

    char* buffer = new char[1000000];

    for (int i = 0; i < 200; i++) {
        PrimesSample >> PrimesSampleArr[i]; }

    // Read returns and option prices data
    ifstream infile;
```

```
app_path = getexepath();
app_path.append("\\RealData.txt");

try { infile.open(app_path.c_str()); }
catch (ios_base::failure e) {
   cout << "No Real Data file found!
   Exception opening/reading/closing file!\\\\n";
   getch();
   return 0;
}

infile.getline(buffer, 1000000);
x = buffer;
infile.getline(buffer, 1000000);
y = buffer;

int count;

// === Parse command line arguments ===
// In the silent model most of parameters are read
// from the command line arguments, namely not f
// rom the Settings file or manual choice on screen
for (count = 1; count < argc; count++)
   if ( string(argv[count]) == "/silent" )
      silent = 1;
// Performance measurement regime
for (count = 1; count < argc; count++)
   if (string(argv[count]) == "/nper")
      performance = 0;
// Estimation regime
for (count = 1; count < argc; count++)
   if (string(argv[count]) == "/nest")
      estimation = 0;

float multiplier;
int devType;

if (silent)
{
   for (count = 1; count < argc; count++) {

      // Which device to use fot computations (CPU or GPU)
      string tmp = string(argv[count]);
      if ( string(argv[count]) == "--device" ) {
         if ((string)argv[count + 1] == "cpu") {
```

```
        devType = 1; }
    else {
        devType = 2; }
}

//"#paths (1 - 16K, 2 - 32K, 3 - 48K, 4 - 64K and etc): ";
if (string(argv[count]) == "/np") {
   GLOBAL_MEMORY_SIZE_Y =
       (int)(32 * StrToFloat((string)argv[count + 1]));
   multiplier = StrToFloat((string)argv[count + 1]);
}

// #Options & #Iterations
// #Options in the basket (1 to 70) From:
if (string(argv[count]) == "/no1")
   NumOfPoints1 = (int)StrToFloat((string)argv[count + 1]);
// To:
if (string(argv[count]) == "/no2")
   NumOfPoints2 = (int)StrToFloat((string)argv[count + 1]);

// #Iterations From:
if (string(argv[count]) == "/ni1")
   MC_iterations1 = (int)StrToFloat((string)argv[count + 1]);
// To:
if (string(argv[count]) == "/ni2")
   MC_iterations2 = (int)StrToFloat((string)argv[count + 1]);

// Meth.Opt.(ASA,SA,TA,GD,EV ',' LM,BFGS as 2nd if necessary)
if (string(argv[count]) == "/mo1")
   Method = (string)argv[count + 1];

// Obj.Fcn (WRSS,likRet,likOpt,likMixed,likOptBSV,likMixedBSV)
if (string(argv[count]) == "/of1")
   ObjFcn = (string)argv[count + 1];

// Meth.Opt.(ASA,EV ',' LM,BFGS as 2nd if necessary)
if (string(argv[count]) == "/mo2")
   Method2 = (string)argv[count + 1];

// Obj.Fcn 2 ( WRSS, likOpt, likOptBSV)
if (string(argv[count]) == "/of2")
   ObjFcn2 = (string)argv[count + 1];

// Random Number Generator ( 1 - Pseudo, 2 - Quasi )
if (string(argv[count]) == "/rng")
```

```
            RNG = (int)StrToFloat((string)argv[count + 1]);

        // Spot price S0
        if (string(argv[count]) == "/S0")
            initPrice = StrToFloat((string)argv[count + 1]);

        // logreturns historical path length
        if (string(argv[count]) == "/plen")
            path_length = (int)StrToFloat((string)argv[count + 1]);

        // clustering parameter
        if (string(argv[count]) == "/m0")
            m0_real = StrToFloat((string)argv[count + 1]);

        // long-run volatility
        if (string(argv[count]) == "/sigma0")
            sigma_real = StrToFloat((string)argv[count + 1]);

        // leverage parameter
        if (string(argv[count]) == "/rho")
            rho_real = StrToFloat((string)argv[count + 1]);

        // risk-premium (ERP)
        if (string(argv[count]) == "/lambda")
            lambda_real = StrToFloat((string)argv[count + 1]);

        // volatility risk-premium (VRP)
        if (string(argv[count]) == "/nu")
            nu_real = StrToFloat((string)argv[count + 1]);

        // AMSM1/AMSM2 model
        if (string(argv[count]) == "/model")
            model = (int)StrToFloat((string)argv[count + 1]);

        // Risk-free interest rate
        if (string(argv[count]) == "/rf")
            interest = StrToFloat((string)argv[count + 1]);

    }
    argc = 5;
}
else
{
    // Pick Platform & Device
    cout << "Device type (1-CPU, 2-GPU,
```

