Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Reduction of Computing Time for Numerical Pricing of European Multi-dimensional Options based on the COS Method

A comparison between MATLAB, C and CUDA

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

DIRK HAZENOOT

Delft, the Netherlands
January, 2016

This page is intentionally left blank.

MSc THESIS APPLIED MATHEMATICS

Reduction of Computing Time for Numerical Pricing of European Multi-dimensional Options based on the COS Method

D. Hazenoot BSc

Delft University of Technology

Responsible professor

Prof.dr.ir. C.W. Oosterlee

Other members of the thesis committee

Dr.ir. R.J. Fokkink

Dr.ir. M.B. van Gijzen

January, 2016 Delft, the Netherlands

This page is intentionally left blank.

Acknowledgment

This thesis has been submitted for the degree of Master of Science in Applied Mathematics at Delft University of Technology. Research for this thesis took place during an internship at the Delft University of Technology. The supervisor of this thesis was Kees Oosterlee, professor at the Numerical Analysis Group of Delft Institute of Applied Mathematics. I express my sincere gratitude to my supervisor Kees Oosterlee for his advice, guidance and encouragement throughout the making of this thesis. Also, I'm very grateful for the support at developing and testing computer codes, which I received from Álvaro Leitao Rodríguez, PhD student of Centrum Wiskunde & Informatica (CWI). Finally, I thank my family and friends for their direct and indirect encouragement throughout my study.

This page is intentionally left blank.

Abstract

Numerical integration methods such as the Fourier-based COS method can be used for efficiently and accurately pricing financial products. The COS method can be applied to options on one underlying stock as well as on multiple underlying stocks. However, this method suffers from an exponential increase in computational complexity as the dimensions increase. In this thesis we research how to reduce the computational time, especially for multi-dimensional options. Firstly, we discuss the COS method. Secondly, we program this method in three different languages, namely MATLAB, C and CUDA. Thirdly, we perform numerical tests: MATLAB- and C-code on a CPU and CUDA-code on a GPU. Lastly, we compare some options for the different computing times of these codes.

Key words: option pricing, European options, multi-dimensional options, COS method, Fourier-cosine series, Fourier-cosine expansion, fast Fourier transform, discrete cosine transform, C, CPU, CUDA, GPU.

Contents

Acronyms and Notation

1. Acronyms

B-S	Black-Scholes (model)
ChF	characteristic function
CUDA	compute unified device architecture
CPU	central processing unit
DCT	discrete cosine transform
DFT	discrete Fourier transform
FCT	fast discrete cosine transform
FFT	fast Fourier transform, Cooley-Tukey
GBM	geometric Brownian motion
GPU	graphics processing unit
i.i.d.	independent and identically distributed
MC	Monte Carlo (simulation)
MJD	Merton's jump diffusion (model)
OOM	out of memory
PDE	partial differential equation
PDF	probability density function
SDE	stochastic differential equation

2. Notation

$[a, b]$, $[\vec{a}, \vec{b}]$	integration range COS method
α	drift of the jump part for the MJD model
$B(t)$	money-savings account
$\vec{c}$	real vector
$\hat{c}$	DCT of $\vec{c}$
C	real array 2D or 3D
$\hat{C}$	DCT of array C
δ	standard deviation of the jump part for the MJD model
Δ	number of stocks in delta-hedging
$D(t, T)$	discount term
$\mathbb{E}^{\mathbb{Q}}[.]$	expectation under risk-neutral measure
ϵ	overall error of the COS method
ϵ^i	i-th error of the COS method
f	probability density function
$\mathcal{F}(t)$	filtration of sub-σ-algebras
$g(S(T))$, $g(\vec{S}(T))$	payoff function of an option
H	exponential function being used by DCT
$\vec{k}$	indexing vector

K	strike price of an option
λ	intensity of the jump part for the MJD model
M	size of array
$\hat{M}$	number of Monte Carlo simulations
$M(t)$	adapted stochastic process
μ, $\overline{\mu}$	drift of a stochastic process
$\vec{\mu}$	drift vector
N	sample size of COS method
$N(t)$	Poisson process
$\hat{N}(x)$	cumulative normal density function
ω, $\vec{\omega}$	variable of the characteristic function
P	payoff matrix
q	continuous dividend yield
Q	sample size of payoff coefficients $V_{\vec{k}}$
r	instantaneous risk-free interest rate
R	random variable with a normal distribution $\mathcal{N}(\mu, \sigma)$
$\hat{R}$	random variable with PDF $f(x)$
$\mathrm{Re}\{.\}$	real part of complex number
ρ	correlation matrix for the Black-Scholes model
ρ^J	correlation matrix of the jump part for the MJD model
$S(t)$	price of the underlying stock at time t
$\vec{S}(t)$	vector of prices of the underlying stocks at time t
σ	volatility of a stochastic process
Σ	covariance matrix
$\overline{\sigma}$	diffusion of a stochastic process
$\vec{\sigma}$	volatility vector
t	start time of option, smaller or equal than T
T	maturity or expiry time of an option
Δt	difference between time T and time t
$v(S(t), t)$	value of an option at time t
$v(\vec{S}(t), t)$	value of a multi-dimensional option at time t
$v(S(T), T)$	value of an option at time T, the payoff of the option
$v(\vec{S}(T), T)$	value of a multi-dimensional option at time T, the payoff of the option
$V_{\vec{k}}$	DCT of payoff array P
$W(t)$	Brownian motion (Wiener process)
$\hat{x}$	value of log-asset price at initial time.
ξ^i	i-th cumulant of log-asset price
$\hat{y}$	value of log asset price at maturity time.
$X(t), Y(t)$	stochastic processes
Z	random variable with a normal distribution $\mathcal{N}(0, 1)$

Chapter 1

Introduction

Computation times for the pricing of multi-asset options increase significantly when the number of underlying assets increases. In financial markets, where time is of the essence, this increase is a huge problem. Hence, this more or less restricts the business of the parties involved. Therefore, we perform research on the basis of the COS method on how we can speed up the computations and mitigate this problem. Our research has several objectives, namely:

- Extend the COS method mathematically to higher dimensions.
- Program MATLAB-, C- and CUDA-code up to three dimensions.
- Test and compare the computation times for some multi-dimensional options.

This thesis is organized as follows: in chapter 2 we start with a short history of options and option pricing. Then we introduce some financial terminology and discuss some mathematical definitions, theorems and lemmas. Furthermore, we discuss the Black-Scholes model and the Merton's jump diffusion model. Next, we pay attention to the three major groups of numerical option pricing methods, namely Monte Carlo simulation, partial differential equations (PDE) methods [43] and Fourier-based methods [17, 33]. Finally, we focus on the order of convergence.
This thesis is based on the COS method: a Fourier-based option pricing technique. Therefore, in chapter 3 we briefly discuss the Fourier transform, Fourier series, Fourier cosine series, the discrete Fourier transform, fast Fourier transform and discrete Fourier cosine transform.
In chapter 4, we start our research with the COS method developed by Fang and Oosterlee [17]. We derive this method for one dimension, discuss the payoff coefficients, focus on the truncation range, give an error analysis and show the calculation complexity.
In chapter 5, we mathematically extend the one-dimensional (1D) COS method to n dimensions. Then, we derive the nD COS formula. As in the previous chapter, we discuss payoff coefficients, truncation range, error analysis and calculation complexity.
In chapter 6, we focus on parallel implementation. The COS method is an assembly of different functions. The most important functions of this assembly, in the sense of time consumption, are the payoff function, discrete cosine tranform (DCT), characteristic function and dot product. We have developed specific codes in the programming languages MATLAB, C and CUDA which enable us to measure the calculation time of these four functions. Thus, we can judge whether it is useful to implement these functions in a parallel way.
In chapter 7, we present and compare the results of our numerical experiments. To perform these experiments, we have developed specific codes in the environments MATLAB, C and CUDA, in order to program the COS method. Our tests show substantial time differences between these three programming languages.
Finally, chapter 8 contains the conclusions of this thesis.

Chapter 2

Preliminaries

In this chapter we give a short overview of option pricing. We pay attention to its history, present some basic terms, give definitions and theorems, discuss the famous Black-Scholes model[1], focus on different important groups of numerical methods, and explain the order of convergence. These subjects provide a base for our research.

2.1 History

Futures and options trace their roots back to antiquity. In ancient times, for example, Romans used contingent claim contracts in shipping. The ancient Greek mathematician and philosopher Thales of Miletus (624-546 BC) bought an option on olive presses in the off-season; in the season he executed his right and rented these presses profitably [35]. Future markets can be traced back to the Middle Ages. They were originally developed to meet the needs of merchants and farmers. But the first thoroughly organized future exchange was established in the 18th century in Ōsaka-shi (Dōjima Rice Exchange) [34]. In the 19th century other exchanges were founded in Chicago, Frankfurt, New York and London.

The theory of option pricing also has a long history. On March 29, 1900 the French mathematician Louis Bachelier published his thesis Théorie de la Spéculation, a pioneering analysis of the stock and option markets. Assuming that stock prices follow a Brownian motion, he drafted an option pricing formula. His thesis deeply influenced the development of stochastic calculus and mathematical finance. Courtault et al. therefore consider this date the birth of mathematical finance [11].

Since then, numerous researchers have contributed to this theory. In the 1950s, Samuelson, Osborne and others replaced Bachelier's Brownian motion with the geometric Brownian motion. However, their option pricing theories contained ad hoc elements, and even their creators felt vaguely dissatisfied [20]. In 1973, Fisher Black and Myron Scholes developed a mathematical model for the pricing of stock options which has become known as the Black-Scholes model [3]. This model is based on the possibility to create a dynamic hedge position, consisting of a long position in the stock and a short position in the option, whose value will not depend on the price of the stock. The B-S model is widely considered as a significant breakthrough in attacking the option pricing problem, because this problem was reduced to solving a partial differential equation[4, 27]; the solution has a closed-form. This form means that this model makes it possible to calculate a theoretical estimate of European-style options very rapidly.

In the same year, 1973, the Chicago Board Option Exchange (CBOE) was founded. Soon afterwards, all over the world new exchanges opened their doors. Therefore, the paper of Black and Scholes and the creation of the CBOE can be seen as the most important elements leading to a boom in the use and trading of options over the last decades.

[1] also known as Black-Scholes-Merton model because of Merton's substantial contribution[27].

The idealistic conditions and model assumptions of the B-S model limit its use. In 1976, Merton [28] pointed out that 'abnormal' variations in the price, due to the arrival of important information about the stock that has more than a marginal effect on the price, must be added to the B-S model. Merton's theory was modeled with so-called jump processes. Other authors also observed skewness and excess kurtosis in log returns of market data, i.e. these log returns deviate from the normal distribution of Brownian motion. Therefore, a more general distribution was needed. In the early 1990s, there was a great revival of interest in the theory of stochastic processes due to new developments and novel applications, particularly option pricing in mathematical finance [1]. Many authors presented new models driven by stochastic processes with the important features of Brownian motion, i.e. independent and stationary increments, and with an infinitely divisible distribution. An important class of these underlying stochastic processes is the class of extended exponential Lévy processes. Examples are the Variance Gamma (VG) model, the Normal Inverse Gaussian (NIG) model, the Generalized Hyperbolic model and the CGMY model[9].

