CALIBRATION OF FINANCIAL MODELS BASED ON AUTOMATIC DIFFERENTIATION AND HIGH-PERFORMANCE COMPUTING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2018

By
Grzegorz Kozikowski
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

CALIBRATION OF FINANCIAL MODELS BASED ON AUTOMATIC DIFFERENTIATION AND HIGH-PERFORMANCE COMPUTING
Grzegorz Kozikowski
A thesis submitted to the University of Manchester
for the degree of Master of Philosophy, 2018

Stochastic models are commonly used in quantitative finance to describe the dynamics of the derivatives market. As the market quotes are constantly changing, the models need to be calibrated to make real-time investment decisions. This can involve the sensitivity calculation to support the calibration process and investment portfolio management. For investment portfolios consisting of thousands of assets and options, the sensitivity calculation and calibration process are computationally expensive.

This thesis presents a number of approaches to sensitivity calculation and model calibration utilizing high-performance computing architectures and Automatic Differentiation that improve performance and accuracy in financial modeling when compared to finite differences and pathwise methods. A parallel Monte-Carlo engine has been developed using the Adjoint methods for the first-order sensitivity calculation and model calibration This addresses the sensitivity and calibration problem for general stochastic differential models. The engine supports multi-/many-core CPU, GPU and distributed computing architectures. The work utilizes a graph representation and overloading operators to express general stochastic differential models. The sensitivities for the model calibration are calculated in parallel via a single simulation by the Adjoint method with the gradient computation cost being 1.8x that of function evaluation. The computational experiments consider both the Heston model and Heston with term-structure. These show that the engine improves performance by up to two orders of magnitude when compared to a sequential version. A hardware implementation has been developed for the Heston model calibration via the Adjoint on FPGA. The work also shows performance improvement of up to two orders of magnitude when compared to a sequential implementation. A parallel non-linear least squares optimization framework using Automatic Differentiation has been developed. This utilizes a graph representation and overloaded operator techniques to express the general objective and constraint functions. The framework supports multi-/many-core architectures such as GPUs and Intel Xeon/Xeon-Phi. The computational experiments consider the semi-closed form Heston model with the Gauss-Kronrod integration. These show performance improvement of 8.4x on GPU (OpenCL) and 7x on (CUDA) vs a sequential OMP (OpenMP) implementation. A Xeon-Phi implementation improves performance by 34x when compared to a single-thread implementation.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Glossary

The following table presents an explanation of financial terms mentioned in this work [Hul12]

Name	Description
American Option	An option that can be exercised at any time during its life
Arbitrage	A trading strategy that takes advantage of two or more securities being mispriced relative to each other
Ask Price	The price that a dealer is offering to sell an asset
Barrier Option	An option whose payoff depends on whether the path of the underlying asset has reached a barrier (i.e. a certain predetermined level)
Bid Price	The price that a dealer is prepared to pay for an asset
Bid-Ask Spread	The amount by which the ask price exceeds the bid price
Calibration	Method for implying a model's parameters from the prices of actively traded assets
Call Option	An option to buy an asset at a certain price by a certain date
Derivative	An instrument whose price depends on, or is derived from the price of another asset
Discount Rate	The annualized dollar return on a Treasury bill or similar instrument expressed as a percentage of the final face value
Euler-Maruyama method	A method for the approximate numerical solution of a stochastic differential equation
European option	An option that can be exercised only at the end of its life
Exercise (Strike) Price	The price at which the underlying asset may be bought or sold in an option contract
Expiration date	The end of life of a contract
Feller condition	the condition for which the volatility of the Heston model is strictly positive
Implied volatility	Volatility implied from an option price using the Black-Scholes formula
Interest Rate	Interest Rate defines the amount of money which is paid to a borrower by the lender and is a key factor in pricing interest rate derivatives (contracts whose value depends on the interest rate fluctuations)
Kantorovich Graph (DAG)	a graph representing chain-rule of the function subsequent operations necessary to evaluate the value of a function
Maturity	the end of the life of a contract
Newton-Raphson Method	An iterative procedure for solving nonlinear equations
Option	a contract that gives the right to buy or sell an asset
Payoff	The cash realized by the holder of an option or other derivative at the end of its life
Put Option	An option to sell an asset for a certain price by a certain date
RMSE	Root Mean Square Error
Strike price	the price at which the asset may be bought or sold in an option contract (also called the exercise price)
Volatility	A measure of the uncertainty of the return realized on an asset
Volatility Skew	A term used to describe the volatility smile when it is nonsymmetrical
Volatility Smile	The variation of implied volatility with strike price
Volatility Surface	A table showing the variation of implied volatilities with strike price and time to maturity
Wiener Process	A process where the change in a variable during each short period of time of length t has a normal distribution with a mean equal to zero and a variance equal to t

List of abbreviations

The following table presents a list of abbreviations used in this thesis.

Name	Description
AD	Automatic Differentiation
CPU	Computational Processing Unit
DAG	Directed Acyclic Graph
FFT	Fast Fourier Transform
FPGA	Field Programming Gateway Array
GPU	Graphics Processing Unit
HPC	High-Performance Computing
MC	Monte-Carlo simulation
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PDE	Partial Differential Equation
RMSE	Root Mean Square Error
SDE	Stochastic Differential Equation

Table 2: List of abbreviations

Chapter 1

Introduction

This thesis presents approaches to financial derivatives pricing, the Greeks' calculation and model calibration using numerical methods such as Automatic Differentiation, Monte-Carlo (MC) methods and High-Performance computing (HPC) platforms such as FPGA cards, GPUs, multi-core and many-core processors. The presented approaches utilize overloading operator techniques and graph processing to support general stochastic models. This combination improves performance and accuracy of financial option pricing, the Greeks' calculation and model calibration by up to two orders of magnitude. The work can find application in real-time risk management systems and derivatives trading platforms.

Over the last 30 years derivatives – financial instruments whose value depends on underlying assets such as stocks, stock indices, interest rates, foreign exchange rates and commodities such as oil, gold, silver – have become increasingly important in finance. The amount of outstanding derivatives positions was close to $500 trillion by January 2016 [otc16]. These contracts are traded on the over-the-counter markets. The fundamental derivatives contract is an option which gives the client the right to buy or sell the given commodity at a certain time with the price negotiated at the present time. The negotiated price is called the exercise or strike price. The date of the contract is known as the expiration date or maturity. There are two types of options: call, which gives the investor an opportunity to buy the asset, and put, which is to sell the traded instrument. Further, the call and put options can be sold and purchased.

In addition, there are varieties of options such as American, European, Asian and barrier options. American options can be exercised at any time until expiration date; European options can be exercised at the expiration date only; The Asian options depend on the average price of the underlying asset over a certain period of time; the barrier options depend on whether the underlying asset has exceed the predetermined price. When buying a call or put option, a buyer is referred to as having a long position. When selling a call or put option, a seller is referred to as having a short position. The derivatives market uses option price makers to ensure market liquidity. Option market makers quote a bid and ask price – the bid price is the price at which the option market maker is prepared to buy an option, the ask price is the price at which, the market maker is prepared to sell [Hul12].

There are several factors affecting an option price such as: the current stock price, the strike price, the time to expiration, the volatility of the stock price, the risk-free interest rate and the dividends. The volatility of the stock price measures a degree of changes of the stock price over time. Over the last three decades, several models have been derived to describe the dynamics of options on various underlying assets such as European options, interest rate options, etc. The simplest approach, binomial tree, based on constructing a binomial tree was introduced by Cox, Ross and Rubinstein in 1979 [CRR79]. The binomial tree, also known as the lattice model, describes an evolution of the option price in discrete time. This consists of the nodes representing the stock price and option price. In 1973, Black, Merton and Scholes derived the model for option pricing based on stochastic calculus [BS73]. The model assumes that the stock price follows a geometric Brownian motion with the drift μ and the variance rate σ – the process with log-normal distribution. The European option price is calculated as the discounted value of the difference between the strike price and the stock price at expiration date. This model assumes that the volatility of the commodity is constant over time. In risk management, implied volatility is calculated – the volatility for which the Black-Scholes formula produces the market option price. As the volatility of commodities is non-constant over-time, the Heston model was introduced, which assumes that volatility follows a stochastic mean-reverting process [Hes93]. Further, many derivations were introduced using the Jumps Poisson process to express commodity price jumps caused by various events [Gat06]. There are several methods for solving the Heston model such as Monte-Carlo simulation, finite difference methods and the maximum likelihood method. Monte-Carlo simulation involves random numbers to produce many different paths of the option price. The arithmetic average of the payoffs is the estimated value of the option price. Finite-difference methods solve the Partial Differential Equations (PDE) by using the difference equation [Gla04]. The maximum likelihood method maximizes the likelihood function [MPZ08a]. Further, the European option price for the Heston model can be derived analytically by using the characteristic function of the option price distribution. This involves integration of a complex formula and can utilize the FFT methods, or quadrature-integration methods.

One of the investment strategies using options is hedging, which allows financial institutions to neutralize risks of market movements by contracting commodity price. Hedging often involves application of mathematical modelling to the derivatives market to measure financial risks. The mathematical modelling in the derivatives market can be used to price options for different maturities and strikes and to calculate the Greeks – risk sensitivities that measure how the model parameters affect the option price. Financial institutions use the Greeks to manage and hedge investment portfolios (to compensate loss on the market by gains made on another market and stabilize portfolio value).

As an example, consider an investor wishing to neutralize his portfolio from further commodity price movements by delta-hedging [Hul12]. For this purpose, he calculates the Greek that measures how the change of the underlying asset price affects the option price (delta). Based on this value, he constantly adjusts his investment positions to cover further gains (or loss) on the stock market by loss (or gains) on derivatives market/options. This means that his overall portfolio value will remain constant for a short time (as option prices and stock prices permanently fluctuate). To mitigate risk of movements on his portfolio, he must constantly re-calculate the Greeks and rebalance the positions. For portfolios consisting of thousands of underlying assets and options traded by banks every day, frequent rebalancing based on the Greeks (via Monte-Carlo simulation) is computationally demanding.

Further, option pricing and the Greeks' calculation for a given asset on the market require the model calibration to the historical market-data. There is need to determine such a set of model parameters for which the model accurately expresses past market-quotes. The option model calibration is analytically expensive as this often requires the gradient calculation the first-order Greeks – the first-order derivatives of the option payoff with respect to the input parameters) – to fit hundreds of market quotes and thousands of different underlying assets every day [Hul12]. Most existing solutions for the Greeks' calculation are based on the pathwise, likelihood or finite differences methods, in which computational cost of the gradient is at least proportional to the number of model parameters x the time of a single Monte-Carlo simulation [Gla04].

1.1 Aims and Objectives

This thesis aims at:

- performance and accuracy improvement in model computation, sensitivity evaluation and model calibration in financial applications by utilizing high-performance computing platforms and numerical methods such as Automatic Differentiation;
- flexible definition of various models and discretization schemes used in derivatives pricing and hedging. This aims at utilization of overloading operator techniques and graph processing

The work focuses on the Heston model calibration (the Heston with the Feller condition, the Heston model with Jumps, the Heston model with term-structures and the semi-closed form Heston model solution).

1.2 Contribution

The contribution of this work is performance and accuracy improvement in financial model calibration and sensitivity calculation. For this purpose, several calibration frameworks have been implemented that combine numerical methods as Automatic Differentiation and HPC platforms. The calibration frameworks have been tested with a financial case-study: Heston-model calibration.

The detailed aspects of the contribution include:

- A parallel MC engine for the first-order sensitivity calculation and model calibration using the Adjoint.

  This work utilizes a graph representation and overloading operators to express general stochastic models. The sensitivities for the model calibration are calculated in parallel via a single simulation by the Adjoint method with the gradient computation cost being 1.8x that of function evaluation. The framework supports multi-, many-core and distributed architectures such as GPUs, Intel-Xeon/Xeon-Phi with the frameworks CUDA, OpenCL, OpenMP with Xeon-Phi and OpenMPI.

  The engine provides an interface that allows a platform-independent model definition. The computational experiments consider the Heston model and Heston with term-structure. These show that the engine improves performance by up to two orders of magnitude when compared to a sequential version. This is provided as a static and dynamic library.

- Heston model calibration using the Adjoint and MC methods on FPGA - a MC engine for the expected value computation, the first-order Greeks calculation and model calibration. The sensitivities are calculated in parallel via a single simulation by the Adjoint - an Automatic Differentiation method with the gradient computation cost being 1.8 x that of function evaluation. The gradient is applied to the optimization methods to calibrate the model to market data. The framework supports FPGA cards with Maxeler technology. The computational experiments consider a financial case-study: the Heston model calibration.

- Parallel non-linear least squares optimization framework using Automatic Differentiation – this work presents a parallel non-linear least squares optimization framework using Automatic Differentiation. This approach utilizes a graph representation and overloaded operator techniques to express the general objective and constraint functions. The framework computes values and the gradient of the objective/constraint function at many different points in parallel. The gradient for the non-linear least squares optimization is calculated in parallel with the computational cost being twice that of function evaluation. The gradient is applied to the non-linear least squares optimization methods. The framework supports multi-, many-core architectures such as GPUs, Intel Xeon/Xeon-Phi with the frameworks CUDA, OpenCL, OpenMP with Xeon-Phi. The computational experiments consider the semi-closed form Heston model with the Gauss-Kronrod integration method.

1.3 Structure

The rest of this thesis is structured as follows: Chapter 2 presents a review of related literature; Chapter 3 presents the technical context of the work; Chapter 4 presents parallel frameworks for financial sensitivity calculation and model calibration; Chapter 5 presents computational experiments for the Heston model calibration on High-Performance Computing platforms; Chapter 6 presents conclusions and further work.

Chapter 2

Literature review

2.1 Introduction

This section provides a literature review concerning Automatic Differentiation, Monte-Carlo (MC) methods, model calibration and high-performance computing (HPC) in the context of scientific and industrial applications. This is divided into three sections corresponding to the contribution areas:

1. Section 2.2 reviews work on Automatic Differentiation (AD), MC simulation and HPC applied to scientific and industrial applications;
2. Section 2.3 reviews research on financial simulation and model calibration using FPGA cards;
3. Section 2.4 reviews advances on AD tools and their application in optimization.

2.2 Parallel Monte-Carlo (MC) engine for the first-order sensitivity calculation and model calibration using the Adjoint

Monte Carlo (MC) simulation is a procedure for sampling random scenarios for the input process and evaluating the expected value converging to the correct result. This method is widely used to solve stochastic models for which the analytical solutions are too complex to derive. This is used in finance for pricing and hedging complex derivatives contracts. Unfortunately, MC simulation is computationally expensive when calculating the output result, with the computational cost increasing with the number of scenarios. Furthermore, the sensitivity and model calibration is computationally demanding. The calibration can utilize the gradient information to support local optimization methods.

Several differentiation methods can be used with MC simulation to evaluate the first-order sensitivities to support model calibration. The simplest, the finite difference (FD) method, requires two MC simulations that differ in a small change in a single input model parameter [FHP$^+$12]. Based on these MC results, the sensitivity is calculated as the ratio of the output change to the small change in the input parameter. Unfortunately, this method is both computationally expensive (the first-order sensitivities require (2 x model parameters) x the cost of a single MC simulation) and inaccurate if the change is too big. The likelihood method utilizes the probability density functions, therefore, they can be applied to non-smooth models but often produce inaccurate results with large variance. Pathwise methods are based on AD with the forward order. They require as many MC simulations as the number of model parameters to evaluate the sensitivities. The pathwise method for the sensitivities has a computational cost generally smaller than finite differences. The drawback of this technique is complexity in the case of models with a large number of independent variables [FHP$^+$12].

Work in [Zha13] presents a HPC engine for the *Value at Risk* (VaR) computation using the MapReduce model. The Map function splits the dataset into small data chunks and distributes these to processing nodes. The Reduce function collects the results from the nodes and generates the output result. This paradigm is used to parallelize MC simulation for the VaR. Performance experiments have been performed on a cluster of 4 nodes. The execution time of the sequential MC simulation was around 532 seconds;the parallel MC simulation computes VaR in around 115 seconds.

Work in [Mih15] presents approximations to option sensitivities for stochastic volatility models. These utilize a sequential MC techniques for the latent state in a Hidden Markov Model. These techniques are applied to the Greeks' computation using the likelihood ratio method. Further, the work develops a modified explicit Euler scheme for SDEs with non-Lipschitz continuous drift or diffusion.

Work in [Mor06] presents a parallel option pricing framework for interest rate derivatives via the Hull-White trinomial interest rate lattice model. This applies vectorization and data parallelization techniques available in Fortran. The parallel pricing includes the classical backward induction and MC simulation. The implementation supports distributed computing with shared memory.

Work in [Ras16] presents an approach to the Dupire's deterministic local volatility function calibration. The work compares performance for five different calibration methods for the local volatility function. Further, the work presents approaches to evaluating the first-order derivatives: complex-step derivative approximation, automatic differentiation forward mode and reverse mode. These are utilized to calculate the Greeks for *Credit Value Adjustment*(CVA) for an interest rate swap. The work utilizes FADBAD++ – an AD tool for the derivatives calculation. The work does not support the derivatives calculation on parallel and distributed architectures.

Work in [Doa10] presents a HPC engine for MC simulations for financial pricing applications. This introduces a grid programming framework for derivatives pricing. The framework supports fault-tolerance, load balancing, dynamic task distribution and deployment mechanism for heterogeneous grid architectures. Further, the work presents an implementation of a Classification MC algorithm for high-dimensional American option pricing. This is scaled up to 64 processors in a grid environment.

Work in [PSLvL16] presents application of the automatic differentiation forward mode to solving mass transfer equations. This applies the Newton method for solving non-linear equations. AD was combined with block and band compression for efficiently computing the Jacobian. For the derivatives computation, ADOL-C was utilized. The work compares the forward AD mode with FD method and analytical derivations for the first-order derivatives. The approach using the analytical derivations for the Jacobian was two times faster than the forward AD mode. The work does not utilize the Adjoint method. This does not support parallel processing on HPC platforms.

Work in [DZ13] presents an implementation for stochastic volatility model calibration on a multi-core CPU cluster. The implementation utilizes shared and distributed packages in Python. The computational experiments consider the Heston and Bates models. The experiments have been performed on a cluster of 32 dual socket Dell PowerEdge R410 nodes providing 256 cores. The speed-up achieved is around 139x against the sequential version. This reduces overall time taken to calibrate 1024 SPX options by a factor of 37x.

Work in [Cap11] gives an approach to calculating sensitivities of stochastic differential processes. The underlying idea is an application of the Adjoint to each scenario of the MC simulation for solving stochastic differential equations (SDE). The computational cost of the sensitivity calculation via the Adjoint is independent of the number of input parameters of the stochastic model. Derivatives are calculated by utilizing the Adjoint in the rules required to calculate the final result (chain-rule). Considering a system depending on 60 parameters and providing the SDE solution in 1 minute, this typically computes the gradient in less than 4 minutes. Unfortunately, this work does not exploit parallel architecture.

Work in [GHR$^+$16] presents an approach to the gradient calculation using the Adjoint techniques on GPU architectures. This utilizes vector and matrix structures to store intermediate partial derivatives. The work studies performance for four cost functions: sum of sigmoids, linear leasts-squares, maximum entropy models and Cholesky decomposition. The computational experiments have been performed on an Intel Xeon e-5-1620 v2 (3.7 GHz) with 64 GB of DDR3 RAM memory and an NVIDIA GeForce GTX Titan Black. The speedup achieved on GPU cards is around $7.5x \pm 4.4$ vs. a sequential implementation on CPU. In this implementation, the multiple threads located on GPU cannot create their own tapes.

Work in [SZAW15] presents parallel approaches to option pricing utilizing the lattice model and the MC method. The work has been implemented in CUDA and OpenCL. This utilizes the Black-Scholes model for option pricing. Benchmarks have been performed for the option pricing via the Black-Scholes model. Computational experiments have been done on an Intel Dual Core Xeon processor with 3.4GHz with 16GB RAM memory and NVIDIA GeForce GTX 980 with 4GB RAM memory. Pricing of an option with 100000 timesteps on a CPU via the lattice method requires around 23K ms whereas a parallel version computes option prices in around 3.5K ms. The MC simulation for 1000000 paths and 1000 timesteps takes around 186 seconds on a CPU. The parallel GPU version of the MC simulation calculates the option prices in around 6 seconds for 1000000 scenarios and 100 timesteps on GPU. The work does not allow the sensitivity computation and does not support general stochastic models.

Work in [MF12] applies AD to the quadrature – numerical integration to calculate the sensitivities of the integral. There are included derivations for truncation errors. The results hold when the integrand is one degree higher continuously differentiable than the sufficient for convergence of its quadrature. The work is tested with a tetrachoric correlation estimation example in Matlab. This utilizes rectangle, midpoint and Simpson's rule for integration. The work uses forward mode automatic differentiation for the derivatives calculation. The work does not support automatic differentiation in reverse mode.

Work in [JY11a] considers application of the Adjoint to MC simulation for the gamma matrix computation for multidimensional financial derivatives including Asian baskets and cancelable swaps. Numerical results show that computation of all $n(n+1)/2$ gammas in the LMM takes approximately $n/3$ times as long as computing the price.

Work in [JY11b] presents the application of the Adjoint techniques to the Delta computation of interest rate derivatives. The work studies constant maturity swap rate market model and co-initial swap-rate market model. The work shows that the computational complexity of the algorithm is proportional to the number of rates x the number of factors per step.

Work in [CL14] presents an application of automatic differentiation and the Implicit Function Theorem to sensitivity calculation and model calibration for the credit default swaps and credit default index swaptions. The computational experiments for the credit default swaps shows performance improvement by up to 50x for 18 spread tenors and interest rate instruments. The combination of automatic differentiation and the Implicit Function Theorem allows computation of interest rate and credit spread risk in 20 % less time than computation cost of option price. The work does not support processing on parallel architectures.

Work in [LL14] addresses an implementation for pricing Asian Options on an Intel Xeon-Phi Coprocessor architecture. This utilizes a closed-form analytical solution for the geometric average Asian option and MC method for the arithmetic average Asian option. The work was compared with a single-Asian option example included in CUDA SDK 6.0 on an NVIDIA Tesla K40c. Experimentation shows that the pricing of a 2-year contract with 252 time steps and 1M scenarios takes around one fifth of a second on an Intel Xeon-Phi Coprocessor 7210p. The GPU implementation calculates the Asian option price in around 0.469 second. The work does not support graph processing to define general stochastic models on a multi-core Xeon-Phi architecture. Further, this does not support the sensitivity calculation.

Work in [FBI06] applies the Adjoint in seismology. The changes to displacement field and flow patterns at the surface can be quantified with respect to the changes of density, viscosity or elastic coefficients. The work derives the equations for the scalar wave dynamics in two dimensions. A numerical case-study shows that the Adjoint focuses near to the location of a parameter perturbation at the same rate when the original wavefront reaches that location and the Adjoint is far more efficient than finite difference approximations. This does not rely on the existence of Green's functions or transposes of a differential operator. To locate parameter perturbations, the least-square function with the Adjoint is utilized. The method can be applied to non-linear operators using the Navier-Stokes equations. Work [DLS10] presents an implementation of the Adjoint MC method in Geant4 – a toolkit for simulation of the passage of particles through matter and the GRAS module (Geant4 Radiation Analysis for Space). This aims at precise computation of space radiation effects on electronic components, solar panels and optical devices in complex payload and satellite geometry. It compares the Pathwise forward and the Adjoint methods to analyze two test geometries presenting the components. The numerical results show that the Adjoint improves performance by up to four orders of magnitude when considering the sensitive volume size equal to 1 mm.

The Adjoint with MC simulation of the European option was applied to the Heston calibration in [KMS09]. For the optimization process, the standard sequential quadratic programming methods with the Gauss-Newton approximation of the Hessian were utilized. Tests were performed on a 3 GHz CPU and 2 GB memory. As expected, the algorithm reduced the number of MC simulations for different partial derivatives (usually evaluated by FD approximations). The presented approach improves the calibration time for a typical equity market model with time-dependent model parameters calibrated by the FD method from over three hours (approx 200 minutes) to less than ten minutes. Unfortunately the sequential solution is strongly dependent on the starting points as the problem is non-convex, thus it converges to the nearest local minimum. The accuracy experiments showed that the Adjoint implementation produces the same result as the FD approach. The implementation was only executed on a single CPU core, hence, it may be worth investigating parallel computing for the MC-based calibration.

Calibration of the Heston stochastic volatility model using filtering and maximum likelihood methods is presented in [MPZ08b]. A standard steepest ascent method for a non-convex maximum likelihood function was utilized to optimize the maximum likelihood function. As evaluation of the maximum likelihood function is computationally demanding, parallel architectures were used to boost performance. The implementation was tested on a IBM SP4 machine with 32 processors. The speedup factor was equal to 15x for 16 processors used. The accuracy of the calibration of European options was about 10^{-5} (the value of the Root Mean Square Error (RMSE) function).

Previous work by the thesis author [KK13] examines application of the Adjoint to evaluation of the first-order Greeks using the standard MC simulation. This utilizes parallel GPU architectures such as NVIDIA CUDA to boost performance. The first-order Greeks are calculated through the MC simulation for Black-Scholes and the Heston model. This uses derivations for specific financial models and cannot be applied to general SDE models. The parallel experiments are compared with a sequential implementation and run on an NVIDIA Tesla C2070 with 448 cores. They show an improvement of approximately two orders of magnitude for both the Black-Scholes and the Heston first-order Greeks.

2.3 Heston model calibration using the Adjoint and MC methods on FPGA

Work [dSSK$^+$11] presents a hardware implementation of an MC simulation for the Heston model for European options. The approach developed in VisualHDL on a Xilinx Virtex-5 doubles performance compared with a CPU equipped with an Intel Xeon CPU W335 3.07 Ghz and 8 GB RAM. Unfortunately, the implementation does not support the Greeks calculation and model calibration.

Work [SRMLB$^+$13] explores an FPGA implementation of an MC method to price Asian options via the Black-Scholes model on an FPGA Altera Stratix-V card. The approach has been developed in the Impulse C environment supporting floating-point arithmetic on the FPGA. For generating normal distribution random samples, the Mersenne Twister with Box-Muller transform has been utilized. The results show performance improvement of the FPGA implementation by 504x when compared to an execution on a single-core implementation (Intel i7 850 2.8 Ghz) and approx. 150x when compared with a 4-core implementation supported by OpenMP. The implementation only investigates the Black-Scholes model which does not allow risk sensitivities calculation and model calibration.

Work [TB08] explores an FPGA engine for solving the Black-Scholes model via MC simulation. This is benchmarked on a Maxwell Supercomputer equipped with 64 FPGA Xilinx Virtex-4 XC4VSX55 cards and compared against a 32 core CPU cluster. The speed-up achieved was around 750x compared to the CPU cluster. This work only presents the Black-Scholes model and does not evaluate the Greeks.

Work of [JLT11] addresses a FPGA framework to solve option pricing formulas via different methodologies: MC, FD, quadrature method and binomial trees. As case studies both European and American options are considered. The MC simulation European options on an FPGA is 41x faster compared with an 8 core Intel Xeon CPU processor. This work does not allow risk sensitivity calculation and model calibration.

Studies presented in [NAW08] explore a Quasi MC method for option pricing using Brownian paths. The performance results in a 50x speed-up on an FPGA (Altera Stratix III EP3SE260-3) compared with an Intel Xeon 3 GHz CPU. This work does not support more complicated option pricing models, such as the Greeks' calculation and calibration to market data.

2.4 Parallel non-linear least-squares optimization framework using Automatic Differentiation

There are several tools that support the gradient calculation via AD. These generally present two approaches for the recording chain-rule: source-code transformation or operator overloading.

Work in [WG10] presents an open-source package in C/C++ for AD. The work supports overloading operator techniques to define differentiable functions. This allows the first and higher-order derivatives calculation in forward and reverse automatic differentiation mode. This supports MPI and OMP parallel programs. Unfortunately, this does not support many-core and GPU architectures.

Work in [HP13] presents an AD tool that supports the adjoint derivatives and tangent calculation. The software allows differentiation of function written in C++ and Fortran. The software does not support differentiation on parallel architectures.

Work in [Hog14] presents an approach to the first-order derivatives calculation in reverse-mode AD. The work utilizes Expression C++ templates. The work utilizes overloaded operator techniques to build a graph representation of a model function. The work has been incorporated into the Adept library. Benchmarks for four different numerical algorithms show the performance improvement against other operator-overloading libraries. ADOL-C is 5-7 times slower than Adept, CppAD is 7-9 times more expensive and Sacador is 2.6-8 times slower than Adept. The work reduces memory usage being 1.3-7.7 times more efficient. The work does not support complex arithmetic and differentiation on parallel computing architectures.

Work in [FSA$^+$12] presents an AD framework for solving statistical parameter estimation problems. This contains an optimization module for non-linear models with a large number of parameters. The software provides functionality of analysis of uncertainty of estimated parameters via a Markov-chain MC method. The framework utilizes overloading operator techniques to record chain-rule of the model function. The software transforms the source-code with the model definition into C++. Next, the C++ code is compiled and linked with the AD library to create a binary program. Unfortunately, the software does not support differentiation on HPC architectures.

Work in [GHR$^+$16] presents an approach to the gradient calculation utilizing the Adjoint methods on GPU architectures. this uses vector and matrix data structures to store intermediate partial derivatives The work studies performance for four cost functions: sum of sigmoids, linear least-squares, maximum entropy models and Cholesky decomposition. The computational experiments have been performed on an Intel Xeon e-5-1620 v2 (3.7 GHz) with 64 GB of DDR3 RAM memory and an NVIDIA GeForce GTX Titan Black. The performance improvement on GPU is around 7.5x $\pm$ 4.4 vs. a sequential implementation on CPU. In this implementation, the multiple-threads located on GPU cannot create their own tapes – data structures to store intermediate results and the first-order sensitivities.

Work [ABFK11] proposes an approach to the Heston model calibration using global optimization methods. The Heston model is evaluated by using the Fourier cosine method on CUDA. The model has been evaluated for many sample points distributed with quasi-distribution over the parameter space. Further, the Levenberg-Marquardt method has been applied to the most feasible points (the points for which RMSE is low). The gradient for optimization was evaluated using the FD method. Performance studies have been done on a GPU server with two NVIDIA GPU C1060 and one GTX 260. The implementation was benchmarked with a varied number of generated random samples (from 256 to 65536). The experiments have been carried out for one option set (256 options). Performance improvement has been achieved for 65536 samples; execution times on a GPU were 50x times faster than on a CPU. The optimization algorithm does not utilize differentiation techniques as the Adjoint to improve performance.

Previous work by the thesis author [KK13] presents an application of the Adjoint techniques to the first-order sensitivity calculation via the standard MC simulation. This utilizes parallel GPU architectures to improve performance. The first-order sensitivities are calculated through the MC simulation for Black-Scholes and Heston model. This uses derivations for specific financial models and cannot be applied to general SDE functions. The parallel experiments are compared with a sequential implementation and run on an NVIDIA Tesla C2070 with 448 cores. They show performance improvement of approximately two orders of magnitude for both the Black-Scholes and the Heston first-order sensitivity computation.

Previous work by the thesis author in [KK12] presents a parallel approach to evaluating the gradient and Hessian with both forward and reverse AD modes. The library implemented in C++ uses a generic programming paradigm to dynamically parametrize the data-type. Overloaded operator techniques are used to build a graph representation for factorable functions. The graph representation is transferred to GPU and further processed in parallel. Each thread block evaluates and differentiates a single function. The performance experiments were carried out on an Athlon 2.8 GHz processor with an NVIDIA GF 9800 GT supporting Tesla architecture. The speed-up is equal to 180x for functions consisting of 64 nodes (the gradient calculation). The execution times for the Hessian calculation were 280x faster on a GPU. The implementation supports interval arithmetic.

2.5 Summary

In this chapter, the literature review has been presented. The section 2.2 focused on application of Monte-Carlo simulation, Automatic Differentiation and High-Performance Computing platforms to various scientific and industrial applications. This showed the Adjoint technique reduces the cost of the sensitivity calculation when compared to pathwise forward and finite difference methods. The computational cost of the gradient calculation via the Adjoint is independent of the number of input parameters for the stochastic models. The input model for various applications can be represented as a sequence of elementary arithmetic operations. This can utilize overloading operator techniques and graph processing to allow the differentiation via AD.

The section 2.4 investigated automatic differentiation frameworks and the semi-closed form Heston model calibration. The automatic differentiation tools can utilize two approaches for the chain-rule recording: source-code transformation and overloading operator techniques. The work [GHR$^+$16] presented a GPU approach to the gradient computation. This showed the performance improvement by the factor of $7.5\text{x} \pm 4.4$ vs. a sequential implementation. The work [ABFK11] showed that the GPU cards improve performance of the semi-closed form Heston model calibration calculated via the FD method by the factor of 50 x.

Chapter 3

Technical context

3.1 Introduction

This chapter provides overview of technical aspects of this work to contextualise the developments and contributions of the thesis. This is organized as follows:

- Section 3.2 introduces a directed acyclic graph - graph representation that can be used to express a composite function
- Section 3.3 presents fundamentals of automatic differentiation for the gradient computation using Forward and Reverse (the Adjoint) modes.
- Section 3.4 presents fundamentals of non-linear least squares optimization.
- Section 3.5 concerns Monte-Carlo simulation
- Section 3.6 briefly introduces high-performance computing platforms
- Section 3.7 presents financial models: Heston model, Heston model with Jumps, Heston model with term-structure used as case-studies.

3.2 Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a data structure formed by a collection of nodes and directed edges without a directed cycle. Each node is connected to another, such that there is no way to start at a node v, follow any sequence and end at the same node. Each task can operate on a vector of input data, process the intermediate results and return the output data vector. The DAG structure can represent a topology of arithmetic operations required to evaluate a function.

3.3 Automatic Differentiation

Background

Automatic Differentiation (AD) is a set of algorithmic routines to accurately and efficiently compute derivatives of a composite function [KK12]. This approach allows the derivatives computation with machine accuracy [1] [FHP+12].

AD exploits the fact that each composite function can be interpreted as a sequence (chain-rule) of the elementary operators (as addition, multiplication or exponential, etc.) required for its evaluation. To explain AD, the Black-Scholes formula is used as an example [FHP+12].