```
3–CPU(platf.1), 4–GPU(platf.1)): ";
cin >> devType;

if (devType == 1) {
   argc = 5;
   argv[1] = "--platformId";;
   argv[2] = "0";
   argv[3] = "--device";
   argv[4] = "cpu";
}
if (devType == 2) {
   argc = 5;
   argv[1] = "--platformId";;
   argv[2] = "0";
   argv[3] = "--device";
   argv[4] = "gpu";
}
if (devType == 3) {
   argc = 5;
   argv[1] = "--platformId";
   argv[2] = "1";
   argv[3] = "--device";
   argv[4] = "cpu";
}
if (devType == 4) {
   argc = 5;
   argv[1] = "--platformId";
   argv[2] = "1";
   argv[3] = "--device";
   argv[4] = "gpu";
}
if (devType == 5) {
   argc = 5;
   argv[1] = "--platformId";
   argv[2] = "2";
   argv[3] = "--device";
   argv[4] = "cpu";
}
if (devType == 6) {
   argc = 5;
   argv[1] = "--platformId";
   argv[2] = "2";
   argv[3] = "--device";
   argv[4] = "gpu";
}
```

```
    // Further parameters are manually inputted
    // from the screen using keyboard
    // Number of sample paths
    cout << "#Trajectories (1 - 16K, 2 - 32K,
    3 - 48K, 4 - 64K and etc): ";
    cin >> multiplier;
    GLOBAL_MEMORY_SIZE_Y = (int)(32 * multiplier);


    // #Options & #Iterations
    cout << "#Options in the basket (1 to 70) From:                    ";
    cin >> NumOfPoints1;
    cout << "To:                                                       ";
    cin >> NumOfPoints2;

    cout << "#Iterations From:                                         ";
    cin >> MC_iterations1;
    cout << "To:                                                       ";
    cin >> MC_iterations2;

    // Optimization method
    cout << "Meth.Opt.(ASA,SA,TA,GD,EV ',' LM,BFGS as 2nd if necessary) : ";
    cin >> Method;

    // Objective function
    cout << "Obj.Fcn (WRSS,likRet,likOpt,likMixed,likOptBSV,likMixedBSV): ";
    cin >> ObjFcn;

    // Method of optimization for sequential optimization
    // namely global first, local second
    cout << "Meth.Opt.(ASA,EV ',' LM,BFGS as 2nd if necessary)          : ";
    cin >> Method2;

    // Objective function
    cout << "Obj.Fcn 2 ( WRSS, likOpt, likOptBSV)                      : ";
    cin >> ObjFcn2;

    // Type of random number generator
    cout << "Random Number Generator ( 1 - Pseudo, 2 - Quasi )         : ";
    cin >> RNG;

}

// If used real data from RealData.txt
```

```
if (data_gen == 0) {
   infile.getline(buffer, 1000000); // maturities

   real_1d_array tmp = buffer;
   number_of_C = (int)tmp.length();

   DataPointArr = new double *[number_of_C];
   for (int i = 0; i<number_of_C; i++) DataPointArr[i] = new double[3];

   DataPointRead(DataPointArr, number_of_C);
}

// If simulated and estimated only log-returns data
if (RNG == 1) { max_maturity = (int)1; }

// If simulated and estimated option prices data
if (RNG == 2) {
   float strike; float interest;
   DataPoint( NumOfPoints2, &max_maturity, &strike, &interest);
   //max_maturity = (int)NumOfPoints2 * 3;
}  // NumOfPoints2 - NumOfPoints1 ?

cout << endl;

// Memory allocation for estimation results arrays
int       itRange = MC_iterations2 - MC_iterations1 + 1;
double ** m0 = new double *[itRange];
for (int i = 0; i<itRange; i++) m0[i] = new double[IP];

double ** sigma = new double *[itRange];
for (int i = 0; i<itRange; i++) sigma[i] = new double[IP];

double ** rho = new double *[itRange];
for (int i = 0; i<itRange; i++) rho[i] = new double[IP];

double ** lambda = new double *[itRange];
for (int i = 0; i<itRange; i++) lambda[i] = new double[IP];

double ** nu = new double *[itRange];
for (int i = 0; i<itRange; i++) nu[i] = new double[IP];

double ** err = new double *[itRange];
for (int i = 0; i<itRange; i++) err[i] = new double[IP];

// Create file with logs of the experiment
```

```
_int64 t1; _int64 TotalTimer = (_int64)0;
 StartTimer( &t1 ); StartTimer(&TotalTimer);