Fourier analysis can be applied to pricing European options on assets driven by Lévy processes. A common method is to derive an integral representation using the Fourier transform. "This blends perfectly with Lévy processes, since the representation involves the characteristic function of the random variables, which is explicitly provided by the Lévy-Khintchine formula" [31, p. 38].

Despite the recent crisis in financial markets, research into fast, accurate, robust and easy to calibrate option pricing techniques is still very much alive. In 2008 and 2009, Fang and Oosterlee published their papers on their Fourier-based COS method[17, 18]. In 2011, developing their idea, Ding et al. presented three new methods (FCOS, FSIN and FSER)[15]. In 2012, Ruijter and Oosterlee extended the COS method to higher dimensions[33]. The COS method is therefore the core business of our research.

2.2 Option terminology

Before we start our research we introduce some financial terminology that will be used in this thesis.

In the last decades derivatives have become increasingly important in financial markets. A derivative can be described as a financial instrument whose value is derived from the value of one or several underlying assets, such as stocks, bonds and commodities. Well-known derivatives are futures and options. A future contract is an agreement to buy or sell an asset at a certain time in the future for a certain price. An option contract is a similar agreement. The main difference between these contracts is that the holder of an option does not have to exercise his right.

In this thesis we focus on stock options. A stock option is a derivative whose value is dependent on the price of a stock (a share in a company). There are two basic types of options: call and put options. A *call* option gives the holder of the option the right, but not the obligation, to buy a stock by a certain date for a certain price. A *put* option gives the holder of the option the right, but not the obligation, to sell a stock at a certain date for a certain price. The date specified in the contract is known as the expiration date or the maturity date. The price specified in the contract is known as the exercise price or the strike price [21].
With an American option, exercise can take place at any time during the life of the option and the exercise price is fixed. For a European option, exercise can only take place at the expiration date and the exercise price is also fixed. When early exercise may be restricted to certain dates the option is known as a Bermudan option. These options have standard, well-defined properties and are traded actively. Their prices are quoted by exchanges or by brokers on a regular basis.
Besides standard options ("vanillas") many nonstandard options exist. These options with more complex features, created by financial engineers for a number of reasons, are termed exotic products [21]. These options are traded over-the-counter, a market where traders are usually financial institutions, corporations and fund managers. The OTC market is an important alternative to regular option exchanges. The prices of these "exotics" are not quoted.
A standard call or put option is written on one underlying risky stock. We call this derivative a one-dimensional option. But it is also possible to write an option on two or more underlying risky stocks. We call these contracts multi-dimensional options ("rainbows" [21]).

The payoff of a European one-dimensional call option is based on the difference between the price of the stock $S(t)$ at maturity time T and a fixed strike price K. The current time is $t = 0$. The holder of (a long position in) a European call option will exercise his option when $S(T)$ exceeds K, i.e. $S(T) - K \geq 0$; in this case the option is in-the-money. When, at expiry time T, $S(T) - K \leq 0$, the option is worthless. The holder will not execute his option right (see figure 2.1 left side), the option is out-of-the-money and its value equals zero. When $S(T) = K$ the option is at-the-money.

The payoff of (a long position in) a European call option $v(S(t), t)$ at time T is thus

$$g_{call}(S(T)) = (S(T) - K)^+ = \max(S(T) - K, 0).$$

The holder of (a long position in) a European put option will exercise his option when $S(T)$ is less than K i.e. $K - S(T) \geq 0$. When, at expiry time T, $K - S(T) \leq 0$, the option is worthless and the holder will not execute his option right (see figure 2.1 right side).

The payoff of (a long position in) a European put option $v(t, S(t)$ at time T is thus

$$g_{put}(S(T)) = (K - S(T))^+ = \max(K - S(T), 0).$$

A put and a call option are related when they have the same properties. The put-call parity specifies a relationship of calls and puts with the identical strike price K and the expiry time T. A very important feature is that this parity is model-independent and applies at all times. The put-call parity is given by

$$v_{call}(S(t), t) = v_{put}(S(t), t) + S(t) - Ke^{-r(T-t)}$$

where r is the risk-free interest rate.

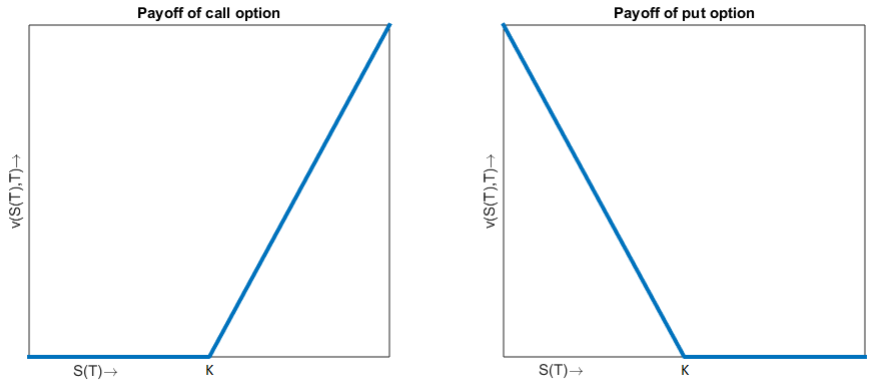

Figure 2.1: Payoff functions of long call (left) and long put (right)

2.3 Definitions

This section contains the definitions which are important for our thesis. Note that we often use descriptions of authors such as Shreve[36] and Applebaum[1, 2].

Definition 1. Stochastic process
For the definition of a stochastic process we quote Applebaum [1, p. 1336]: "... a stochastic process is a family of random variables

$$\{X(t), t \geq 0\}$$

defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and taking values in a measurable space $(E, \mathcal{E})$. Here Ω is a set (the sample space of possible outcomes), $\mathcal{F}$ is a σ-algebra of subsets of Ω (the events), and $\mathbb{P}$ is a positive measure of total mass 1 on $(\Omega, \mathcal{F})$ (the probability). E is sometimes called the state space. Each $X(t)$ is an $(\mathcal{F}, \mathcal{E})$ measurable mapping from Ω to E and should be thought of as a random observation made on E at time t."

Definition 2. Brownian motion
For the definition of a Brownian motion we quote Shreve [36, p. 94]: "Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space. For each $\omega \in \Omega$, suppose there is a continuous function $W(t)$ of $t \geq 0$, that satisfies $W(0) = 0$ and that depends on ω. Then $W(t)$, $t \geq 0$, is a Brownian motion if for all

$$0 = t_0 < t_1 < t_2 < \ldots < t_m$$

the increments

$$W(t_1), W(t_2) - W(t_1), \ldots, W(t_m) - W(t_{m-1})$$

are independent and each of these increments is normally distributed with

$$\mathbb{E}[W(t_{i+1}) - W(t_i)] = 0$$
$$\mathrm{Var}(W(t_{i+1}) - W(t_i)) = t_{i+1} - t_i."$$

Definition 3. Stochastic Differential Equation
For the definition of a stochastic differential equation we also quote Shreve [36, p. 264]: "A stochastic differential equation is an equation of the form

$$dX(u) = \overline{\mu}(X(u), u)du + \overline{\sigma}(X(u), u)dW(u). \tag{2.1}$$

Here $\overline{\mu}(u, x)$ and $\overline{\sigma}(u, x)$ are given functions, called the *drift* and *diffusion*, respectively. In addition to this equation, an *initial condition* of the form $X(t) = x$, where $t \geq 0$ and $x \in \mathbb{R}$, is specified."
The SDE (2.1) is short-hand notation for

$$X(t) = x,$$
$$X(T) = X(t) + \int_t^T \overline{\mu}(u, X(u))du + \int_t^T \overline{\sigma}(u, X(u))dW(u).$$

Solving a SDE is finding a stochastic process which satisfies both equations.

Definition 4. Lévy process
Applebaum defines the Lévy process as follows [1, p. 1337]: "A Lévy process $X = \{X(t), t \geq 0\}$ is a stochastic process satisfying the following axioms:

- X has independent and stationary increments,
- Each $X(0) = 0$ (with probability 1),
- X is stochastically continuous i.e. for all $a > 0$ and for all $s \geq 0$, $\lim_{t \to s} \mathbb{P}(|X(t) - X(s)| > a) = 0$.

The first axiom is the most important. Independent means that given any finite ordered sequence of times

$$0 \leq t_1 < t_2 < \ldots < t_m < \infty,$$

the increments

$$X(t_1), X(t_2) - X(t_1), \ldots, X(t_m) - X(t_{m-1}),$$

are independent. Stationary means that for any $0 \leq s < t < \infty$, $X(t) - X(s)$ has the same distribution as $X(t - s)$."

Definition 5. Characteristic function of a Lévy process
The characteristic function of a Lévy process $X(t)$ is given by:

$$\phi_t(\omega) = \mathbb{E}[\exp(i\omega X(t))]$$

where $\mathbb{E}[.]$ is the expectation.

Formula 1. Lévy-Khintchine formula
The characteristic function of a Lévy process can be written with its triple (μ, σ, ν) with $\mu \in \mathbb{R}$ the drift, $\sigma \geq 0$ the Gaussian variance and ν the Lévy density

$$\phi_t(u) = \exp\left(t\left[i\mu u - \frac{1}{2}\sigma^2 u^2 + \int_{\mathbb{R}} (e^{iux} - 1 - iux1_{\{|x|<1\}}\nu(dx))\right]\right), \tag{2.2}$$

where the Lévy density statifies

- $\nu(0) = 0$,
- $\int_{\mathbb{R}} \min(1, |x|^2)\nu(dx) < \infty$.

For Formula (2.2) we follow [16, p. 3].

Definition 6. Exponential Lévy process
Let $X(t)$ be a Lévy process, then an exponential Lévy process $Y(t)$ is given by

$$Y(t) = Y(0)\exp(X(t)), \quad t \geq 0. \tag{2.3}$$

Definition 7. Geometric Brownian motion
A geometric Brownian motion process $S(t)$ satisfies the following SDE [36, p. 264]

$$dS(t) = \mu S(t)dt + \sigma S(t)dW(t) \quad \text{with} \quad S(0) = S_0, \tag{2.4}$$

where $\mu, \sigma \in \mathbb{R}$, $\mu S(t)$ is the drift and $\sigma S(t)$ is the diffusion. With Itô's lemma it can be proven that GBM is an exponential Lévy process and the underlying Lévy process $X(t)$ is a Brownian motion with drift.