Example Consider the Black-Scholes model and its recursive dependency of commodity price from time t to time $t+1$.

$$S_{t+1} = S_t \cdot (1 + r \cdot dt + \sigma \cdot \sqrt{dt} \cdot Z) \tag{3.1}$$

Suppose, we want to evaluate the Black-Scholes single path from time t to $t+1$. For this purpose, we derive a sequence (see Table 3.1a) (chain-rule) of elementary operations needed to evaluate the underlying asset price at time $t+1$.

$$\begin{aligned} f_1 &= r \cdot dt \\ f_2 &= 1 + f_1 \\ f_3 &= \sqrt{dt} \\ f_4 &= \sigma \cdot f_3 \\ f_5 &= f_4 \cdot Z \\ f_6 &= f_2 + f_5 \\ f_7 &= S_t \cdot f_6 \end{aligned}$$

(a) Chain-rule of the Black-Scholes model

This sequence can be expressed graphically using a DAG (see Figure 3.1) [FHP+12]. The edges represent a dependency between subsequent elementary operators or input

[1]Computational devices use binary systems to represent floating-point arithmetic. The floating-point number is represented by using three numbers: sign, exponent and mantissa. Due to the finite number of bits for exponent and mantissa representation, the numbers are rounded to the nearest, toward zero, toward plus/minus infinity [IEE08]

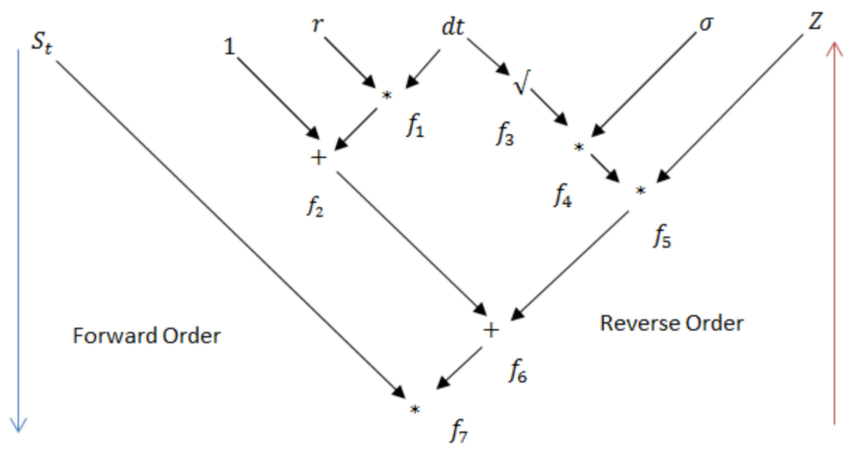


Figure 3.1: Chain-rule of the Black-Scholes model as a DAG

parameters; the nodes are identified by f_i – intermediate functions to evaluate the final result. As can be seen, each f_i is dependent on some previously computed f_j or model parameters $(i > j)$. It is worth noting that Black-Scholes requires 7 elementary operations to evaluate the underlying asset price at time $t+1$ from time t with the forward order. Intuitively, in order to calculate the composite function, we start computations from independent variables (S_t,..., Z) through intermediate operations (f_1,...,f_6) to the final operator (f_7). By interpreting such a chain-rule with forward-order (or reverse) and applying basic differentiation routines for elementary operators, AD computes derivatives of the composite function. These derivatives are accurately evaluated and subject only to rounding and not discretization error. This makes AD particularly attractive when compared to standard numerical differentiation methods, such as finite-differences [Cap11]. AD has two basic modes: Forward (Pathwise) and Reverse (the Adjoint) [FHP+12].

Pathwise methods

The pathwise method computes derivatives with the forward order and calls differentiation routines for elementary functions.

Example Let us perform differentiation of the function from f_1 to f_7 (with the forward order).

First, the derivatives df_1 and df_3 are equal to zero as f_1 and f_3 are independent of σ. Next, the derivative of df_4 is $d\sigma \cdot f_3 + \sigma \cdot df_3$. because f_4 contains σ. Note,

$f_1 = r \cdot dt$	$df_1 = 0$
$f_2 = 1 + f_1$	$df_2 = df_1$
$f_3 = \sqrt{dt}$	$df_3 = 0$
$f_4 = \sigma \cdot f_3$	$df_4 = d\sigma \cdot f_3 + \sigma \cdot df_3$
$f_5 = f_4 \cdot Z$	$df_5 = df_4 \cdot Z$
$f_6 = f_2 + f_5$	$df_6 = df_2 + df_5$
$f_7 = S_t \cdot f_6$	$df_7 = S_t \cdot df_6$

(a) Chain-rule of Black-Scholes (Pathwise methods)

$f_1 = r \cdot dt$	$\overline{f_7} = \frac{df_7}{df_7} = 1$
$f_2 = 1 + f_1$	$\overline{f_6} = \frac{df_7}{df_6} \cdot \overline{f_7} = S_t \cdot \overline{f_7}$
$f_3 = \sqrt{dt}$	$\overline{S_t} = \frac{df_7}{dS_t} \cdot \overline{f_7} = f_6 \cdot \overline{f_7}$
$f_4 = \sigma \cdot f_3$	$\overline{f_2} = \frac{df_6}{df_2} \cdot \overline{f_6} = \overline{f_6}$
$f_5 = f_4 \cdot Z$	$\overline{f_5} = \frac{df_6}{df_5} \cdot \overline{f_6} = \overline{f_6}$
$f_6 = f_2 + f_5$	$\overline{f_4} = \frac{df_5}{df_4} \cdot \overline{f_5} = Z \cdot \overline{f_5}$
$f_7 = S_t \cdot f_6$	$\overline{Z} = \frac{df_5}{dZ} \cdot \overline{f_5} = \overline{f_5}$
	$\overline{f_3} = \frac{df_4}{df_3} \cdot \overline{f_4} = \sigma \cdot \overline{f_4}$
	$\overline{\sigma} = \frac{df_4}{d\sigma} \cdot \overline{f_4} = f_3 \cdot \overline{f_4}$
	$\overline{dt} = \frac{df_3}{ddt} \cdot \overline{f_3} = \frac{1}{2\sqrt{dt}} \cdot \overline{f_3}$
	$\overline{f_1} = \frac{df_2}{df_1} \cdot \overline{f_2} = \overline{f_2}$
	$\overline{dt} += \frac{df_1}{ddt} \cdot \overline{f_1} = r \cdot \overline{f_1}$
	$\overline{r} = \frac{df_1}{dr} \cdot \overline{f_1} = dt \cdot \overline{f_1}$

(b) Chain-rule of Black Scholes (Adjoint methods)

that this derivation includes previously evaluated and stored df_3. In the next step we differentiate f_5 with respect to f_4 (previously evaluated). Processing this sequence, the final partial derivative of underlying asset price (df_7) is computed with respect to σ (see Table 3.2a). The gradient computation by the pathwise method is proportional to the number of independent input parameters.

The Adjoint method

The Adjoint performs computations in a reverse manner starting from the final operation. This approach requires evaluation and storage of the function value and all intermediate results (DAG nodes).

Consider the following example using the above DAG to evaluate the gradient of f_7.

Evaluation	Pathwise	Adjoint
7	$45+7=52$	$13+7=20$

Table 3.1: Number of arithmetic operations required to evaluate the gradient of the Black Scholes model

Example Let us differentiate the final operation f_7 with respect to f_7. As expected, $\frac{df_7}{df_7}$ is equal to 1. Assuming the values of all intrinsic functions (f_1,..., f_6) are known, f_7 can be differentiated with respect to S_t – the left branch and f_6 – the right branch of the DAG graph. The first derivative: $\frac{df_7}{dS_t} = \frac{d(f_6 \cdot S_t)}{dS_t} = f_6$: Analogously: $\frac{df_7}{df_6} = \frac{d(f_6 \cdot S_t)}{dS_t} = S_t$ In the next stage, $\frac{df_6}{df_2}$ and $\frac{df_6}{df_5}$ are calculated: $\frac{df_6}{df_2} = \frac{d(f_2+f_5)}{df_2} = 1$ and $\frac{df_6}{df_5} = \frac{d(f_2+f_5)}{df_5} = 1$ By multiplying the above results by the previously evaluated $\frac{df_7}{df_6}$ we have: $\frac{df_6}{df_2} \cdot \frac{df_7}{df_6} = \frac{df_7}{df_2} = S_t$ $\frac{df_6}{df_5} \cdot \frac{df_7}{df_6} = \frac{df_7}{df_5} = S_t$ In the third step, we compute derivatives f_5 with respect to f_4 and f_5 with respect to Z. Then: $\frac{df_5}{df_4} = \frac{d(f_4 \cdot Z)}{df_4} = Z$ and $\frac{df_5}{dZ} = \frac{d(f_4 \cdot Z)}{dZ} = f_4$ To obtain $\frac{df_7}{df_4}$ and $\frac{df_7}{dZ}$ let us multiply the above results by value $\frac{df_6}{df_5} \cdot \frac{df_7}{df_6}$ from the previous step, then: $\frac{df_5}{df_4} \cdot \frac{df_6}{df_5} \cdot \frac{df_7}{df_6} = \frac{df_7}{df_4} = Z \cdot S_t$ $\frac{df_5}{dZ} \cdot \frac{df_6}{df_5} \cdot \frac{df_7}{df_6} = \frac{df_7}{dZ} = f_4 \cdot S_t$ In the next stage, we differentiate f_4 with respect to σ and f_4 with respect to f_3 in an analogous manner and multiply by the above results, giving: $\frac{df_4}{d\sigma} \cdot \frac{df_5}{df_4} \cdot \frac{df_6}{df_5} \cdot \frac{df_7}{df_6} = \frac{df_7}{d\sigma} = Z \cdot S_t \cdot f_3 = Z \cdot S_t \cdot \sqrt{dt}$ and $\frac{df_4}{df_3} \cdot \frac{df_5}{df_4} \cdot \frac{df_6}{df_5} \cdot \frac{df_7}{df_6} = \frac{df_7}{df_3} = \sigma \cdot Z \cdot S_t$ Repeating this processing flow for all DAG nodes, the gradient of f_7 is evaluated.

As can be seen, this procedure requires only one sweep through the chain-rule to calculate the gradient, thus, involving fewer computations than the Pathwise method

Table 3.2b shows a complete chain-rule for the gradient of the Black-Scholes by the Adjoint ($\overline{f_i}$ denotes the partial derivative $\frac{df_7}{df_i}$)

Table 3.1 gives the number of necessary arithmetic operations for the Pathwise and the Adjoint method (including the cost of function evaluation).

Mathematical fundamentals

In order to present the principles of the components of the AD algorithm [FHP$^+$12]; we assume that:

- $x_1, x_2, \ldots, x_n$ denote independent variables;
- $y_i = f_i(f^i_{left}, f^i_{right}) = f^i_{left} \circ f^i_{right}$ – a differentiable function f_i considered as a composition of two operands f^i_{left}, f^i_{right} where $1 \leq left, right \leq i$ and an intrinsic operator $\circ$ as $+$, $-$, $*$, $/$, etc. If $\circ$ is an unary operator (*sin*, *cos*, *ln*, etc.) then f_i is only dependent on f^i_{left};

- $f_1, \ldots, f_{k-1}$ – intermediate intrinsic functions;
- f_k – the final intrinsic function required to evaluate f;
- y_i – intermediate results, $y_i = f_i(f^i_{left}, f^i_{right})$ or $y_i = f_i(f^i_{left})$ for $1 \leq i \leq n$.

$$\begin{bmatrix} x_1 \\ x_2 \\ . \\ x_n \\ y_1 \\ . \\ y_k \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \\ f_1(f^1_{left}, f^1_{right}) \\ . \\ f_k(f^k_{left}, f^k_{right}) \end{bmatrix} \tag{3.2}$$

Evaluation of the gradient of f relies on differentiation of the subsequent intrinsic functions with respect to each input variable.

As a result, we have the Jacobian matrix:

$$J_f(x) = \left(\frac{\partial f_i}{\partial x_j}\right)_{k \cdot n} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ . & . & . \\ \frac{\partial f_k}{\partial x_1} & \cdots & \frac{\partial f_k}{\partial x_n} \end{bmatrix} \tag{3.3}$$

Forward mode – gradient

Consider the function dependent on one or two functions immediately preceding in the DAG, we have: $y_{curr} = f_{curr}(f_{left}, f_{right})$ or $y_{curr} = f_{curr}(f_{left})$

Differentiating the subsequent functions, we obtain the results defined as follows:

- for binary operators:

$$f_{curr}(f_{left}, f_{right}) = f_{left}\, op\, f_{right}\ , \tag{3.4}$$

$$\frac{\partial f_{curr}}{\partial x_i} = \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_{left}}{\partial x_i} + \frac{\partial f_{curr}}{\partial f_{right}} \cdot \frac{\partial f_{right}}{\partial x_i}\ , \tag{3.5}$$

- for unary operators:

$$\begin{aligned} f_{curr}(f_{left}) &= op\, f_{left} \,, \\ \frac{\partial f_{curr}}{\partial x_i} &= \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_{left}}{\partial x_i} \,. \end{aligned}$$

Note that:

$$\frac{\partial x_i}{\partial x_i} = 1 \,.$$

Having computed the derivatives of this list, the final result denotes the partial derivative of the input function.

Reverse mode (the Adjoint method) – gradient The second AD method for the gradient calculation is known as the Adjoint Mode done in a reverse manner.

This is better suited to functions of many input variables. To explain the Adjoint method, we consider the relation below:

$$\overline{f_{curr}} = \frac{\partial f_{curr}}{\partial f_i} \cdot \overline{f_{curr}} \,, \tag{3.6}$$

where $\overline{f_i} = \frac{\partial f_k}{\partial f_i}$ and $\overline{f_{curr}} = \frac{\partial f_k}{\partial f_{curr}}$. The index *curr* corresponds to the function which is directly dependent on the operations denoted by sub-index *i* (as can be seen in the DAG 3.1); additionally for $curr = k$ we assume: $\overline{f_{curr}} = \frac{\partial f_{curr}}{\partial f_{curr}} = 1$ Taking into account the previous equation, the formulas for left and right partial derivatives can be derived.

$$\frac{\partial f_k}{\partial f_{left}} = \sum \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_k}{\partial f_{curr}} \,, \qquad \frac{\partial f_k}{\partial f_{right}} = \sum \frac{\partial f_{curr}}{\partial f_{right}} \cdot \frac{\partial f_k}{\partial f_{curr}} \,. \tag{3.7}$$

Having carried out the above operations, the values of the partial derivatives are represented by nodes of the independent variables. As a result, we evaluate the gradient in a single DAG sweep.

3.4 Non-linear least-squares optimization

Non-linear least squares optimization is used to determine a set of parameters for which the function fits the observation data.

Let us define a mean squares error function as:

$$MSE = f(x) = \frac{1}{2} r(x) r(x)^T \tag{3.8}$$

The general non-linear least squares optimization problem can be formulated as follows:

$$\begin{aligned} \min_{x \in R^n} \quad & \frac{1}{2} r(V(x), M) r(V(x), M)^T \\ \text{subject to:} \quad & p(x) = 0 \\ & q(x) \leq 0 \\ & l \leq x \leq u \end{aligned} \tag{3.9}$$

where:

- $x = (x_1, x_2, ..., x_K)$ is a set of K input parameters
- V is the model function vector for N observations: $V_i(x) = V_i(x_1, x_2, ..., x_K)$ where $i = 1, ..., N$
- r is a residuum vector: $r_i(x, M_i) = V_i(x) - M_i$ for $i = 1, ..., N$
- $p(x)$ and $q(x)$ are inequality and equality constraint functions
- l and u are lower and upper bounds for the input parameters

The least-squares error function may be non-convex and have multiple local minima. The gradient-based optimization methods for the non-linear least squares functions utilize the gradient information V_i with respect to each parameter x_k.

3.5 Monte-Carlo simulation

Monte-Carlo (MC) simulation is the most efficient approach to determining the results of integral functions that are too complicated to be solved analytically, for example option pricing models. Its computational effort increases approximately linearly with the number of random samples, while the complexity of analytical solutions tends to increase exponentially [Hul12].

The key idea of MC simulation is a production of many random different scenarios (paths) and evaluation of the further expected value converging to the correct results with the number of paths. For financial models describing the evolution of an underlying asset price or volatility, the MC method assumes that each scenario is a sample payoff calculated and discounted at the interest rate:

$$C_i = e^{-rT} \cdot max(S_T - K, 0) \tag{3.10}$$

where C_i is the payoff – the option price along the i-th path, S_T is a commodity price at time T according to the i-th scenario, and K denotes the strike price (the previously negotiated price at which the commodity is traded at time T).

The expected value of the option price is equal to the average of all the discounted payoffs (M denotes the number of different payoff scenarios – paths), as below [Gla04]:

$$v_M = \mathrm{E}(\Phi(C_i)) \approx \frac{\sum_{i=0}^{M} \Phi(C_i)}{M} \tag{3.11}$$

Further, if we perform differentiation operations for the expected value, we must take into account all sample paths [Gla04]:

$$\frac{dv_M}{d\theta} = \mathrm{E}(\frac{d\Phi(C_i)}{d\theta}) \approx \frac{\sum_{i=0}^{M} \frac{d\Phi(C_i)}{d\theta}}{M} \; . \tag{3.12}$$

These values, known as the Greeks, measure the impact of model factors on the option price and are fundamental in risk management and hedging.

3.6 High-Performance Computing

High-Performance Computing is the use of parallel processing for running complex and/or lengthy applications more quickly. This can utilize hardware architectures such as GPUs, multi-/many-core processors and FPGA cards.

3.6.1 OpenMP framework

OpenMP is an application framework supporting multi-platform programming with shared memory [Ope13]. It consists of compiler directives, library routines and environment variables that allows running vectorized[2]. code on multi-core architectures. OpenMP standards uses fork-join model for parallelization. The main thread, termed the *Master*, forks into a specified number of threads. Program tasks are assigned to threads. The runtime environment system assigns threads to different processors located on multi-/many-core CPUs. This paradigm is suitable to execute `for` and `while` loops without data-dependency between subsequent iterations. After `for` loop execution, threads join back to the main thread. OpenMP supports clauses for data specification inside the paralelized loop. The data can be shared within a parallel region – all

[2]Vectorization is a technique to process a vector of data in a single-clock cycle.

threads have access to the same data or private data which means all threads process a local copy of the data [Ope13]. Further, reduction operations are supported. When the parallelized `for` loop ends, all private variables can be summed into a global shared variable. This is particularly useful for *embarrassingly parallel* problems [3], hence 8 double-precision numbers or 16 single-precision numbers can be processed in a single clock cycle. Xeon-Phi supports two modes of code execution: native mode, where the standalone program is directly run on the coprocessor; offload mode, where the code region is run on the coprocessor. Xeon-Phi technology utilizes OpenMP or OpenCL framework to run the code on the Xeon-Phi coprocessor. Xeon-Phi provides instructions to transfer data to/from the coprocessor, offload and native mode configuration and data-alignment [Int16].

3.6.3 CUDA technology

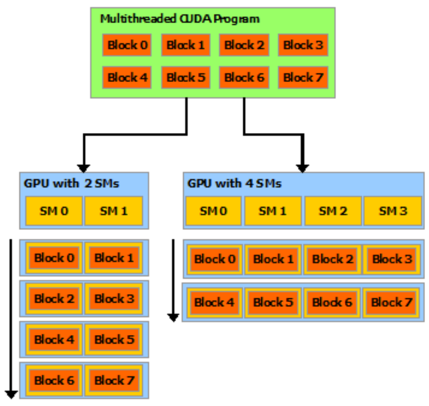
Introduction

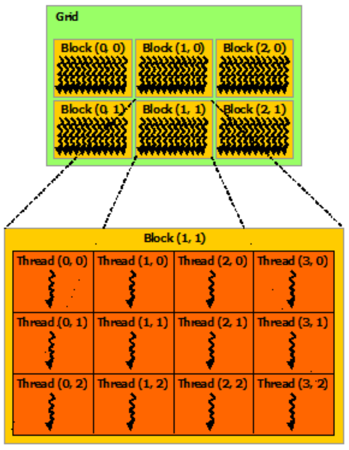
CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model supporting general-purpose computations on NVIDIA GPU cards. This platform is well-suited for embarrassingly parallel problems. To specify a computational problem, CUDA allows kernel function definition. The kernel function is processed by many threads on different streaming GPU processors. Each streaming processor operates on a thread-block – a group of 1024 threads sharing memory resources. Hence, performance is scalable with the number of available streaming processors. Figure 3.2a shows a sample graph representing execution with 2SMs and 4SMs. Thread-blocks can be identified into one-dimensional, two dimensional or three

[3]The computation problem that can be split into a number of independent tasks without data dependency and communication between them [HS12] as MC simulation. OpenMP allows using synchronization instructions to indicate the sequential part of the code inside the `for` loop. This defines atomic operations (in a multi-threading system, an atomic operation is one that cannot be interrupted during processing by another thread), critical sections (the critical region is executed by only one thread) and barrier functions (all threads must wait before any is allowed to proceed). Additionally, for performance improvement of processing data, the OpenMP defines directives for data vectorization.

3.6.2 Xeon-Phi framework

Xeon-Phi is a many-core hardware architecture designed for embarrassingly parallel problems . A parallelized program is run on an Intel Xeon-Phi Coprocessor providing up to 61 cores that simultaneously compute 244 tasks (threads), with a performance peak of about 1.2 teraFLOPS ($1.2 \cdot 10^{12}$ FLOPs). Each core is based on an Intel Pentium 3 architecture. Xeon-Phi technology supports 512-bt SIMD (Single Instruction-Multiple Data) instructions [Int16] dimensional indexes. These can be utilized when processing large data of vectors, matrices or volumes. Figure 3.2b shows the thread hierarchy. The total number of threads is a number of threads per block times the number of thread-blocks [NVI16].





(a) NVIDIA CUDA technology— Scalability with the number of streaming processors

(b) NVIDIA CUDA technology— Thread-hierarchy

Synchronization methods within threads

CUDA technology supports synchronization instructions to avoid race conditions or hazards while reading from or writing to memory.

- a local barrier for threads within a single thread-block
- a global barrier within all threads located on the thread grid

These instructions can find applications in a sequential part of the code such as reduction of the vector elements.

Memory model

CUDA allows access to several memory spaces located on GPU cards. Each thread has access to its own memory located in registers. Further, thread blocks can operate on the shared-memory located close-to-chip. The size of the shared-memory depends on the compute capability of a device (SM version). For the SM 2.1, the shared memory size is 64kB per thread-block. All threads have access to global memory space located in the GPU memory.

Execution model

CUDA differentiates code run on the CPU from the code executed on the GPU. The program run on the CPU is termed the *host*; the kernel program performed on GPU is termed the *device*. CUDS supports an asynchronous execution on multiple devices by using streams. The streams allow asynchronous data transfer to/from device and kernel calls.

3.6.4 OpenCL framework

OpenCL (Open Computing Language) is a framework designed to simplify cross-platform programming, providing a common low-level API for many-core architectures, GPUs and FPGAs [Khr16]. OpenCL specifies a programming language based on C99.

3.6.5 Maxeler technology

The Maxeler perating system (MaxelerOS) is a computational framework supporting processing on FPGA cards developed by vendors such as Xilinx. This offers an approach to parallelism known as multi-scale dataflow processing [max14].

In this paradigm, the application is considered as a dataflow graph that consists of elementary nodes corresponding to the arithmetic operations and variables. The graph is processed from the independent nodes through intermediate nodes ending up at the final node. Each node contains a data corresponding to a single-arithmetic operation. The dataflow graph is computed by a reconfigurable Dataflow Engine (DFE) consisting of thousands of cores [max14]. Each node is processed by a single dataflow core. Maxeler multi-scale (pipeline) extensions allow processing arrays/vectors of nodes to optimally utilize FPGA capacity. The pipelining strategy supports loop level parallelism of vectors via spatial and pipelined processing. Flexible definition of primitive data types as floating-point or fixed-number types [max14] is supported. The floating-point data can be defined by using various range of bits for exponent and mantissa (8 bits for mantissa and 24 bits for the exponent corresponds to the single-precision floating-point type; 11 bits for mantissa and 53 for the exponent represent double-precision floating-point numbers). These bits are further mapped to a host CPU application written in C. Fixed-point types are specified by the number of bits to represent integer numbers. Accelerated Maxeler applications consist of three components:

- Kernel – describes computation flow structurally by DAGS. DAGS consist of several types of node, such as arithmetic operations, values, stream offsets, multiplexers (condition instructions, counters, input/output operations). The kernel graphs are directly mapped to hardware and computed by dataflow engines.
- Manager configuration – interconnects a host CPU to an FPGA card. This includes input output stream information and memory accessing. The Manager usually calls the Kernel instantiation.
- Host application – the host CPU application that calls the FPGA configuration and transfers data from RAM to FPGA memory. The host application is also responsible for data streams between CPU and FPGA.

This overall application is managed by the MaxelerOS. Additionally, multiple dataflow engines are connected to linearly scale the problem with the number of available DFEs [max14].

3.6.6 OpenMPI framework

OpenMPI (Open Message Passing Interface) is a message passing interface supporting distributed computing. This defines an API that allows inter-process communication within the nodes in a cluster. The API offers point-to-point communication functionality, message broadcasting [ope16].

3.7 Financial case-studies

3.7.1 The Heston Model

The model forecasting behaviour of commodity prices is often expressed as a stochastic process (changes of the value are uncertain over time and are dependent on some probability factor). Besides the commodity price, this process allows prediction of the further volatility of the asset. This is called the *stochastic volatility model* [Gat06].

The Heston model is an extension of the Black-Scholes model that proposes the underlying asset price follows the various volatilities (observable on the market) over time (V_t is a stochastic process) [AWVdS13]. The commodity price of Heston satisfies the following stochastic equation [Gat06]

$$dS_{t+1} = (r-q)S_t dt + \sqrt{V_t} S_t \cdot W_t^1 \ . \tag{3.13}$$

where $r-q$ is its expected rate of return, V_t denotes a volatility and $W(t)$ is a Wiener process (the Brownian motion - a stochastic process whose increments over time have normal distribution and are independent of the previous process evolution). The non-constant volatility is expressed as a mean-reverting stochastic process of the form [AWVdS13]:

$$dV_{t+1} = \kappa \cdot (\theta - V_t) \cdot dt + \sigma \cdot \sqrt{V_t} dW_t^2 \ . \tag{3.14}$$

where κ denotes the mean reversion of volatility, θ the long-term variance, σ the volatility of volatility and V_0 the initial volatility. dW_t^1 and dW_t^2 are correlated random variables with normal distribution (the correlation factor is equal to ρ). To ensure positive volatility, the Feller condition needs to be taken into account [Gat06].

$$2 \cdot \kappa \cdot \theta - \sigma^2 > 0 \tag{3.15}$$

3.7.2 The Heston Model with Jumps

The Heston model can be extended (on the Poisson processes) to express more accurately the market quotes [Gat06]. In the Heston model with jumps, the SDE is as follows:

$$dS_{t+1} = (r - q - \lambda \mu_J) S_t dt + \sqrt{V_t} S_t \cdot W_t^1 + J_t dN_t \ . \tag{3.16}$$

where $N = N_t, t \geq 0$ is an independent Poisson process with the intensity parameter $\lambda > 0$, J_t is the percentage jump size that is lognormally distributed over time with unconditional mean μ_J. The standard deviation of $log(1+J_t)$ is σ_J:

$$log(1+J_t) \sim Normal(log(1+\mu_J) - \frac{\sigma_J^2}{2}, \sigma_J^2) \tag{3.17}$$

3.7.3 Heston model with term-structure

The Heston model with term structure assumes that the time-dependent parameters are piecewise constant [Li09]. This model expresses options with different maturities more accurately. The stock price SDE for the Heston model with term structure is as follows:

$$dS_{t+1} = (r-q) S_t dt + \sigma_t \sqrt{V_t^{TS}} S_t \cdot W_t^1 \ . \tag{3.18}$$

and the volatility SDE is:

$$dV_{t+1}^{TS} = \lambda_t (1 - V_t^{TS}) dt + \alpha_t \sqrt{V_t^{TS}} S_t \cdot W_t^1 \ . \tag{3.19}$$

The parameters λ_t, α_t and σ_t are retrieved from the standard Heston model by using the formulas: $\sigma_t = \sqrt{\theta}$, $\alpha_t = \frac{V_t}{\sqrt{\theta}}$, $V_t^{TS} = \frac{V_t}{\theta}$

The semi-closed form for the Heston model

The semi-closed form of the Heston model for European call options is given as [Gat06]:

$$C(S_t, V_t, t, T) = S_t P_1 - K e^{-r(T-t)} P_2 \tag{3.20}$$

where the value of P_1 and P_2 is calculated by equation

$$P_j(x, V_t, T, K) = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty Re\left(\frac{e^{-i\phi \ln(K)} f_j(x, V_t, T, \phi)}{i\phi} \right) d\phi \tag{3.21}$$

$$x = \ln(S_t)$$

$$f_j(x, V_t, T, \phi) = \exp\{C(T-t, \phi) + D(T-t, \phi)V_t + i\phi x\}$$

$$C(T-t, \phi) = r\phi i\tau + \frac{a}{\eta^2}\left[(b_j - \rho\eta\phi i + d_j)\tau - 2\ln\left(\frac{1 - g e^{d_j \tau}}{1-g}\right)\right]$$

$$D(T-t, \phi) = \frac{b_j - \rho\eta\phi i + d_j}{\eta^2}\left(\frac{1 - e^{d_j\tau}}{1 - g e^{d_j \tau}}\right)$$

$$g = \frac{b_j - \rho\eta\phi i + d_j}{b_j - \rho\eta\phi i - d_j}$$

$$d_j = \sqrt{(\rho\eta\phi i - b_j)^2 - \sigma^2(2u_j\phi i - \phi^2)}$$

$$u_1 = \frac{1}{2},\ u_2 = -\frac{1}{2},\ a = k\theta,\ b_1 = k + \lambda - \rho\eta,\ b_2 = k + \lambda$$

for j=1,2, where,

$$u_1 = \frac{1}{2},\ u_2 = -\frac{1}{2},\ a = k\theta,\ b_1 = k + \lambda - \rho\eta,\ b_2 = k + \lambda$$

The parameter λ can be eliminated under the risk neutral measure, i.e., underEMM^Q,

$$dV_t = \kappa^*(\theta^* - V_t)dt + \sigma\sqrt{V_t}dW_t^2 \tag{3.22}$$

where,

$$\kappa^{*=}\kappa+\lambda,\ \theta^{*}=\frac{\kappa\theta}{\kappa+\lambda}$$

therefore, the parameters in the close-form solution can be expressed as:

$$a=\kappa^{*}\theta^{*},\ b_1=\kappa^{*}-\rho\eta,\ b_2=\kappa^{*}$$

Financial sensitivity calculation

For risk management and hedging investment portfolios, traders evaluate the Greeks to measure how the model parameters affect future commodity price and option price. The impact on the option value is evaluated to quantify the different aspects of risk. When risk is acceptable, no adjustment is made to the investment portfolio; if it is unacceptable, an appropriate position for either the underlying asset or option contract is taken.

For risk management and hedging investment portfolios, traders evaluate the Greeks to measure how the model parameters affect future commodity price and the option price [Hul12].

The impact on the option value is evaluated to quantify the different aspects of risk. When risk is acceptable, no adjustment is made to the investment portfolio; if risk is unacceptable, an appropriate position for either the underlying asset or option contract is taken.

The underlying concept of the Greeks' calculation combines symbolic differentiation and the Adjoint.

Considering the first-order Greeks, the recursive formulas (along each scenario via MC simulation) must be derived.

$$S_{t+1}=F_t(S_t,\mu,V_t)=S_te^{(r-\frac{1}{2}V_t)dt+\sqrt{V_tdt}}\cdot Z_1 \tag{3.23}$$

$$V_{t+1}=G_t(\kappa,\theta,\sigma,V_t)=V_t+\kappa\cdot(\theta-V_t)dt+\sigma\cdot\sqrt{V_tdt}Z_2 \tag{3.24}$$

Table 3.2a shows the recursive formulas for the first-order Greeks through a single path of the Heston model.

As can be observed, the Greeks at time $t+1$ are dependent on the previously evaluated Greeks at time t and the partial derivatives of G and F.

Based on these recursive dependencies of underlying commodity price and volatility, the Adjoint can be utilized to calculate the partial derivatives of F and G. Further,

The Greek	Recursive equation (the first-order)
$\frac{dS_{t+1}}{dS_0}$	$\frac{dF_t(S_t)}{dS_t} \cdot \frac{dS_t}{dS_0}$
$\frac{dS_{t+1}}{dr}$	$\frac{dF_t(S_t)}{dS_t} \cdot \frac{dS_t}{dr} + \frac{dF_t(r)}{dr}$
$\frac{dS_{t+1}}{dV_t}$	$\frac{dF_t(V_t)}{dV_t} \cdot \frac{dV_t}{dV_0} + \frac{dF_t(S_t)}{dS_t} \cdot \frac{dS_t}{dV_0}$
$\frac{dS_{t+1}}{d\kappa}$	$\frac{dF_t(V_t)}{dV_t} \cdot \frac{dV_t}{d\kappa} + \frac{dF_t(S_t)}{dS_t} \cdot \frac{dS_t}{d\kappa}$

(a) Formulas for first-order Greeks of the underlying asset

these results are used to update the final first-order Greeks.

3.8 Summary

In this chapter, the technical context of the work has been presented. This focused on DAG – directed acyclic graph representation used to represent a composite function. Further, the Automatic Differentiation methods with the forward and reverse mode for the gradient calculation have been presented. Next the non-linear least squares optimization is presented. Further, the Monte-Carlo method has been presented. Next, the non-linear least squares optimization has been introduced. Further, the overview of High-Performance technologies has been presented. Further, the financial-case studies: the Heston model, the Heston model with Jumps, the Heston model with term-structure have been introduced. In the next chapter, there are presented parallel frameworks for financial sensitivity calculation and model calibration.