// Create unique file name containing some experiments settings
// and data/time of the experiment
time_t t        = time(0);    // get time now
 struct tm * now = localtime( & t );

string tt1 = IntToStr( (int)(64 * 32 * 8 * multiplier) );
string tt2 = "(" + IntToStr( (int)NumOfPoints1 ) + "," +
   IntToStr((int)NumOfPoints2) + ")";
string tt3 = IntToStr( MC_iterations1 );
string tt4 = IntToStr( MC_iterations2 );

string tt;
if (silent)
{
   tt = "Simulations\\Opt=" + tt2 + "_Seed=(" + tt3 +
      "," + tt4 + ")_Trj=" + tt1 + "_" + Method + "_"
      + ObjFcn + "_" + Method2 + "_"
      + ObjFcn2 + "_" + IntToStr(RNG) + "_"
      + IntToStr(now->tm_year + 1900) + '-'
      + IntToStr(now->tm_mon + 1) + '-'
      + IntToStr(now->tm_mday) + " "
      + IntToStr(now->tm_hour) + "-"
      + IntToStr(now->tm_min) + "-"
      + IntToStr(now->tm_sec) + " plen = "
      + IntToStr(path_length) + ".txt";
}
else {
   tt = "Simulations\\Opt=" + tt2 + "_Seed=(" +
      tt3 + "," + tt4 + ")_Trj=" + tt1 + "_" + Method + "_"
      + ObjFcn + "_" + Method2 + "_"
      + ObjFcn2 + "_" + IntToStr(RNG) + "_"
      + IntToStr(now->tm_year + 1900) + '-'
      + IntToStr(now->tm_mon + 1) + '-'
      + IntToStr(now->tm_mday) + " "
      + IntToStr(now->tm_hour) + "-"
      + IntToStr(now->tm_min) + "-"
      + IntToStr(now->tm_sec) + ".txt";
}

ofstream simulations(getexepath().append(tt).c_str());

// Log some input settings
```

```
simulations << "#Trajectories,  " <<
   64 * 32 * 8 * multiplier <<endl;
simulations << "#Options,       " <<
   NumOfPoints1   << "," << NumOfPoints2   <<endl;
simulations << "#IterationsFrom," <<
   MC_iterations1 << "," << MC_iterations2 <<
   endl <<endl;

simulations << endl << "Objective FCN : " <<
   ObjFcn << endl << endl;

// Parse Settings.ini file
ParseSettings( simulations, settings );

simulations << endl << endl;

//   Data Generation (x - returns, y - options prices)
// data_gen = 0:  r <= NULL/Real;                        y <= RealData.txt
// data_gen = 1:  r <= simulated each experiment new;  y <= simulated once
// data_gen = 2:  r <= simulated once;                   y <= simulated once
// data_gen = 3:  r <= NULL/Real                         y <= simulated once
// data_gen = 4:  r <= simulated each experiment new;  y <= RealData.txt
// data_gen = 5:  r <= simulated once;                   y <= RealData.txt

SeedNumber = 1;

// Asset returns data vector
r          = new double[path_length];

// Read real returns data from file RealData.txt (fifth line)
if (data_gen == 0) {
   infile.getline(buffer, 1000000); // skip line with id
   infile.getline(buffer, 1000000); // skip option prices
   infile.getline(buffer, 1000000); // skip maturities

   stringstream ss(buffer);
   vector<string> vect;

   while (ss.good())
   {
      string substr;
      getline(ss, substr, ',');
      // Read and append to vect the value after comma
      vect.push_back(substr);
   }
```

```
    // Rewrite values from string vector to float vector
    for (int i = (int)vect.size() - path_length; i < (int)vect.size(); i++)
    {
        r[i - ((int)vect.size() - path_length)] = StrToFloat(vect.at(i));
    }
}

// Returns data generation
if ((data_gen == 2) || (data_gen == 5)) {
    AMSM_logreturns(
        path_length, m0_real, sigma_real, rho_real, lambda_real,
        model, khat, (double)initPrice, (double)interest, r);
}

// Options prices generation
if (((data_gen == 1) || (data_gen == 2)) || (data_gen == 3)) {
    // Initialization and launch of GPU kernel
    if (clMonteCarloAMSM.initialize() != SDK_SUCCESS)
        return SDK_FAILURE;

    // Parse command line options
    if (clMonteCarloAMSM.parseCommandLine(argc, argv))
        return SDK_FAILURE;

    if (clMonteCarloAMSM.isDumpBinaryEnabled())
        return clMonteCarloAMSM.genBinaryImage();
    else
    {
        // Memory allocation
        if (clMonteCarloAMSM.setupMonteCarloAMSM() != SDK_SUCCESS)
            return SDK_FAILURE;
    }

    if (clMonteCarloAMSM.setupCL() != SDK_SUCCESS)
        return SDK_FAILURE;

    // Simulate cross-section of option prices
    y = AMSM_option_data(
        3.f, (float)m0_real, 0.95f, (float)rho_real,
        (float)sigma_real, (float)lambda_real, (float)nu_real,
        model, NumOfPoints1, NumOfPoints2, settings);

    if (clMonteCarloAMSM.cleanup() != SDK_SUCCESS)
            return SDK_FAILURE;
```

```
}

// Time-consumption measurement
if (performance) {
    // Pick a seed of RNG
    SeedNumber = MC_iterations1;
    seed = PrimesSampleArr[SeedNumber - 1];