Lemma 1. Itô's lemma
Let S(t), $t \geq 0$, be a stochastic process

$$dS(t) = \overline{\mu}(S(t), t)dt + \overline{\sigma}(S(t), t)dW(t), \quad \text{with} \quad S(0) = s, \tag{2.5}$$

and let $h(S(t), t)$ be a function for which the partial derivatives, $\frac{\partial h}{\partial t}(S(t), t)$, $\frac{\partial h}{\partial S(t)}(S(t), t)$ and $\frac{\partial^2 h}{\partial S^2(t)}(S(t), t)$, are defined and continuous. Then, for every $t \geq 0$ holds

$$dh(S(t), t) = \frac{\partial h}{\partial t}(S(t), t)dt + \frac{\partial h}{\partial S(t)}(S(t), t)dS(t) + \frac{1}{2}\frac{\partial^2 h}{\partial S^2(t)}(S(t), t)dS(t)dS(t). \tag{2.6}$$

Substitute $dS(t)dS(t) = (\overline{\mu}dt + \overline{\sigma}dW(t))(\overline{\mu}dt + \overline{\sigma}dW(t)) \approx \overline{\sigma}^2 dt$ then

$$dh(S(t), t) = \frac{\partial h}{\partial t}(S(t), t)dt + \frac{\partial h}{\partial S(t)}(S(t), t)dS(t) + \frac{1}{2}\overline{\sigma}^2(S(t), t)\frac{\partial^2 h}{\partial S^2(t)}(S(t), t)dt, \tag{2.7}$$

which can be written in short-hand notation as

$$dh = \frac{\partial h}{\partial t}dt + \frac{\partial h}{\partial S}dS(t) + \frac{1}{2}\overline{\sigma}^2\frac{\partial^2 h}{\partial S^2}dt. \tag{2.8}$$

For further explanation we refer to Shreve [36, p. 147].

Example 1. Geometric Brownian motion
If $S(t)$ in (2.5) satisfies the GBM with drift $\overline{\mu} = rS(t)$, volatility $\overline{\sigma} = \sigma S(t)$ and $h(S(t), t) = \log(S(t))$ then $\frac{\partial h}{\partial t} = 0$, $\frac{\partial h}{\partial S} = 1/S(t)$ and $\frac{\partial^2 h}{\partial S^2} = -1/S^2(t)$. In combination with (2.8) this gives a Brownian motion with drift

$$\begin{aligned} d\log(S(t)) &= \frac{1}{S(t)}dS(t) - \frac{1}{2}\frac{\sigma^2 S^2(t)}{S^2(t)}dt \\ &= \left(r - \frac{1}{2}\sigma^2\right)dt + \sigma dW(t), \end{aligned} \tag{2.9}$$

where $(r - \frac{1}{2}\sigma^2)$ is the drift and σ is the volatility. Define $X(t) = \log(S(t))$ and $x = X(0) = \log(S(0))$. From (2.9) it follows that $X(t)$ is normally distributed with mean $x + (r - \frac{1}{2}\sigma^2)t$ and variance $\sigma^2 t$. As a result, $S(t)$ is log-normally distributed. For more details we refer to Oosterlee [30].

Definition 8. Characteristic function 1D GBM
Define random variable $Z \sim N(0, 1)$. The characteristic function of random variable Z reads

$$\phi(\omega) = \mathbb{E}[e^{iwZ}] = e^{-0.5\omega^2}.$$

Proof: see appendix A.
To calculate the characteristic function of a random variable $R \sim N(\mu, \sigma^2)$ note that $R = \mu + \sigma Z$ where $Z \sim N(0, 1)$. The characteristic function of R reads

$$\begin{aligned}\phi(\omega) &= \mathbb{E}[e^{i\omega R}] = \mathbb{E}[e^{i\omega(\mu+\sigma Z)}] = e^{i\mu\omega}\mathbb{E}[e^{iw(\sigma Z)}] \\ &= \exp\left(i\mu\omega - \frac{1}{2}\sigma^2\omega^2\right).\end{aligned}$$

At time t, if $x = 0$ then $X(t)$ is normally distributed with mean $(r - \frac{1}{2}\sigma^2)t$ and standard deviation $\sigma\sqrt{t}$. The characteristic function of $X(t)$ is:

$$\begin{aligned}\phi_t(\omega) &= \mathbb{E}[e^{i\omega X(t)}] \\ &= \exp\left(t\left[i(r - \frac{1}{2}\sigma^2)\omega - \frac{1}{2}\sigma^2\omega^2\right]\right).\end{aligned} \tag{2.10}$$

If $x \neq 0$ then the stochastic process $X(t)$ have mean $x + (r - \frac{1}{2}\sigma^2)t$ and standard deviation $\sigma\sqrt{t}$. Then, the characteristic function reads

$$\begin{aligned}\phi_t(\omega|x) &= \mathbb{E}[e^{i\omega X(t)}] \\ &= \exp(ix\omega)\exp\left(t\left[i(r - \frac{1}{2}\sigma^2)\omega - \frac{1}{2}\sigma^2\omega^2\right]\right).\end{aligned}$$

Definition 9. Characteristic function nD GBM
For pricing options by the nD COS method under nD geometric Brownian motion we need the nD characteristic function. This function reads, see [33]:

$$\phi_t(\vec{\omega}|\vec{x}) = \exp(i\vec{x}'\vec{\omega})\exp\left(t\left[i\vec{\mu}'\vec{\omega} - \frac{1}{2}\vec{\omega}'\Sigma\vec{\omega}\right]\right), \tag{2.11}$$

$$\Sigma_{ij} = \sigma_i\sigma_j\rho_{ij}, \tag{2.12}$$

$$\mu_i = r - \frac{1}{2}\sigma_i^2, \tag{2.13}$$

where r is the instantaneous risk-free interest rate, $\vec{\mu}$ is the drift vector, Σ the covariance matrix, $\vec{\sigma}$ the volatility vector, ρ the correlation matrix and $\vec{x}$ the initial condition vector.

Definition 10. Arbitrage
We quote Shreve [36, p. 230]: "An arbitrage is a portfolio value process $X(t)$ satisfying $X(0) = 0$ and also satisfying for some time $T > 0$

$$\mathbb{P}(X(T) \geq 0) = 1, \quad \mathbb{P}(X(T) > 0) > 0."$$

Theorem 1. First fundamental theorem of asset pricing
We quote Shreve [36, p. 231]: "If a market model has a risk-neutral probability measure it does not admit arbitrage."
Proof. For a proof we refer to Shreve [36].

Definition 11. Complete market model
We quote Shreve [36, p. 231]: "A market model is complete if every derivative security can be hedged." (I.e. for every derivative one can make a replicating portfolio with other financial products which has the same value).

Theorem 2. Second fundamental theorem of asset pricing
We quote Shreve [36, p. 232]: "Consider a market model that has a risk-neutral probability measure. The model is complete if and only if the risk-neutral probability measure is unique."
Proof. For a proof we refer to Shreve [36].

Definition 12. Martingale
We quote Shreve [36, p. 74]: "Let $(\Omega, F, \mathbb{P})$ be a probability space, let T be a fixed positive number, and let $F(t)$, $0 \geq t \geq T$, be a filtration of sub-σ-algebras of F. Consider an adapted stochastic process $M(t)$, $0 \geq t \geq T$. If

$$\mathbb{E}[M(t)|F(s)] = M(s) \quad \text{for all } 0 \geq s \geq t \geq T,$$

we say this process is a martingale. It has no tendency to rise or fall."

Definition 13. Money-savings account
Let $B(t)$ be the value of a bank account at time $t \geq 0$. Assume $B(0) = 1$, an initial investment of 1, and assume that the bank account evolves according to

$$dB(t) = r(t)B(t)dt, \quad B(0) = 1,$$

where r(t) is the risk-free interest rate. Then it follows that

$$B(t) = e^{\int_0^t r(s)ds}$$

An initial investment of $B(0)$ yields in the money-savings market to a value of $B(t)$ at time t.
We define the discount term by

$$D(t,T) = \frac{B(t)}{B(T)} = e^{-\int_t^T r(s)ds} \quad t \leq T.$$

If $r(t)$ is constant, then the discount term becomes

$$D(t,T) = e^{-r(T-t)} \quad t \leq T.$$

Definition 14. Risk-neutral probability measure
We quote Shreve [36, p. 228]: "A probability measure $\tilde{\mathbb{P}}$ is said to be risk-neutral if

1. $\tilde{\mathbb{P}}$ and $\mathbb{P}$ are equivalent (i.e. for every $A \in F$, $\mathbb{P}(A) = 0$ if and only if $\tilde{\mathbb{P}}(A) = 0$), and
2. under $\tilde{\mathbb{P}}$, the discounted stock price $D(0,t)S_i(t)$ is a martingale for every $i = 1, \ldots, n$."

Lemma 2. Risk neutral valuation formula
The value of an option $v(S(t), t)$, where t is time and $S(t)$ is the underlying stock price, can be written as the discounted expectation of its payoff under the risk neutral measure $\mathbb{Q}$ [36, p.218].

$$v(S(t), t) = \mathbb{E}^{\mathbb{Q}}[e^{-\int_t^T r(s)ds} v(S(T), T)|\mathcal{F}(t)], \quad 0 \leq t \leq T \tag{2.14}$$

where $\mathcal{F}(t)$ is the filtration at time t.

2.4 Black-Scholes model

Black and Scholes derived a linear parabolic partial differential equation to calculate the value of European options. For an explanation of its derivation we follow Wilmott [42]. Assume that the underlying stock price is modeled as a GBM. Use Π to denote the value of a portfolio of one long option position and a short position in some quantity Δ, delta, of the underlying stock,

$$\Pi = v(S(t), t) - \Delta S(t). \tag{2.15}$$

The change of the portfolio value over time is given by

$$d\Pi = dv(S(t), t) - \Delta dS(t).$$

The differential of v can be derived from Itô's lemma (2.8), written as

$$dv = \frac{\partial v}{\partial t}dt + \frac{\partial v}{\partial S}dS(t) + \frac{1}{2}\frac{\partial^2 v}{\partial S^2}dS(t)dS(t).$$

If $S(t)$ is modeled as a GBM model, then the portfolio with deterministic terms dt and stochastic terms $dS(t)$ changes as follows

$$d\Pi = \frac{\partial v}{\partial t}dt + \frac{\partial v}{\partial S}dS(t) + \frac{1}{2}\sigma^2 S^2(t)\frac{\partial^2 v}{\partial S^2}dt - \Delta dS(t). \tag{2.16}$$

The risk ($dS(t)$ terms) in this portfolio can be eliminated by delta-hedging. If

$$\Delta = \frac{\partial v}{\partial S} \tag{2.17}$$

and substituting (2.17) into (2.16) we find that (2.16) becomes deterministic

$$d\Pi = \frac{\partial v}{\partial t}dt + \frac{1}{2}\sigma^2 S^2(t)\frac{\partial^2 v}{\partial S^2}dt. \tag{2.18}$$

The value of the portfolio does not depend on the value of the stock price any longer. According to the no-arbitrage principle, a portfolio without risk increases in value with the risk-free interest rate. A risk-free portfolio is governed by the following dynamics

$$d\Pi = r\Pi dt. \tag{2.19}$$

Substituting (2.15), (2.17) and (2.18) into (2.19) we find that

$$\frac{\partial v}{\partial t}dt + \frac{1}{2}\sigma^2 S^2(t)\frac{\partial^2 v}{\partial S^2}dt = r\left(v - S(t)\frac{\partial v}{\partial S}\right)dt.$$

Dividing by dt and rearranging, we get the Black-Scholes PDE

$$\boxed{\frac{\partial v}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 v}{\partial S^2} + rS\frac{\partial v}{\partial S} - rv = 0.} \tag{2.20}$$

The payoff is the value of an option at expiry time T, i.e. the final condition of the Black-Scholes PDE.