Chapter 4

High-Performance frameworks for financial risk management

4.1 Introduction

This chapter presents high-performance approaches for financial risk management. This includes:

- Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint
- Heston model calibration using the Adjoint and Monte-Carlo methods on FPGA
- Parallel non-linear least squares optimization framework using Automatic Differentiation

4.2 Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint

4.2.1 Introduction

Monte-Carlo (MC) simulation is a widely used method to solve processes that involve uncertainty or that are too complex to solve analytically [Cha13]. When using MC simulation, the sensitivity and calibration analysis of the models needs to be measured: how is the model sensitive to changes, what and how do the model parameters affect output, and how far is the model is from the real answer. This process is computationally expensive when solving models with millions of different scenarios. The models are usually expressed as analytical recursive equations in which the solution at time $t+1$ depends on the model parameters and the state at time t [Cha13]. The model parameters need to be determined by calibrating the model to observations. Therefore, the model can be generalized as a function of input parameters and the previous solution. This function can be expressed as a sequence of operations with specified dependencies and graphically represented as a Directed Acyclic Graph (DAG)[FHP$^+$12]. The graph nodes represent the variables or intermediate and final operators, whereas the edges represent their dependencies. This abstraction can be effectively used with parallel and distributed architectures to represent general models.

This section presents a parallel engine for Monte-Carlo (MC) for the expected value computation, the first-order sensitivity calculation and the model calibration using a Directed Acyclic Graph (DAG) representation. This supports overloaded operator techniques, polymorphism and objected oriented design to ensure a flexible model definition. Additionally, the DAG abstraction allows a cross-platform definition that is independent of particular high-performance architectures. User code for specific application can be run on high-performance platforms without modification and recompilation.

The sensitivities are calculated on parallel and distributed platforms via a single simulation of the Adjoint – an Automatic Differentiation method with the gradient computation cost being 1.8x times that of function evaluation. The gradient is further used in unconstrained and constrained optimization methods to calibrate the model to the observed data. The engine supports multi-/many-core and distributed computing architectures such as GPUs and Intel-Xeon-Phi with the frameworks OpenMP, CUDA, OpenCL, OpenMP with Xeon-Phi. Further this includes an OpenMPI module that allows simulation with the above architectures over the cloud. A static and dynamic library are provided.

The engine is demonstrated using a financial case-study: the first-order sensitivity calculation and model calibration for the Heston model, the Heston model with Jumps and the Heston model with term-structure for derivatives pricing and hedging. The Heston model is considered with constraints to ensure the positive volatility calculation. The computational cost of the gradient calculation is reduced by a factor of 16x when compared to the Finite Difference methods. Further, use of parallel/distributed architectures improves performance by approx. 100x on two GPU NVIDIA K40 cards vs. a single-thread implementation on an Intel Xeon X5650.

4.2.2 Overview

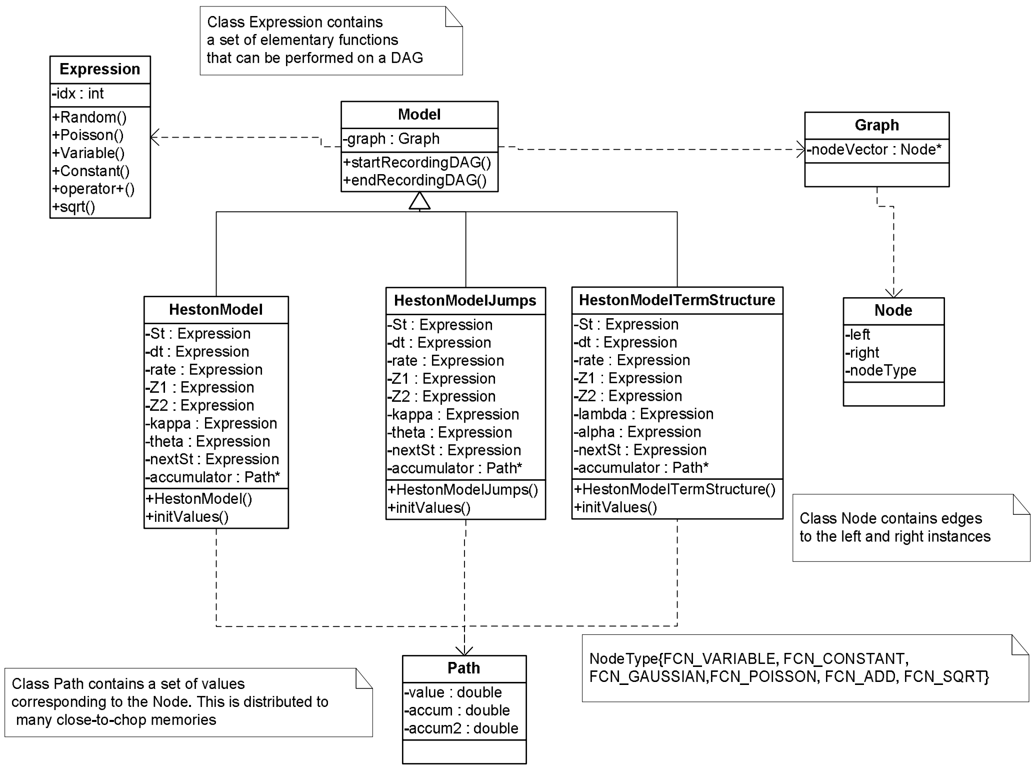
The parallel engine for MC simulation using DAG processing consists of the following modules:

- An integration module with DAG components that integrates the HPC components with the model definitions and calls MC functions on a specific architecture;
- A HPC module supporting multi-/many-core architectures via OpenMP, OpenMP with Xeon-Phi, CUDA, OpenCL framework for MC simulation and AD processing using DAGs;
- A distributing computing module supporting OpenMPI functions that enables parallel simulation and calibration on a cluster;
- A calibration module supporting constrained and unconstrained optimization using AD;
- A market-data module that connects to the SQL database to fetch observations for the calibration process [1];
- A test-suite module that allows performance, calibration benchmarks and tex-report generation.

The engine is compiled as a shared/static library. This provides an API that allows a platform-independent model definition and execution on parallel/distributed computing. The HPC module is designed according to the SIMD (Single Instruction Multiple Data) paradigm. This means that a single graph representation [2] is utilized by many threads and cores available on multi-/many-core architectures. Additionally, each thread/core operates on the *Path* – the separate data structure located in the close-to-chip memory to produce the intermediate partial results. The path contains the intermediate model results and the first-order derivatives. The functions processing a single DAG representation with the forward and reverse order utilize vectorization and multi-threading.

[1]The market-data module uses SQLite for the connection and database query execution [sql16]

[2]a single DAG contains a set of instructions required to evaluate the specific stochastic model



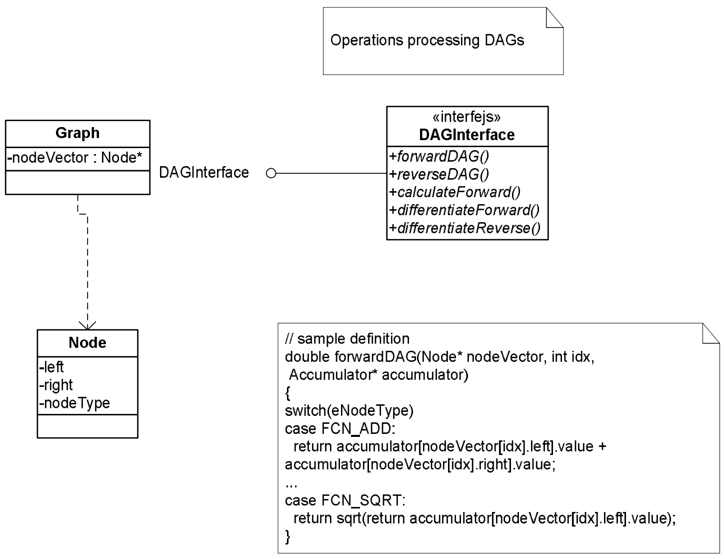
Class Expression contains a set of elementary functions that can be performed on a DAG

Class Node contains edges to the left and right instances

NodeType{FCN_VARIABLE, FCN_CONSTANT, FCN_GAUSSIAN,FCN_POISSON, FCN_ADD, FCN_SQRT}

Class Path contains a set of values corresponding to the Node. This is distributed to many close-to-chop memories

Figure 4.1: Architecture – Model Definition



Operations processing DAGs

```
// sample definition
double forwardDAG(Node* nodeVector, int idx,
 Accumulator* accumulator)
{
switch(eNodeType)
case FCN_ADD:
  return accumulator[nodeVector[idx].left].value +
accumulator[nodeVector[idx].right].value;
...
case FCN_SQRT:
  return sqrt(return accumulator[nodeVector[idx].left].value);
}
```

Figure 4.2: DAG processing on HPC platforms

4.2.3 Architecture

Directed Acyclic Graph (DAG) representation

The objective of the parallel DAG library for MC simulation is a flexible definition of various graphs representing process models with processing on HPC architectures. This requires an appropriate data model design compatible with the parallel frameworks and architectures. To allow a flexible definition and execution on parallel/distributed platforms the objected-oriented paradigm and generic design have been utilized. The software is additionally integrated with external functions to process DAG operations for specific applications (DAG differentiation, DAG reduction). This approach allows a DAG definition by using:

- *String* representation – a sequence of characters to formulate the model;
- *Expression* representation – data structures representing the DAG nodes; these additionally utilize overloaded operator techniques to define the model and its constraints.

DAG as a String This data model enables integration with external languages and software for mathematical modelling. The first stage is a definition of elementary algebra functions such as binary and unary operators: addition, multiplication, sin, cos and their relative priorities. The priority array with the supported functions is further utilized to interpret the DAG – a string passed by the user. The function parsing the graph model recognizes elementary algebra operators, left and right brackets and adds the lexemes [3]. The lexeme expresses a node of a DAG. For execution, memory is allocated to build a DAG; then the array of all the lexemes expressing operators and nodes is processed by the function – *createDAG*. In this phase, the *Expression*, a data structure, is utilized to build a graph.

DAG as a set of expressions The model utilizing expression structures allows integration with C++ software. The underlying concept is to employ overloaded operator techniques and polymorphism to define a DAG. The specific graph definition class inherits from the base class – *Model*, that is responsible for memory allocation. The nodes are represented by objects of the class – *Expression*, and intermediate algebra

[3] A lexeme is an abstract unit of dictionary[Alf06] which can be used in the context of language or algebra definition

functions. The overloaded operator techniques allow the graph to be interpreted at run-time. Figure 4.1 presents a data model and a sample graph definition for the Heston model.

Numerical inputs/outputs and hardware configuration To properly process DAGs on parallel architectures, an appropriate connection of numerical values with nodes of the graph is required. For this purpose, the class *Path* is implemented. This contains a set of numerical values referring to a single node. The *Path* is processed in parallel by multiple threads and cores located on the multi-/many-core architectures [4] The library supports different HPC platforms and their parameters are configured via the class *HardwareParameters*. This contains a set of parameters such as the number of threads, the number of machines, and the device id or the number of block per thread grid. The class *OutputParameters* is initialized with the results of the DAG processing, such as estimatedValue, firstSensitivities etc.

DAG interface – evaluation and automatic differentiation functions processing DAGs

The library provides a set of functions to process DAGs. These are applied in the context of the first-order sensitivity calculation. The functions processing DAGs are implemented on the device and are optimized in terms of performance and memory usage. They utilize vectorization and multi-threading techniques. Figure 4.2 shows a DAG interface model and a sample definition of the function processing DAGs.

HPC module

The concept of the HPC module for the first-order sensitivities is a combination of numerical methods, such as the MC method and the Adjoint, and HPC systems to improve the performance and accuracy of the MC simulation, its sensitivity calculation and model calibration. The first-order sensitivities via MC are parallelized with respect to the number of scenarios. Each scenario is calculated and differentiated through the Adjoint method on a single core on GPU or thread on a multi-core architecture. The MC simulation module supports definition of various financial models, various discretization schemes and different data types by using a DAG representation. The first

[4]The Xeon-Phi implementation utilizes vectorization functions that simultaneously process either 8 or 16 paths in offload mode. The parallelized part of code is copied to the Xeon-Phi coprocessor. The GPU implementation processes each warp – a set of 32 threads simultaneously operating 32 paths.

stage is recording the discretization scheme (Euler or Milstein) by using expressions or string representations. At this stage, the overloaded operators at a high level call low-level routines that create the DAG representing the processing flow inputs/intermediate variables/outputs and their data-dependency (nodes). This graph representation of discrete models is transferred to the HPC platforms to enable the symbolic differentiation (the Adjoint). The next step is the model parameter initialization with numeric values (the numeric values are assigned to appropriate graph nodes). Further, the number of paths and steps per single simulation and the input/output model arguments are passed to the main function running MC simulation and the sensitivity computation. The first stage of parallel scenario processing by core or thread is the uniform random generation. The implementation supports various random number generators (pseudo/quasi generators) and libraries such as NAG [Gro15], CURAND [NVI13], MKL [Int16], and MW64X [Tho14]. The second step is DAG evaluation (calculation of the next MC step). Then, the reverse sweep via Adjoint can be utilized on the DAG to calculate the partial first-sensitivities. After, the partial sensitivities can be updated to produce the final derivatives at the timestep. When the MC simulation is completed, the partial results and sensitivities are collected to calculate the estimated value. The graph 4.3 presents the processing flow on HPC architectures.

CUDA & OpenCL The MC simulation on GPU is performed by calling kernel routines 1. The kernel function takes the DAG graph representing the discretization model, its size, and the total number of paths and the number of steps as parameters. This produces the final result, and the first-order sensitivities. Each thread calculates a single MC scenario and first-order sensitivities via the Adjoint. The GPU thread iterates the DAG calculation and differentiation sequence until the final step. Next, the inner loops process the DAG with the forward order (evaluation) and the backward order (the Adjoint) to produce the first-order sensitivities. After this, the final first-order sensitivities can be updated. The final stage is collection (reduction) of the partial results and evaluation of the estimated values. This utilizes synchronization and barrier methods. The pseudocode 1 shows a parallel algorithm on GPU.

Xeon-Phi The Xeon-Phi implementation utilizes OpenMP (OMP) and Xeon-Phi instructions to parallelize DAG processing. Each thread exploits vectorization techniques to evaluate and differentiate 16 or 8 scenarios at once represented by single or double-precision numbers respectively. The outer loop uses OMP pragmas and

Algorithm 1 GPU Kernel implementation

```
Require: DAG, dagSize, numPaths, numSteps, numOptions, inputDataVector, outputDataVector
Ensure: outputDataVector contains model results and the first-order sensitivities
 1: pathId $\leftarrow$ getNumberOfThread()
 2:                                              $\triangleright$ DAG initialization
 3: for step $\leftarrow$ 1, numSteps do
 4:     for graphId $\leftarrow$ 1, dagSize do
 5:         EvaluateDAG(DAG, graphId)
 6:     end for
 7:     for graphId $\leftarrow$ dagSize, 1 do
 8:         AdjointDAG(DAG, graphId)
 9:     end for
10:          $\triangleright$ Update the first-order sensitivities using recursive formulas
11: end for
12:                                     $\triangleright$ Reduction within thread block
```

iterates through a number of scenarios. The next iteration is to process a single scenario through evolution time. Two inner loops iterate the DAG to produce the results (forward order) and sensitivities (reverse order). The results are stored in 8 double-precision or 16 single-precision number vectors. This approach maximizes performance on Xeon-Phi Coprocessor [5]. The pseudocode 2 shows the parallel algorithm for the Xeon-Phi Coprocessor.

OpenMPI The OpenMPI module integrates the modules supporting parallel processing. This allows distributed simulation and model calibration in a cluster. This consists of the master node and the computation nodes. The master node sets initial values such as the total number of paths, the number of steps, and MC mode and hardware configuration for the model and distributes them to the nodes. Each node computes the number of paths divided by the total number of nodes available. Each node then sends the output results back to the master node. The master node then calculates the overall expected value result. The listing 4.5 shows the processing flow for the option pricing via OpenMPI. The listing 4.8 shows the processing flow for the model calibration using OpenMPI.

[5]Intel Xeon-Phi Coprocessor maximizes performance when vectors consisting of 8 or 16 elements are processed at once [Int16]

Algorithm 2 Xeon-Phi implementation

Require: DAG, dagSize, numPaths, numSteps, numOptions, inputDataVector, outputDataVector
Ensure: outputDataVector contains model results and the first-order sensitivities

```
1:                                                        ▷ DAG initialization
2:                                                     ▷ pragma omp parallel for
3: for pathId ← 1, numPaths do
4:     for step ← 1, numSteps do
5:         for graphId ← 1, dagSize do
6:                                                   ▷ the for loop vectorized
7:             for vectorId ← 1, 8 do
8:                 EvaluateDAG(DAG, graphId)
9:             end for
10:        end for
11:        for graphId ← dagSize, 1 do
12:                                                  ▷ the for loop vectorized
13:            for vectorId ← 1, 8 do
14:                AdjointDAG(DAG, graphId)
15:            end for
16:        end for
17:                    ▷ Update the first-order sensitivities using recursive formulas
18:    end for
19: end for
20:                                                ▷ Reduction within thread block
```

4.2.4 Deployment Process

The engine provides functionality that allows a platform-independent model definition and execution on HPC without the core re-compilation process. User code with the model definition and processing flow is linked with shared/dynamic libraries. The external code should include the model definition: its constraints, boundary conditions and processing flow. The user's class with the model inherits from the base class – `Model`. The class Model initializes DAG structures. During user code compilation the model is transformed into the DAG representation. The listing 5.17 presents a sample model definition. The listing 4.6 shows the parameter definition. The listing 4.7 shows the processing flow for the model calibration.

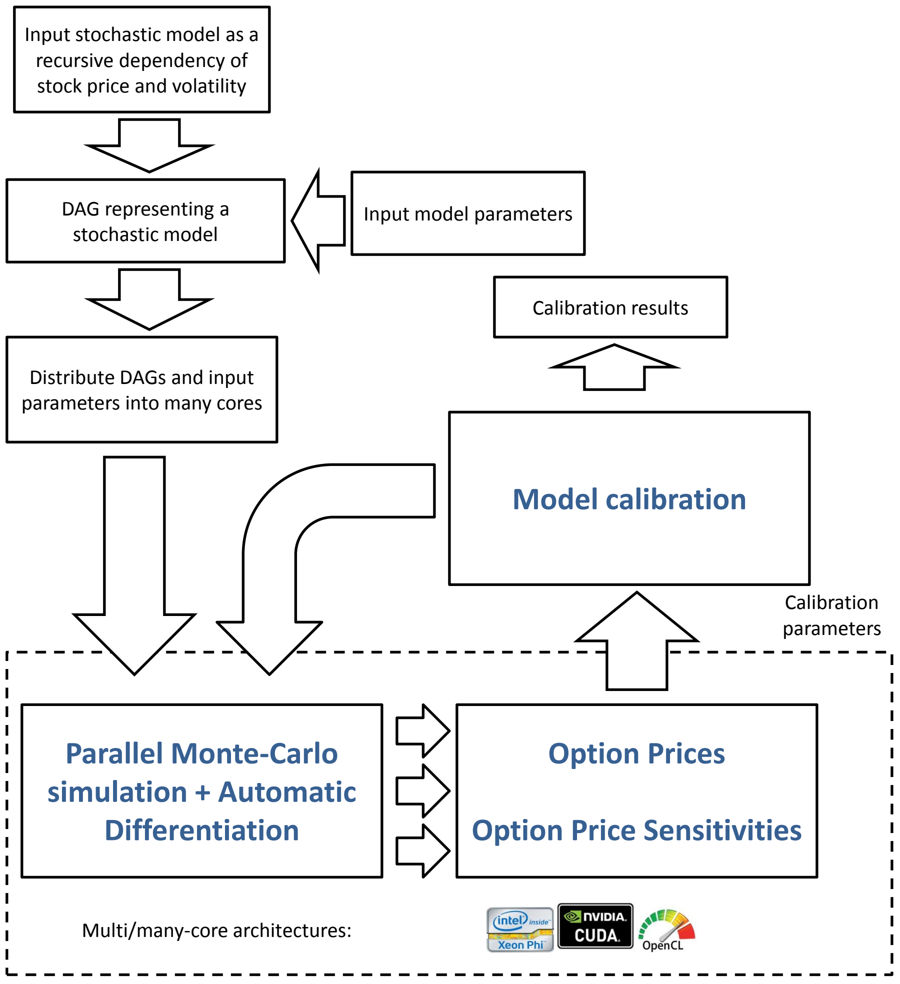

Input stochastic model as a recursive dependency of stock price and volatility
DAG representing a stochastic model
Input model parameters
Calibration results
Distribute DAGs and input parameters into many cores
Model calibration
Calibration parameters
Parallel Monte-Carlo simulation + Automatic Differentiation
Option Prices
Option Price Sensitivities
Multi/many-core architectures:
intel inside Xeon Phi
NVIDIA CUDA
OpenCL

Figure 4.3: Architecture – High-Performance Engine for Monte-Carlo simulation and model calibration

4.2.5 Summary

In this section, the architecture of the Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint has been presented. The work utilizes overloading operator and graph processing to support general stochastic differential models. The work can be applied to various industrial and scientific applications. Computational experiments for the Parallel Monte-Carlo engine for the first-order sensitivities calculation and model calibration using the Adjoint are presented in Chapter 5.

```
// import headers from the SDK library
#include <Model.h>
// Heston model -- definition
class HestonModel : public Model
{
public:
Key key;
Expression St;
...
Expression constraint;
void initModel()
{
// Recording Expressions and building a graph structure
this->startRecord();
Expression::Variable(Zs);
...
nextSt = St * exp(((rate - q) - factor * Vtt) * dt + sqrt
   (Vtt * dt) * Zs);
// constraint definition
constraint = 2 * kappa * theta - sigma * sigma;
this->endRecord();
}
};
```

Figure 4.4: HPC Engine for MC simulation – Model Definition

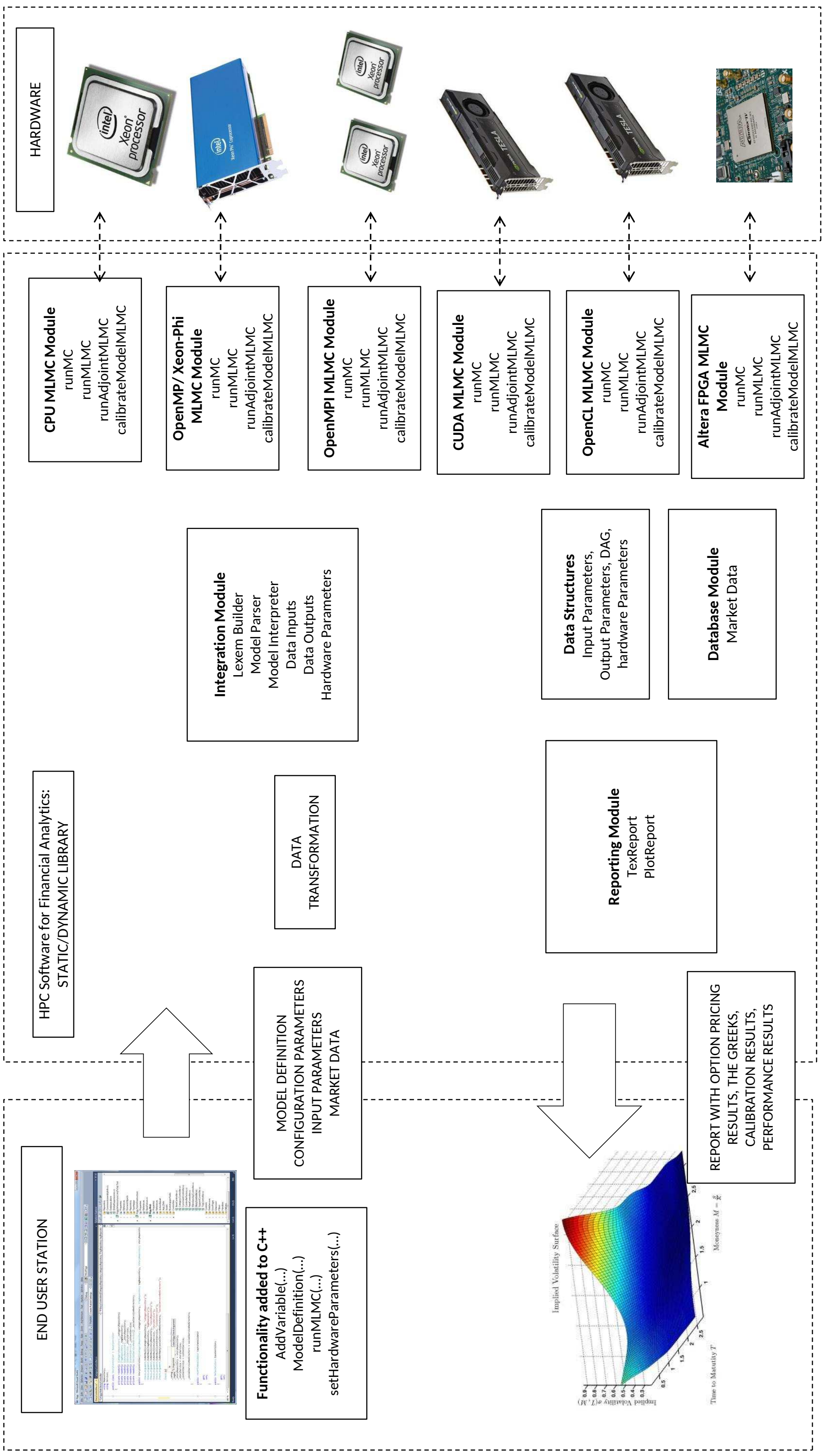


Figure 4.5: Architecture – High-Performance Engine for Monte-Carlo simulation and model calibration

```
// import headers from the SDK library
#include <Model.h>
// Heston model -- definition
class HestonModel : public Model
{
public:
Key key;
...
void initParameters()
{
// Calibration parameter initialization
key.nodeId = sigma.idx;
values[key].value = 0.3;
values[key].isCalibrated = true;
values[key].numEquation  = VOLATILITY_EQUATION;
bounds[key].lb = 0.1;
bounds[key].ub = 1.0;
variableNames[key] = "sigma";
...
}
};
```

Figure 4.6: HPC Engine for MC simulation – Parameter definition

```
// import headers from the SDK library
#include <HPCEngine.h>
#include "HestonModel.h"
// available types: double, Dual, HyperDual, Interval
define DATA_TYPE double;
main()
{
HestonModel hestonModel;
MCEngine mcEngine;
DiffMode diffMode = Adjoint;
HardwareParameters hardwareParameters;
// OpenMP, OpenMPI, Xeon-Phi, FPGA, GPU
hardwareParameters.platform = H_CUDA;
...
// Option Price/Implied Volatility Matrix
OptionPriceData optionPriceData("data/data.txt");
// the Greeks' calculation
mcEngine.runSimulation(hestonModel, mcMode, diffMode);
// calibration
mcEngine.runCalibration(hestonModel, optionPriceData);
// report
reportTex("report/report.tex");
};
```

Figure 4.7: HPC Engine for MC simulation – Processing flow

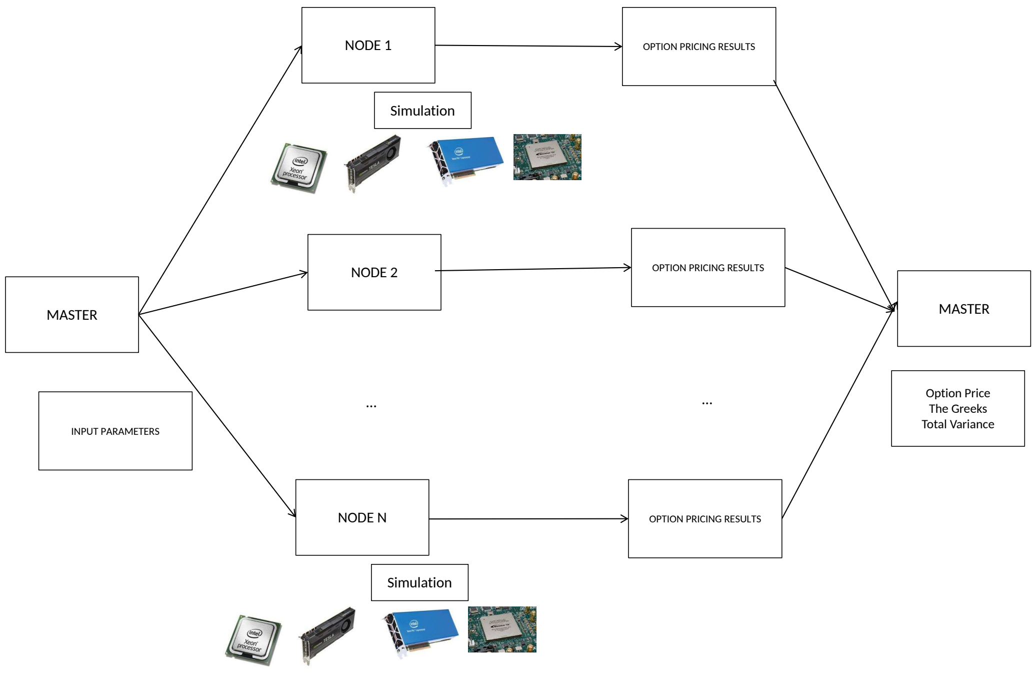


Figure 4.8: Architecture – HPC Engine for Monte-Carlo simulation and model calibration

4.3 Heston model calibration using the Adjoint and MC methods on FPGA

4.3.1 Introduction

Pricing derivatives, calculating the Greeks and calibrating the models to market data are computationally-demanding problems in the financial and insurance sector [Gla04]. To manage investment portfolios, financial institutions need to solve complex stochastic differential models via Monte-Carlo (MC) simulation. When using MC simulation, pricing, sensitivity calculation and model calibration via simulation is computationally expensive. Furthermore, the output results are burdened by numerical error if finite-differences and likelihood methods are used. This section presents a Monte-Carlo (MC) engine for the expected value computation, the first-order Greeks' calculation and model calibration using FPGA cards. The sensitivities are calculated in parallel via a single simulation by the Adjoint – an Automatic Differentiation method with

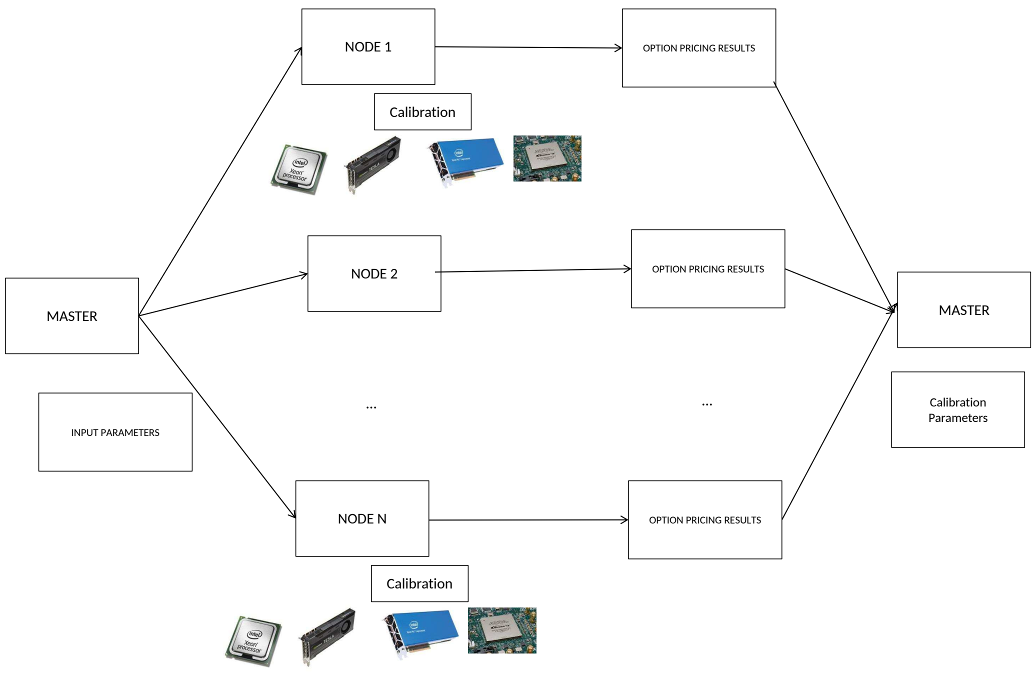


Figure 4.9: Architecture – HPC Engine for Monte-Carlo simulation and model calibration

the gradient computation cost being 1.8 times that of function evaluation. The gradient is applied to the optimization methods to calibrate the model to the market data. The framework supports FPGA cards with Maxeler technology. The computational experiments consider a financial case study: the Heston model calibration.

4.3.2 Overview

The FPGA engine for the financial option pricing, sensitivity calculation and model calibration via MC simulation consists of the host module written in C++, the hardware Maxeler application with Kernel and the Manager configuration. The hardware implementation is written in the MaxJ language [max14]. The FPGA implementation uses hand-coded routines for option pricing and the model gradient evaluation.

Algorithm 4 FPGA kernel implementation

```
 1: procedure HESTONMODELKERNEL
 2:     DFEVector carriedSt ← newInstance()
 3:     DFEVector carrieddStGreek ← newInstance()
 4:     DFEVector carriedAccSt ← newInstance()
 5:     DFEVector previousSt ← init(control, carriedSt)
 6:     DFEVector previousdStGreek ← init(control, carrieddStGreek)
 7:     DFEVector previousAccSt ← init(control,
 8:     carriedAccSt)
 9:                         ▷ Evaluation and differnentiation of the chain-rule
10:     inputs.St ← previousSt
11:     DFECVector newSt ← nextStockPrice(inputs, chainrule, Zvar)
12:     AdjointStockPrice(inputs, chainrule, derchainrule, greeks)
13:     DFEVector newPrice ← Payoff(control, newSt, Strike)
14:     DFEVector newGreekSt ← greeks.St * previousdStGreek
15:     DFEVector newAccSt ← previousAccSt + (control.lastPoint ? carriedAccSt :
    0)
16:     carriedSt ← stream.offset(newSt, -bsdeLoopLength)
17:     carrieddStGreek ← stream.offset(newGreekSt, -bsdeLoopLength)
18:     carriedAccSt ← stream.offset(newAccSt, -bsdeLoopLength)
19: end procedure
```

4.3.3 Architecture – dataflow implementation on Maxeler technology

The idea of the dataflow implementation is to utilize a recursive equation for the scenario evolution from time t to time $t+1$. The recursive equation can be differentiated in the Adjoint mode to calculate the partial sensitivities from time t to time $t+1$. The hand-coded routines are included in the Kernel function. The kernel function presents processing flow. Based on the kernel code, a DFE graph is produced. This is further replicated based on the number of pipelines and clocks. The computation and differentiation of a scenario from time t to time $t+1$ requires multiple clock ticks to produce the intermediate output at $t+1$ step. This number of clock ticks is termed the pipeline depth. The pipeline depth needs to be set as an offset in order to refer to the next iteration in a stream.