    // Initialization and launch of GPU kernel
    if (clMonteCarloAMSM.initialize() != SDK_SUCCESS)
        return SDK_FAILURE;

    // Parse command line options
    if (clMonteCarloAMSM.parseCommandLine(argc, argv))
        return SDK_FAILURE;

    if (clMonteCarloAMSM.isDumpBinaryEnabled())
        return clMonteCarloAMSM.genBinaryImage();
    else
    {
        // Memory allocation
        if (clMonteCarloAMSM.setupMonteCarloAMSM() != SDK_SUCCESS)
            return SDK_FAILURE;
    }

    if (clMonteCarloAMSM.setupCL() != SDK_SUCCESS)
        return SDK_FAILURE;

    // Logging
    simulations << endl << "m0_real = " << m0_real <<
    " sigma_real = " << sigma_real << " rho_real = " <<
    rho_real << " lambda_real = " << lambda_real <<
    " nu_real = " << nu_real << " S0 = " << initPrice <<
    " AMSM" << model << " SeedNumber = " <<
    SeedNumber << endl;

    Cross_section_prices(
        3, m0_real, 0.95, rho_real, sigma_real, lambda_real, nu_real,
        model, NumOfPoints1, NumOfPoints2);

    if (clMonteCarloAMSM.cleanup() != SDK_SUCCESS)
        return SDK_FAILURE;
}
```

```
string Method_tmp = Method;

// Estimation experiment with different seeds
if (estimation) {
   for (int SimN = MC_iterations1; SimN <= MC_iterations2; SimN++)
   {
      counter1 = 0;
      SeedNumber = SimN;

      // MLE data generation
      if ((data_gen == 1) || (data_gen == 4)) {
         AMSM_logreturns(
            path_length, m0_real, sigma_real, rho_real, lambda_real,
            model, khat, (double)initPrice, (double)interest, r);
      }

      if (clMonteCarloAMSM.initialize() != SDK_SUCCESS)
         return SDK_FAILURE;

      // Parse command line options
      if (clMonteCarloAMSM.parseCommandLine(argc, argv))
         return SDK_FAILURE;

      if (clMonteCarloAMSM.isDumpBinaryEnabled())
         return clMonteCarloAMSM.genBinaryImage();
      else
      {
         // Memory allocation
         if (clMonteCarloAMSM.setupMonteCarloAMSM() != SDK_SUCCESS)
            return SDK_FAILURE;
      }

      if (clMonteCarloAMSM.setupCL() != SDK_SUCCESS)
         return SDK_FAILURE;

      if (SimN == MC_iterations1) {
         simulations << DevInf << endl << endl;
      }

      // create and initialize timer
      StartTimer(&Timer);

      if (SimN == MC_iterations1) cout << endl << tt << endl << endl;

      // Simulations counter
```

```
cout << "Number of Simulation : " << SimN <<
" in (" << MC_iterations1 << "," << MC_iterations2 << ")" << endl;

seed = PrimesSampleArr[SeedNumber - 1];

// Save current array of estimates in tmp arrays
m0_tmp = m0_real; sigma_tmp = sigma_real;
rho_tmp = rho_real; lambda_tmp = lambda_real;
nu_tmp = nu_real;
Stage = 1;
Par_number = (int)LoBoundary.length();
//diffstep   = diffstep2;

//  ==Global optimization==
if (Method_tmp.find(",") != std::string::npos) {
   if (Method_tmp.find("ASA") != std::string::npos) { Method = "ASA"; }
   if ((Method_tmp.find("SA") != std::string::npos) &&
      (Method_tmp.find("ASA") == std::string::npos)) {
      Method = "SA";
   }
   // extremely imprecise
   if (Method_tmp.find("TA") != std::string::npos) { Method = "TA"; }
   // extremely imprecise
   if (Method_tmp.find("GD") != std::string::npos) { Method = "GD"; }
   if (Method_tmp.find("EV") != std::string::npos) { Method = "EV"; }

}

// Write arrays with search region boundaries and initial points
// in arrays used by ASA
if (Method.find("ASA") != std::string::npos) {
   for (int i = 0; i < Par_number; i++) { LBoundary[i] = LoBoundary[i]; }
   for (int i = 0; i < Par_number; i++) { UBoundary[i] = UpBoundary[i]; }
   for (int i = 0; i < Par_number; i++) { IPoint[i] = IPo[i]; }

   // Run external ASA optimization subroutine
   main2(0, argv);
}

// If method is TA, GD or EV run it in the search region hypercube
// (LoBoundary, UpBoundary)
if (Method.find("TA") != std::string::npos) {
   optTA(optimum, maxCalculations, seed, LoBoundary, UpBoundary); }
if (Method.find("GD") != std::string::npos) {
   optGD(optimum, maxCalculations, seed, LoBoundary, UpBoundary); }
```