Theorem 3. (Feynman-Kac Theorem)
The Feynman-Kac theorem is known in different versions. We use the version of Oosterlee et al. [30]. Given a money-savings account, modeled by $dB(t) = rB(t)dt$, with a constant risk-free interest rate r. Let $v(S(t), t)$ be a sufficiently differentiable function of stock price $S(t)$ and time t. Suppose that $v(S(t), t)$ satisfies the following PDE, with general drift term, $\overline{\mu}(S(t), t)$, and volatility term, $\overline{\sigma}(S(t), t)$:

$$\frac{\partial v}{\partial t} + \frac{1}{2}\overline{\sigma}^2(S(t), t)\frac{\partial^2 v}{\partial S^2} + \overline{\mu}\,(S(t), t)\,\frac{\partial v}{\partial S} - rv(S(t), t) = 0, \quad \text{(PDE)} \tag{2.21}$$

with final condition $v(S(T),T)$ given.
The unique solution for $v(S(t),t)$ at time $t<T$ reads

$$v(S(t),t)=e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}[v(S(T),T)|F(t)],\quad \text{(expectation)} \tag{2.22}$$

where the expectation is taken under the measure $\mathbb{Q}$, with respect to a process $S(t)$, defined by:

$$dS(t)=\overline{\mu}(S(t),t)dt+\overline{\sigma}(S(t),t)dW^{\mathbb{Q}}(t),\quad \text{for}\quad t>0\quad \text{(SDE)} \tag{2.23}$$

Proof. For a short outline of a proof we also refer to Oosterlee [30].

The PDE (2.21) in the Feynman-Kac theorem becomes the Black-Scholes PDE when we substitute for $\overline{\sigma}$ the term σS and for $\overline{\mu}$ the term rS. By using the Feynman-Kac theorem we can determine the value of the option by solving the expectation (2.22).

The dynamics of the underlying asset price under the risk-neutral measure are given by

$$dS(t)=rS(t)dt+\sigma S(t)dW^{\mathbb{Q}}(t).$$

The interest rate r is fixed and the underlying stock price is a GBM process. Therefore, the option value can be written as

$$v(S(t),t)=e^{-r(T-t)}\int_{-\infty}^{\infty}v(S(T),T)f\left(S(T)|S\left(t\right)\right)dt$$

where $v(S(T),T)$ is the payoff function and $f(S(T)|S(t))$ the probability density function of $S(T)$ given $S(t)$. From example 1. it is known that $\log(S(t))$ is normally distributed. This means that the PDF is known. With the help of changing variables the integral can be solved analytically.

Lemma 3. Black-Scholes Formula
The value of a European call option $v(S(t),t)$ under GBM asset dynamics at time t is given by

$$v(S(t),t)=S(t)\hat{N}(d_1)-Ke^{-r(T-t)}\hat{N}(d_2), \tag{2.24}$$

with

$$d_1=\frac{\log\left(\frac{S(t)}{K}\right)+\left(r+\frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}},\quad d_2=\frac{\log\left(\frac{S(t)}{K}\right)+\left(r-\frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}},$$

where $S(t)$ is the value of the asset at time t, N is the cumulative normal density function, K is the strike price, r is the risk-free interest rate, $T-t$ is the time to maturity and σ is the volatility of the asset. The function N is known as

$$\hat{N}(x)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x}e^{-\frac{1}{2}z^2}dz. \tag{2.25}$$

Proof. For a proof of the B-S Formula (2.24) we refer to the appendix of chapter 14 of Hull [21].

From the put-call parity it follows that the value of a European put option $v(S(t),t)$ under GBM at time t is given by

$$v(S(t),t)=Ke^{-r(T-t)}\hat{N}(-d_2)-S(t)\hat{N}(-d_1) \tag{2.26}$$

2.5 Merton's Jump Diffusion model

In 1976, based on Black-Scholes, Merton [28] developed a jump diffusion model (MJD model). In the MJD model 'normal' vibrations in price are modeled by a GBM and 'abnormal' vibrations (jumps) in

price, as a result of the arrival of important new information about the stock, are modeled as a compound Poisson process. These two stochastic processes are assumed to be independent.
We follow Tankov and Voltchkova [40]. A stochastic jump diffusion process can be written as follows

$$X(t) = \mu t + \sigma W(t) + \sum_{i=1}^{N(t)} Y_i, \tag{2.27}$$

where $\mu t + \sigma W(t)$ is a stochastic process with drift and diffusion, $N(t)$ is a Poisson process with mean arrival rate λ distributed with an exponential distribution and the Y_i are i.i.d. random variables with distribution $f(x)$.

The characteristic function of (2.27) reads

$$\mathbb{E}[\exp(iuX(t))] = \exp\left(t\left[i\mu u - \frac{\sigma^2 u^2}{2} + \lambda \int_{\mathbb{R}} (e^{iux} - 1) f(dx)\right]\right).$$

If there is no positive probability of immediate ruin then the jumps (Y_i) in MJD are modeled as a normal distribution $\mathcal{N}(\alpha, \delta^2)$. Then, this characteristic function reads

$$\mathbb{E}[\exp(iuX(t))] = \exp\left(t\left[i\mu u - \frac{\sigma^2 u^2}{2} + \lambda\left\{\exp\left(i\alpha u - \frac{\delta^2 u^2}{2}\right) - 1\right\}\right]\right). \tag{2.28}$$

The exponential Lévy process of the asset is defined as

$$\begin{aligned} S(t) &= S(0)\exp\left(X(t)\right) \\ &= S(0)\exp\left(\mu t + \sigma W(t) + \sum_{i=1}^{N(t)} Y_i\right) \end{aligned}$$

The SDE for the asset price reads

$$\frac{dS(t)}{S(t)} = (\mu - \lambda\kappa)dt + \sigma dW(t) + (e^Y - 1)dN(t), \tag{2.29}$$

where $\kappa = \mathbb{E}[e^Y - 1]$. The average jump size measured as a percentage of the asset price. Let $\log(S(t))$ to be a martingale under the risk-neutral measure then the total drift is $(r - \frac{\sigma^2}{2} - \lambda\kappa)$.
Merton gives an analytical formula for a European 1D call option where the underlying asset is modeled as SDE (2.29). This analytical formula is called the MJD formula and reads

$$v(S(t), t) = \sum_{n=0}^{\infty} \frac{e^{-\lambda'\Delta t}(\lambda'\Delta t)^n}{n!} f_n, \tag{2.30}$$

where $\lambda' = \lambda(1+\kappa)$, f_n are the B-S option values given the number of jumps is n hence $\sigma_n^2 = \sigma^2 + n\delta^2/\Delta t$ and $r_n = r - \lambda\kappa + n\log(1+\kappa)/\Delta t$.

nD MJD model

Let $\vec{S}(t)$ be the stock prices at time t. The stock prices $\vec{S}(t)$ are modeled by the following SDE

$$dS_i(t) = (r - \lambda\kappa_i)S_i(t)dt + \sigma_i S_i(t)dW_i(t) + (e^{Y_i} - 1)S_i(t)dN(t), \quad i = 1, \ldots, n \tag{2.31}$$

where r is the instantaneous risk-free interest rate, λ is the mean number of arrivals per unit time, $\kappa_i := \mathbb{E}[e^{Y_i} - 1]$, e^{Y_i} is the random variable giving the percentage left after a jump has occurred, $N(t)$ is a Poisson process with mean arrival rate $\lambda > 0$ and Y_i are normally distributed jumps with mean α and standard deviation δ. $W_i(t)$ and $N(t)$ are independent processes [33, p. B658].

The log-processes $X_i(t) := \log(S_i(t))$ reads

$$dX_i(t) = \left(r - \frac{1}{2}\sigma_i^2 - \lambda\kappa_m\right)dt + \sigma_i dW_i(t) + Y_i dN(t).$$

For the derivation of (2.31) see [30, p. 73].

Definition 15. Characteristic function nD MJD
The characteristic function reads [33, p. B658]

$$\phi(\vec{\omega}|\vec{x}) = \exp(i\vec{x}'\vec{\omega})\phi_{Merton}(\vec{\omega}),$$

where

$$\phi_{Merton}(\vec{\omega}) = \exp\left(t\left[i\vec{\mu}'\vec{\omega} - \frac{1}{2}\vec{\omega}'\Sigma\vec{\omega} + \lambda\left\{\exp\left(i\vec{\alpha}'\vec{\omega} - \frac{1}{2}\vec{\omega}'\Sigma^J\vec{\omega}\right) - 1\right\}\right]\right),$$

and

$$\mu_i = \left(r - \frac{1}{2}\sigma_i^2 - \lambda\kappa_i\right), \quad \Sigma_{ij} = \sigma_i\sigma_j\rho_{ij}, \quad \Sigma_{ij}^J = \delta_i\delta_j\rho_{ij}^J,$$

where ρ is the correlation matrix of the diffusion process and ρ^J the correlation matrix of the jump part.

2.6 Numerical methods

The PDE of the classic Black-Scholes model can thus be solved analytically (closed-form solutions). However, for more complex PDEs this analytical technique is not available. Therefore, researchers have developed other valuation techniques. An import class among other techniques is that of the numerical methods. These methods can be roughly divided into three major groups.

The first group is Monte Carlo simulation. Hull [21, p. 804] describes this simulation as a procedure for randomly sampling changes in market variables in order to value a derivative. The use of this simulation technique is attractive, because it is reliable, the mathematics can be very basic[2], better accuracy can easily be achieved by running more simulations, the simulations can often be changed without much work and the curse of dimensionality does not occur. However, this method has a great disadvantage: it takes a lot of computing time. According to Wilmott [42, p. 685] it is slow in comparison with the finite-difference solution of a PDE for problems up to three dimensions. That is why this simulation technique is often used to price high-dimensional options; sometimes it is the only practical one.
This technique can also be used for the pricing of European-style options, not only for one-dimensional options, but for higher-dimensional options as well.