The algorithm 4 shows the concepts of the kernel implementation for the Heston model. Figure 4.10 presents the DFE graph for the option pricing and sensitivity computation for the Heston model. This graph is iteratively processed by DFE cores. Table 4.1 shows the resource utilization after the compilation process on an FPGA.

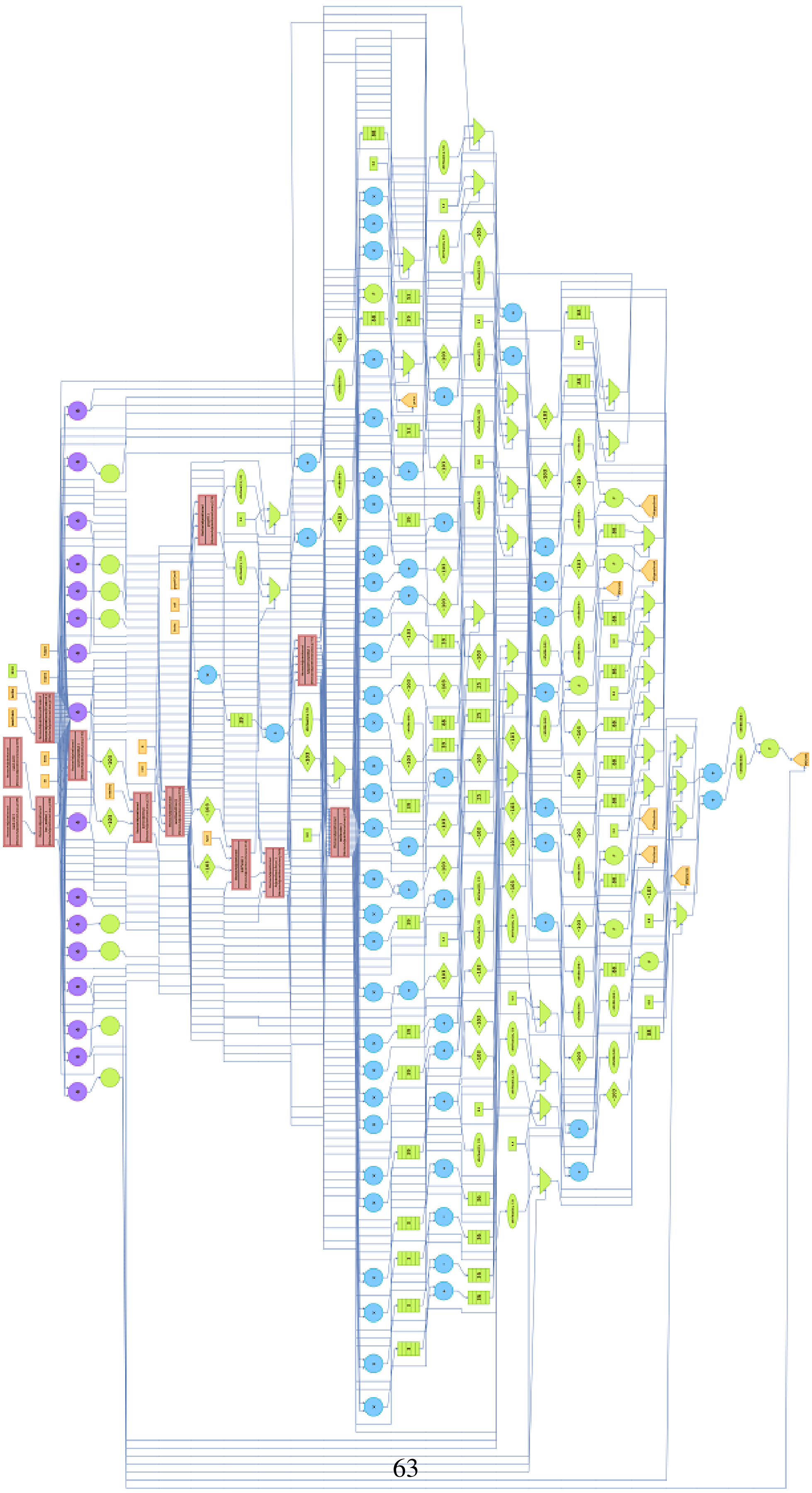


Figure 4.10: DFE graph for the Heston model

Model	FPGA (2 pipes)	FPGA (4 pipes)
Logic utilization:	110119 / 262400 (41.97%)	207380 / 262400 (79.03%)
Primary FFs:	212034 / 524800 (40.40%)	402153 / 524800 (76.63%)
Secondary FFs:	9214 / 524800 (1.76%)	16947 / 524800 (3.23%)
Multipliers (18x18):	940 / 3926 (23.94%)	1866 / 3926 (47.53%)
DSP blocks:	534 / 1963 (27.20%)	1060 / 1963 (54.00%)
Block memory:	1091 / 2567 (42.50%)	2099 / 2567 (81.77%)

Table 4.1: Resource utilization for the FPGA implementation

4.3.4 Summary

In this section the architecture of the Heston model calibration on FPGA has been presented. The work utilizes the differentiation routines for the Heston-model. This can find application in option pricing and hedging of investment portfolios consisting of thousands of options in real-time. The computational experiments are presented in Chapter 5.

4.4 Parallel non-linear least squares optimization framework using Automatic Differentiation

4.4.1 Introduction

The dynamics of real-time processes can be simplified and expressed as mathematical functions depending on several factors. The mathematical models need to be fitted to the observed data to determine the model parameters for which the model accurately expresses reality. For this purpose, the least squares error function can be used. The least squares function is expressed as a sum of the squared differences between the model function and observation data. The optimization process – minimization of the least-squares function – can utilize the gradient information. This is computationally expensive and involves the function value and gradient computation for many different input parameters.

This section proposes a parallel non-linear least-squares optimization framework using Automatic Differentiation (AD) methods. This approach utilizes a graph representation and overloaded operator techniques to express the general objective and constraint functions. The engine allows the objective/constraint function value calculation and the gradient computation for many different points in parallel. The gradient is calculated on parallel architectures via the Adjoint method with the cost being 2x of that of function evaluation. The gradient is applied to the non-linear least squares optimization methods. The framework supports multi-/many-core architectures such as GPUs, Intel Xeon/Xeon-Phi with the frameworks CUDA, OpenCL, OpenMP with Xeon-Phi.

The non-linear least squares optimization framework is benchmarked using a financial case-study: a semi-closed form Heston model calibration using the Gauss-Kronrod integration method.

4.4.2 Architecture

The parallel non-linear least-squares optimization engine using AD consists of the following modules:

- A module that records operations for the input objective and constraint function and builds a graph representation;
- An optimization module supporting constrained and unconstrained optimization; this calls HPC functions with the DAGs representing objective and constraint functions;
- A HPC module supporting AD computation on multi-/many-core architectures via OpenMP, OpenMP with Xeon-Phi, CUDA, OpenCL
- A market data module that connects to the SQL database to fetch observations for the calibration process [6]

The first stage is an objective and constraint function definition as a template function. Further, the objective and constraint functions are called with the Expression structure. During the programme execution, the Expression structure using overloading operator techniques records all arithmetic operations. These operations are stored in the DAG structure. Further, there is memory allocated for the Tape data structure – a data structure to store intermediate results and the first-order sensitivities.

[6]The market data module uses the SQLite library for database connection and query exectution. [sql16]

The optimization engine is compiled as a shared library. This provides an API that allows a platform-independent function definition and execution on parallel computing architectures.

AD Engine – evaluation and differentiation functions

The framework provides functions to process DAGs: `evaluateDAG` to calculate the function value and `reverseDAG` with automatic differentiation routines to evaluate the gradient. They utilize vectorization and multi-threading techniques.

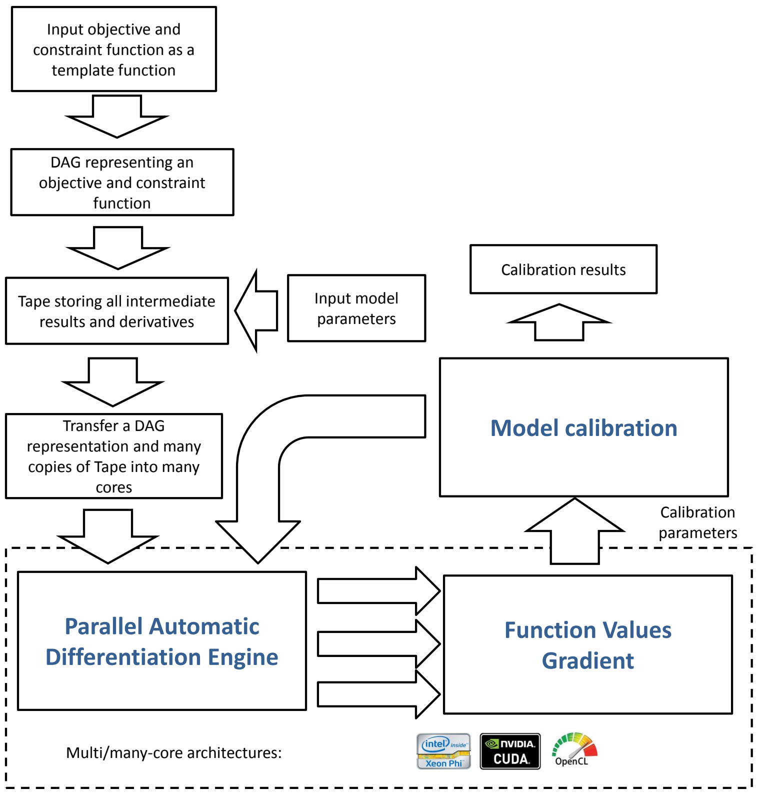

Input objective and constraint function as a template function
DAG representing an objective and constraint function
Tape storing all intermediate results and derivatives
Input model parameters
Calibration results
Transfer a DAG representation and many copies of Tape into many cores
Model calibration
Calibration parameters
Parallel Automatic Differentiation Engine
Function Values Gradient
Multi/many-core architectures:
intel inside Xeon Phi
NVIDIA CUDA
OpenCL

Figure 4.11: Architecture – HPC engine for Non-linear least squares optimization

CUDA & OpenCL

The GPU kernel function takes the DAG representing an objective function, the size of graph, and the number of options. Each thread calculates an the objective function value and the gradient for a single option. Each thread operates on its own Tape. A Tape with the intermediate results is stored in global memory. The pseudocode 5 describes a parallel algorithm on GPU.

Algorithm 5 GPU Kernel implementation

```
Require: DAG, dagSize, tape, numOptions
Ensure: f, fjac
    optionIdx ← getNumberOfThread() + blockDimX * blockIdx                ▷ DAG
    initialization;
    for graphId ← 1, dagSize do
        EvaluateDAG(DAG, tape[optionIdx], graphId)
    end for
    for graphId ← dagSize, 1 do
        AdjointDAG(DAG, tape[optionIdx], graphId)
    end for    ▷ Update f on the model results    ▷ Update fjac on the gradient results
```

Algorithm 6 Xeon-Phi implementation

```
Require: DAG, dagSize, tape, numOptions
Ensure: f, fjac
    for optionIdx = 0, optionIdx ¡ numOptions do                ▷ DAG initialization;
        for graphId ← 1, dagSize do
            EvaluateDAG(DAG, tape[optionIdx], graphId)
        end for
        for graphId ← dagSize, 1 do
            AdjointDAG(DAG, tape[optionIdx], graphId)
        end for
    end for    ▷ Update f on the model results    ▷ Update fjac on the gradient results
```

Xeon-Phi

The Xeon-Phi implementation utilizes OpenMP (OMP) and Xeon-Phi instructions to parallelize DAG processing. The outer loop uses OMP pragmas and iterates through the number of options. Two inner loops iterate the DAG to produce the objective function value (forward order) and the gradient (reverse order). The results are stored in 8 double-precision or 16 single precision number vectors. The pseudocode 6 shows a parallel code for Xeon-Phi Coprocessor.

4.4.3 Deployment Process

The optimization engine provides functionality that allows a platform-independent objective and constraint function definition and execution on HPC without re-compilation. The user code with the objective and constraint function definition is linked with shared libraries. The external code should include the objective and constraint function definitions using templates, boundary conditions and processing flow. The generic objective and constraint functions are used with the Expression structure. This is used to record all intrinsic operations. During user code compilation the objective and constraint functions are transformed to the DAG representation. The listing 5.17 presents a sample function definition. The listing 4.13 shows the processing flow.

4.4.4 Summary

In this section, the architecture of the parallel non-linear least squares optimization framework using the Adjoint has been presented. The work utilizes overloading operator techniques to build a graph structure for the objective and constraint functions. The computational experiments are presented in Chapter 5.

```
// Objective Function
template<class Value>
class HestonFormula
{
public:
Complex<Value> Hestf(ComplexType<Value> &phi, ..., int type);
Value HestonPIntegrand(Value phi, Value *params, int type);
Value HestonP(int type, Value& kappa, ..., Value &K)
{
Integral<Value> integral;
...
Value area = integral.GetIntegrationResult(params, type);
}

Value HestonCallQuad(Value &kappa, ..., Value &K)
{
Value result;
return result
}
};
```

Figure 4.12: Non-linear least-squares optimization framework – Objective Function Definition

```
#include <ADEngine.h>
#include <OptimizationEngine.h>
#include "HestonModel.h"
main()
{
OptimizationEngine optimizationEngine;
OptionPriceData marketOptionPriceData;
HestonFormula<Expression> hestonFormula;
ObjectiveFunction objFunction(10000);
objFunction.startRecord();
Expression x[5];
x[0].setCalibration();
...
x[4].setCalibration();
x[0] = 0.4;
...
x[4] = 0.02;
Bound bounds[6];
bounds[0].lb = 0;
bounds[5].ub = 1;
Expression maturity;
...
maturity = 1.0;
Expression f;
f = hestonFormula.HestonCallQuad(x[0], ..., strike);
optimizationEngine.runCalibration(&objFunction, x, bounds, f,
strike, maturity, &marketOptionPriceData, isDifferentiation,
Adjoint, &executionTime);
}
```

Chapter 5

Computational experiments

5.1 Introduction

This chapter presents computational experiments for the Heston model calibration on HPC. This is divided into the following sections:

- Section 5.2.3 provides a description of the market data used for the Heston model calibration;
- Section 5.2 presents performance, accuracy and calibration results for the Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint;
- Section 5.3 presents computational experiments for the FPGA implementation of the Heston model calibration using the Adjoint for the sensitivity calculation;
- Section 5.4 presents both performance and calibration results for the semi-closed form Heston model calibration using a parallel non-linear least squares optimization framework;
- Section 5.5 summarises the computational experiments;

5.2 Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint

5.2.1 Overview

In this section, there are presented the computational results for the Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint. This includes a computational environment and market-data description. Further, there are presented the input data for the calibration process. Next, there are presented performance results for multi-/many-core and distributed architectures. Further, there are presented the calibration results for the Heston model.

5.2.2 Computational Environment

Studies have evaluated option pricing/first-order derivatives and model calibration via the Heston model computed by MC methods. These compare the Adjoint techniques with Finite Differences. Computational experiments have been performed on the following system configurations:

1. OpenMP & OpenMPI: 32x Intel Xeon X5650 2.66 Ghz with 6 cores supporting execution of 12 threads with 48 GB RAM
2. OpenMP with Xeon-Phi & OpenMPI: 2x Intel Xeon-Phi Coprocessor 7120p
3. CUDA and OpenCL: 2x NVIDIA Kepler K40 with 2880 cores
4. an Intel Core i7-4810MQ CPU 2.80GHz with 8GB RAM memory

Total speedup S_{total} is measured as follows:

$$S_{total} = S_{nodes} \cdot S_{HPC} \cdot S_{method} \tag{5.1}$$

where S_{nodes} is speedup achieved by using a cluster of HPC devices, S_{HPC} denotes speedup on a HPC platform and S_{method} is speedup from applying the numerical method. The computational experiments consider performance comparison with the QuantLib Library 1.9 [qua16].

	V_0	κ	θ	σ	ρ	l	μ
Value	0.02	0.4	0.1	0.3	0.3	0.1	0.1
Lower bound	10^{-16}	10^{-3}	10^{-3}	10^{-16}	-0.99	0.001	0.001
Upper bound	2.0	10.0	1.0	1.0	0.99	1.0	1.0

Table 5.1: Lower and upper bounds for the Heston model calibration

5.2.3 Market Data

For the financial case-study, the calibration experiments consider datasets representing options on stock assets and equity indices. The option prices are published on Yahoo Finance [1] Each dataset considers call/put options with different strikes and expiration dates. The dataset module extracts the option market data and inserts it into a database. The observation data is a mid-price between the ask and bid option prices. For the calibration process, the dataset module loads the option-price matrix into a vector that is used in the optimization process. The experimentation considers the market option matrix for the following indexes and commodities:

- S&P 500 index
- Dow Jones Industrial Average index
- BP

Based on the option price matrix, the following implied volatility has been obtained [Hul12].

5.2.4 Input Data

The input parameters for the Heston model are: S_0 = 100, r=0.005, q = 0. The model calibration via MC simulation has been tested with the following configuration: 300 timesteps, 10000 paths. Table 5.1 shows the input values, lower and upper bounds for the optimization.

The computational experiments for the QuantLib and the implementation for the Heston model calculation using OpenMP are as follows: S_0=1.0, V_0=0.01, κ=0.4, θ=0.1, σ=0.3, ρ=0.3, r=0.005, K= 0.5 and maturity=1.0.

[1]Option market data source – Yahoo Finance `http://finance.yahoo.com`.

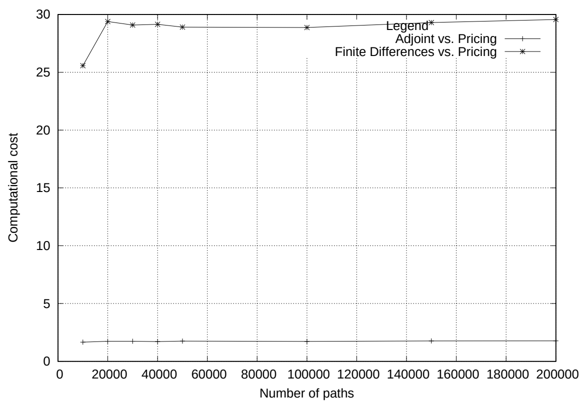


Figure 5.1: Performance results – Differentiation vs. Pricing

5.2.5 Performance results

Figure 5.1 shows the results from comparing the Adjoint and Finite differences methods for the first-order sensitivity calculation with pricing via MC simulation. As expected, the Adjoint improves gradient calculation performance: the finite difference method requires around 30x the time for option pricing simulation whereas the gradient is calculated in 1.8x that of the function evaluation. The Adjoint improves performance of the sensitivity calculation by 16x for the Heston model when compared to differentiation via FD.

The Monte-Carlo simulation is an embarrassingly parallel computation problem. The theoretical speedup is proportional to the number of processors utilized when there is no communication between the processors.

$$S^{nodes}_{theoretical} = n_{processors} \tag{5.2}$$

where $n_{processors}$ is the number of processors utilized.

The presented work contains a sequential part of code required for calculating the sum of option prices and the sensitivities. Therefore the speedup is less than the factor $n_{processors}$. The performance analysis includes the comparison of the achieved speedups with the maximum theoretical Monte-Carlo speedups on a different number of cores/nodes utilized. The percentage:

$$\eta = \frac{S^{nodes}}{S^{nodes}_{theoretical}} \cdot 100\% \tag{5.3}$$

measures the speedup vs. the theoretical maximum speedup.

The computational percentage cost spent on a sequential code is calculated as below:

$$
\begin{aligned}
percentage_{nodes} &= \frac{t_{nodes} - \frac{t_{1node}}{S^{nodes}_{theoretical}}}{t_{nodes}} \cdot 100\% \\
&= (1 - \frac{t_{1node}}{t_{nodes}} \cdot \frac{1}{S^{nodes}_{theoretical}}) \cdot 100\% \\
&= (1 - S_{nodes} \cdot \frac{1}{S^{nodes}_{theoretical}} \cdot 100\% \\
&= (1 - \frac{S_{nodes}}{S^{nodes}_{theoretical}}) \cdot 100\%
\end{aligned}
$$

These metrics are used to analyse performance of parallel implementations using multi-core and distributed computing architectures.

Multi-core architectures (OpenMP)

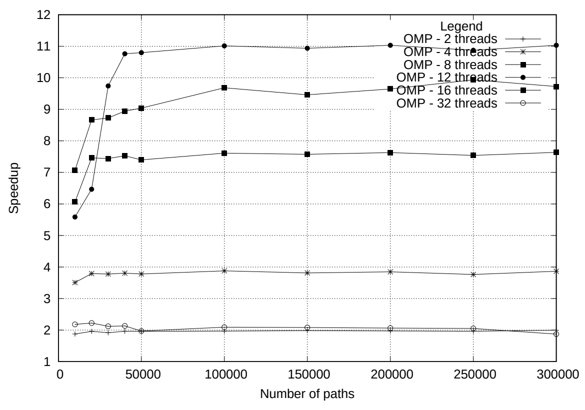
Figure 5.2a compares the speedup of the OpenMP implementation with different number of threads vs. a single-thread implementation. The performance for the implementations using from 2 to 12 threads increases with the number of threads utilized. [2] The maximum speedup can be observed for the implementation using 12 threads as the Intel Xeons X5650 supports up to 12 thread execution at once. The speedup for 12 threads is around 11x for a range from 40000 paths to 300000 paths. With a greater number of threads (16 and 32), the performance decreases as there is more threads used than 12. The table 5.3 presents the ratio of the achieved speedup vs. the maximum theoretical speedup. This shows that for the number of threads varying from 2 to 12 and the number of paths over 100 000 the performance achieves over 90% of the maximum theoretical speedup. For a greater number of threads, the η decreases as there is more threads than available cores. The table 5.4 shows the percentage cost spent on a sequential code, etc. The figure 5.2b presents the performance results for OpenMPI implementation using 2, 4 and 8 nodes vs a single-node implementation. The performance for the implementation using 8 nodes is around 7x. The table shows the ratio of the achieved speedup vs. the maximum theoretical speedup. For the implementation using 2 and 4 nodes and the number of paths over 100000, the performance achieves over 90% of the maximum theoretical speedup. The OpenMPI execution time

[2]The OpenMP code includes a sequential part of code required for calculating the sum of option prices and the sensitivities. Therefore, the speedup is less than the factor of 1x the number of threads.

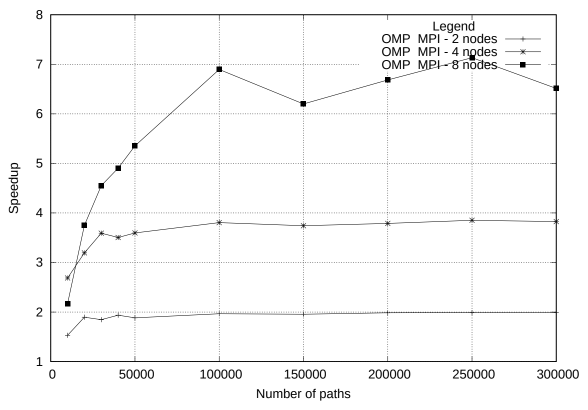
Number of paths	QuantLib	Execution Time (ms)	MC (1 thread)	Execution Time (ms)	Speedup
10000	0.501176	16467	0.501833	1334	12.34x
20000	0.501205	32789	0.503463	2569	12.76x
30000	0.502282	48981	0.503326	4000	12.25x
40000	0.501915	65412	0.502969	5202	12.57x
50000	0.502064	81656	0.502887	6530	12.50x
100000	0.502567	165286	0.502953	13004	12.71x

Table 5.2: Performance comparison of the sequential implementation for the Heston model calculation with the QuantLib Library 1.9. Tests have been performed on an Intel Core i7-4810MQ CPU 2.80GHz with 8GB RAM memory

measurements include times required for message passing to the nodes and receiving collected results from the nodes. The table presents 5.2 the performance comparison with the QuantLib library 1.9.



(a) Performance results for the gradient calculation via the Adjoint – OMP (a multithread implementation) vs. OMP (a single-thread implementation)



(b) Performance results for the gradient calculation via the Adjoint – OMP vs. OMP (a single node)

GPU (CUDA & OpenCL)

Figure 5.2a shows performance results for the gradient calculation of the Heston model on an NVIDIA K40 card compared with an OMP version (a single thread implementation). The maximum speed-up achieved is 58x for 20000 paths. Figure 5.2b compares results on 2 GPUs vs 1 GPU. Speedup increases with the number of scenarios starting from 50000; speed-up is 1.9x for 150000 scenarios. Figure 5.2 shows results for the OpenCL implementation. The speedup for the OpenCL implementation is around 58x

Number of paths	OMP 2 threads	OMP 4 threads	OMP 8 threads	OMP 12 threads	OMP 16 threads	OMP 32 threads
10000	93.79	87.66	75.77	46.54	44.17	6.81
20000	97.97	94.76	93.27	53.89	54.15	6.94
30000	95.82	94.38	93.01	81.18	54.54	6.62
40000	98.14	95.07	94.07	89.68	55.89	6.66
50000	97.93	94.49	92.48	89.98	56.49	6.16
100000	98.34	96.96	95.12	91.77	60.52	6.53
150000	99.04	95.31	94.69	91.15	59.12	6.5
200000	98.61	96.18	95.35	91.93	60.29	6.45
250000	98.09	94.06	94.23	90.62	62.09	6.41
300000	99.68	96.64	95.45	91.92	60.78	5.86

Table 5.3: η – the achieved speedup vs. the maximum theoretical speedup x 100 %.

Number of paths	OMP 2 threads	OMP 4 threads	OMP 8 threads	OMP 12 threads	OMP 16 threads	OMP 32 threads
10000	6.21	12.34	24.23	53.46	55.83	93.19
20000	2.03	5.24	6.73	46.11	45.85	93.06
30000	4.18	5.62	6.99	18.82	45.46	93.38
40000	1.86	4.93	5.93	10.32	44.11	93.34
50000	2.07	5.51	7.52	10.02	43.51	93.84
100000	1.66	3.04	4.88	8.23	39.48	93.47
150000	0.96	4.69	5.31	8.85	40.88	93.50
200000	1.39	3.82	4.65	8.07	39.71	93.55
250000	1.91	5.94	5.77	9.38	37.91	93.59
300000	0.32	3.36	4.55	8.08	39.22	94.14

Table 5.4: MC: $percentage_{nodes}$ – the computational percentage cost spent on a sequential code

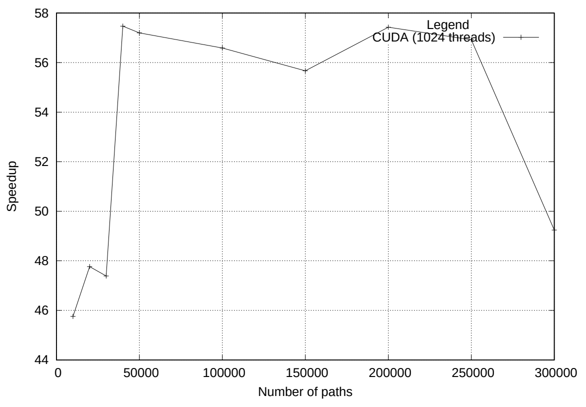
Number of paths	OMPI 2 nodes	OMPI 4 nodes	OMPI 8 nodes
10000	76.59	67.19	27.15
20000	94.84	79.87	46.88
30000	92.31	89.78	56.88
40000	96.85	87.61	61.30
50000	94.17	89.91	66.91
100000	98.26	95.14	86.20
150000	97.77	93.56	77.49
200000	99.27	94.71	83.55
250000	99.38	96.28	89.17
300000	99.57	95.64	81.41

Table 5.5: MC: η – the achieved speedup vs. the maximum theoretical speedup x 100 %.

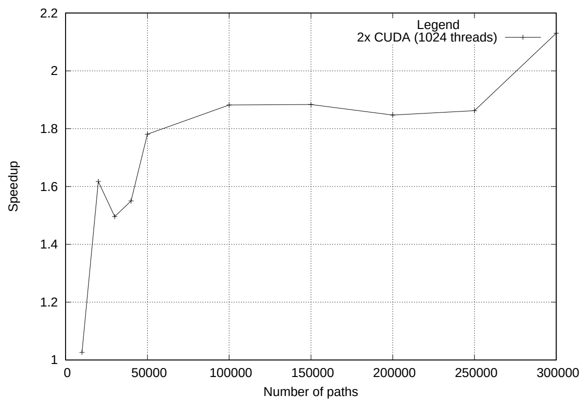
Number of paths	OMPI 2 nodes	OMPI 4 nodes	OMPI 8 nodes
10000	23.41	32.81	72.85
20000	5.16	20.13	53.12
30000	7.69	10.22	43.12
40000	3.15	12.39	38.70
50000	5.83	10.09	33.09
100000	1.74	4.86	13.80
150000	2.23	6.44	22.51
200000	0.73	5.29	16.45
250000	0.62	3.72	10.83
300000	0.43	4.36	18.59

Table 5.6: $percentage_{nodes}$ – the computational percentage cost spent on a sequential code

for 150000 paths.



(a) Performance results for the gradient calculation via the Adjoint – CUDA NVIDIA K40 vs. OMP (a single-thread implementation)



(b) Performance results for the gradient calculation via the Adjoint – 2x NVIDIA K40 vs. NVIDIA K40

Many-core architectures (Xeon-Phi)

Figure 5.3a shows the performance results for the Xeon-Phi version. [3] The maximum speed-up is 22x for the implementation using 32 threads. Figure 5.3b presents results for the Xeon-Phi implementation with OpenMPI.

[3]The Xeon-Phi contains a sequential part of code that is for calculating the sum of option prices and the sensitivities. Therefore, the speedup is less than the factor of 1x the number of threads.

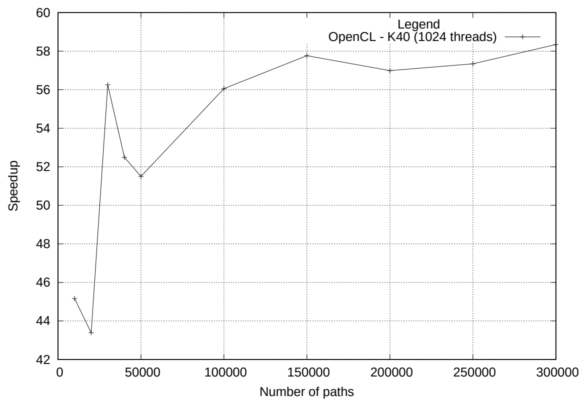
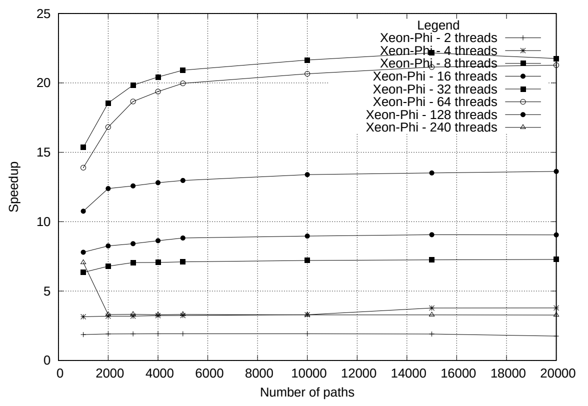
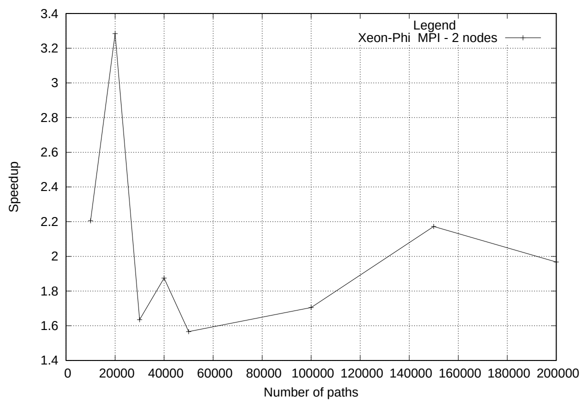


Figure 5.2: Performance results – OpenCL NVIDIA K40 vs. OMP (a single-thread implementation)



(a) Performance results for the gradient calculation – Xeon-Phi (a multithread implementation) vs. Xeon-Phi (a single-thread implementation)



(b) Performance results for the gradient calculation via the Adjoint – Xeon-Phi vs. Xeon-Phi (a single node)

5.2.6 Calibration results

Table 5.7 presents the execution times in milliseconds for the Heston model calibration on different HPC platforms; in the brackets are the number of major iterations and minor iterations during optimization. Table 5.8 shows the calibration results for a single-thread implementation. Table 5.9 presents calibration results for the Heston model and Heston model with term-structure. The Heston volatility matrix is retrieved via the Newton-Raphson method from the model prices.

Model	OpenMP (1 thread) – Simulation	OpenMP (1 thread) – Calibration	OpenMP (8 threads) – Simulation	OpenMP (8 threads) – Calibration	Xeon-Phi (32 threads) – Simulation	Xeon-Phi (32 threads) – Calibration	GPU (CUDA – K40) – Simulation	GPU (CUDA – K40) – Calibration
Heston model	229255	229289 (17, 37)	79455	79479 (10, 20)	125635	125652 (4, 11)	6747	17256 (6, 21)
Heston model with constraints	186243	186271 (11, 33)	91555	91581 (8, 23)	194338	194359 (4, 15)	6155	15869 (7, 27)
Heston with Jumps	192442	192464 (7, 32)	124853	124886 (11, 46)	218180	218213 (4, 11)	–	–

Table 5.7: Simulation and calibration times during the calibration process for 10000 paths

Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.013234	0.01459	0.0126333	0.0144067
κ	10	0.00100863	1.08172	5.00904
θ	0.0185984	0.225559	0.00139981	0.018675
σ	0.400961	0.292118	0.664814	0.366409
ρ	-0.995326	-0.856153	-0.999982	-0.964993
l	–	–	0.0274284	0.0810467
μ	–	–	0.19921	0.016398
RMSE (volatility)	1.90948	1.89477	3.51522	1.95436
RMSE (option price)	0.404118	0.501432	0.628826	0.424208
Number of major/minor iterations	17/37	11/33	7/32	34/131

Table 5.8: Calibration results for 10000 paths

Parameter	Value
V_0	1.00614e-16
λ	0.001
α	0.001
σ_t	1.11022e-16
ρ	-1
RMSE (volatility)	2.39836
RMSE (option price)	0.12457
Number of major iterations	3
Number of minor iterations	13

(a) Heston model

Maturity	0.00273973	0.0219178
V_0	1.00149	1
λ	0.540665	0.540665
α	0.662946	0.0134989
σ_t	0.225595	0
ρ	-0.122158	-0.122108
RMSE (volatility)	0.406747	0.406747
RMSE (option price)	0.0844884	0.0844884
Number of major iterations	4	4
Number of minor iterations	30	30

(b) Heston model with term structure

Table 5.9: Calibration results for 23 call options with 2 maturities – 10000 paths, 100 timesteps

SPX 500 index – Call options

Table 5.10 shows the calibration results for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition.