```
if (Method.find("EV") != std::string::npos) {
   optEV(optimum, maxCalculations, seed, LoBoundary, UpBoundary); }

bool StochOpt = false;
// If stochastic global optimization by SA, TA, GD, EV was conducted
// then print out  the optimum on screen, log it in the file,
// write it in arrays with results (m0, sigma,...)
if ((Method.find("SA") != std::string::npos) ||
   (Method.find("TA") != std::string::npos) ||
   (Method.find("GD") != std::string::npos) ||
   (Method.find("EV") != std::string::npos)
   ) {
   StochOpt = true;

   //        if (printout) {
   cout << "INITIAL POINT " << SimN
      << ":" << optimum[0]
      << "," << optimum[1]
      << "," << optimum[2]
      << "," << optimum[3]
      << "," << optimum[4]
      << endl;
   //        }

   simulations << "INITIAL POINT " << SimN
      << "," << optimum[0]
      << "," << optimum[1]
      << "," << optimum[2]
      << "," << optimum[3]
      << "," << optimum[4]
      << "," << BestRSS << "," << seed << "," <<
      devType << "," << Method << endl;

   m0[      SimN - MC_iterations1][0] = optimum[0];
   sigma[   SimN - MC_iterations1][0] = optimum[1];
   rho[     SimN - MC_iterations1][0] = optimum[2];
   lambda[  SimN - MC_iterations1][0] = optimum[3];
   nu[      SimN - MC_iterations1][0] = optimum[4];

   err[     SimN - MC_iterations1][0] = BestRSS;

   m0_tmp = optimum[0];
   sigma_tmp = optimum[1];
   rho_tmp = optimum[2];
}
```

```
// ==Local optimization methods==
if (Method_tmp.find(",") != std::string::npos) {
   if (Method_tmp.find("LM") != std::string::npos) {
      Method = "LM"; }
   if (Method_tmp.find("BFGS") != std::string::npos) {
      Method = "BFGS"; }
}

// If global optimization was used then take its optimum
// as initial point for local optimization method
real_1d_array c;
if (StochOpt &&
   ((Method.find("LM") != std::string::npos) ||
   (Method.find("BFGS") != std::string::npos))) {
   string       IPoString;
   IPoString = "[" +
      FloatToStr(optimum[0]) + "," +
      FloatToStr(optimum[1]) + "," +
      FloatToStr(optimum[2]) + "]";
   const char * IPoChar = IPoString.c_str();
   c = IPoChar;
}
else {
   c = IPo;
}

// Levenberg-Marquardt local optimization method from AlgLib
if (Method.find("LM") != std::string::npos) {
   //This criterion guarantees that algorithm will
   //stop only near the minimum, independently
   //if how fast / slow we converge to it.
   //Second and third criteria are less reliable
   //because sometimes algorithm makes small
   //steps even when far away from minimum.
   double epsf = 0;
   double epsx = 0;
   double epsg = 0.0000000000000001;
   //diffstep = 1.0e-4;  used from settings
   ae_int_t maxits = (ae_int_t)maxCalculations / (int)5;

   minlmstate state;
   minlmreport rep;
   //ae_int_t acctype = 0; // acceleration is switched off
```

```
// LM method is developed directly for minimization of
// sums of squares, so the squares are directly used as
// as an inputted rather then whole WRSS sum
if (ObjFcn.find("WRSS") != std::string::npos) {
   minlmcreatev(
      NumOfPoints2 - NumOfPoints1 + 1,
      c,
      diffstep,
      state);
}
else {
   // If likelihood is an objective function
   minlmcreatev(1, c, diffstep, state); }

// Set boundaries, stop criterions, scale of parameters
// and etc
minlmsetbc(state, LoBoundary, UpBoundary);
minlmsetcond(state, epsg, epsf, epsx, maxits);
minlmsetscale(state, s);
minlmsetacctype(state, acctype);

// Run optimization
minlmoptimize(state, function1_fvec);
// Collect the results
minlmresults(state, c, rep);

// Write the optimum in the corresponding arrays
m0[SimN - MC_iterations1][0] = c[0];
sigma[SimN - MC_iterations1][0] = c[1];
rho[SimN - MC_iterations1][0] = c[2];

err[SimN - MC_iterations1][0] = (double)state.f;

m0_tmp = c[0]; sigma_tmp = c[1]; rho_tmp = c[2];

if (Par_number == 4) {
   if (!lambda_external) {
      lambda[SimN - MC_iterations1][0] = c[3];
      lambda_tmp = c[3];
   }
   else {
      nu[SimN - MC_iterations1][0] = c[3];
   }
}
```

```
    if (Par_number == 5) {
        lambda[SimN - MC_iterations1][0] = c[3];
        nu[SimN - MC_iterations1][0] = c[4];
    }

    // Log the result of current iteration
    simulations << SimN << "," << c[0] << "," <<
    c[1] << "," << c[2] << "," << c[3] << "," <<
    c[4] << "," << state.f << "," << seed << "," <<
    devType << "," << Method << endl;