The second group includes PDE methods. Finizio [19, p. 421] describes PDEs as equations that involve partial derivatives of an unknown function, where the unknown function depends on two or more independent variables. Famous examples are the wave equation and the heat equation. The wave equation is written as:

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2},$$

where $u(x,t)$ is the vertical position at location x on the string at time t, and the constant c relates to the tension in the string and how springy it is [37]. This formula of D'Alembert (1746) involves (second order) derivatives of u. Since these are partial derivatives, it is a PDE.

The heat equation is written as:

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2},$$

[2] $v(S(t),t) = \mathbb{E}^{\mathbb{Q}}[e^{-\int_t^T r(s)ds}v(S(T),T)|\mathcal{F}(t)] \approx e^{-r(T-t)}\frac{1}{N}\sum_{i=1}^N v(S^i(T),T)$, Surkov [39, p. 13]

where $u(x,t)$ is the temperature of a metal rod at position x and time t, considering the rod to be infinitely thin, and the constant α is the diffusivity [37, p. 151]. Since this equation also involves partial derivatives of u, it is a PDE.

Various numerical methods for solving PDEs have been developed, such as the finite difference method (FDM), the finite element method (FEM), the method of lines (MOL) and the finite volume method (FVM). However, according to Wilmott, although FDMs form the dominant approach to numerical solutions of PDEs, they start to become slow and cumbersome at higher (three and more) dimensions [42, p. 661].

The third group is formed by numerical integration techniques, i.e. the study of how to find the numerical value of an integral [12]. In this study the process of approximating an integral from values of the integrand is of central interest. In this process integration points and the weighted values of these points are important.

These techniques often rely on the characteristic function, i.e. the Fourier transform of the probability density function of the underlying asset [6, 17, 33]. For speeding up their computational time an efficient algorithm which is called the fast Fourier transform is often used. A relatively new Fourier-based technique is the COS method, developed by Fang and Oosterlee [17]. This technique is based on the idea to replace the PDF of the underlying stock price with its Fourier-cosine series expansion [16]. They showed that their method is highly efficient.

2.7 Orders of convergence

For a competitive method for pricing of financial derivatives accuracy, speed and convergence play an important role. The cohesion between the approximation error and the number of integrand evaluations is called convergence. A method is usually considered superior when the error becomes smaller at a faster rate when taking more integrand evaluations.

Boyd [5, p. 25] notes that it is useful to have precise definitions for classifying the rates of convergence for series. These rates are all asymptotic definitions based on the behavior of the series coefficients for large m. He warns that these definitions may be highly misleading if applied to small or moderate m. For our research several of these definitions are important.

Definition 16. Algebraic index of convergence
Boyd [5, p. 25] defines the algebraic index of convergence, aic, as the largest number for which

$$\lim_{m\to\infty} |u_m| m^{aic} < \infty, \quad m >> 1,$$

where u_m are the coefficients of the series.
Boyd also gives an alternative definition: if the coefficients of a series are u_m and if

$$u_m \sim \mathcal{O}(1/m^{aic}), \quad m >> 1,$$

then aic is the algebraic index of convergence, u_m decays asymptotically.

Definition 17. Exponential convergence
We follow Boyd [5, p. 25]. The coefficients of the series are said to have the property of exponential convergence if the algebraic index of convergence aic is unbounded - in other words if the coefficients u_m decrease faster than $1/m^{aic}$ for any finite power of aic.
Boyd also gives an alternative definition: if

$$u_m \sim O(\exp(-\overline{q} \cdot m^{eic})), \quad m >> 1,$$

with $\overline{q}$ a constant for some $eic > 0$, then the series has exponential convergence.

Definition 18. Exponential index of convergence
Boyd [5, p. 26] defines the exponential index of convergence *eic* by:

$$eic = \lim_{m\to\infty} \frac{\log|\log|u_m||}{\log(m)}.$$

Definition 19. Rates of Exponential Convergence
We quote Boyd [5, p. 26]. "A series, whose coefficients are u_m, is said to have the property of supergeometric, geometric or subgeometric convergence depending upon whether

$$\lim_{m\to\infty} \log(|u_m|)/m = \begin{cases} \infty & \text{supergeometric} \\ constant & \text{geometric} \\ 0 & \text{subgeometric."} \end{cases}$$

Boyd clarifies these concepts of convergence by way of two graphs.

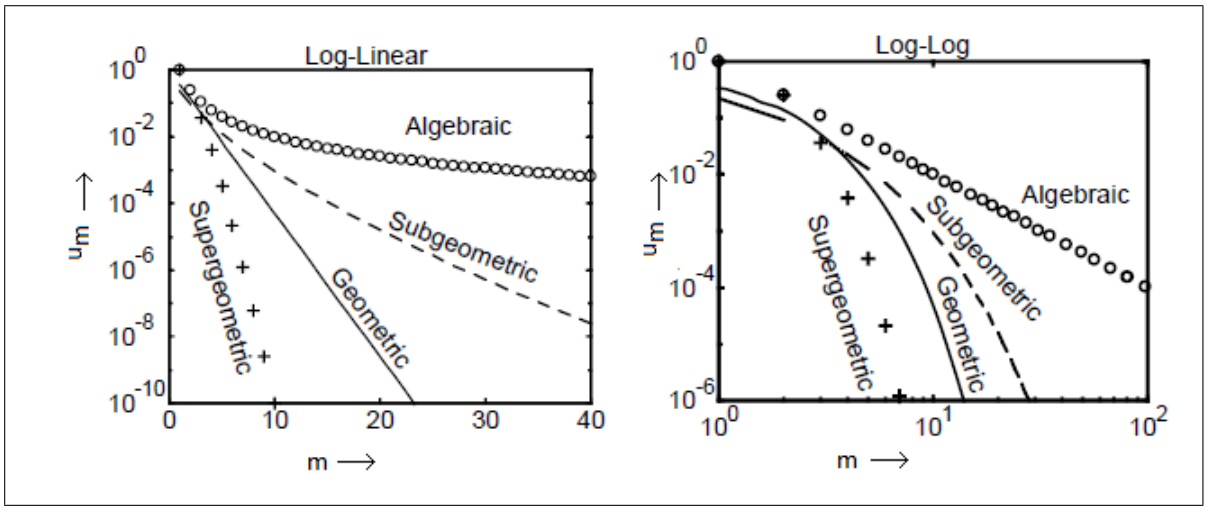

Figure 2.2: Four different convergences [5, p. 27-28]

We shall see that the overall error of the one-dimensional COS method has an exponential convergence and that the overall error of the n-dimensional COS method ($n \geq 2$) has an algebraic convergence.

Chapter 3

Fourier Transform

An asset price process with independent and stationary increments can be identified by its characteristic function. Therefore, in the 1990s, the Fourier transform, a widely used and well understood mathematical tool from physics and engineering disciplines, was introduced into mathematical finance.

In this chapter we discuss the Fourier transform. Firstly, we give some definitions and properties. Secondly we show the path from Fourier series to Fourier cosine series. Thirdly, we connect the characteristic function with the Fourier cosine series. Finally, we focus on different discrete transforms in particular higher dimensional discrete cosine transforms and fast cosine transforms.

3.1 Definitions and properties

For the definition of the Fourier transform and the inverse Fourier transform we follow Deng [14].

Definition 20. Fourier transform and inverse Fourier transform
Let $p(x)$ be a piecewise continuous real function over $\mathbb{R}$ which satisfies the integrability condition

$$\int_{-\infty}^{\infty} |p(x)|dx < \infty.$$

The Fourier transform of $p(x)$ is defined by

$$\hat{p}(\omega) = \int_{-\infty}^{\infty} e^{i\omega x} p(x)dx, \quad \omega \in \mathbb{R}, \tag{3.1}$$

where $i = \sqrt{-1}$ is the imaginary unit. The inverse Fourier transform of $\hat{p}(\omega)$ is given by

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-i\omega x} \hat{p}(\omega)d\omega, \quad x \in \mathbb{R}. \tag{3.2}$$

Definition 21. Characteristic function of a random variable
Let $\hat{R}$ be a random variable having PDF $f(x)$. $f(x)$ is a non-negative function and satisfies the integrability condition

$$\int_{-\infty}^{\infty} f(x)dx = 1 < \infty.$$

The Fourier transform of $f(x)$, i.e. the ChF, is written as $\phi(\omega)$ and is defined by

$$\phi(\omega) = \mathbb{E}[e^{i\omega \hat{R}}] = \int_{-\infty}^{\infty} e^{i\omega x} f(x)dx, \quad \omega \in \mathbb{R}. \tag{3.3}$$

The PDF is the inverse Fourier transform of the ChF. The ChF and the PDF thus form a Fourier pair.

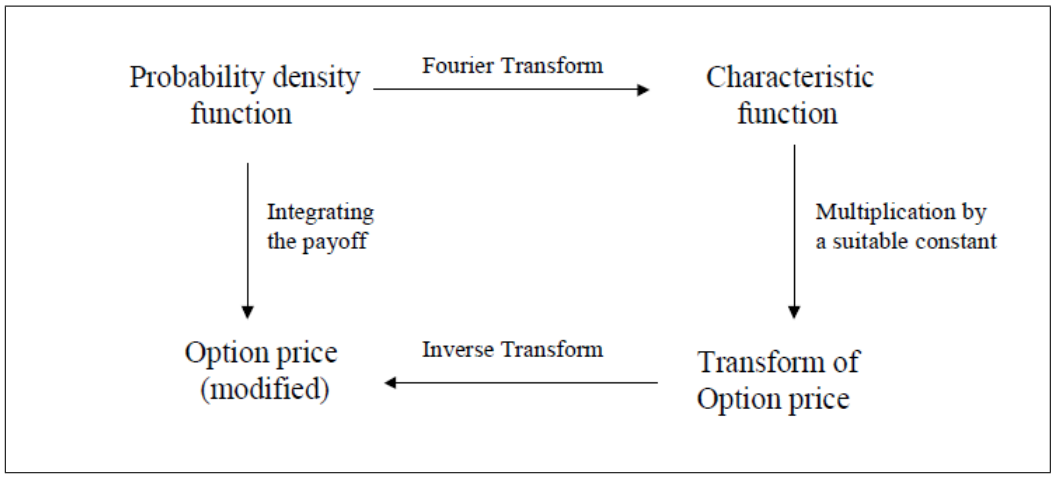

Figure 3.1: Fourier transform scheme.

The Fourier transform technique in option pricing is displayed schematically in Figure 3.1 [13].

Properties of characteristic functions
Characteristic functions possess several properties, see [14]. We only mention:

- if $a, b \in \mathbb{R}$ and Zv, Rv are random variables with $Rv = a \cdot Zv + b$ then

$$\phi_{Rv}(\omega) = e^{ib\omega}\phi_{Zv}(a\omega) \tag{3.4}$$

  where ϕ_{Zv}, ϕ_{Rv} are the characteristic functions of Zv respectively Rv.