The computational experiments include the RMSE (Root-Mean Square Error) for the Heston model volatility and the Heston model option price. The computational experiments consider the calibration to 123 call S&P options with 10 maturities selected by the following database query. The Heston model has been calculated via MC sim-

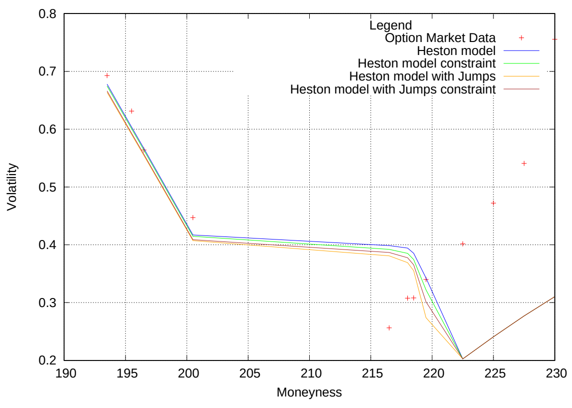
```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='SPY' AND IMPLIED_VOLATILITY > 0 AND
    OPTION_TYPE=0 AND MATURITY < 1.0 AND OPTION_PRICE > 0
    AND MATURITY_ID < 10 ORDER BY OPTION_TYPE, MATURITY,
    STRIKE ASC
```

Figure 5.3: Database query for call option price matrix for S&P 500 index

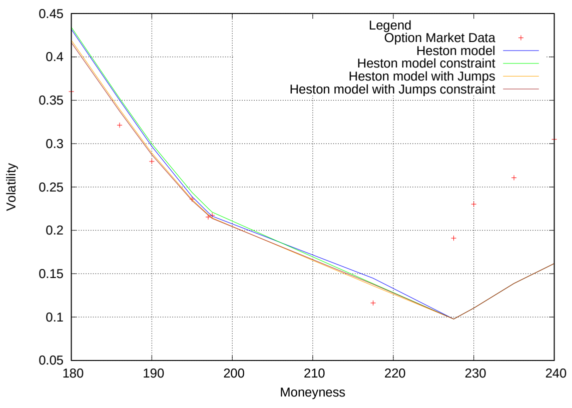
Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.0102997	0.00993562	0.00934736	0.00964807
κ	0.0941267	0.146015	1.97319	0.277915
θ	0.385344	0.0170663	0.0384247	0.175766
σ	0.266227	0.425135	0.544621	0.485939
ρ	-0.941072	-0.867355	-0.679457	-0.707878
l	–	–	0.0419667	0.620926
μ	–	–	0.0524088	0.00391578
RMSE (volatility)	1.25921	1.22782	1.23759	1.23969
RMSE (option price)	0.389978	0.538844	0.359756	0.3673
Number of major/minor iterations	20/52	29/47	26/89	12/42
Simulation time	150297	131371	265713	248769
Calibration time	150347	131415	265777	248823

Table 5.10: Calibration results for 10000 paths – SPX 500 call options

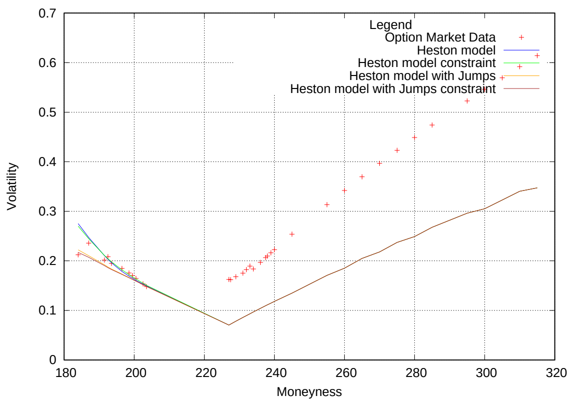
ulation with 100 timesteps. The best fitting to the option matrix for the S&P index is for the Heston model with Jumps. The RMSE error for the Heston model with Jumps is 0.359756. The best volatility fitting to the implied volatility for the S&P index is for the Heston model with the Feller condition. The RMSE volatility error for the Heston model with the Feller condition is equal to 1.122782. The figures 5.4a, 5.4b, 5.4c, 5.4d, 5.4e, 5.4f, 5.4g, 5.4h, 5.4a, 5.4b show the volatility slices for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition. Figure 5.4 presents the implied volatility surface for the S&P index for the call options. Figures 5.5a, 5.5b, 5.5c, 5.5d presents the model volatility surfaces after the calibration process.



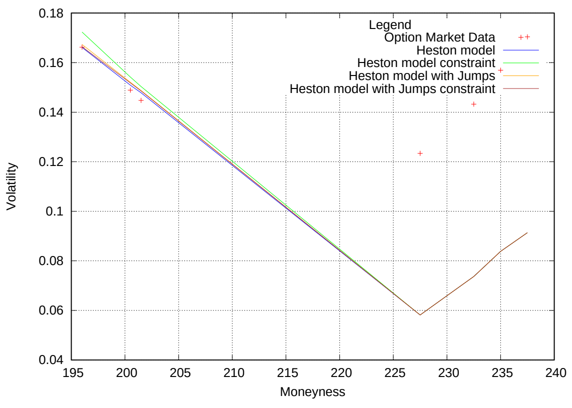
(a) SPY 500 call options, maturity: 0.00273973



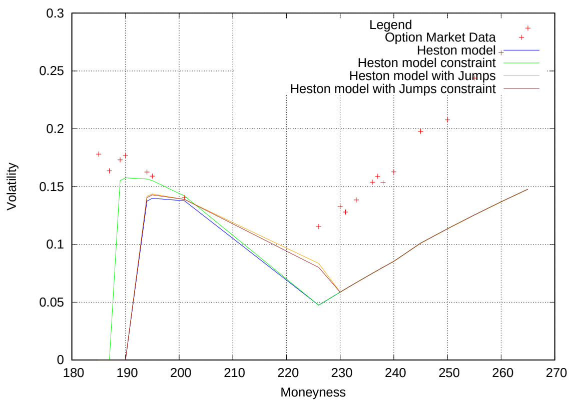
(b) SPY 500 call options, maturity: 0.0219178



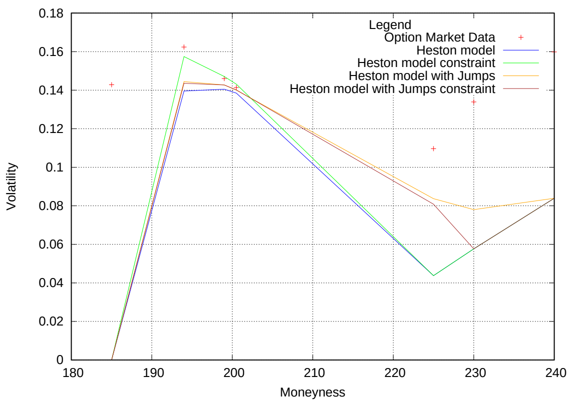
(c) SPY 500 call options, maturity: 0.0410959



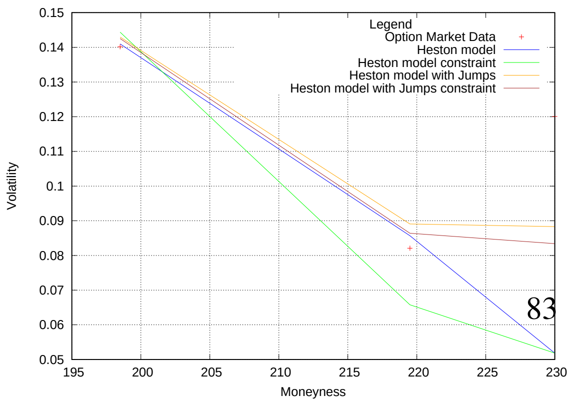
(d) SPY 500 call options, maturity: 0.060274



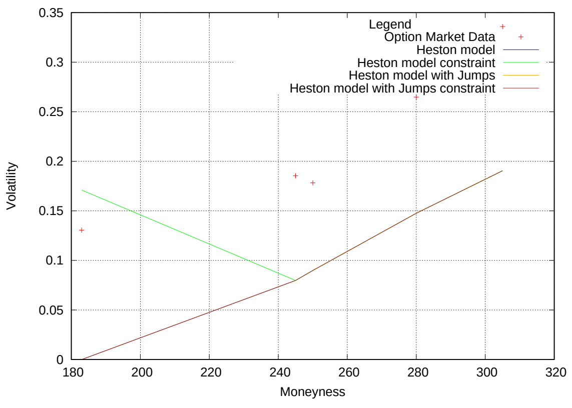
(e) SPY 500 call options, maturity: 0.0767123



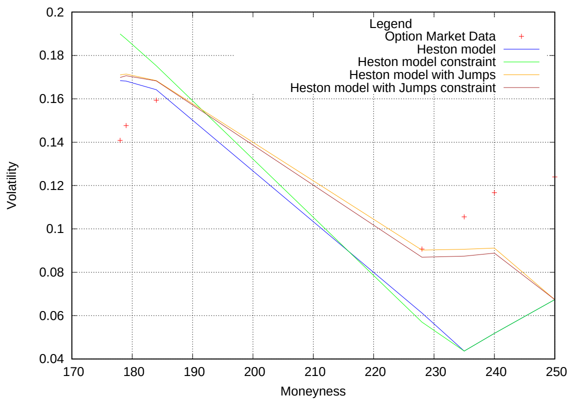
(f) SPY 500 call options, maturity: 0.0794521



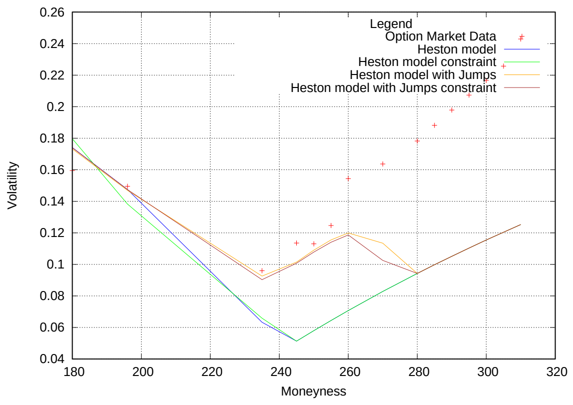
(g) SPY 500 call options, maturity: 0.0986301



(h) SPY 500 call options, maturity: 0.117808



(a) SPY 500 call options, maturity: 0.213699



(b) SPY 500 call options, maturity: 0.290411

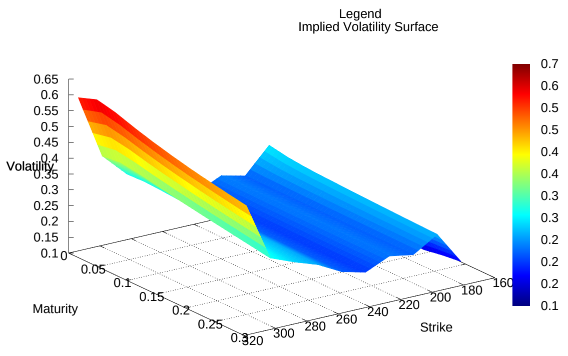
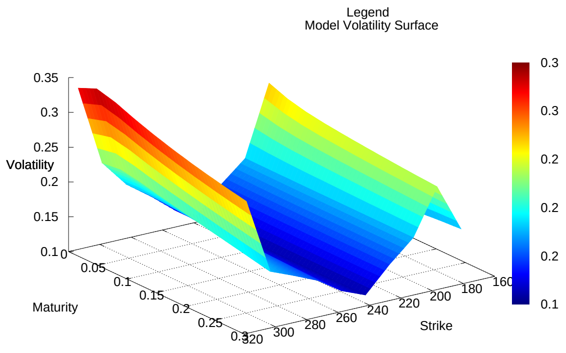
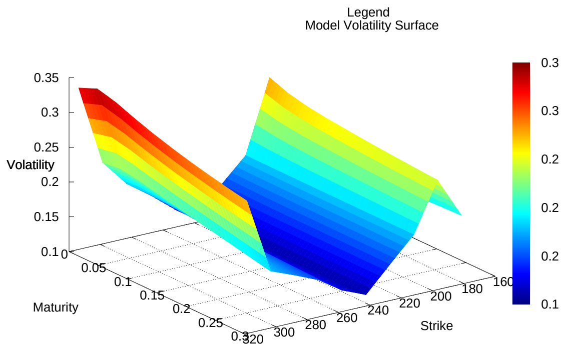


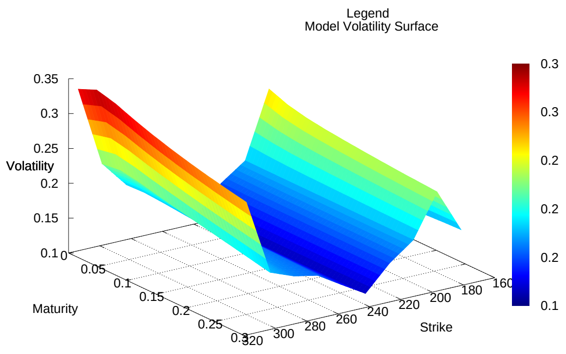
Figure 5.4: Implied volatility



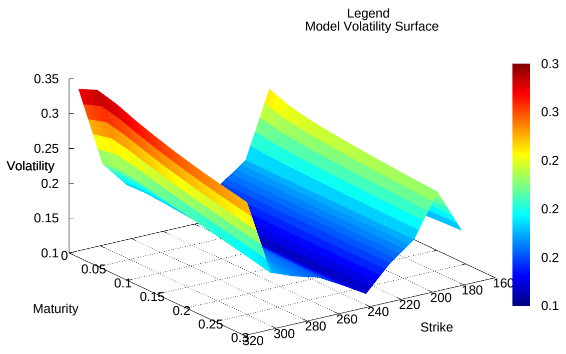
(a) Heston model volatility



(b) Heston model volatility with constraint



(c) Heston model with Jumps



(d) Heston model with Jumps and constraint

SPX 500 index – Put options

Table 5.11 presents the calibration results for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition.

The computational experiments consider the calibration to 197 put S&P options with 10 maturities selected from the database with option market data by the following database query:

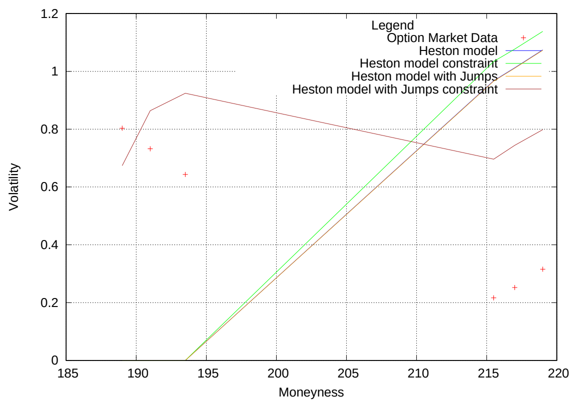
The Heston model has been computed via MC simulation with 100 timesteps. The best fitting to the option matrix for the S&P index is for the Heston model with Jumps. The RMSE error for the Heston model with Jumps is equal to 43.598. The best volatility fitting to the implied volatility for the S&P index is for the Heston model with the Feller condition. The RMSE volatility error for the Heston model with the Feller condition is equal to 3.83677. Figures 5.6a, 5.6b, 5.6c, 5.6d, 5.6e, 5.6f, 5.6g, 5.6h, 5.6a, 5.6b present the volatility slices for the Heston mode, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition respectively. Figure 5.6 shows the implied volatility surfaces for the S&P index for the put options. Figures 5.7a, 5.7b, 5.7c, 5.7d show the model volatility surfaces after the calibration process.

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='SPY' AND IMPLIED_VOLATILITY > 0 AND
   OPTION_TYPE=1 AND MATURITY < 1.0 AND OPTION_PRICE > 0
   AND MATURITY_ID < 10 ORDER BY OPTION_TYPE, MATURITY,
   STRIKE ASC
```

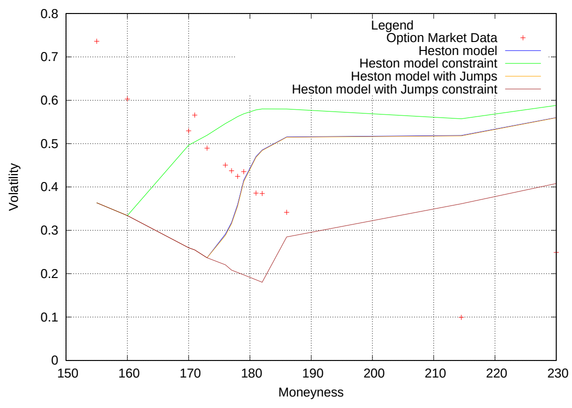
Figure 5.5: Database query for put option price matrix for S&P 500 index

Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.138201	0.163232	0.137632	0.0572952
κ	0.00100255	0.00227608	0.001	2.77897
θ	0.00281572	0.760289	0.001	0.457629
σ	1	0.997356	1	0.99869
ρ	0.986304	0.725009	0.979791	0.734252
l	–	–	1	0.0543364
μ	–	–	0.001	0.190406
RMSE (volatility)	4.07049	3.83677	4.06867	3.93335
RMSE (option price)	43.6379	44.6466	43.598	45.4714
Number of major/minor iterations	12/29	3/14	5/15	7/25
Simulation time	123457	60812	98342	432041
Calibration time	123495	60834	98372	432102

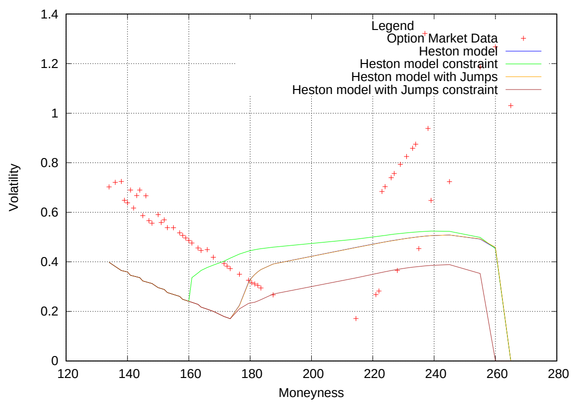
Table 5.11: Calibration results for 10000 paths – SPX 500 put options



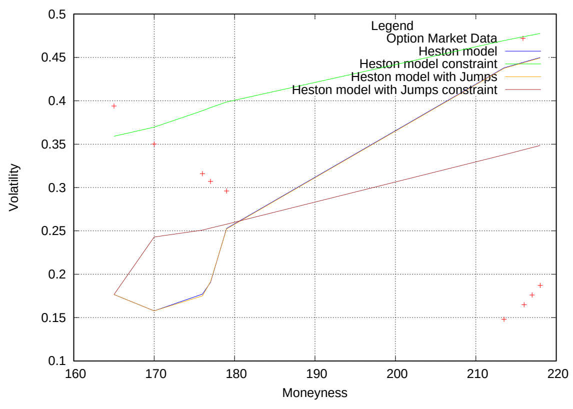
(a) SPY 500 put options, maturity: 0.00273973



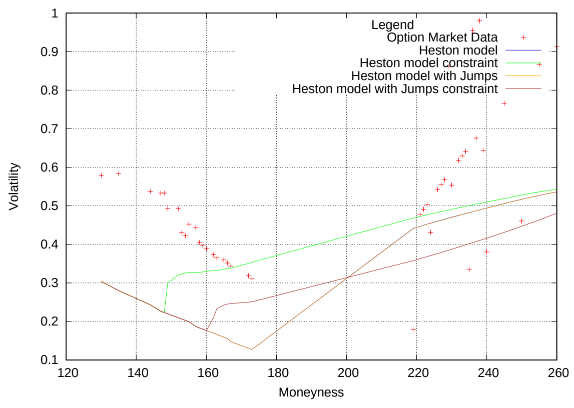
(b) SPY 500 put options, maturity: 0.0219178



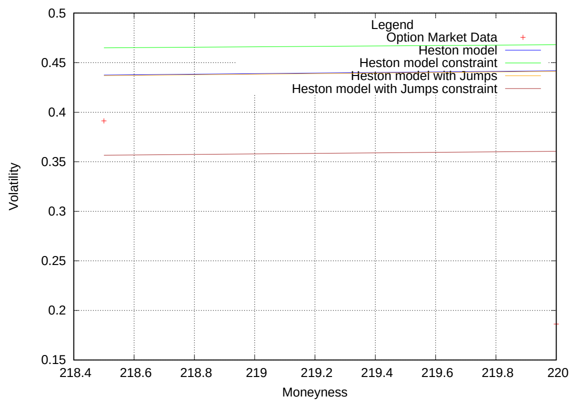
(c) SPY 500 put options, maturity: 0.0410959



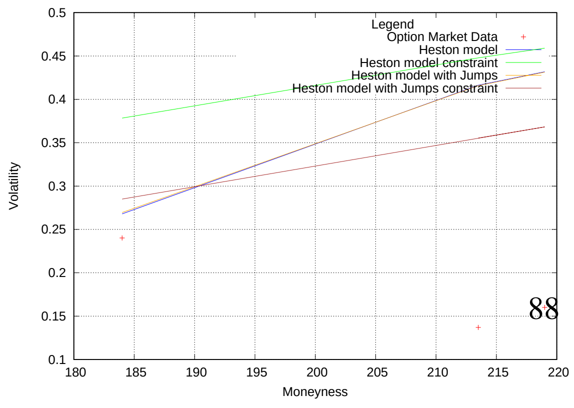
(d) SPY 500 put options, maturity: 0.060274



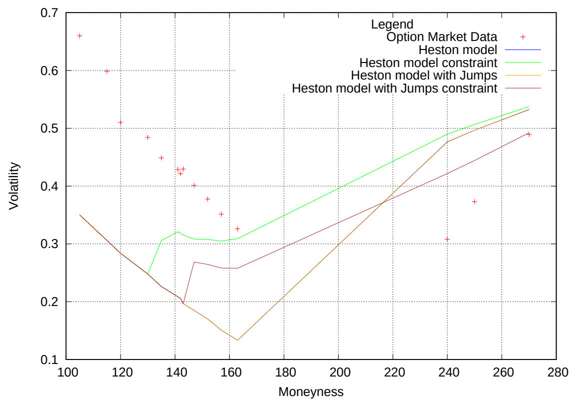
(e) SPY 500 put options, maturity: 0.0767123



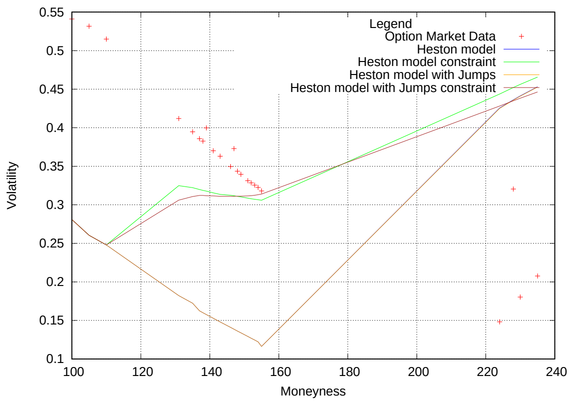
(f) SPY 500 put options, maturity: 0.0794521



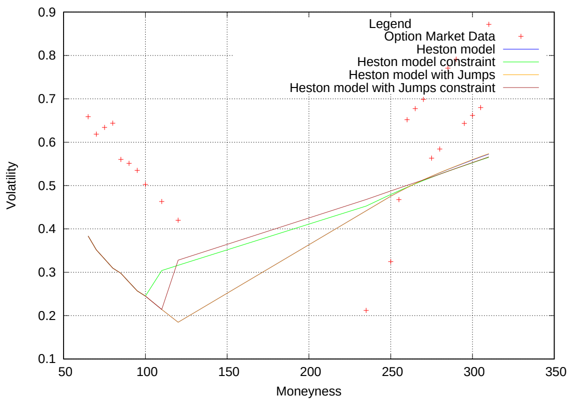
(g) SPY 500 put options, maturity: 0.0986301



(h) SPY 500 put options, maturity: 0.117808



(a) SPY 500 put options, maturity: 0.213699



(b) SPY 500 put options, maturity: 0.290411

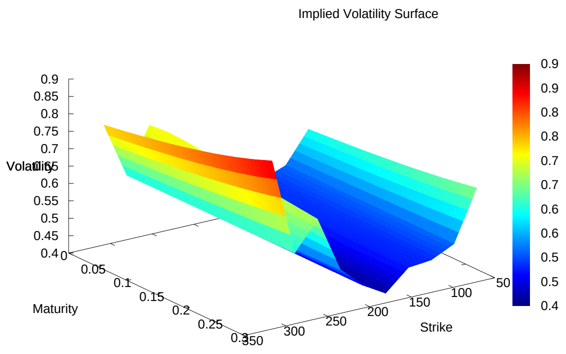
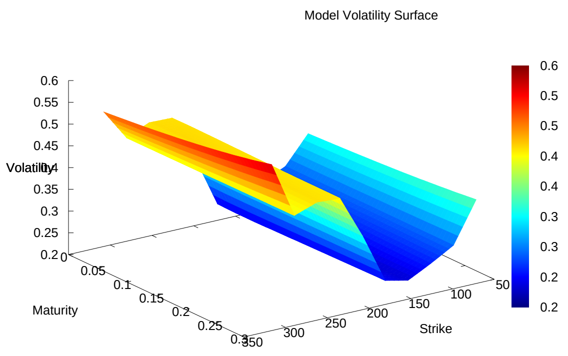
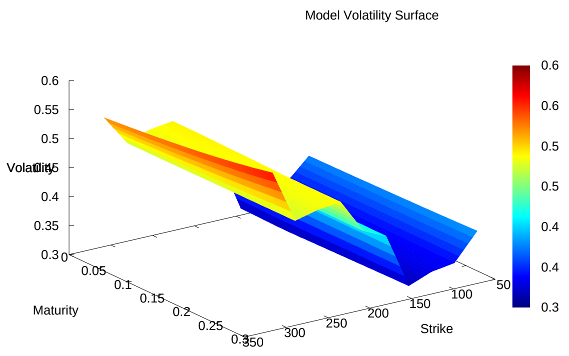


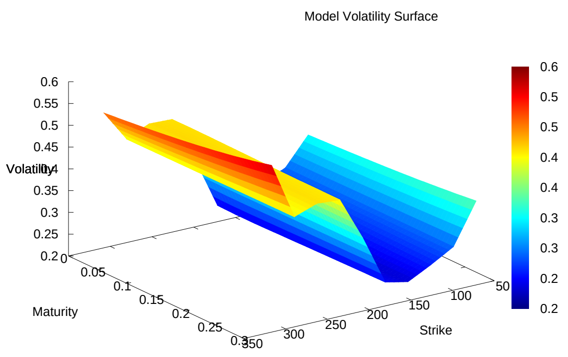
Figure 5.6: Implied volatility



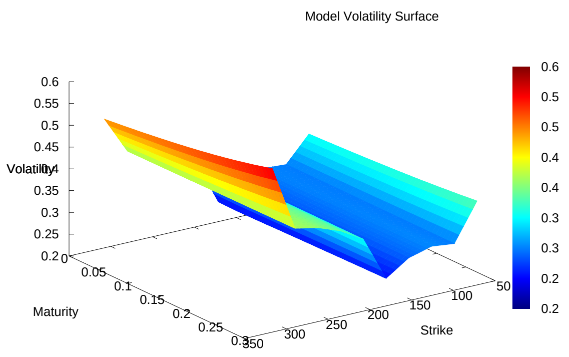
(a) Heston model volatility



(b) Heston model volatility with constraint



(c) Heston model with Jumps



(d) Heston model with Jumps and constraint

Dow-Jones Industrial Average index – Call options

Table 5.12 shows the calibration results for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition.

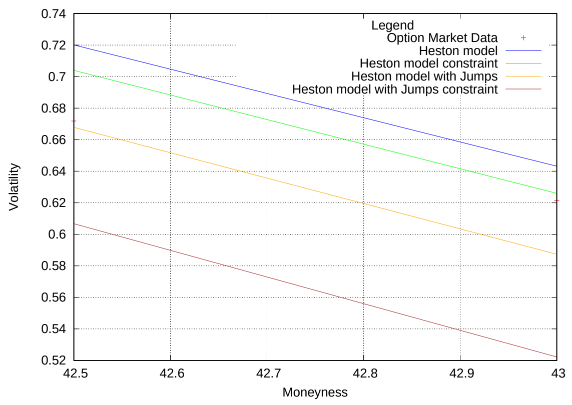
The computational experiments consider the calibration to 91 call Dow-Jones Industrial Average options with 8 maturities selected by the following database query: The Heston model has been calculated via MC simulation with 100 timesteps. The best option price and the volatility fitting is for the Heston model without the Feller condition. The RMSE error for the Heston model without the Feller condition is equal to 4.23393. The RMSE volatility error is equal to 2.1095. Figures 5.8a, 5.8b, 5.8c, 5.8d, 5.8e, 5.8f, 5.8g, 5.8h present the volatility slices for the models. Figure 5.8 shows the implied volatility surface for the Dow-Jones Industrial Average index for the call options. Figures 5.9a, 5.9b, 5.9c, 5.9d show the model volatility surfaces after the calibration.

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='DXJ' AND IMPLIED_VOLATILITY > 0 AND
    OPTION_TYPE=0 AND MATURITY < 1.0 AND OPTION_PRICE > 0
    AND MATURITY_ID < 8 ORDER BY OPTION_TYPE, MATURITY,
    STRIKE ASC
```

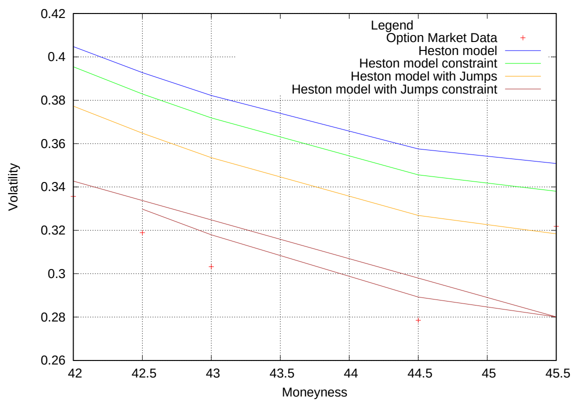
Figure 5.7: Database query for call option price matrix for Dow-Jones Industrial Average index

Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.0806155	0.0762379	0.0668164	0.0521738
κ	0.001	0.001	1.61214	2.03301
θ	0.00100253	0.001	1	0.35412
σ	1	0.9994	0.3149	1
ρ	-0.960712	-1	-1	-0.999515
l	–	–	0.001	0.001
μ	–	–	0.001	0.001
RMSE (volatility)	2.1095	2.12821	2.15412	2.35541
RMSE (option price)	4.23393	4.2479	4.27255	4.27849
Number of major/minor iterations	18/39	8/16	8/39	8/27
Simulation time	151377	70779	71178	251238
Calibration time	151424	70806	71203	251282

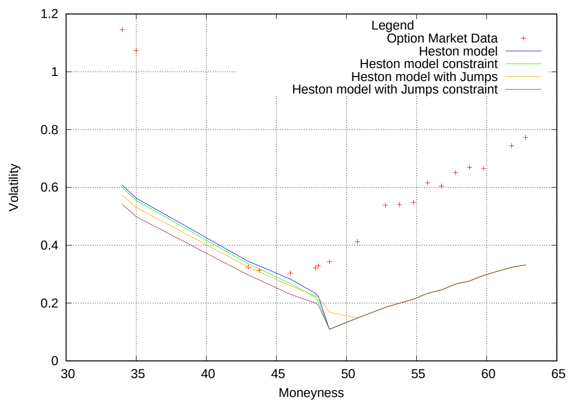
Table 5.12: Calibration results for 10000 paths – Dow-Jones Industrial Average index call options



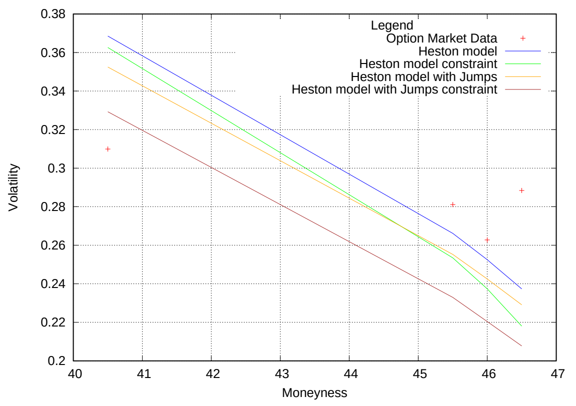
(a) Dow-Jones Industrial Average call options, maturity: 0.00273973



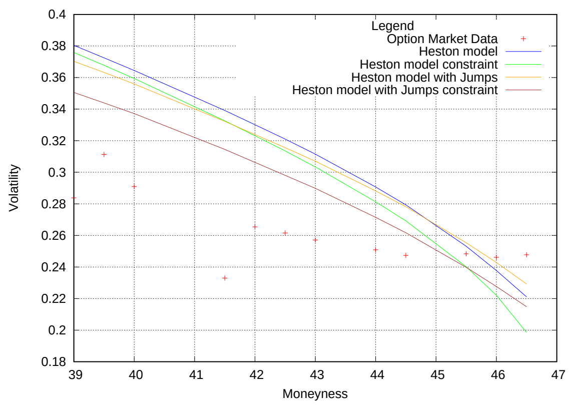
(b) Dow-Jones Industrial Average call options, maturity: 0.0219178