    // Print on screen the result of current iteration
    if (printout) {
        printf("term. type = %d\n", int(rep.terminationtype));
        cout << "============================
        ================================" <<
        endl << endl;
    }
}

// BFGS local optimization method from AlgLib
if (Method.find("BFGS") != std::string::npos) {
    //This criterion guarantees that algorithm will
    //stop only near the minimum, independently
    //if how fast / slow we converge to it.
    //Second and third criteria are less reliable
    //because sometimes algorithm makes small
    //steps even when far away from minimum.
    double epsf = 0;
    double epsx = 0;
    double epsg = 0.000000001;
    //diffstep = 1.0e-4;  used from settings
    ae_int_t maxits = (ae_int_t)maxCalculations / (int)10;

    minlbfgsstate  state;
    minlbfgsreport rep;

    // Set boundaries, stop criterions, scale of parameters
    // and etc
    minlbfgscreatef(
        Par_number,  // number of optimized parameters
        Par_number,
        c,
        diffstep,
        state);
```

```
    minlbfgssetscale(state, s);
    minlbfgssetprecscale(state);
    //minlbfgssetprecdiag(  state, d);
    minlbfgssetcond(state, epsg, epsf, epsx, maxits);

    // Run optimization
    minlbfgsoptimize(state, function1_func);
    // Collect the results
    minlbfgsresults(state, c, rep);

    // Write the optimum in the corresponding arrays
    m0[SimN - MC_iterations1][0] = c[0];
    sigma[SimN - MC_iterations1][0] = c[1];
    rho[SimN - MC_iterations1][0] = c[2];

    err[SimN - MC_iterations1][0] = (double)state.f;

    m0_tmp = c[0]; sigma_tmp = c[1]; rho_tmp = c[2];

    if (Par_number == 4) {
       if (!lambda_external) {
          lambda[SimN - MC_iterations1][0] = c[3];
          lambda_tmp = c[3];
       }
       else {
          nu[SimN - MC_iterations1][0] = c[3];
       }
    }

    if (Par_number == 5) {
       lambda[SimN - MC_iterations1][0] = c[3];
       nu[SimN - MC_iterations1][0] = c[4];
    }

    // Log the result of current iteration
    simulations << SimN << "," << c[0] << "," <<
     c[1] << "," << c[2] << "," << c[3] << "," <<
     c[4] << "," << state.f << "," << seed <<
     "," << devType << "," << Method << endl;
}

//Method "SPSA"
if (Method.find("SPSA") != std::string::npos) {
    /*
    int dim = 3, mu = 2 * dim;
```

```
    float ** bounds = new float * [dim];
    for(int i=0; i<dim; i++) bounds[i] = new float [2];

    bounds[0][0] = 2.2f;
    bounds[0][1] = 2.7f;

    bounds[1][0] = 1.1f;
    bounds[1][1] = 1.3f;

    bounds[2][0] = 0.85f;
    bounds[2][1] = 0.99f;

    //bounds[0][0] = 0.9f;
    //bounds[0][0] = 0.9f;

    ACiD(dim, mu, bounds);

    delete[](bounds);
    */
    int dim = 2;
    float * theta = new float[dim];
    // Initialization
    float a = 10.f, c_par = /*0.00005f*/0.00025f,
    chat_par = 0.01f, A = 2.f, alpha = 0.602f/* 1.f */,
    gamma = 0.101f/* 1/6 */;
    theta[0] = 1.8f; theta[1] = 0.01f;
    // theta[2] = 0.8f;//, theta[3] = 0.01f;//1.35148f;

    // Run optimization
    SPSA(a, c_par, chat_par, A, alpha, gamma, theta, 4000, dim);

    delete[](theta);
}

// ===Sequential optimization===
// In this case risk premium is estimated
// after model parameters)

// Find out number of parameters by the length of
// search region boundary vector
Par_number = (int)LoBoundary2.length();
Stage = 2;

double epsf = 0.0000000001;
double epsx = 0.0;
```

```
double epsg = 0.0;

// If Levenberg–Marquardt is used for the second stage
if (Method2.find("LM") != std::string::npos) {
   ae_int_t maxits = (ae_int_t)maxCalculations;// / (int)5;

   minlmstate state;
   minlmreport rep;
   //ae_int_t acctype = 1; // acceleration is switched off

   // Set boundaries, stop criterions, scale of parameters
   // and etc
   if (ObjFcn2.find("WRSS") != std::string::npos) {
      minlmcreatev(
         NumOfPoints2 - NumOfPoints1 + 1,
         c2,
         diffstep2,
         state);
   }
   else { minlmcreatev(1, c2, diffstep2, state); }

   minlmsetbc(state, LoBoundary2, UpBoundary2);
   minlmsetcond(state, epsg, epsf, epsx, maxits);
   minlmsetscale(state, s2);
   minlmsetacctype(state, acctype);