- random variables Zv and Rv have the same distribution function if and only if they have the same characteristic function.

3.2 Fourier series

A Fourier series is an expansion of a periodic function $f(x)$ in terms of an infinite sum of sines and cosines. As such, a Fourier series exploits the orthogonality relationships of sine and cosine functions, see [7].
A function $f(x)$ supported on the domain $x \in [-\pi, \pi]$ can be written as its Fourier series by

$$f(x) = \frac{1}{2}A_0 + \sum_{n=1}^{\infty} A_n \cos(nx) + \sum_{n=1}^{\infty} B_n \sin(nx),$$
$$A_0 := \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)dy, \quad A_n := \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)\cos(ny)dy, \quad B_n := \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)\sin(ny)dy.$$

For more details see [5]. Often, it is unnecessary to use the full Fourier series. If a function $f(x)$ is symmetric, $f(x) = f(-x)$ for all x, then all B_n terms will be zero. The series with only A_0 and A_n terms is known as the "Fourier cosine series":

$$f(x) = \frac{1}{2}A_0 + \sum_{n=1}^{\infty} A_n \cos(nx), \quad A_0 := \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)dy, \quad A_n := \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)\cos(ny)dy.$$

The COS method is based on the Fourier cosine series.

3.3 Fourier cosine series with a Fourier transform

A function $f(x)$ supported on the domain $x \in [a, b]$ can be written as its Fourier cosine series by

$$f(x) = \frac{1}{2}A_0 + \sum_{k=1}^{\infty} A_k \cos\left(k\pi \frac{x-a}{b-a}\right) \tag{3.5}$$

$$= \sum_{k=0}^{\infty}{}' A_k \cos\left(k\pi \frac{x-a}{b-a}\right), \tag{3.6}$$

where A_k is defined as

$$A_k := \frac{2}{b-a}\int_a^b f(y)\cos\left(k\pi\frac{y-a}{b-a}\right)dy.$$

The symbol $\sum'$ is a summation where the first element is multiplied by one half. The coefficients A_k can be approximated using the Fourier transform of $f(y)$. This can be done with Euler's formula. This formula connects cosine and sine functions. The following identity holds

$$e^{ix} = \cos(x) + i\sin(x), \quad x \in \mathbb{R}.$$

Therefore, a cosine-function can be written as follows

$$\cos\left(k\pi\frac{y-a}{b-a}\right) = \mathrm{Re}\left\{\exp\left(ik\pi\frac{y-a}{b-a}\right)\right\},$$

where Re is the real part of a complex number. Putting this equation into the equation for A_k gives

$$A_k = \frac{2}{b-a}\int_a^b f(y)\mathrm{Re}\left\{\exp\left(ik\pi\frac{y-a}{b-a}\right)\right\}dy = \frac{2}{b-a}\mathrm{Re}\left\{\int_a^b f(y)\exp\left(ik\pi\frac{y-a}{b-a}\right)dy\right\}.$$

When taking the correct ω, the formula of A_k and the Fourier transform of $f(x)$ differ only on the integration range. Suppose the integral over the whole domain is a good approximation of the integral of the interval. The coefficients A_k can be written as

$$\begin{aligned} A_k &= \frac{2}{b-a}\mathrm{Re}\left\{\int_a^b f(y)\exp\left(i\left(\frac{k\pi}{b-a}\right)y - \frac{ik\pi a}{b-a}\right)dy\right\} \\ &\approx \frac{2}{b-a}\mathrm{Re}\left\{\int_{\mathbb{R}} f(y)\exp\left(i\left(\frac{k\pi}{b-a}\right)y - \frac{ik\pi a}{b-a}\right)dy\right\} \\ &\approx \frac{2}{b-a}\mathrm{Re}\left\{\phi\left(\frac{k\pi}{b-a}\right)\exp\left(-\frac{ik\pi a}{b-a}\right)\right\}. \end{aligned}$$

3.4 Discrete Fourier transform and fast Fourier transform

The discrete Fourier transform is one of the specific forms of Fourier analysis. As such, it transforms one vector into another. The DFT requires a finite sequence of numbers, so that a computer can process this sequence.

Definition 22. Discrete Fourier transform
Let $\vec{c} = (c_0, \ldots, c_{M-1}) \in \mathbb{C}^M$ be a vector of M complex numbers. The DFT of $\vec{c}$ is then defined as

$$\mathrm{DFT}(\vec{c})_k = \sum_{j=0}^{M-1} c_j e^{-\frac{2\pi i}{M}jk}, \quad k = 0, \ldots, M-1$$

where $\mathrm{DFT}(\vec{c}) = (\mathrm{DFT}(\vec{c})_0, \ldots, \mathrm{DFT}(\vec{c})_{M-1}) \in \mathbb{C}^M$.

This DFT can be written as a matrix-vector multiplication $\mathrm{DFT}(\vec{c}) = G_M \vec{c}$ with

$$(G_M)_{ij} = e^{-2\pi i \frac{ij}{M}},$$

where G_M is called the Fourier-matrix. This implies a number of multiplications of M^2 and a number of additions of $M(M-1)$. The arithmetic complexity is $\mathcal{O}(M^2)$ [44].

The fast Fourier transform (FFT) is an efficient algorithm designed to compute the DFT by minimizing the number of multiplications and summations. The most common FFT is the Cooley-Tukey algorithm [10]. If $M = 2^p$, with p a natural number, then the DFT can be calculated with this algorithm with a process of order

$$\mathcal{O}(M \log_2(M)) = \mathcal{O}(pM).$$

We refer to [7] and [44].

3.5 Discrete Fourier cosine transform

We only can use the COS method if we are able to determine the Fourier cosine transformation of the payoff function of the contingent claim. The Fourier cosine transformation of the payoff of a European vanilla option can be solved analytically. But in the case of other payoff functions, an analytic solution is often not available. Therefore, the payoff functions require a numerical approximation. For this approximation we typically use the discrete Fourier cosine transformation. In this section, we present the DCT for higher dimensions.
For all dimensions the programming languages MATLAB and C are equipped with a DCT subroutine or a DCT subroutine is available. But this is not the case for the computer language CUDA. However, it is possible to convert the DCT into a DFT. This conversion makes it possible to perform calculations for the DCT by means of the FFT.

Definition 23. DCT Type II
Let $\vec{c} = (c_0, \ldots, c_{Q-1}) \in \mathbb{R}^Q$ be the input vector, then the DCT of $\vec{c}$ is given by

$$\mathrm{DCT}(\vec{c})_k = 2 \sum_{n=0}^{Q-1} c_n \cos\left(\frac{\pi(2n+1)k}{2Q}\right), \quad 0 \leq k \leq Q-1 \tag{3.7}$$

where $\mathrm{DCT}(\vec{c}) = (\mathrm{DCT}(\vec{c})_0, \ldots, \mathrm{DCT}(\vec{c})_{Q-1}) \in \mathbb{R}^Q$.

If the calculations of the DCT are performed by the FFT, then this kind of calculation is called the fast cosine transform (FCT) algorithm.

Definition 24. Fast Cosine Transform (1D)
To calculate the DCT of $\vec{c} = (c_0, \ldots, c_{Q-1}) \in \mathbb{R}^Q$, where Q is even, we rewrite (3.7) as a FCT which reads

$$\mathrm{FCT}(\vec{c})_k = 2\mathrm{Re}\left[H_{4Q}^k \mathrm{FFT}(\breve{c})_k\right], \quad 0 \leq k \leq Q-1, \tag{3.8}$$

where

$$\breve{c}_n = \begin{cases} c_{2n}, & n = 0, \ldots, \left\lfloor \frac{Q-1}{2} \right\rfloor, \\ c_{Q-2n-1}, & n = \left\lfloor \frac{Q+1}{2} \right\rfloor, \ldots, Q-1, \end{cases} \tag{3.9}$$

and

$$H_{4Q}^k = \exp\left(\frac{-i\pi k}{2Q}\right). \tag{3.10}$$

Proof. See Appendix B.3.

Definition 25. Two-dimensional DCT
Let C be a matrix of size $Q \times Q$ of real numbers. The 2D DCT of C is given by

$$\text{DCT}(C)_{k,l} = 4 \sum_{n=0}^{Q-1} \sum_{m=0}^{Q-1} C_{n,m} \cos\left(\frac{\pi(2n+1)k}{2Q}\right) \cos\left(\frac{\pi(2m+1)l}{2Q}\right), \qquad 0 \leq k, l \leq Q-1.$$

Interchanging gives

$$DCT(C)_{k,l} = 2 \sum_{m=0}^{Q-1} \left[2 \sum_{n=0}^{Q-1} C_{n,m} \cos\left(\frac{\pi(2n+1)k}{2Q}\right)\right] \cos\left(\frac{\pi(2m+1)l}{2Q}\right), \quad 0 \leq k, l \leq Q-1.$$

We observe that a two-dimensional DCT consists of a 1D-cosine transformation for each row and a 1D-cosine transformation for each column. Also, for more than two dimensions this approach does not change. Independent of the dimension, the problem is converted into a number of 1D-cosine transformations.

For the 2D COS method, discussed in Appendix C, the payoff coefficients V_{k_1,k_2} are given by

$$V_{k_1,k_2} \approx \sum_{n_1=0}^{Q-1} \sum_{n_2=0}^{Q-1} \tfrac{2}{b_1-a_1} \tfrac{2}{b_2-a_2} g(y_1^{n_1}, y_2^{n_2}) \cos\left(k_1\pi \tfrac{y_1^{n_1}-a_1}{b_1-a_1}\right) \cos\left(k_2\pi \tfrac{y_2^{n_2}-a_2}{b_2-a_2}\right) \Delta y_1 \Delta y_2, \quad 0 \leq k_1, k_2 \leq Q-1.$$

The midpoint-rule integration gives us

$$y_i^{n_i} = a_i + (2n_i+1)\frac{b_i-a_i}{Q} \quad \Delta y_i = \frac{b_i-a_i}{Q} \quad i = 1, 2,$$

$$\begin{aligned} V_{k_1,k_2} &\approx \frac{2}{Q}\frac{2}{Q} \sum_{n_1=0}^{Q-1} \sum_{n_2=0}^{Q-1} g(y_1^{n_1}, y_2^{n_2}) \cos\left(k_1\pi \frac{2n_1+1}{2Q}\right) \cos\left(k_2\pi \frac{2n_2+1}{2Q}\right), \\ &= \frac{4}{Q^2} \sum_{n_1=0}^{Q-1} \sum_{n_2=0}^{Q-1} g(y_1^{n_1}, y_2^{n_2}) \cos\left(\frac{\pi(2n_1+1)k_1}{2Q}\right) \cos\left(\frac{\pi(2n_2+1)k_2}{2Q}\right). \end{aligned}$$

So, to calculate the V_{k_1,k_2} coefficients we can use the two-dimensional DCT Type II. However, we have to divide the answers by the known factor Q^2.