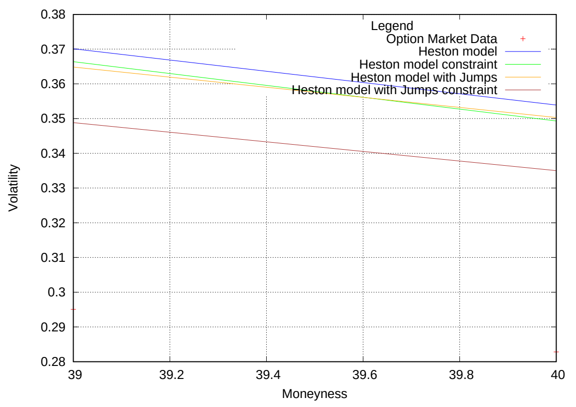
(c) Dow-Jones Industrial Average call options, maturity: 0.0410959



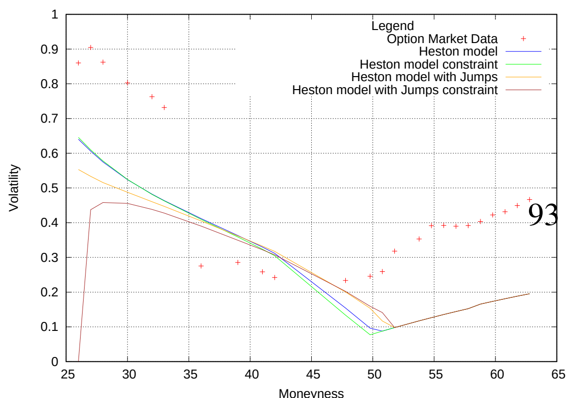
(d) Dow-Jones Industrial Average call options, maturity: 0.060274

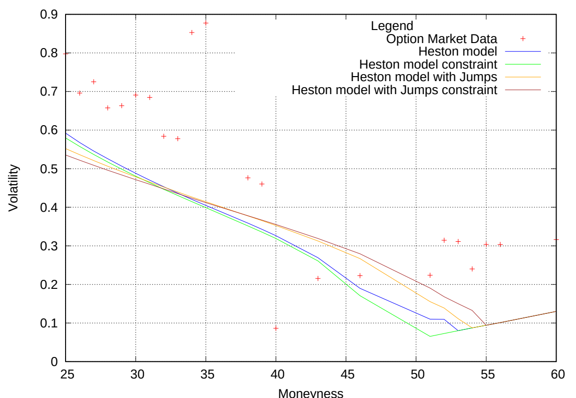


(e) Dow-Jones Industrial Average call options, maturity: 0.0794521



(f) Dow-Jones Industrial Average call options, maturity: 0.0986301





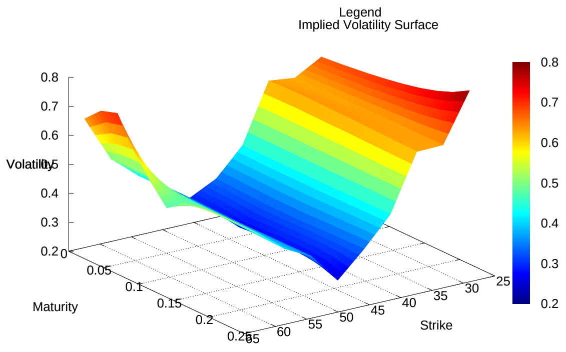
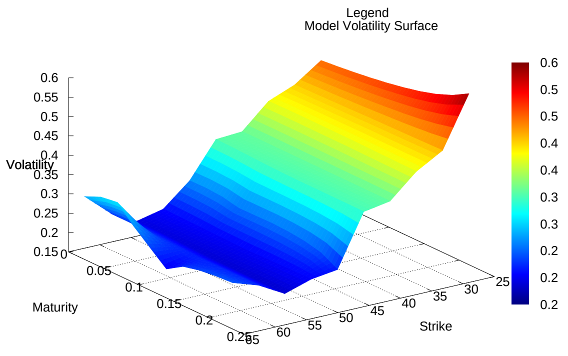
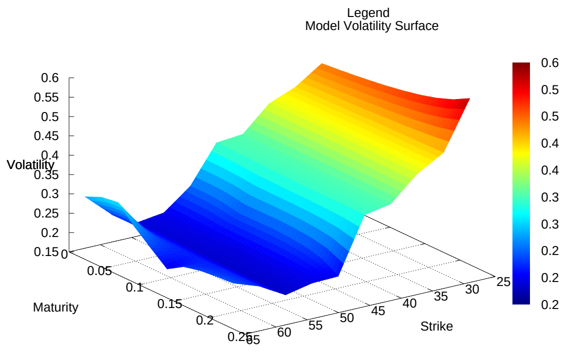


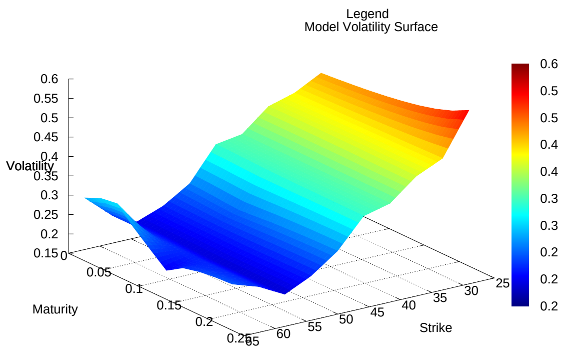
Figure 5.8: Implied volatility



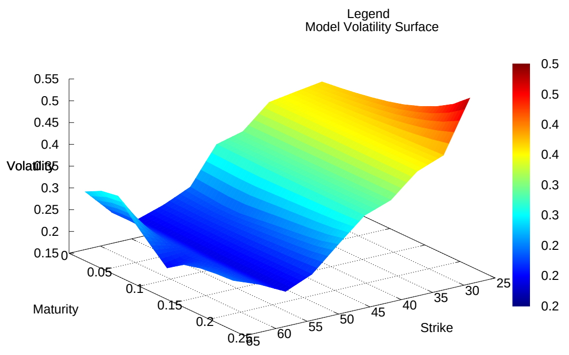
(a) Heston model volatility



(b) Heston model volatility with constraint



(c) Heston model with Jumps



(d) Heston model with Jumps and constraint

Dow-Jones Industrial Average index – Put options

Table 5.13 presents the calibration results for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition.

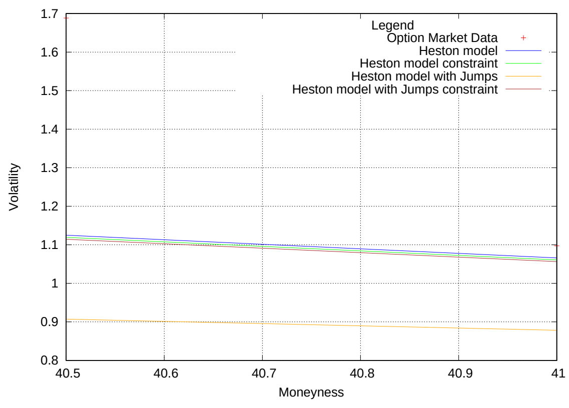
The computational experiments consider the calibration to 84 put Dow-Jones Industrial Average options with 8 maturities selected by the following database query: The Heston model has been computed via MC simulation with 100 timesteps. The best option price and volatility fitting for the Dow-Jones Industrial Average index is for the Heston model without the Feller condition. In this case, the RMSE option price error is 3.99134 and the RMSE volatility error is 2.04302. Figures 5.10a, 5.10b, 5.10c, 5.10d, 5.10e, 5.10f, 5.10g, 5.10h show the volatility slices for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition repsectively. Figure 5.10 presents the implied volatility surface for Dow-Jones Industrial Average index for the put options. Figures 5.11a, 5.11b, 5.11c, 5.11d present the model volatility surfaces.

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='DXJ' AND IMPLIED_VOLATILITY > 0 AND
   OPTION_TYPE=1 AND MATURITY < 1.0 AND OPTION_PRICE > 0
   AND MATURITY_ID < 8 ORDER BY OPTION_TYPE, MATURITY,
   STRIKE ASC"
```

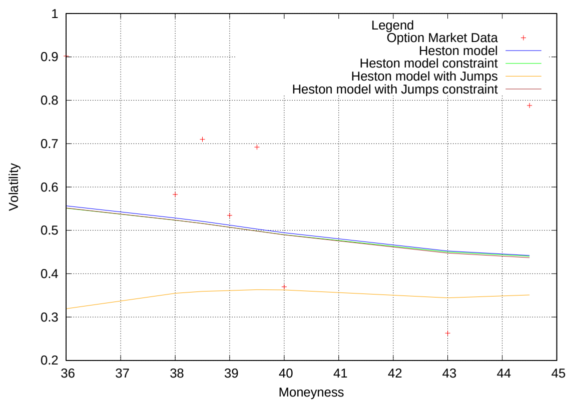
Figure 5.9: Database query for put option price matrix for Dow-Jones Industrial Average index

Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.127332	0.12559	0.0744969	0.124055
κ	0.142927	0.00362353	0.001	0.264662
θ	0.0861626	0.002271	0.291645	0.323881
σ	1	0.658093	1	1
ρ	-0.380209	-0.516055	0.300063	-0.382532
l	–	–	0.001	0.001
μ	–	–	0.001	0.001
RMSE (volatility)	2.04302	2.08609	2.7641	2.05014
RMSE (option price)	3.99134	4.01513	4.38301	3.99476
Number of major/minor iterations	14/44	8/36	2/11	12/42
Simulation time	85755	49021	32809	140431
Calibration time	85787	49044	32824	140483

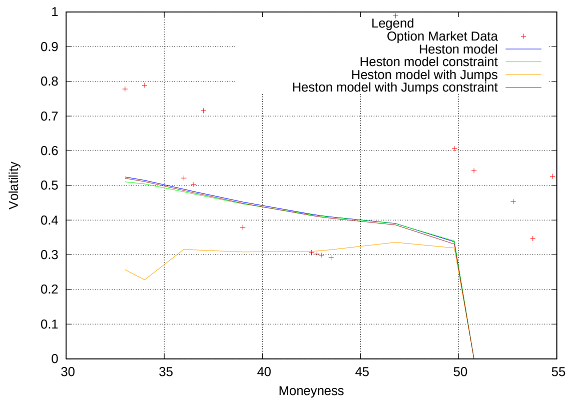
Table 5.13: Calibration results for 10000 paths – Dow-Jones Industrial Average index put options



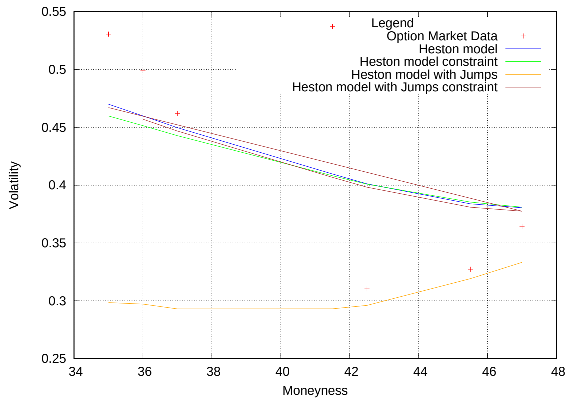
(a) Dow-Jones Industrial Average put options, maturity: 0.00273973



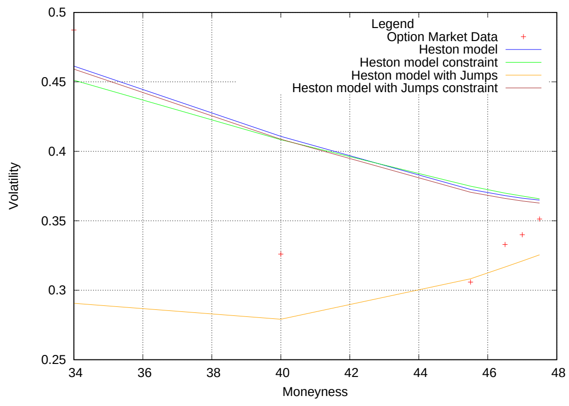
(b) Dow-Jones Industrial Average put options, maturity: 0.0219178



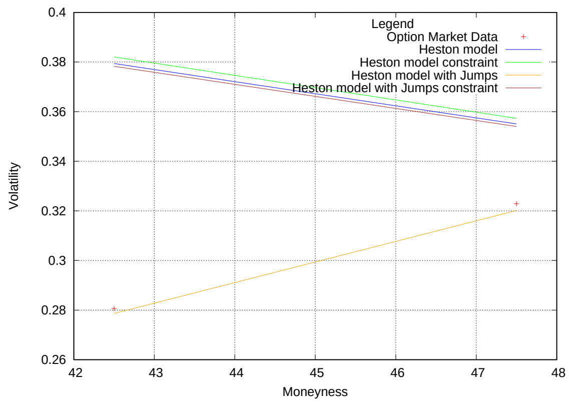
(c) Dow-Jones Industrial Average put options, maturity: 0.0410959



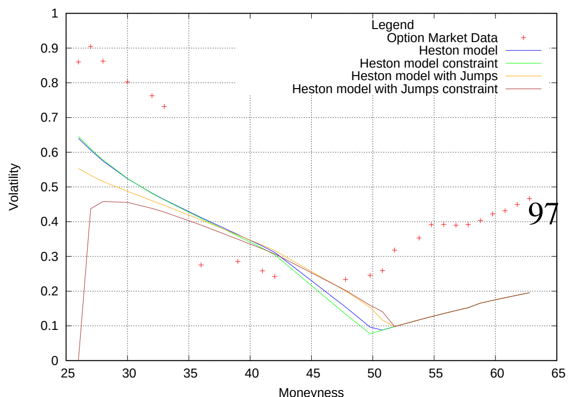
(d) Dow-Jones Industrial Average put options, maturity: 0.060274

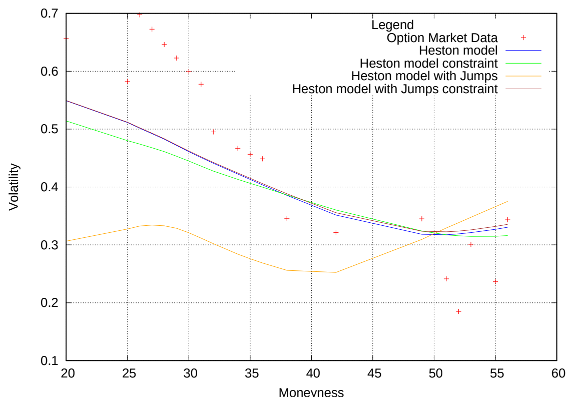


(e) Dow-Jones Industrial Average put options, maturity: 0.0794521



(f) Dow-Jones Industrial Average put options, maturity: 0.0986301





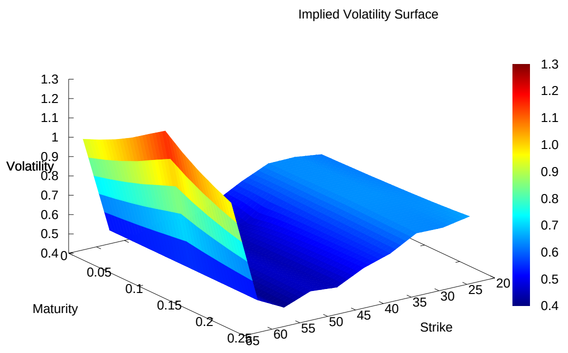
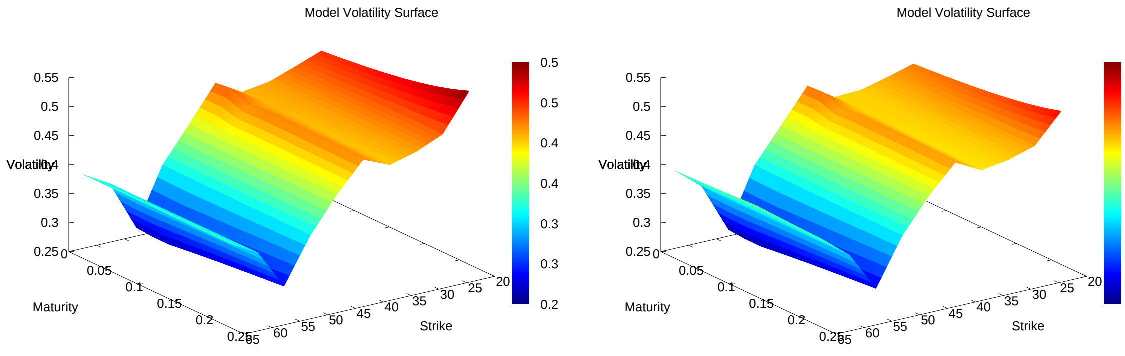
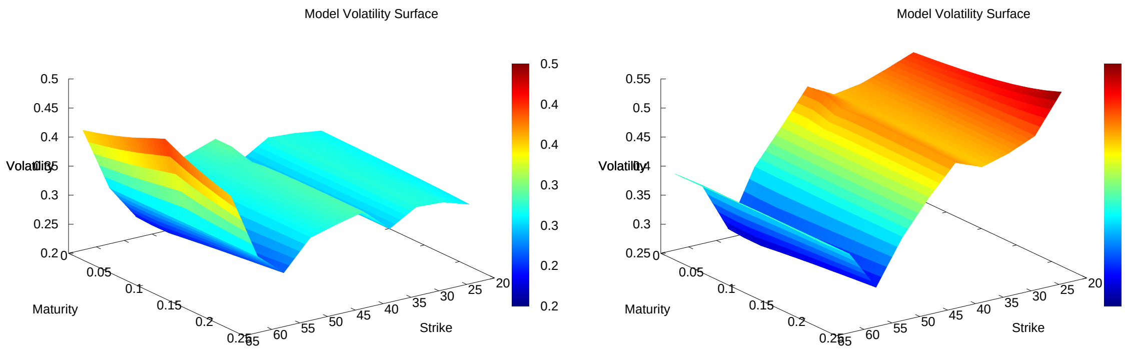


Figure 5.10: Implied volatility



(a) Heston model volatility (b) Heston model volatility with constraint



(c) Heston model with Jumps (d) Heston model with Jumps and constraint

BP – Call options

Table 5.14 shows the calibration results for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps and the Heston model with Jumps and the Feller condition.

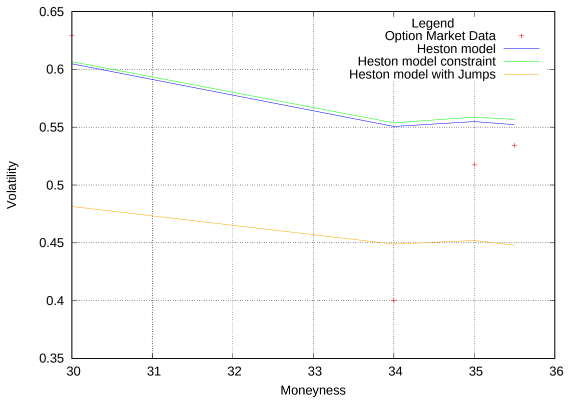
The computational experiments consider the calibration to 44 call BP options with 7 maturities selected by the following database query: The best option price fitting is for the Heston model without the Feller condition. In this case, the RMSE error is 0.896763. The best volatility fitting is for the Heston with the Feller condition. The RMSE volatility error is 1.0649. Figures 5.12a, 5.12b, 5.12c, 5.12d, 5.12e, 5.12f, 5.12g show the volatility slices. Figure 5.12, shows the implied volatility for the BP call options. Figures show 5.13a, 5.13b, 5.13c, 5.13d show the model volatility surfaces. 44 options, 8 maturities 100 steps

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
     IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
     COMMODITY_NAME='BP' AND IMPLIED_VOLATILITY > 0 AND
    OPTION_TYPE=0 AND MATURITY < 1.0 AND OPTION_PRICE > 0
    AND MATURITY_ID < 8 ORDER BY OPTION_TYPE, MATURITY,
    STRIKE ASC"
```

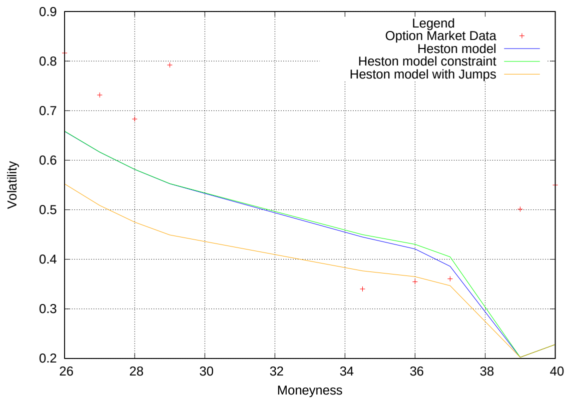
Figure 5.11: Database query for call option price matrix for BP

Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.190154	0.192258	0.116731	0.15636
κ	6.26606	9.25539	0.001	1.56702
θ	0.001	0.0463328	0.00100049	0.0178398
σ	1	1	0.52634	1
ρ	-1	-0.999077	-0.999998	-1
l	–	–	0.001	0.132913
μ	–	–	0.00100076	0.001
RMSE (volatility)	1.1	1.0649	1.26031	1.1514
RMSE (option price)	0.896763	0.91674	1.16979	0.951865
Number of major/minor iterations	15/44	24/67	3/16	8/34
Simulation time	119851	140776	38743	166898
Calibration time	119888	140820	38764	166950

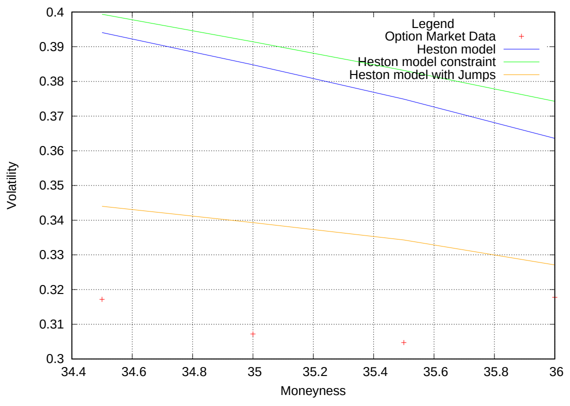
Table 5.14: Calibration results for 10000 paths – BP call options



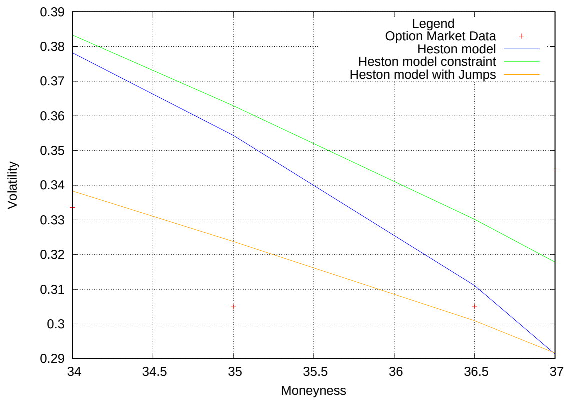
(a) BP call options, maturity: 0.0219178



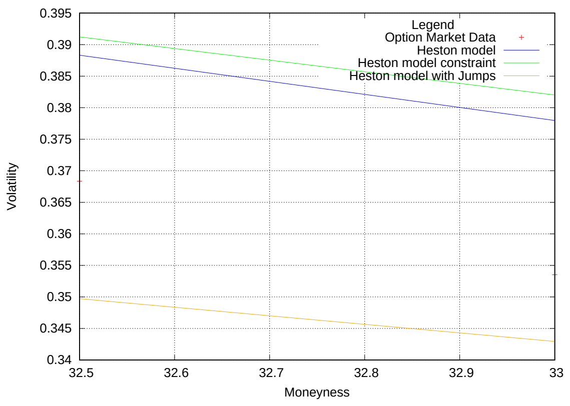
(b) BP call options, maturity: 0.0410959



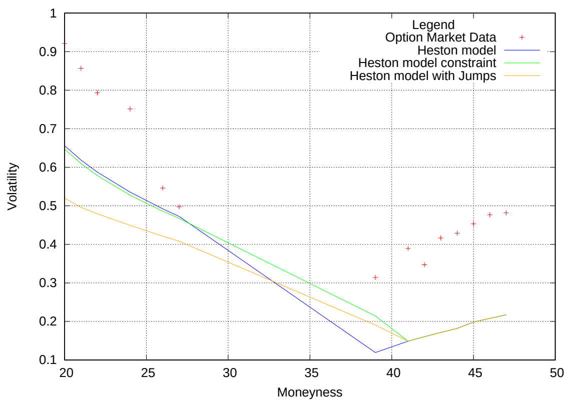
(c) BP call options, maturity: 0.060274



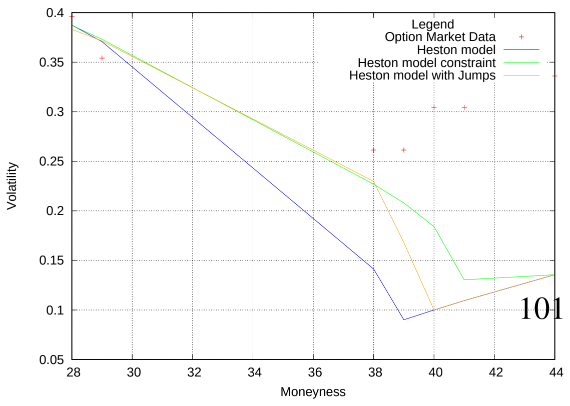
(d) BP call options, maturity: 0.0794521



(e) BP call options, maturity: 0.0986301



(f) BP call options, maturity: 0.117808



(g) BP call options, maturity: 0.213699

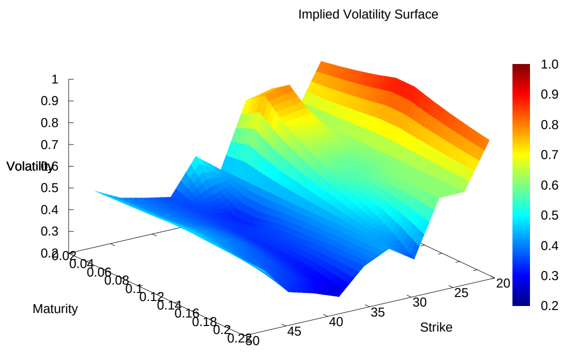
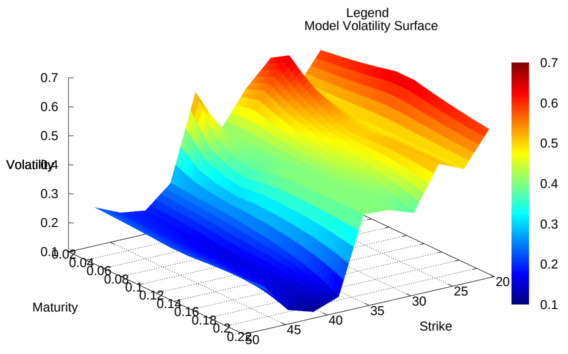
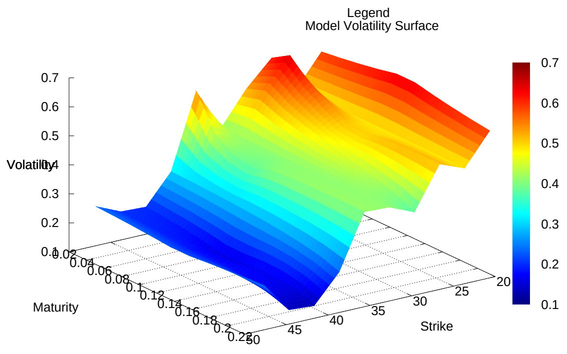


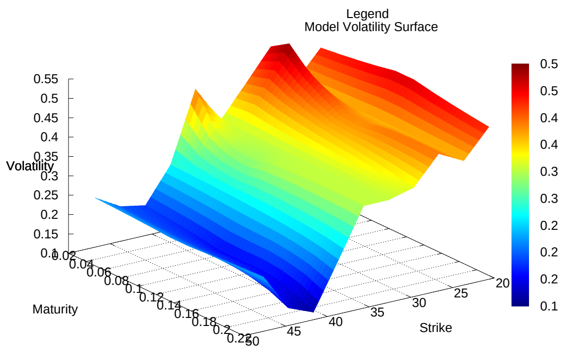
Figure 5.12: Implied volatility



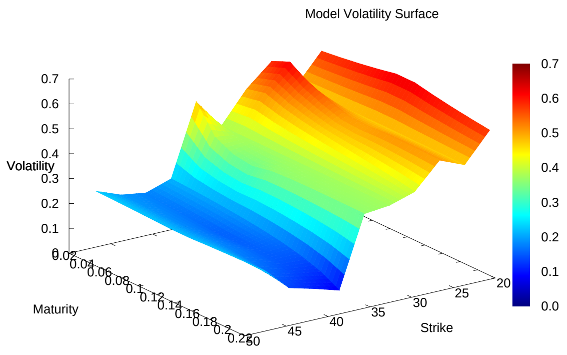
(a) Heston model volatility



(b) Heston model volatility with constraint



(c) Heston model with Jumps



(d) Heston model with Jumps and constraint

BP – Put options

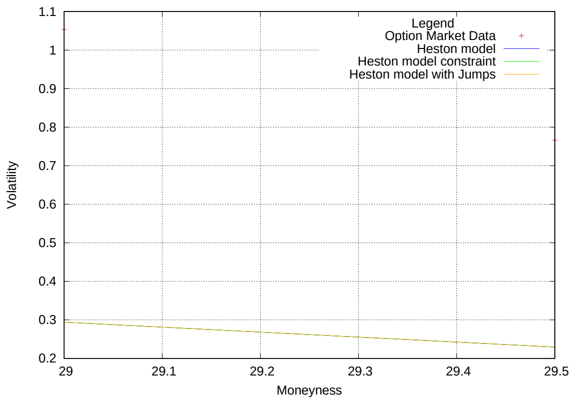
The calibration results are presented in Table 5.15. The best fitting is for the Heston without the Feller condition. In this case, the RMSE error is 1.52412. The best volatility fitting is for the Heston model with the Feller condition. The RMSE volatility error is 2.26233. Figures 5.14a, 5.14b, 5.14c, 5.14d, 5.14e, 5.14f, 5.14g present the volatility slices for the models. Figure 5.14 presents the implied volatility for the BP put options. Figures 5.15a, 5.15b, 5.15c, 5.15d present the model volatility surfaces. The experiments consider 42 BP put options with 7 maturities selected by the following database query:

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='BP' AND IMPLIED_VOLATILITY > 0 AND
   OPTION_TYPE=1 AND MATURITY < 1.0 AND OPTION_PRICE > 0
   AND MATURITY_ID < 8 ORDER BY OPTION_TYPE, MATURITY,
   STRIKE ASC"
```

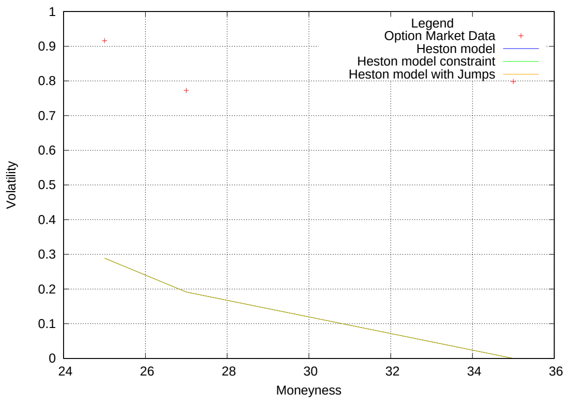
Figure 5.13: Database query for put option price matrix for BP

Parameters	Heston model	Heston model with constraints	Heston model with Jumps	Heston model with Jumps with constraints
V_0	0.0198473	0.0126823	0.012435	0.00859615
κ	0.340051	0.78193	0.997242	1.13231
θ	1	0.654892	0.500051	0.524023
σ	1	1	0.68194	0.987115
ρ	0.0782293	-0.0485706	0.212111	-0.0863248
l	–	–	0.001	0.001
μ	–	–	0.001	0.001
RMSE (volatility)	2.27855	2.26233	2.36248	2.26803
RMSE (option price)	1.52412	1.52484	1.54388	1.52668
Number of major/minor iterations	48/90	10/34	9/27	17/39
Simulation time	317892	65671	155068	377065
Calibration time	317983	65692	155105	377157

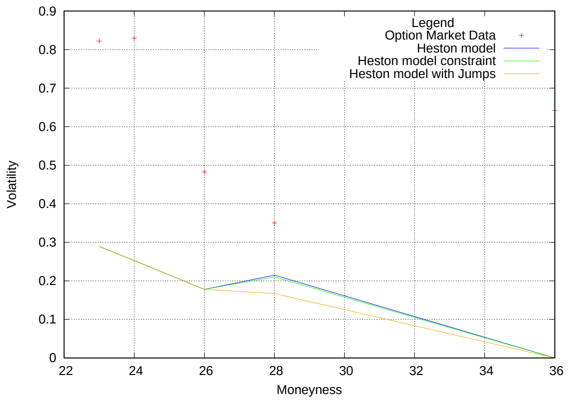
Table 5.15: Calibration results for 10000 paths – BP put options



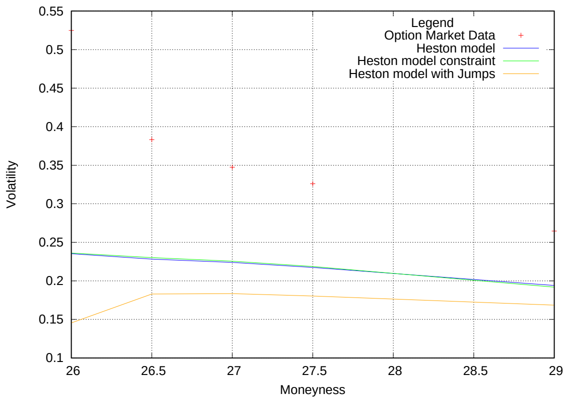
(a) BP put options, maturity: 0.0219178



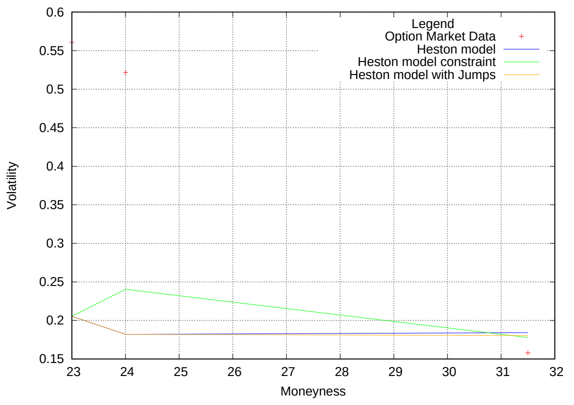
(b) BP put options, maturity: 0.0410959



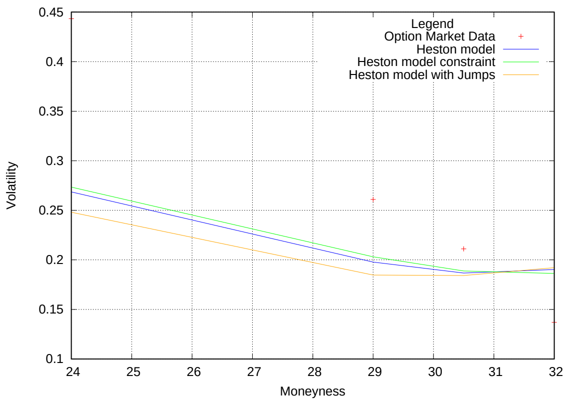
(c) BP put options, maturity: 0.060274



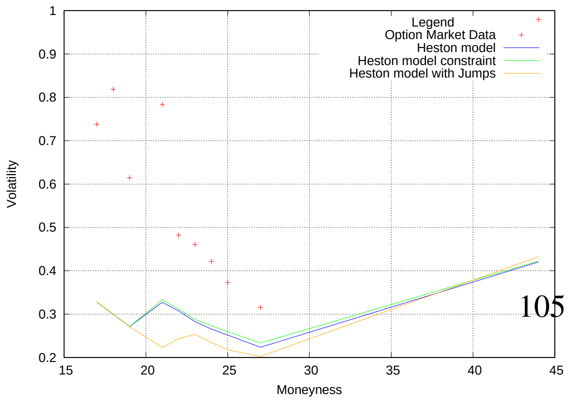
(d) BP put options, maturity: 0.0794521



(e) BP put options, maturity: 0.0986301



(f) BP put options, maturity: 0.117808



(g) BP put options, maturity: 0.213699

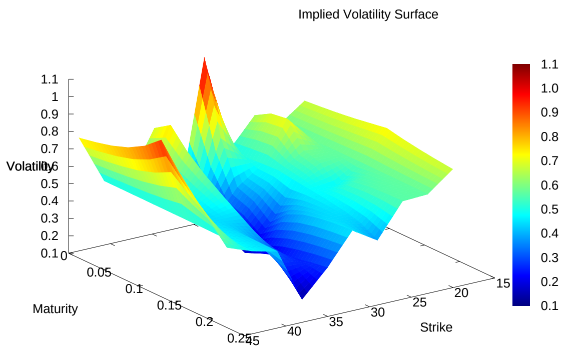
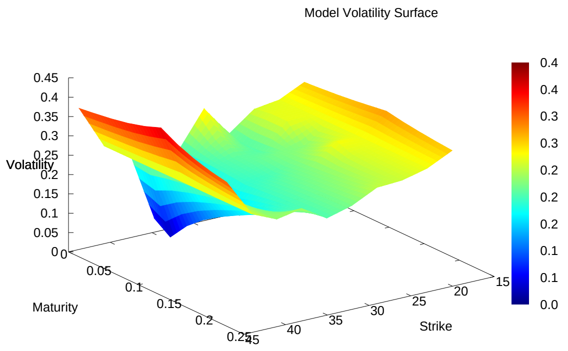
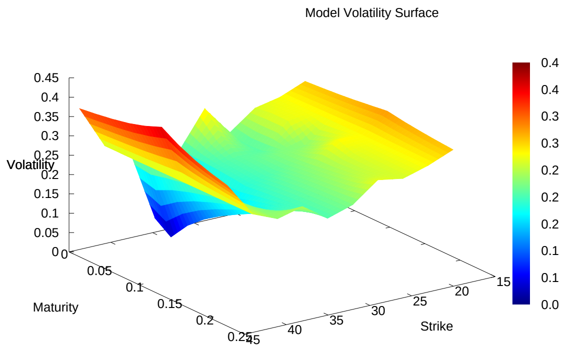


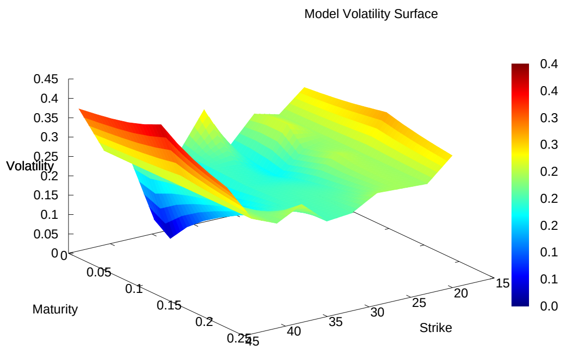
Figure 5.14: Implied volatility



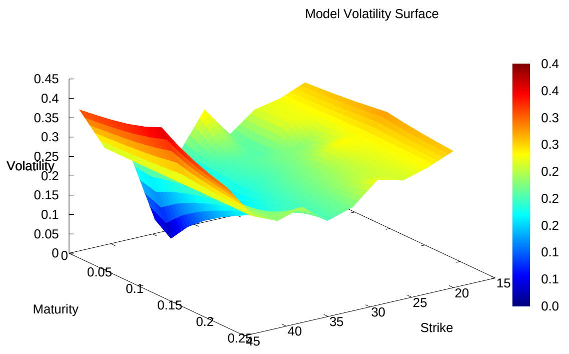
(a) Heston model volatility



(b) Heston model volatility with constraint



(c) Heston model with Jumps



(d) Heston model with Jumps and constraint

5.2.7 Summary

Computational experiments show that the first-order sensitivity are calculated via the Adjoint in 1.8x that of the function evaluation for the Heston model. The Adjoint method improves the sensitivity calculation by the factor of 16x vs. finite difference methods. The maximum speedup for the OpenMP implementation is around 11x for 12 threads. The GPU implementation improves performance by the factor of 58x vs. a single-thread OMP implementation. The multi-GPU version improves performance by two orders of magnitude vs. a single-thread OMP implementation.

5.3 Heston model calibration using the Adjoint and MC methods on FPGA

5.3.1 Overview

In this section, the computational results for the Heston model calibration on FPGA are presented. This include a computational environment and input data description.

5.3.2 Computational Environment

Experiments have evaluated the MC simulation for the Heston model evaluation/first-order sensitivities and model calibration. These compare the Adjoint methods with Finite Differences. Tests were performed on the following hardware configurations:

1. OpenMP: Intel Xeon X5650 2.66 Ghz with 48 GB RAM
2. FPGA: Maxeler MAIA technology with an Xilinx Virtex FPGA card

Total speedup S_{total} is measured as follows

$$S_{total} = S_{HPC} \cdot S_{method} \tag{5.4}$$

where S_{HPC} denotes the speedup on a HPC platform. S_{method} is the speedup from applying the numerical method.

5.3.3 Input Data

The performance benchmarks for the Greeks' calculation have been done for the following input parameters for the Heston model: S_0 = 100, r=0.005, q = 0. The Heston

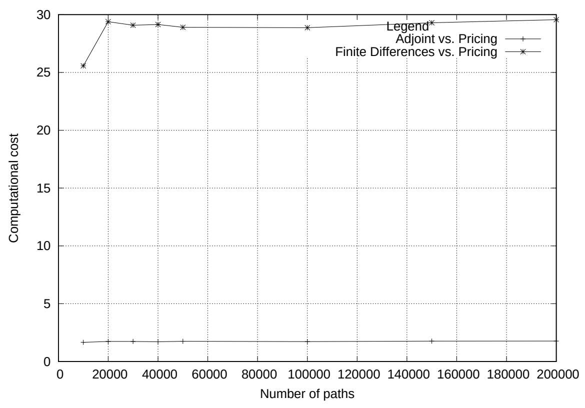
	V_0	κ	θ	σ	ρ	l	μ
Value	0.02	0.4	0.1	0.3	0.3	0.1	0.1
Lower bound	10^{-16}	10^{-3}	10^{-3}	10^{-16}	-0.99	0.001	0.001
Upper bound	2.0	10.0	1.0	1.0	0.99	1.0	1.0

Table 5.16: Lower and upper bounds for the Heston model calibration

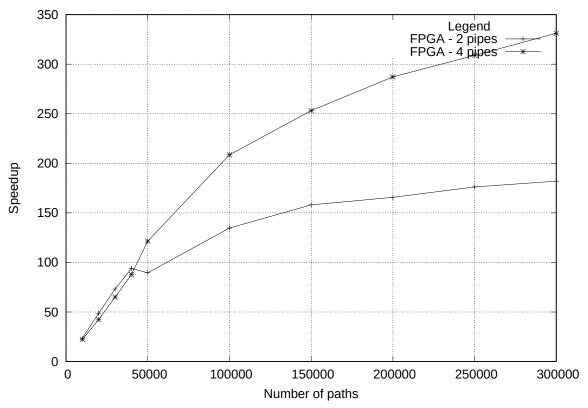
model calibration has been tested with the following configurations: 300 timesteps, 10000 paths. Table 5.16 presents the input values, lower and upper bounds for the optimization.

5.3.4 Performance results

Figure 5.15a presents a performance comparison for the Adjoint and Finite Difference methods for the first-order sensitivities compared with pricing via MC simulation. The Adjoint improves the gradient calculation. The gradient computation cost via the Adjoint is calculated at around 1.8x the time required for option pricing; the gradient via Finite Difference methods requires around 30x the time required for option pricing.



(a) Performance results – Differentiation vs. Pricing



(b) Performance results for the Greeks' calculation via the Adjoint - FPGA vs. OMP (a single-thread implementation)

The execution times on FPGA have been compared with a single-thread implementation presented in the previous section.

Maxeler FPGA cards

Figure 5.15b shows the performance results for the FPGA implementation using 2 and 4 pipes. For the 2 pipeline implementation, the maximum speedup is 180x for 300,000 paths when compared to a single-thread implementation using OMP. The 4-pipelined implementation improves performance by up to 330x compared with a single-thread implementation using OMP.

Calibration results

Table 5.17 shows the execution times in milliseconds for the Heston model calibration on multi-core and FPGA architectures; in the brackets the number of major and minor iterations during the optimization are given. Table 5.18 shows the calibration results for the Heston model to 101 call SPX options. The Heston model volatility is retrieved via the Newton-Raphson method from the model prices. Calibration using OMP utilizes a single MC simulation for the entire option matrix. Calibration using FPGA uses a single MC simulation per single option.

5.3.5 Summary

The computational experiments show that the FPGA implementation using 4 pipes improves performance of the sensitivity calculation by the factor of 331x vs. a single-thread implementation. For the version using 2 pipes, the speedup is around 181x vs a single-thread implementation.

Model	OpenMP (1 thread) – Simulation	OpenMP (1 thread) – Calibration	OpenMP (8 threads) – Simulation	OpenMP (8 threads) – Calibration	FPGA (2 pipes)– Simulation	FPGA (2 pipes)– Calibration	FPGA (4 pipes) – Simulation	FPGA (4 pipes) – Calibration
Heston model	183468 (9, 20)	183495 (9, 20)	101506 (14, 32)	101537 (14, 32)	91431 (3, 9)	91442 (3, 9)	38031 (2, 9)	38037 (2, 9)
Heston model with constraints	178535 (13, 32)	178562 (13, 32)	44842 (10, 34)	44860 (10, 34)	140379 (4, 18)	140397 (4, 18)	33527 (2, 12)	33534 (2, 12)

Table 5.17: Simulation and calibration times during the calibration process for 10000 paths and 300 timesteps

Parameters	Heston model	Heston model with constraints
V_0	0.012286	0.0200341
κ	0.001	10
θ	0.001	0.00299607
σ	0.436461	0.357109
ρ	-1	-0.987813
RMSE (volatility)	2.16467	2.08891
RMSE (option price)	0.396623	0.28162
Number of major iterations	9	13
Number of minor iterations	20	32

Table 5.18: Calibration results for 10000 paths

5.4 Parallel non-linear least squares optimization framework using Automatic Differentiation

5.4.1 Overview

In this section, the computational results for the parallel non-linear least squares optimization framework using Automatic Differentiation are included. This contains a computational environment and input data description. Further, there are presented performance results. Next, the accuracy results for the sensitivity calculation are included. Further, there are presented the calibration results.

5.4.2 Computational Environment

The parallel nonlinear least-squares optimization framework has been tested with a financial case study – the Heston model calibration. Experimentation was performed on the following hardware configurations:

1. OpenMP: Intel Xeon X5650 2.66 Ghz with 6 cores supporting execution of 12 threads with 48 GB RAM
2. OpenMP with Xeon-Phi & OpenMPI: Intel Xeon-Phi Coprocessor 7120p
3. CUDA and OpenCL: NVIDIA Kepler K40 with 2880 cores
4. Intel Core i7 -4810MQ 2.80 GHz with 8 GB RAM

5.4.3 Input Data

The input parameters for the Heston model calibration are: S_0 = 100, r=0.005, q = 0. The sensitivity calculation has been performed for the following input parameters: V_0 = 0.01, κ = 0.4, θ = 0.1, σ = 0.2, ρ = 0.3, S_0 = 50, K=50, r=0.005, q = 0, maturity = 0.5. Table 5.19 shows the input values, lower and upper bounds for the optimization. The DAG size for the semi-closed form Heston model is 10653 nodes. For the integral computation in the semi-closed form Heston model, the Gauss Kronrod integration method has been used.

	V_0	κ	θ	σ	ρ
Value	0.02	0.4	0.1	0.2	0.3
Lower bound	10^2	10^{-2}	10^{-2}	10^{-2}	-0.99
Upper bound	2.0	5.0	1.0	1.0	0.99

Table 5.19: Lower and upper bounds for the Heston model calibration

Number of options	NAG (Heston model)	NAG (FD)	Optimization Framework (Heston model)	Optimization Framework (FD)	Optimization Framework (AD)
100	13.5	172.75	62.5	422.75	126.75
200	29	355.5	132.75	866.5	263.5
300	43	532	205	1332	397.5
400	57.5	705.75	272	1785.25	542.5
500	72.25	883.5	328.75	2255.75	652.25
1000	145	1765.5	657.25	4414.75	1308.5

Table 5.20: The computation times of option prices, sensitivities via the Finite Difference methods and FD. The computational experiments have been performed on an Intel Core i7 -4810MQ 2.80 GHz with 8 GB RAM.

5.4.4 Performance results

Figure 5.15a compares the gradient computation via the Adjoint and Finite Differences with pricing for the semi-closed form Heston model. The gradient calculation cost via the Adjoint method is around 2x of that of function evaluation.

Multi-core architectures (OpenMP)

Performance experiments on HPC platforms consider the gradient calculation with different numbers of options. Figure 5.15b presents the speedup of the OpenMP implementation with a different number of threads vs. a single-thread implementation. The maximum speedup is around 9x for the implementation using 12 threads. With a greater number of threads the speedup decreases, as there is more threads than available cores. Table 5.21 shows the ratio of the achieved speedup vs. the maximum theoretical speedup.

GPU (CUDA & OpenCL)

Figure 5.15a shows performance results for the gradient calculation for a different number of options on GPU compared with a single-thread CPU implementation. The

Number of options	OMP 2 threads	OMP 4 threads	OMP 8 threads	OMP 12 threads	OMP 16 threads	OMP 32 threads
10000	80.25	77.12	66.43	50.75	34.65	14.30
20000	85.74	83.97	80.19	74.43	43.27	22.77
30000	85.22	82.02	76.43	73.12	44.04	26.40
40000	86.80	83.88	80.01	74.94	44.61	25.38
50000	84.60	81.52	78.72	72.64	44.11	27.04

Table 5.21: η – the achieved speedup vs. the maximum theoretical speedup x 100 %.

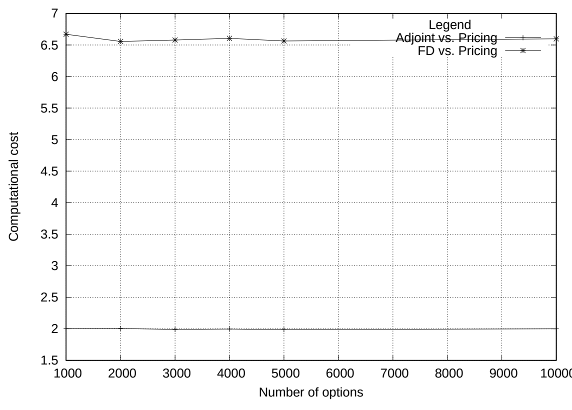
Model	OpenMP (1 thread)	OpenMP (8 threads)	Xeon-Phi (32 threads)	GPU (CUDA)
Heston model	1261 (12, 42)	410 (12, 42)	1904 (12, 42)	9027 (12, 42)
Heston model with constraints	2752 (23, 52)	772 (23, 52)	5184 (23, 52)	26180 (24, 53)

Table 5.22: Execution times for the semi-closed form Heston model calibration using the Adjoint on HPC

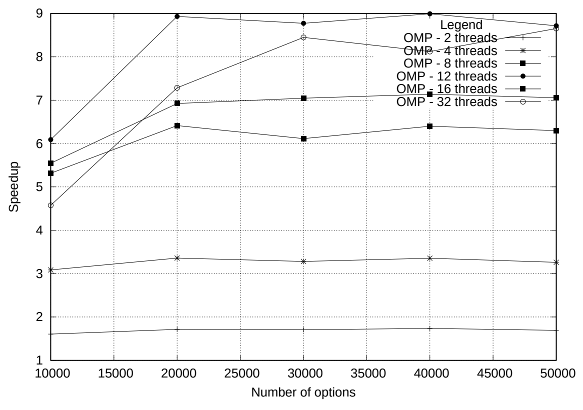
maximum speedup achieved for the CUDA implementation is 8.3x. The OpenCL version improves performance by around 9.6x.

Many-core architectures (Xeon-Phi)

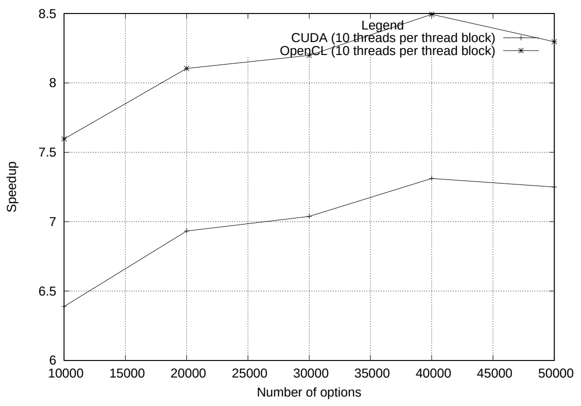
Figure 5.15b shows the performance results for the Xeon-Phi implementation. Maximum speed-up is around 34x for the implementation using 64 threads.



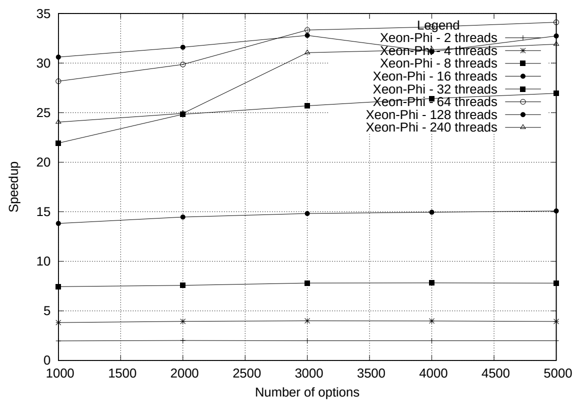
(a) Performance comparison of the option pricing and the gradient calculation via the Adjoint and FD methods



(b) Performance results for the gradient calculation via the Adjoint – OMP



(a) Performance results for the gradient calculation via the Adjoint – CUDA & OpenCL



(b) Performance results for the gradient calculation via the Adjoint – Xeon-Phi

Parameters	Adjoint	FD	NAG (FD)
C	1.91932	1.91932	1.90982
$\frac{dC}{\kappa}$	1.0895	1.0895	1.0883
$\frac{dC}{\theta}$	5.09296	5.09296	5.08076
$\frac{dC}{\sigma}$	-0.528465	-0.528463	-0.578122
$\frac{dC}{\rho}$	0.00570983	0.00571134	-0.0310216
$\frac{dC}{dV_0}$	47.4117	47.4117	47.0814
$\frac{dC}{dS_0}$	0.504303	0.504303	0.504299
Execution Time (ms)	1	5	2

Table 5.23: The sensitivity calculation for the semi-closed form Heston model – DAG size (10653 nodes)

5.4.5 Accuracy results

Table 5.23 shows the option prices and the Greeks' values calculated via the Adjoint and Finite Difference methods for the implemented semi-closed form Heston model. This is compared with the Greeks and option price computed via the NAG library. The Adjoint improves performance of the Greeks' computation by 5x when compared to finite differences.

5.4.6 Calibration results

Table 5.22 shows the execution times in microseconds for the Heston model calibration on different HPC platforms for the S&P call options; in the brackets the number

[1]Tests performed on Intel Core i7 4810MQ 2.80 GHz with 8 GB RAM

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='SPY' AND IMPLIED_VOLATILITY > 0 AND
   OPTION_TYPE=0 AND MATURITY < 1.0 AND MATURITY_ID < 10
   AND OPTION_PRICE > 0 ORDER BY OPTION_TYPE, MATURITY,
   STRIKE ASC"
```

Figure 5.15: Database query for call option price matrix for S&P 500 index

of major iterations and minor iterations during the calibration are given. Tables 5.24, 5.25, 5.26 present the calibration results for the Heston model calibration. The performance experiments for the calibration process consider the S&P option matrix from 02/06/2016 with 128 call options.

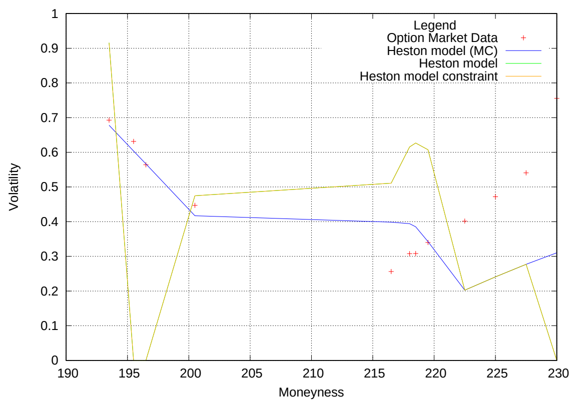
The calibration results for the semi-closed form Heston model have been compared to the Heston model solution via MC simulation.

SPX 500 index – call options

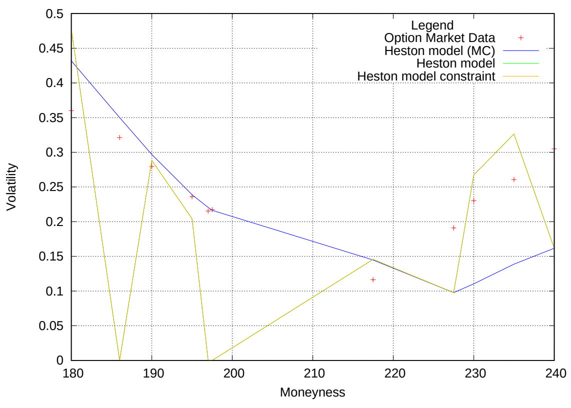
The computational experiments for the S&P index consider 123 options with 10 maturities. The option market data was selected by the following database query. The MC simulation for the Heston model has been performed with 100 timesteps. Figures 5.16a, 5.16b, 5.16c, 5.16d, 5.16e, 5.16f, 5.16g, 5.16h, 5.16a, 5.16b present the volatility slices for the call S&P options with 10 maturities. Figure 5.16a shows the implied volatility for the S&P index. Figure 5.16b presents the Heston model volatility (MC). Figures 5.16c, 5.16d show the implied volatility surfaces for the semi-closed form Heston model solutions.

Parameters	Heston model (MC)	Heston model	Heston model with constraints
V_0	0.0102997	0.0197162	0.0197156
κ	0.0941267	5	5
θ	0.385344	0.01	0.01
σ	0.266227	0.269685	0.269633
ρ	-0.941072	-0.99	-0.99
RMSE (volatility)	1.25921	1.73922	1.73922
RMSE (option price)	0.389978	1.17	1.17
Number of major/minor iterations	20/52	21/44	33/74
Simulation time	150297	–	–
Calibration time	150347	2529	5622

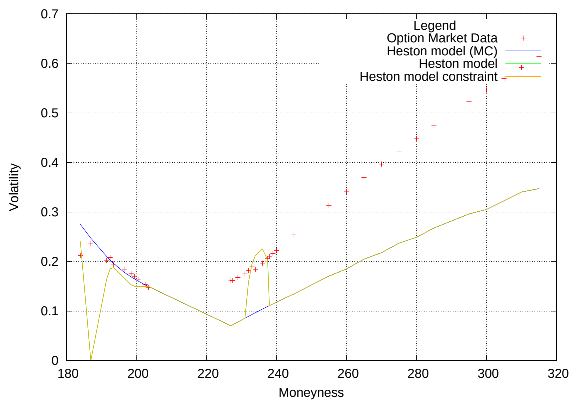
Table 5.24: Calibration results for 10000 paths – SPX 500 call options



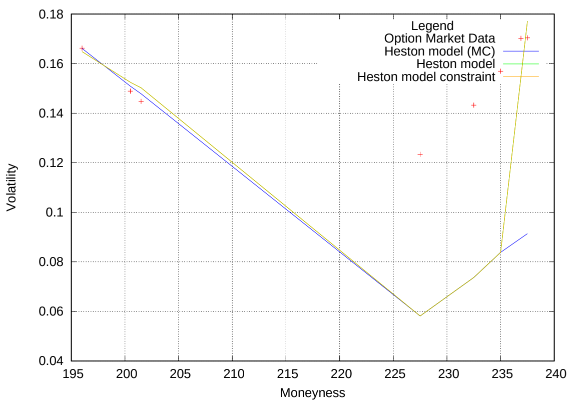
(a) SPY 500 call options, maturity: 0.00273973



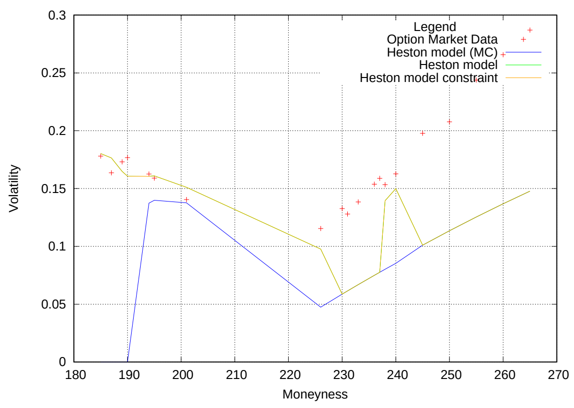
(b) SPY 500 call options, maturity: 0.0219178



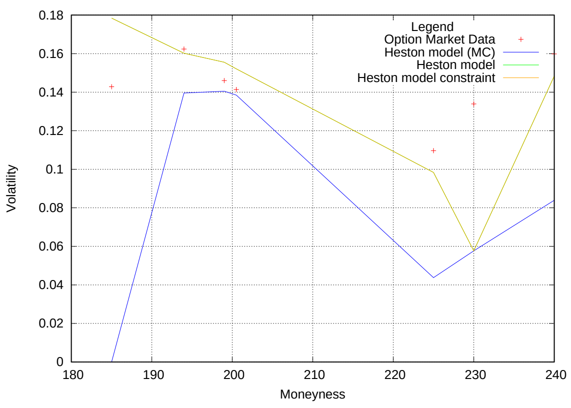
(c) SPY 500 call options, maturity: 0.0410959



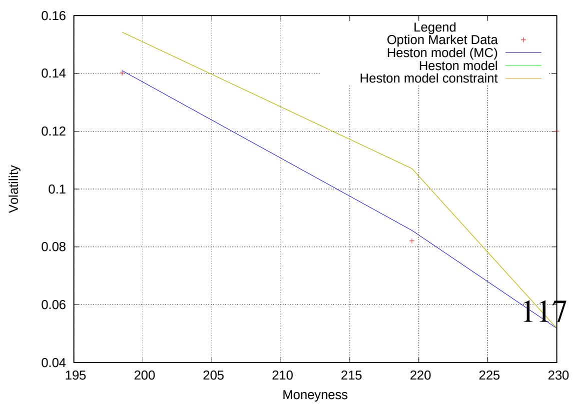
(d) SPY 500 call options, maturity: 0.060274



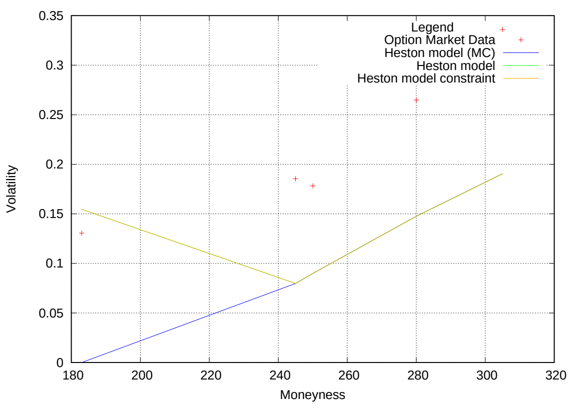
(e) SPY 500 call options, maturity: 0.0767123



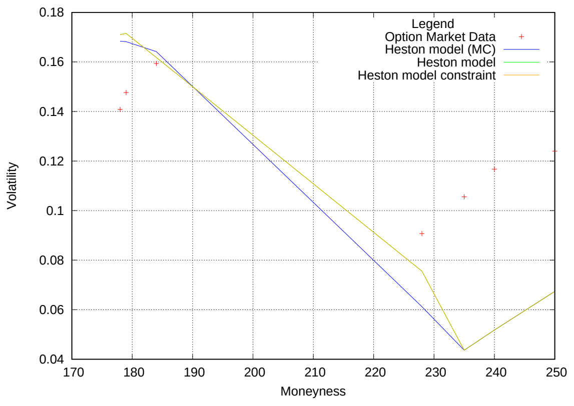
(f) SPY 500 call options, maturity: 0.0794521



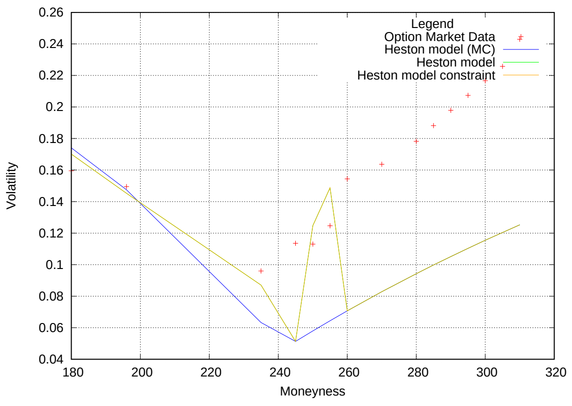
(g) SPY 500 call options, maturity: 0.0986301



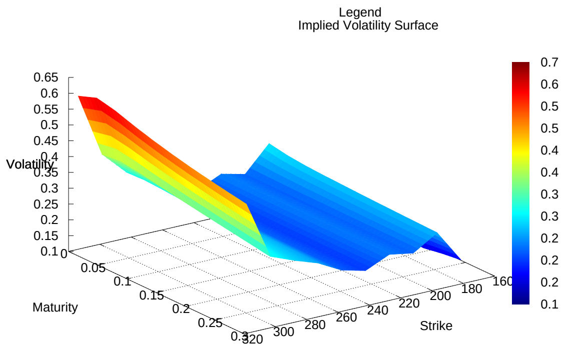
(h) SPY 500 call options, maturity: 0.117808



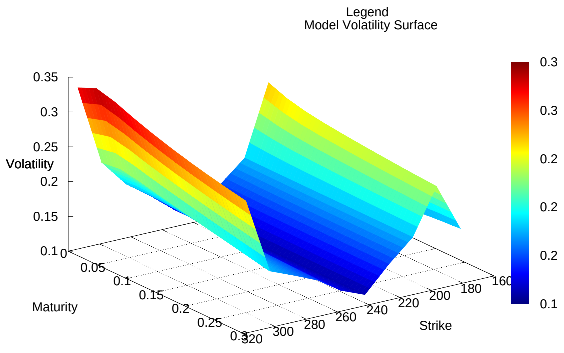
(a) SPY 500 call options, maturity: 0.213699



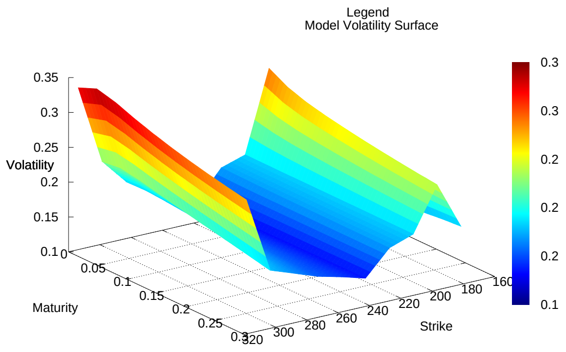
(b) SPY 500 call options, maturity: 0.290411



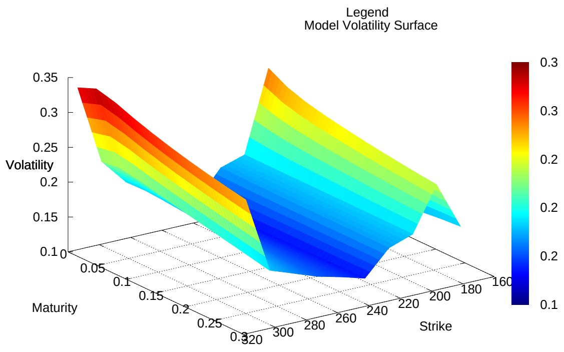
(a) Implied volatility



(b) Heston model volatility (MC)



(c) semi-closed form Heston model Heston model



(d) semi-closed form Heston model Heston model with constraint

Dow-Jones Industrial Average index – call options

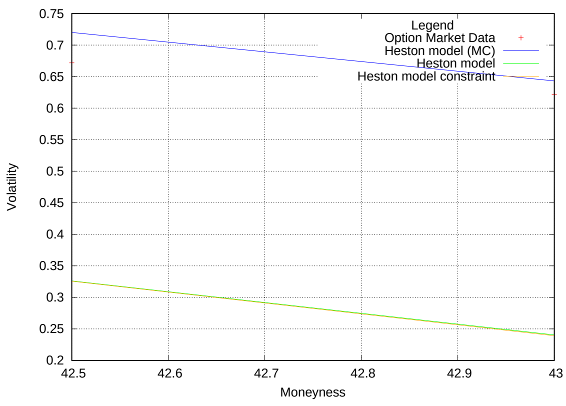
The computational experiments consider the Dow-Jones Industrial Average index with 91 call options with 8 maturities selected by the following database query: The MC simulation for the Heston model has been performed with 100 timesteps. Figures 5.17a, 5.17b, 5.17c, 5.17d, 5.17e, 5.17f, 5.17g, 5.17h present the volatility slices for the call Dow-Jones Industrial Average options with 8 maturities. Figure 5.17a presents the implied volatility surface for the Dow-Jones Industrial Average. Figure 5.17b presents the Heston model volatility (MC). Figures 5.17c, 5.17d present the implied volatility surfaces for the semi-closed form Heston model solutions.