   // Run optimization
   minlmoptimize(state, function1_fvec2);
   // Collect the results
   minlmresults(state, c2, rep);

   // Write the optimum in the corresponding arrays
   if (Par_number == 1) {
      if (lambda_external) {
         lambda[SimN - MC_iterations1][0] = c2[0];
      }
      else {
         nu[SimN - MC_iterations1][0] = c2[0];
      }
   }

   if (Par_number == 2) {
      lambda[SimN - MC_iterations1][0] = c2[0];
      nu[SimN - MC_iterations1][0] = c2[1];
   }
```

```
    // Log the result of current iteration
    simulations << SimN << "," <<
    lambda[SimN - MC_iterations1][0] << "," <<
    nu[SimN - MC_iterations1][0] << ", , , ," <<
    seed << "," << devType << "," << Method << endl;

    if (printout) {
      cout << "=========================
      ==============================
      ====" << endl << endl;
    }
  }

  // If BFGS is used for the second stage
  if (Method2.find("BFGS") != std::string::npos) {

    ae_int_t maxits = (ae_int_t)maxCalculations / (int)15;

    minlbfgsstate  state;
    minlbfgsreport rep;

    // Set boundaries, stop criterions, scale of parameters
    // and etc
    minlbfgscreatef(
      Par_number,
      Par_number,
      c2, diffstep2, state);
    minlbfgssetscale(state, s2);
    minlbfgssetprecscale(state);
    //minlbfgssetprecdiag(  state, d);
    minlbfgssetcond(state, epsg, epsf, epsx, maxits);

    // Run optimization
    minlbfgsoptimize(state, function1_func2);
    // Collect the results
    minlbfgsresults(state, c2, rep);

    // Write the optimum in the corresponding arrays
    if (Par_number == 1) {
      if (lambda_external) {
        lambda[SimN - MC_iterations1][0] = c2[0];
      }
      else {
        nu[SimN - MC_iterations1][0] = c2[0];
```

```
        }
    }
    if (Par_number == 2) {
        lambda[SimN - MC_iterations1][0] = c2[0];
        nu[SimN - MC_iterations1][0] = c2[1];
    }

    // Log the result of current iteration
    simulations << SimN << "," << lambda[SimN - MC_iterations1][0] <<
    "," << nu[SimN - MC_iterations1][0] << ", , , ," << state.f <<
    "," << seed << "," << devType << "," << Method << endl;
}

// If stochastic optimization (EV) is used for the second stage
if (Method2.find("EV") != std::string::npos) {
    optEV2(
        optimum,
        maxCalculations,
        seed,
        LoBoundary2, UpBoundary2); }

// If stochastic optimization (ASA) is used for the second stage
if (Method2.find("ASA") != std::string::npos) {
    for (int i = 0; i < Par_number; i++) { LBoundary[i] = LoBoundary2[i]; }
    for (int i = 0; i < Par_number; i++) { UBoundary[i] = UpBoundary2[i]; }
    for (int i = 0; i < Par_number; i++) { IPoint[i] = IPo2[i]; }

    main2(0, argv);

    // Print out om=n screen
    cout << "INITIAL POINT " << SimN
        << ":" << optimum[0]
        << "," << optimum[1]
        << ", "
        << ", "
        << ", "
        << endl;

    // Write the optimum in the corresponding arrays
    if (Par_number == 1) {
        if (lambda_external) {
            lambda[SimN - MC_iterations1][0] = optimum[0];
        }
        else {
            nu[SimN - MC_iterations1][0] = optimum[0];
```

```
            }
        }

        if (Par_number == 2) {
            lambda[SimN - MC_iterations1][0] = optimum[0];
            nu[SimN - MC_iterations1][0] = optimum[1];
        }

        // Log the result of current iteration
        simulations << SimN << "," <<
        lambda[SimN - MC_iterations1][0] << "," <<
        nu[SimN - MC_iterations1][0] << ", , , ," <<
        BestRSS << "," << seed << "," << devType <<
        "," << Method << endl;

    }

    // Clean up OpenCL objects
    if (clMonteCarloAMSM.cleanup() != SDK_SUCCESS)
        return SDK_FAILURE;
}

// Calculate error metrics for each calibrated/estimated parameter
StandardErrorsMed(
    m0, 0, itRange, m0_real, &m0_av,
    &FSSE_m0, &RMSE_m0, &med_m0);
StandardErrorsMed(
    sigma, 0, itRange, sigma_real, &sigma_av,
    &FSSE_sigma, &RMSE_sigma, &med_sigma);
StandardErrorsMed(
    rho, 0, itRange, rho_real, &rho_av,
    &FSSE_rho, &RMSE_rho, &med_rho);
StandardErrorsMed(
    lambda, 0, itRange, lambda_real,
    &lambda_av, &FSSE_lambda, &RMSE_lambda, &med_lambda);
StandardErrorsMed(
    nu, 0, itRange, nu_real, &nu_av,
    &FSSE_nu, &RMSE_nu, &med_nu);