Definition 26. Fast cosine transform (2D)
Also for the two-dimensional case a FFT can be used. Define matrix $\breve{C}$ as

$$\breve{C}_{n_1,n_2} = \begin{cases} C_{2n_1,2n_2}, & 0 \leq n_1 \leq \left\lfloor \frac{Q-1}{2} \right\rfloor, & 0 \leq n_2 \leq \left\lfloor \frac{Q-1}{2} \right\rfloor, \\ C_{2Q-2n_1-1,2n_2}, & \left\lfloor \frac{Q+1}{2} \right\rfloor \leq n_1 \leq Q-1, & 0 \leq n_2 \leq \left\lfloor \frac{Q-1}{2} \right\rfloor, \\ C_{2n_1,2Q-2n_2-1}, & 0 \leq n_1 \leq \left\lfloor \frac{Q-1}{2} \right\rfloor, & \left\lfloor \frac{Q+1}{2} \right\rfloor \leq n_2 \leq Q-1, \\ C_{2Q-2n_1-1,2Q-2n_2-1}, & \left\lfloor \frac{Q+1}{2} \right\rfloor \leq n_1 \leq Q-1, & \left\lfloor \frac{Q+1}{2} \right\rfloor \leq n_2 \leq Q-1, \end{cases}$$

The 2D FCT is definied as

$$\text{FCT}(C) = 2\text{Re}\left\{ H_{4Q}^{k_1} \left[H_{4Q}^{k_2} \text{FFT}(\breve{C})_{k_1,k2} + H_{4Q}^{-k_2} \text{FFT}(\breve{C})_{k_1,Q-k_2} \right] \right\} \tag{3.11}$$

Proof. See Appendix B.4.

Definition 27. n-dimensional DCT
Let P be the nD input array then the nD DCT of P is given by

$$\mathrm{DCT}(P)_{k_1,\ldots,k_n} := 2^n \sum_{m_n=0}^{Q-1} \cdots \sum_{m_1=0}^{Q-1} P_{m_1,\ldots,m_n} \cos\left(\frac{\pi(2m_1+1)k_1}{2Q}\right) \ldots \cos\left(\frac{\pi(2m_n+1)k_n}{2Q}\right),$$
$$0 \le k_1, \ldots, k_n \le Q-1.$$

Interchanging gives us

$$\mathrm{DCT}(P)_{k_1,\ldots,k_n} = 2 \sum_{m_n=0}^{Q-1} \left[\ldots 2 \sum_{m_1=0}^{Q-1} P_{m_1,\ldots,m_n} \cos\left(\frac{\pi(2m_1+1)k_1}{2Q}\right) \ldots \right] \cos\left(\frac{\pi(2m_n+1)k_n}{2Q}\right),$$
$$0 \le k_1, \ldots, k_n \le Q-1.$$

For the nD COS method, g is the payoff function and thus the input for the n-dimensional DCT.

$$V_{\vec{k}} = \frac{2^n}{Q^n} \sum_{m_1=0}^{Q-1} \cdots \sum_{m_n=0}^{Q-1} g(\hat{y}_1^{m_1}, \ldots, \hat{y}_n^{m_n}) \cos\left(\frac{\pi(2m_1+1)k_1}{2Q}\right) \ldots \cos\left(\frac{\pi(2m_n+1)k_n}{2Q}\right) \quad 0 \le k_1, \ldots, k_n \le Q-1.$$

So, to calculate the $V_{\vec{k}}$ coefficients we can use the n-dimensional DCT Type II. However, we have to divide the answers by the known factor Q^n.

In anticipation of chapter 7, we note that our numerical experiments add up to three dimensions. Therefore, we do not give the definition for nD FCT , but we limit ourselves to the 3D FCT.

Definition 28. Fast cosine transform (3D)
The 3D FCT of the 3D array C of size $M_1 \times M_2 \times M_3$ is defined as

$$\begin{aligned}\mathrm{DCT}(C)_{k_1,k_2,k_3} :=& 2\mathrm{Re}\Big\{ H_{4M_1}^{k_1} \Big(H_{4M_2}^{k_2} H_{4M_3}^{k_3} \mathrm{FFT}(\breve{C})_{k_1,k_2,k_3} + H_{4M_2}^{k_2} H_{4M_3}^{-k_3} \mathrm{FFT}(\breve{C})_{k_1,k_2,M_3-k_3} \\ &+ H_{4M_2}^{-k_2} H_{4M_3}^{k_3} \mathrm{FFT}(\breve{C})_{k_1,M_2-k_2,k_3} + H_{4M_2}^{-k_2} H_{4M_3}^{-k_3} \mathrm{FFT}(\breve{C})_{k_1,M-k_2,M-k_3} \Big) \Big\},\end{aligned}$$

where $\breve{C}$ is defined in appendix B.5.

The COS method is a Fourier-transform based method. In the next two chapters we focus extensively on this method.

Chapter 4

One-dimensional COS method

The COS method is a deterministic method for pricing European options with one underlying asset. This method is also called one-dimensional COS method or 1D COS method. This numerical method developed by Fang and Oosterlee [17, 18] is based on Fourier cosine series expansions of the discounted expected payoff. The characteristic function of the underlying asset is used to approximate the Fourier coefficients.

In this chapter we discuss the 1D COS method. Firstly, we show the derivation of this method. This derivation leads to the COS formula. Secondly, we focus on the payoff coefficients needed in this formula. Thirdly, we analyze the truncation range. This range is necessary because the COS method only works on a finite domain. Fourthly, we analyze the overall error and we pay attention to the computational complexity. Finally, we derive that the overall error is exponentially decreasing and the calculation complexity is linear.

4.1 Derivation

We start from the risk-neutral valuation formula:

$$v(S(t),t) = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}[v(S(T),T)|S(t)] = e^{-r(T-t)}\int_{\mathbb{R}} v(S(T),T)f(S(T)|S(t))dS(T), \tag{4.1}$$

where t is the initial time, T is the maturity time, $S(t)$ is the price of the underlying asset at time t, $v(S(t),t)$ is the value of the option at time t, r is the constant risk-free interest rate, $\mathbb{E}^{\mathbb{Q}}[.]$ is the expectation under the risk-neutral measure and the function f is the probability density function of $S(T)$ given $S(t)$. At maturity time the value of an option is equal to its payoff.

We insert in (4.1):

$$x := S(t), \quad y := S(T), \quad \Delta t := T - t.$$

Then (4.1) reads

$$v(x,t) = e^{-r\Delta t}\int_{\mathbb{R}} v(y,T)f(y|x)dy. \tag{4.2}$$

In five steps we derive the 1D COS method, which is an approximation of this integral. We follow [16].

Step 1. *Truncate the integration range*
The density function $f(y|x)$ decays to zero very fast as $y \to \pm\infty$ in (4.2). Therefore, $v(x,t)$ can be well approximated by some finite integration range $[a,b] \subset \mathbb{R}$ without losing significant accuracy:

$$v_1(x,t) = e^{-r\triangle t}\int_a^b v(y,T)f(y|x)dy.$$

Remark: $v_1(x,t)$ is an approximation of $v(x,t)$.

Step 2. *Replace the density function by its cosine expansion*
The density function is usually not known; the characteristic function on the other hand is generally known for Lévy processes. That is why we replace the density function by its cosine expansion in y. The cosine expansion of $f(y|x)$ on $[a,b]$ is given by

$$f(y|x) = \sum_{k=0}^{\infty}{}' A_k(x) \cos\left(k\pi \frac{y-a}{b-a}\right), \tag{4.3}$$

where the series coefficients $A_k(x)$ are defined as

$$A_k(x) := \frac{2}{b-a} \int_a^b f(y|x) \cos\left(k\pi \frac{y-a}{b-a}\right) dy.$$

Inserting (4.3) into $v_1(x,t)$, gives

$$v_1(x,t) = e^{-r\triangle t} \int_a^b v(y,T) \sum_{k=0}^{\infty}{}' A_k(x) \cos\left(k\pi \frac{y-a}{b-a}\right) dy. \tag{4.4}$$

The $\sum'$-summation is again a summation where the first term is multiplied by 0.5.

Step 3. *Interchange summation and integration*
We interchange summation and integration in (4.4). We introduce the terms $2/(b-a)$ and $(b-a)/2$ and insert them into (4.4)

$$v_1(x,t) = e^{-r\triangle t} \sum_{k=0}^{\infty}{}' \frac{b-a}{2} A_k(x) \frac{2}{b-a} \int_a^b v(y,T) \cos\left(k\pi \frac{y-a}{b-a}\right) dy, \tag{4.5}$$

and we define the Fourier-cosine series coefficients V_k of $v(y,T)$ on $[a,b]$ as

$$V_k := \frac{2}{b-a} \int_a^b v(y,T) \cos\left(k\pi \frac{y-a}{b-a}\right) dy.$$

Insert V_k into (4.5), gives

$$v_1(x,t) = e^{-r\triangle t} \sum_{k=0}^{\infty}{}' \frac{b-a}{2} A_k(x) V_k. \tag{4.6}$$

The integral over the product of $f(y|x)$ and $v(y,T)$ is now written as a summation of the product of their Fourier-cosine coefficients.

Step 4. *Truncate the series summation*
The summation is further truncated because these coefficients have a rapid decay rate.

$$v_2(x,t) = e^{-r\Delta t} \sum_{k=0}^{N-1}{}' \frac{b-a}{2} A_k(x) V_k. \tag{4.7}$$

The function $v_2(x,t)$ is an approximation of $v_1(x,t)$. It contains an overall error consisting of two approximations compared to $v(x,t)$.

Step 5. *Insert the characteristic function*
We define the characteristic function of $f(y|x)$ on the interval $[a,b]$ by ϕ_A. Take ϕ as the characteristic function of $f(y|x)$ over the domain $\mathbb{R}$. If the characteristic function of $f(y|x)$ is known, then it will

be defined on the whole domain $\mathbb{R}$. The function $f(y|x)$ decays to zero very rapidly outside the domain $[a, b]$. Therefore, ϕ_A will not differ much from ϕ.

$$\begin{aligned} A_k(x) &= \frac{2}{b-a}\int_a^b f(y|x)\cos\left(k\pi\frac{y-a}{b-a}\right)dy \\ &= \frac{2}{b-a}\mathrm{Re}\left\{\phi_A\left(\frac{k\pi}{b-a}|x\right)\exp\left(-i\frac{ka\pi}{b-a}\right)\right\} \\ &\approx \frac{2}{b-a}\mathrm{Re}\left\{\phi\left(\frac{k\pi}{b-a}|x\right)\exp\left(-i\frac{ka\pi}{b-a}\right)\right\}. \end{aligned} \tag{4.8}$$

We introduce $F_k(x)$ defined as

$$F_k(x) := \mathrm{Re}\left\{\phi\left(\frac{k\pi}{b-a}|x\right)\exp\left(-i\frac{ka\pi}{b-a}\right)\right\}. \tag{4.9}$$

We insert $F_k(x)$ into (4.7), which gives the 1D COS pricing formula:

$$\boxed{v(x,t) \approx v_3(x,t) = e^{-r\Delta t}\sum_{k=0}^{N-1}{}' F_k(x)V_k} \tag{4.10}$$

Remark: $v_3(x,t)$ contains an overall error consisting of three approximations compared to $v(x,t)$.