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='DXJ' AND IMPLIED_VOLATILITY > 0 AND
   OPTION_TYPE=0 AND MATURITY < 1.0 AND MATURITY_ID < 8
   AND OPTION_PRICE > 0 ORDER BY OPTION_TYPE, MATURITY,
   STRIKE ASC"
```

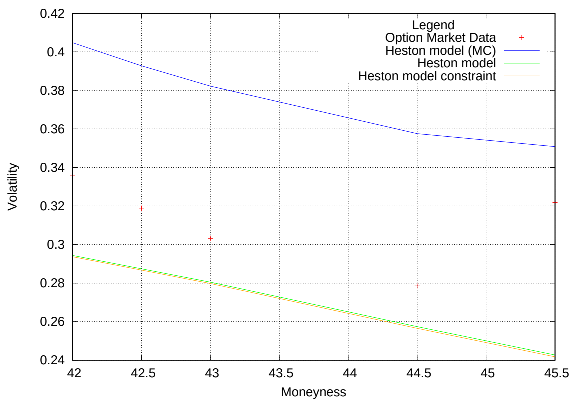
Figure 5.16: Database query for call option price matrix for Dow-Jones Industrial Average index

Parameters	Heston model (MC)	Heston model	Heston model with constraints
V_0	0.0806155	0.0724245	0.0718856
κ	0.001	0.487275	0.5
θ	0.00100253	1	1
σ	1	1	1
ρ	-0.960712	-0.724728	-0.727632
RMSE (volatility)	2.1095	2.93628	2.9353
RMSE (option price)	4.23393	4.37652	4.37651
Number of major/minor iterations	18/39	8/21	17/62
Simulation time	151377	–	–
Calibration time	151424	835	1412

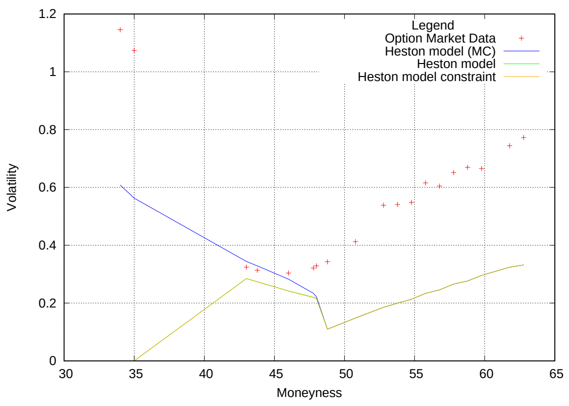
Table 5.25: Calibration results for 10000 paths – Dow-Jones Industrial Average index call options



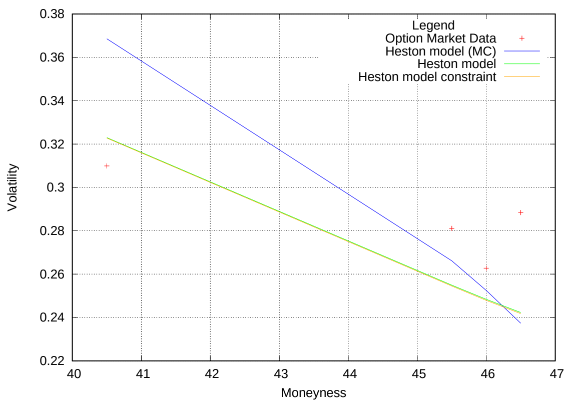
(a) Dow-Jones Industrial Average call options, maturity: 0.00273973



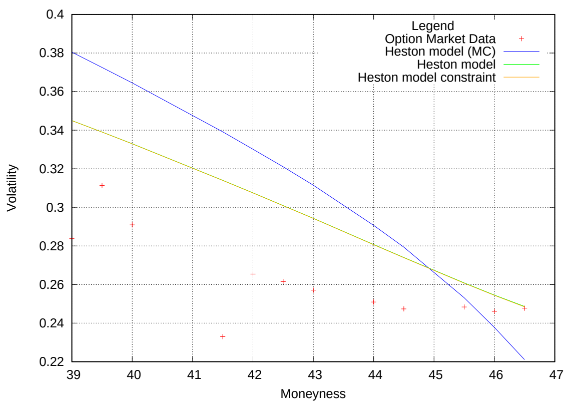
(b) Dow-Jones Industrial Average call options, maturity: 0.0219178



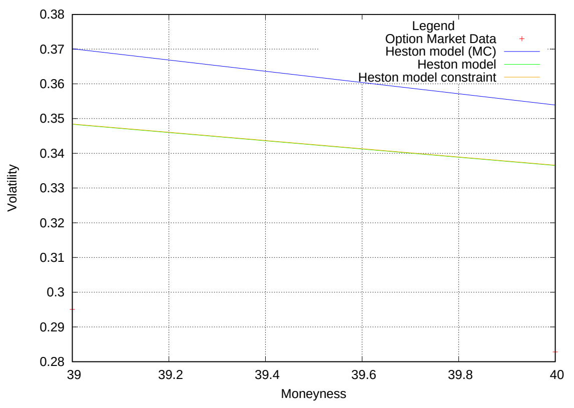
(c) Dow-Jones Industrial Average call options, maturity: 0.0410959



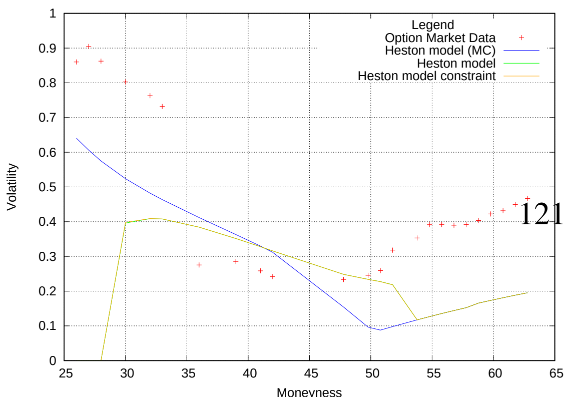
(d) Dow-Jones Industrial Average call options, maturity: 0.060274

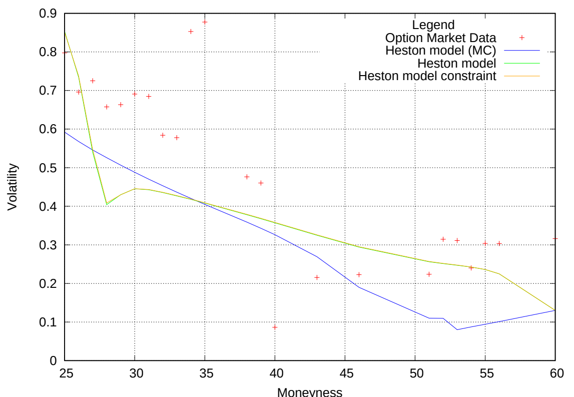


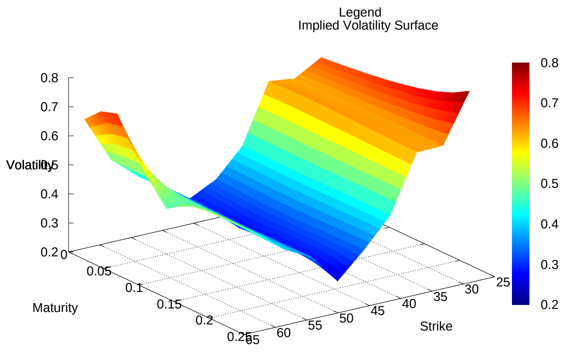
(e) Dow-Jones Industrial Average call options, maturity: 0.0794521



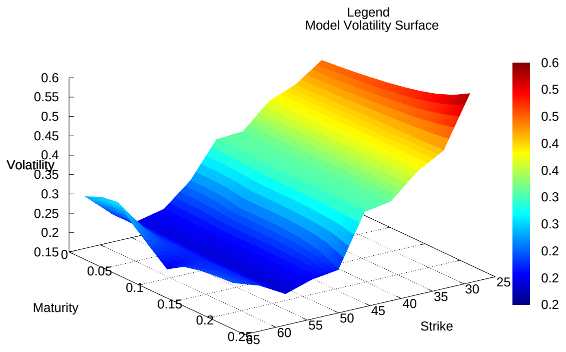
(f) Dow-Jones Industrial Average call options, maturity: 0.0986301



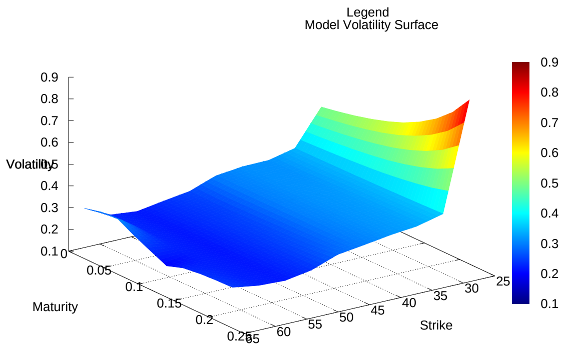




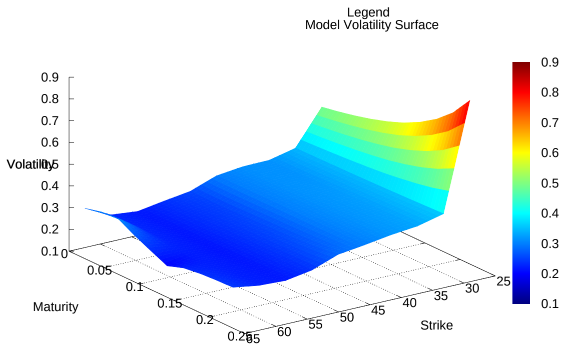
(a) Implied volatility



(b) Heston model volatility (MC)



(c) semi-closed form Heston model Heston model



(d) semi-closed form Heston model Heston model with constraint

BP – call options

The computational experiments consider 44 BP call options with 7 maturities selected by the following database query: The MC simulation for the Heston model uses 100 timesteps. Figures Figures 5.18a, 5.18b, 5.18c, 5.18d, 5.18e, 5.18f, 5.18g present the volatility slices for the call BP options with 7 maturities. Figure 5.18a shows the implied volatility surface for BP options. Figure 5.18b shows the Heston model volatility (MC). Figures 5.18c, 5.18d show the implied volatility surfaces for the semi-closed form Heston model solutions. The calibration results for the semi-closed form Heston model calibration to BP call options were performed on an Intel Core i7-4810MQ CPU 2.80GHz with 8GB RAM memory.

5.4.7 Summary

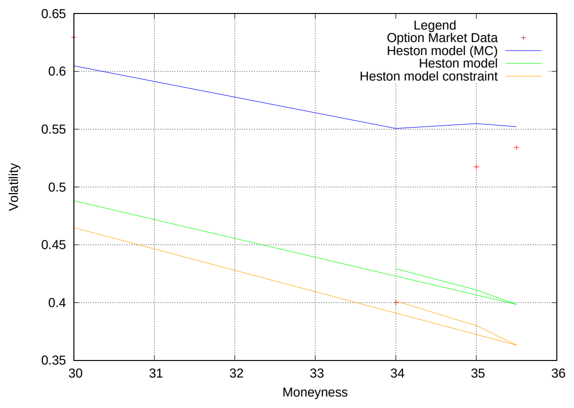
The computational experiments show that OpenMP implementation using 12 threads improves performance by around 9x vs a single-thread version. The CUDA version boosts performance of the sensitivity calculation by 8.3x. The OpenCL version improves performance by around 9.6x. The Adjoint technique reduces the gradient calculation cost by the factor of 5x when compared to finite difference methods.

```
SELECT SPOT, MATURITY, STRIKE, OPTION_TYPE, OPTION_PRICE,
    IMPLIED_VOLATILITY, MATURITY_ID FROM MARKETDATA WHERE
    COMMODITY_NAME='BP' AND IMPLIED_VOLATILITY > 0 AND
   OPTION_TYPE=0 AND MATURITY < 1.0 AND MATURITY_ID < 8
   AND OPTION_PRICE > 0 ORDER BY OPTION_TYPE, MATURITY,
   STRIKE ASC"
```

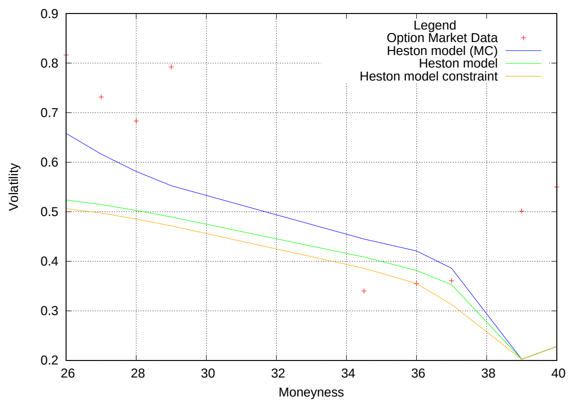
Figure 5.17: Database query for call option price matrix for BP

Parameters	Heston model (MC)	Heston model	Heston model with constraints
V_0	0.190154	0.234736	0.206194
κ	6.26606	5	5
θ	0.001	0.01	0.0963658
σ	1	1	0.981661
ρ	-1	-0.856951	-0.893393
RMSE (volatility)	1.1	1.43475	1.43301
RMSE (option price)	0.896763	1.14301	1.18815
Number of major/minor iterations	15/44	12/31	16/44
Simulation time	119851	–	–
Calibration time	119888	1706	1871

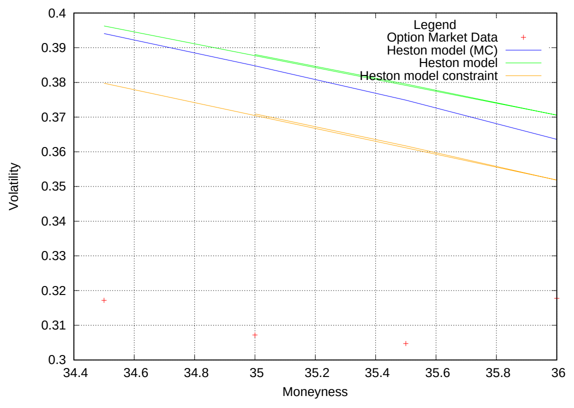
Table 5.26: Calibration results for 10000 paths – BP call options



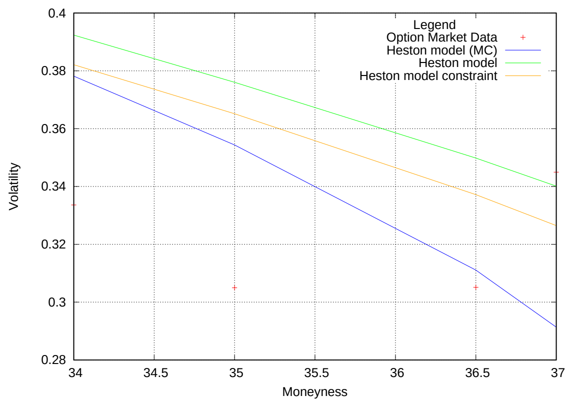
(a) BP call options, maturity: 0.0219178



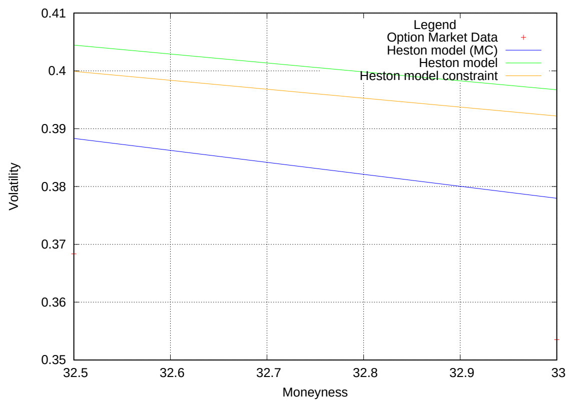
(b) BP call options, maturity: 0.0410959



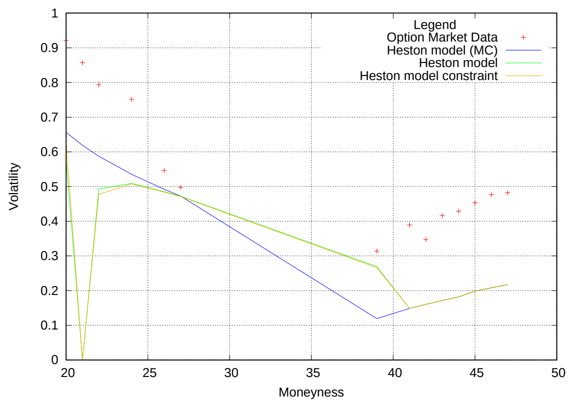
(c) BP call options, maturity: 0.060274



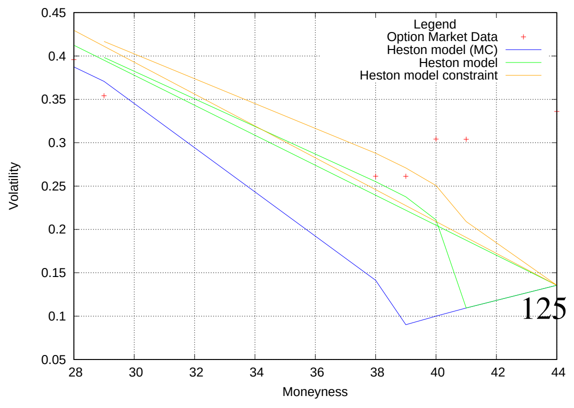
(d) BP call options, maturity: 0.0794521



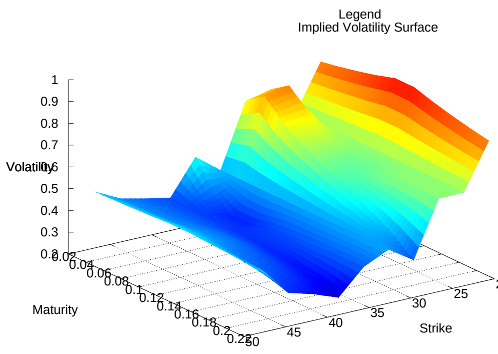
(e) BP call options, maturity: 0.0986301



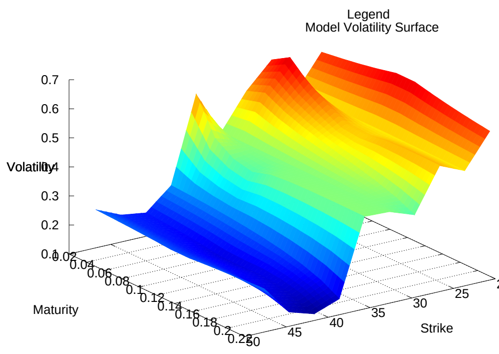
(f) BP call options, maturity: 0.117808



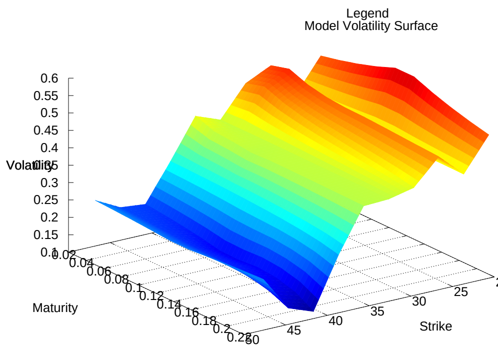
(g) BP call options, maturity: 0.213699



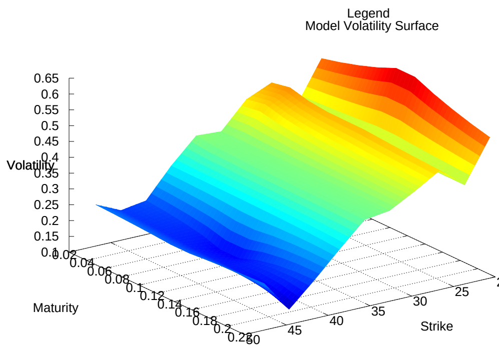
(a) Implied volatility



(b) Heston model volatility (MC)



(c) semi-closed form Heston model Heston model



(d) semi-closed form Heston model Heston model with constraint

5.5 Summary

This chapter presented computational results for the Heston model, the Heston model with the Feller condition, the Heston model with Jumps, the Heston model with Jumps and the Feller condition, the Heston model with the term-structure. The Heston model was calculated by using the semi-closed form solution and Monte-Carlo simulation. The performance experimentation in 5.2 shows that:

1. Parallel Monte-Carlo engine for the first-order sensitivity calculation and model calibration using the Adjoint:

   - The first-order sensitivities are calculated via the Adjoint in 1.8x that of the function evaluation for the Heston model;
   - The Adjoint improves the sensitivity calculation by 16x when compared with the Finite Difference methods for the Heston model;
   - The performance improvement for the OpenMP implementation increases with the number of threads utilized (η for the parallel implementation is over 90% for the number of paths greater than 100000 and the number of threads 2-12;
   - The performance improvement for the OpenMPI implementation increases with the number of nodes utilized (η for the Open MPI implementation is over 90% for the number of paths greater than 100000 and the number of nodes 2-4);
   - The CUDA implementation improves the performance of the sensitivity calculation by up to of 58x for 20000 paths on a single GPU when compared with a single-thread OMP implementation;
   - The multi-GPU version improves performance around by up to 100x on 2 GPUs when compared with a single-thread OMP implementation on a single node;
   - The sequential implementation of the Monte-Carlo simulation for the Heston model computes option prices 12 x faster than the QuantLib library 1.9 on an Intel Core i7-4810MQ CPU 2.80GHz with 8GB RAM memory;

2. Heston model calibration using the Adjoint and MC methods on FPGA:

- The FPGA implementation improves the performance of the Greeks' computation by two orders of magnitude. when compared to a sequential version on a CPU;

3. Parallel non-linear least squares optimization framework using Automatic Differentiation:

    - The performance for the OpenMP implementation increases with the number of threads varying from 2-12 (η for the implementation using from 2-8 threads is over 80% for 40000 calculating option prices);
    - The speedup achieved on a multi-core CPU is around 9x when compared with a single-thread OMP version;
    - The CUDA version boosts performance by 7.3 x when compared to a single-thread OMP version;
    - The OpenCL version improves performance by 8.49x when compared to a single-thread OMP version;
    - The Adjoint reduces the sensitivity calculation cost by 5x when compared to finite difference methods;

Chapter 6

Conclusion

This thesis has presented parallel frameworks that contribute to the performance and accuracy improvement in financial sensitivity computation and model calibration by utilizing high-performance computing platforms, numerical methods as Automatic Differentiation and Monte-Carlo methods. There have been introduced the following approaches:

1. A parallel MC engine for the first-order sensitivity calculation and model calibration using the Adjoint;
2. A hardware implementation of the Heston model calibration using the Adjoint techniques and Monte-Carlo (MC) method was presented;
3. A parallel non-linear least squares optimization framework using Automatic Differentiation;

The developed techniques have been investigated using financial case-studies: the Heston model, the Heston model with Jumps, the Heston model with term-structure for the equity option pricing. The parallel frameworks have been presented in Chapter 4.

- In section 4.1, there has been presented a HPC engine for MC simulation, sensitivity calculation and model calibration using DAG processing and AD. This utilizes graph processing and overloaded operator techniques to support general SDE models. The experiments show that the combination of the Adjoint and graph representation with parallel/distributed systems improves performance and accuracy of model evaluation, sensitivity calculation and calibration process. The performance of the gradient calculation for SDE models is improved by two orders of magnitude with the Adjoint on 2 GPUs (104x) when compared with the Adjoint on a sequential machine. Furthermore, the Adjoint techniques improve performance by up to 16x when compared to finite differences. The library provides a platform-independent API for a flexible model definition and processing flow configuration. This work can more widely be applied to solve various scientific and industrial models.

- In section 4.2, there has been presented an FPGA engine for MC simulation for the Heston model sensitivity calculation and model calibration using the Adjoint methods. This is compared with the OpenMP version; model calibration using OpenMP utilizes a single MC simulation for the entire option matrix. The FPGA implementation uses a single MC simulation per single option. The computational experiments show that the FPGA implementation improve the Greeks' calculation via the Adjoint by up to 330x compared with a single-thread implementation using the OMP framework. Furthermore, the Adjoint methods improve the gradient calculation performance by up to 16x when compared to finite difference methods. This work offers potential for improved performance in managing and hedging investment portfolios consisting of thousands of underlying assets.

- In section 4.4, there has been presented a parallel framework for non-linear least-squares optimization using AD. The framework provides a platform-independent API for flexible objective and constraint function definition. The computational experiments show significant performance improvement of the objective function gradient calculation on parallel architectures when compared to sequential CPU architectures. The speedup achieved for multi-core CPUs is around 9x when compared with a single-thread OMP version; the OpenCL implementation improves performance by 8.49x when compared with a single-thread OMP implementation.; the CUDA version improves performance by 7.3x when compared to a single-thread OMP implementation. Further, the Adjoint reduces a computational cost of the sensitivity calculation by the factor of 5x when compared to finite difference methods.

The work demonstrates performance and accuracy improvement via a combination of HPC platforms, numerical methods such as AD and MC methods with software development techniques such as graph processing and overloaded operator techniques.

The work can be applied in real-time risk management systems, in replicating portfolio systems to support investment decisions involving equity option and interest rate derivatives trading, which can be useful in managing and hedging investment portfolios.

Bibliography

[ABFK11] Michael Aichinger, Andreas Binder, Johannes Fürst, and Christian Kletzmayr. A fast and stable heston model calibration on the gpu. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 431–438, Berlin, Heidelberg, 2011. Springer-Verlag.

[Alf06] Monica S. Lam Ravi Sethi Alfred, V. Aho. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Pearson Education, 2006.

[AWVdS13] Cornelis W. Oosterlee Anthonie Willem Van der Stoep, Lech A. Grzelak. The heston stochastic-local volatility model: Efficient monte carlo simulation. *SSRN*, 2013.

[BS73] Fishcer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637, 1973.

[Cap11] Luca Capriotti. Fast greeks by algorithmic differentiation. *Journal of Computational Finance*, 2011.

[Cha13] Victor Chan. *Theory and Applications of Monte Carlo Simulations*. InTech, 2013.

[CL14] Luca Capriotti and Jacky Lee. Adjoint credit risk management. *Risk Magazine*, 2014.

[CRR79] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229 – 263, 1979.

[DLS10] L. Desorgher, F. Lei, and G. Santin. Implementation of the reverse/adjoint monte carlo method into geant4. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 621(13):247 – 257, 2010.

[Doa10] Viet Dung Doan. *Grid computing for Monte Carlo based intensive calculations in financial derivative pricing applications*. Phd thesis, University of Nice, 2010.

[dSSK$^+$11] Christian de Schryver, Ivan Shcherbakov, Frank Kienle, Norbert Wehn, Henning Marxen, Anton Kostiuk, and Ralf Korn. An energy efficient fpga accelerator for monte carlo option pricing with the heston model. In Peter M. Athanas, Jrgen Becker, and Ren Cumplido, editors, *ReConFig*, pages 468–474. IEEE Computer Society, 2011.

[DZ13] Matthew Dixon and Mohammad Zubair. Calibration of stochastic volatility models on a multi-core cpu cluster. In *Proceedings of the 6th Workshop on High Performance Computational Finance*, WHPCF '13, pages 6:1–6:7, New York, NY, USA, 2013. ACM.

[FBI06] A. Fichtner, H.-P. Bunge, and H. Igel. The adjoint method in seismology: I. theory. *Physics of the Earth and Planetary Interiors*, 157(12):86 – 104, 2006.

[FHP$^+$12] Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors. *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2012.

[FSA$^+$12] David A. Fournier, Hans J. Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N. Maunder, Anders Nielsen, and John Sibert. Ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software*, 27(2):233–249, 2012.

[Gat06] Jim Gatheral. *The volatility surface: The Pracitioner's Guide*. Wiley Finance, 2006.

[GHR$^+$16] Felix Gremse, Andreas Hfter, Lukas Razik, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated adjoint algorithmic differentiation. *Computer Physics Communications*, 200:300 – 311, 2016.

[Gla04] Paul Glasserman. *Monte Carlo methods in financial engineering*. Applications of mathematics. Springer, New York, 2004. Permire parution en dition broche 2010.

[Gro15] Numerical Algorithms Group. *NAG Documentation*, 2015. http://www.nag.co.uk/numeric/CL/nagdoc_cl23/html/FRONTMATTER/manconts.html.

[Hes93] Steven L Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *Review of Financial Studies*, 6(2):327–43, 1993.

[Hog14] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014.

[HP13] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[Hul12] John Hull. *Options, Futures and other Derivatives.* Pearson Education Limited, 2012.

[IEE08] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* August 2008.

[Int16] Intel. *Xeon Phi SDK*, 2016. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[JLT11] Q Jin, W Luk, and DB Thomas. On comparing financial option price solvers on fpga. pages 89–92. IEEE COMPUTER SOC, 2011.

[JY11a] Mark Joshi and Chao Yang. Algorithmic hessians and the fast computation of cross-gamma risk. *IIE Transactions*, 43(12):878–892, 2011.

[JY11b] Mark Joshi and Chao Yang. Efficient greek estimation in generic swap-rate market models. *Algorithmic Finance*, 1(1):17–33, 2011.

[Khr16] Khronos. *OpenCL SDK*, 2016. https://www.khronos.org/opencl/.

[KK12] Grzegorz Kozikowski and Bartlomiej Jacek Kubica. Interval arithmetic and automatic differentiation on GPU using opencl. In *Applied Parallel and Scientific Computing - 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers*, pages 489–503, 2012.

[KK13] Grzegorz Kozikowski and Bartlomiej Jacek Kubica. Parallel approach to monte carlo simulation for option price sensitivities using the adjoint and interval analysis. In *Parallel Processing and Applied Mathematics - 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part II*, pages 600–612, 2013.

[KMS09] C. Kaebe, J. H. Maruhn, and E. W. Sachs. Adjoint-based monte carlo calibration of financial market models. *Finance and Stochastics*, 13(3):351–379, 2009.

[Li09] Jiayuan Li. *The Heston Model with Term Structure*. 2009.

[LL14] S. Li and J. Lin. Many-core programming with asian option pricing. In *High Performance Computational Finance (WHPCF), 2014 Seventh Workshop on*, pages 45–52, Nov 2014.

[max14] *Multiscale Dataflow Programming Tutorial*, 2014.

[MF12] Marina Menshikova and Shaun A. Forth. Automatic differentiation of quadrature. *Optimization Methods and Software*, 27(2):323–335, 2012.

[Mih15] Ivo Mihaylov. *Numerical schemes and Monte Carlo techniques for Greeks in stochastic volatility models*. Phd thesis, Imperial College London, 2015.

[Mor06] Hans Moritsch. *High Performance Computing in FinanceOn the Parallel Implementation of Pricing and Optimization Models*. Phd thesis, Technischen Universitat Wien, 2006.

[MPZ08a] Francesca Mariani, Graziella Pacelli, and Francesco Zirilli. Maximum likelihood estimation of the heston stochastic volatility model using asset and option prices: an application of nonlinear filtering theory. *Optimization Letters*, 2(2):177–222, 2008.

[MPZ08b] Francesca Mariani, Graziella Pacelli, and Francesco Zirilli. Maximum likelihood estimation of the heston stochastic volatility model using asset and option prices: an application of nonlinear filtering theory. *Optimization Letters*, 2(2):177–222, 2008.

[NAW08] Tom Van Court Nathan A. Woods. Fpga acceleration of quasi-monte carlo in finance. *International Conference on Field Programmable Logic and Applications*, pages 335 – 340, 2008.

[NVI13] NVIDIA. *NVIDIA CUDA CURAND Library SDK*, 2013.

[NVI16] NVIDIA. *NVIDIA CUDA SDK*, 2016. `https://developer.nvidia.com/cuda-gpus`.

[Ope13] OpenMP. *OpenMP*, July 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[ope16] *Open MPI Documentation*, 2016.

[otc16] Statistical release - otc derivatives statistics. Technical report, Bank for International Settlements, 2016.

[PSLvL16] Andreas Pttmann, Sebastian Schnittert, Samuel Leweke, and Eric von Lieres. Utilizing algorithmic differentiation to efficiently compute chromatograms and parameter sensitivities. *Chemical Engineering Science*, 139:152 – 162, 2016.

[qua16] *QuantLib library*, 2016. http://quantlib.org/.

[Ras16] Lykke Rasmussen. *Computational Finance – On the search for performance*. Phd thesis, University of Copenhagen, 2016.

[sql16] *SQLite*, 2016. https://www.sqlite.org/.

[SRMLB$^+$13] Diego Sanchez-Roman, Victor Moreno, Sergio Lopez-Buedo, Gustavo Sutter, Ivan Gonzalez, Francisco J. Gomez-Arribas, and Javier Aracil. {FPGA} acceleration using high-level languages of a monte-carlo method for pricing complex options. *Journal of Systems Architecture*, 59(3):135 – 143, 2013.

[SZAW15] Simon Suo, Ruiming Zhu, Ryan Attridge, and Justin Wan. Gpu option pricing. In *Proceedings of the 8th Workshop on High Performance Computational Finance*, WHPCF '15, pages 8:1–8:6, New York, NY, USA, 2015. ACM.

[TB08] Xiang Tian and Khaled Benkrid. Design and implementation of a high performance financial monte-carlo simulation engine on an fpga supercomputer. In Tarek A. El-Ghazawi, Yao-Wen Chang, Juinn-Dar Huang, and Proshanta Saha, editors, *FPT*, pages 81–88. IEEE, 2008.

[Tho14] David Thomas. *The MWC64X Random Number Generator Documentation*, 2014.

[WG10] Andrea Walther and Andreas Griewank. *ADOL-C: A Package for the Automatic Differentiation of Algorithms written in C/C++*, 2010.

[Zha13] Yu Zhao. *High Performance Monte Carlo Computation for Finance Risk Data Analysis*. Phd thesis, Brunel University London, 2013.