// Log error metrics
simulations << endl << "m0_real = " << m0_real <<
" sigma_real = " << sigma_real << " rho_real = " <<
rho_real << " lambda_real = " << lambda_real <<
" nu_real = " << nu_real << " AMSM" << model << endl;
```

```
    simulations << endl;

    simulations << "m0      = " << m0_av << " sigma  = " <<
    sigma_av << " rho     = " << rho_av << " lambda  = " <<
    lambda_av << " nu     = " << nu_av << endl;

    simulations << "FSSE    = " << FSSE_m0 << " FSSE    = " <<
    FSSE_sigma << " FSSE    = " << FSSE_rho << " FSSE    = " <<
    FSSE_lambda << " FSSE    = " << FSSE_nu << endl;

    simulations << "RMSE    = " << RMSE_m0 << " RMSE    = " <<
    RMSE_sigma << " RMSE    = " << RMSE_rho << " RMSE    = " <<
    RMSE_lambda << " RMSE    = " << RMSE_nu << endl;

    simulations << "Median = " << med_m0 << " Median = " <<
    med_sigma << " Median = " << med_rho << " Median = " <<
    med_lambda << " Median = " << med_nu <<
    endl << endl << endl;

    simulations << "Average calibration time : " <<
    StopTimer(TotalTimer) / itRange << endl;

    simulations << "Average option calculation time : " <<
    StopTimer(TotalTimer) / price_calc_counter << endl;

}

// Calculate option prices for whole cross-section
// for the best set of parameters (in the sense of error size)
if (estimation) {
    double best_err = 1000000;
    int best_iter = MC_iterations1;
    for (int i = 0; i < itRange; i++) {
        if (err[i][0] < best_err) {
            best_iter = i;
            best_err = err[i][0]; }
    }

    SeedNumber = MC_iterations1 + best_iter;
    seed       = PrimesSampleArr[SeedNumber - 1];

    // Initialize OpenCL
    if (clMonteCarloAMSM.initialize() != SDK_SUCCESS)
        return SDK_FAILURE;
```

```
    // Parse command line options
    if (clMonteCarloAMSM.parseCommandLine(argc, argv))
        return SDK_FAILURE;

    if (clMonteCarloAMSM.isDumpBinaryEnabled())
        return clMonteCarloAMSM.genBinaryImage();
    else
    {
        // Memory allocation
        if (clMonteCarloAMSM.setupMonteCarloAMSM() != SDK_SUCCESS)
            return SDK_FAILURE;
    }

    if (clMonteCarloAMSM.setupCL() != SDK_SUCCESS)
        return SDK_FAILURE;

    // Log the best estimates
    simulations << endl << "m0_best = " << m0[best_iter][0] <<
    " sigma_best = " << sigma[best_iter][0] << " rho_best = " <<
    rho[best_iter][0] << " lambda_best = " << lambda[best_iter][0] <<
    " nu_best = " << nu[best_iter][0] << " S0 = " << initPrice <<
    " AMSM" << model << " SeedNumber = " << SeedNumber << endl;

    // Run calculation
    Cross_section_prices2(
        3, m0[best_iter][0], 0.95, rho[best_iter][0],
        sigma[best_iter][0], lambda[best_iter][0],
        nu[best_iter][0],
        model, NumOfPoints1, NumOfPoints2,
        simulations);

    // Clean up OpenCL
    if (clMonteCarloAMSM.cleanup() != SDK_SUCCESS)
        return SDK_FAILURE;

}

// Clean up memory
for (int i = 0; i<itRange; i++) delete[] m0[i];
for (int i = 0; i<itRange; i++) delete[] sigma[i];
for (int i = 0; i<itRange; i++) delete[] rho[i];
for (int i = 0; i<itRange; i++) delete[] lambda[i];
for (int i = 0; i<itRange; i++) delete[] nu[i];
for (int i = 0; i<itRange; i++) delete[] err[i];
for (int i = 0; i<number_of_C; i++) delete[] DataPointArr[i];
```

```
    delete[] buffer;
    delete[] r;

    // Close of all opened files
    simulations.close();
    settings.close();
    PrimesSample.close();
    infile.close();

    return SDK_SUCCESS;
}
```

Erklärung zum selbständigen Verfassen der Arbeit.

Ich erkläre hiermit, dass ich meine Doktorarbeit

"Essays on Economic Sentiment Dynamics and Asymmetric Multi-Frequent Models of Financial Volatility"

selbstständig und ohne fremde Hilfe angefertigt habe und dass ich alle von anderen Autoren wörtlich übernommenen Stellen, wie auch die sich an die Gedanken anderer Autoren eng anlehnenden Ausführungen meiner Arbeit, besonders gekennzeichnet und die Quellen nach den mir angegebenen Richtlinien zitiert habe.

Datum Unterschrift