4.2 Payoff coefficients

Before we can use the 1D COS formula we need to know the payoff series coefficients V_k. We defined V_k to be given by the equation

$$V_k = \frac{2}{b-a}\int_a^b v(y,T)\cos\left(k\pi\frac{y-a}{b-a}\right)dy, \tag{4.11}$$

where $v(y,T)$ is the payoff function of the option which depends on the asset price $S(T)$ at time T.
In practice, when the characteristic function of an exponential Lévy process is known it will be the characteristic function of the log-asset price, which is known. Therefore, the payoff function has to be related to the log-asset price. We perform a change of variables to achieve this transformation. Let $\hat{x}$ and $\hat{y}$ be defined as

$$\begin{aligned} \hat{x} &:= \log\left(\frac{S(t)}{K}\right), \\ \hat{y} &:= \log\left(\frac{S(T)}{K}\right), \end{aligned}$$

then $\hat{x}$ is a log-asset process. The payoff of a European call option reads

$$v(\hat{y},T) = \max(K(e^{\hat{y}}-1),0). \tag{4.12}$$

To solve equation (4.11) for a European call option we employ the method described in [17].
For $[c,d] \subseteq [a,b]$ we use two analytic functions:

$$\begin{aligned}
\chi_k(c,d) :&= \int_c^d e^y \cos\left(k\pi\frac{y-a}{b-a}\right) dy \\
&= \frac{1}{1+\left(\frac{k\pi}{b-a}\right)^2}\Bigg[\cos\left(k\pi\frac{d-a}{b-a}\right)e^d - \cos\left(k\pi\frac{c-a}{b-a}\right)e^c \\
&\quad + \frac{k\pi}{b-a}\sin\left(k\pi\frac{d-a}{b-a}\right)e^d - \frac{k\pi}{b-a}\sin\left(k\pi\frac{c-a}{b-a}\right)e^c\Bigg], \qquad (4.13)\\
\psi_k(c,d) :&= \int_c^d \cos\left(k\pi\frac{y-a}{b-a}\right) dy \\
&= \begin{cases} \frac{b-a}{k\pi}\left[\sin\left(k\pi\frac{d-a}{b-a}\right) - \sin\left(k\pi\frac{c-a}{b-a}\right)\right] & \text{for } k \neq 0, \\ (d-c) & \text{for } k = 0. \end{cases} \qquad (4.14)
\end{aligned}$$

Insert (4.12), (4.13) and (4.14) into (4.11), then

$$\begin{aligned}
V_k^{\text{call}} &= \frac{2}{b-a}\int_a^b \max(K(e^{\hat{y}}-1),0)\cos\left(k\pi\frac{\hat{y}-a}{b-a}\right) d\hat{y} \\
&= \frac{2}{b-a}\int_{\max(a,0)}^b K(e^{\hat{y}}-1)\cos\left(k\pi\frac{\hat{y}-a}{b-a}\right) d\hat{y} \\
&= \frac{2}{b-a}K\left\{\chi_k\big(\max(a,0),b\big) - \psi_k\big(\max(a,0),b\big)\right\}. \qquad (4.15)
\end{aligned}$$

The payoff for a European put option is given by

$$v(\hat{y},T) = \max(K(1-e^{\hat{y}}),0). \qquad (4.16)$$

Insert (4.16), (4.13) and (4.14) into (4.11), then

$$V_k^{\text{put}} = \frac{2}{b-a}K\left\{\psi_k\big((a,\min(b,0)\big) - \chi_k\big(a,\min(b,0)\big)\right\}. \qquad (4.17)$$

It is possible that an analytic solution is not available for the V_k coefficients. In that case, a numerical approximation of these coefficients has to be performed. However, this has an impact on the convergence of the COS method.
We note that our Formulas 4.15 and 4.17 differ from the Formulas (24) and (25) of Fang and Oosterlee [17]. They implicitly assume that $a < 0$ for V_k^{call} and $b > 0$ for V_k^{put}. We do not restrict these values.

4.3 Truncation range

We need to choose a finite domain $[a,b]$ such that the truncated integral approximates the infinite integral very well. The integration range $[a,b]$ we used in step 1 of section (4.1) is the rule-of-thumb provided by [17],

$$[a,b] := \left[\hat{x} + \xi^1 - L\sqrt{\xi^2 + \sqrt{\xi^4}}, \hat{x} + \xi^1 + L\sqrt{\xi^2 + \sqrt{\xi^4}}\right], \qquad (4.18)$$

where ξ^1, ξ^2 and ξ^4 are the cumulants and L is a scaling parameter. When L increases, the difference between $v_3(x,t)$ and $v(x,t)$ will be smaller. Unfortunately, when L increases, more terms are needed to reach the same accuracy. A test by [17] shows that $L = 10$ will give fast and accurate results for option pricing with $\Delta t = 0.1$ up to $\Delta t = 10$.

We define the function $momgen(t) = \log\left(\mathbb{E}\left[e^{t\hat{y}}\right]\right), \quad t \in \mathbb{R}$, where $momgen(t)$ is the moment-generating function of random variable $\hat{y}$. The i-th cumulant from equation (4.18) is the values at $t = 0$ of the i-th derivative to t of function $momgen(t)$.

4.4 Error analysis

The derivation of the COS formula introduces various errors. In this section we analyse these errors. We will show that the overall error has exponential convergence when the probability density function, $f(y|x)$, belongs to $\mathbb{C}^{\infty}([a,b] \subset \mathbb{R})$. We use error analysis for the mathematical justification that the COS method is fast and highly accurate. We follow Fang and Oosterlee [17].

1. *The integration range truncation error*
   The first error appears at the truncation of the integration range, see section 4.1. The error can be written as:

$$\epsilon^1 := v(x,t) - v_1(x,t) = e^{-r\Delta t} \int_{\mathbb{R}\setminus[a,b]} v(y,T) f(y|x) dy. \tag{4.19}$$

2. *The series truncation error on* $[a,b]$
   The second error arises at the truncation of the series summation on $[a,b]$. The error can be written as:

$$\epsilon^2 := v_1(x,t) - v_2(x,t) = \frac{b-a}{2} e^{-r\Delta t} \sum_{k=N}^{\infty} A_k(x) V_k. \tag{4.20}$$

3. *The error related to approximating* $A_k(x)$ *by* $F_k(x)$*, see (4.9)*
   The third error arises by inserting the Fourier-cosine transform and can be written as:

$$\epsilon^3 := v_2(x,t) - v_3(x,t) = e^{-r\Delta t} {\sum_{k=0}^{N-1}}' \operatorname{Re}\left\{ \int_{\mathbb{R}\setminus[a,b]} \exp\left(ik\pi \frac{y-a}{b-a} \right) f(y|x) dy \right\} V_k. \tag{4.21}$$

Lemma 4. (Bounding ϵ^3 with integration range truncation)
Error ϵ^3 consists of integration range truncation errors, and can be bounded by:

$$|\epsilon^3| < \overline{Q} |\hat{\epsilon^3}|,$$

where $\overline{Q}$ is some constant independent of N and

$$\hat{\epsilon^3} := \int_{\mathbb{R}\setminus[a,b]} f(y|x) dy.$$

Proof. See Fang [16].

So, the errors ϵ^1 and ϵ^3 are related to the domain $[a,b]$. When this domain is sufficiently large, the overall error is dominated by ϵ^2. For error ϵ^2, the product of $A_k(x)$ and V_k converges faster than $A_k(x)$ or V_k. Therefore,

$$\left| \sum_{k=N}^{\infty} A_k(x) V_k \right| \leq C \sum_{k=N}^{\infty} |A_k(x)|,$$

for some constant C. Hence, error ϵ^2 is dominated by the series truncation error of the density function.

Lemma 5. (Bounding of ϵ^2 with infinitely differential density functions)
Error ϵ^2 converges exponentially in the case of smooth density functions $f(x) \in \mathbb{C}^{\infty}([a,b])$.

$$|\epsilon^2| < \hat{P} e^{-(N-1)\nu}, \tag{4.22}$$

where $\nu > 0$ is a constant and $\hat{P}$ is a term that varies less than exponentially with N.
Proof. See Fang [16], Fang and Oosterlee [17].

Lemma 6. (Bounding of ϵ^2 with discontinuous density functions)
Error ϵ^2 for density functions having discontinuous derivatives can be bounded as follows

$$|\epsilon^2| < \frac{\overline{P}}{(N-1)^{\beta-1}} \tag{4.23}$$

where $\overline{P}$ is a constant and $\beta \geq aic \geq 1$, aic is the algebraic convergence of V_k.
Proof. See Fang [16], Fang and Oosterlee [17].

From the above we conclude that the overall error ϵ converges exponentially to zero for smooth density functions that belong to $\mathbb{C}^{\infty}([a,b] \subset \mathbb{R})$, i.e.

$$\begin{aligned} |\epsilon| &\leq |\epsilon^1| + |\epsilon^2| + |\epsilon^3| \quad \text{(triangle inequality)} \\ &< |\epsilon^1| + \hat{P}e^{-(N-1)\nu} + \overline{Q}|\hat{\epsilon^3}|, \end{aligned}$$

and the overall error converges algebraically when the density function of the underlying process has a discontinuity in one of its derivatives, i.e.

$$\begin{aligned} |\epsilon| &\leq |\epsilon^1| + |\epsilon^2| + |\epsilon^3| \\ &< |\epsilon^1| + \frac{\overline{P}}{(N-1)^{\beta-1}} + \overline{Q}|\hat{\epsilon^3}|. \end{aligned}$$

4.5 Complexity

In the previous section we showed that the decay of the convergence rate of the overall error is exponential. Therefore, the question arises by how much the computational time increases. In this section, we discuss this question. We focus on the complexity of the calculations of the COS method. The complexity can be derived from:

$$v_3(x,t) = e^{-r\Delta t} \sum_{k=0}^{N-1}{}' F_k(x) V_k.$$

The calculation of this sum consists of N coefficients $F_k(x)$ and V_k. The time to calculate any of the $F_k(x)$ or V_k will be the same for every k. Thus, the calculation time of the $F_k(x)$ elements is $\mathcal{O}(N)$ and the calculation time of the V_k elements is also $\mathcal{O}(N)$, where $\mathcal{O}$ denotes the order. The summation can also be done in $\mathcal{O}(N)$. Therefore, the total calculation complexity is $\mathcal{O}(N)$. From $\mathcal{O}(N)$ complexity it follows that the calculation complexity is linear with the number of terms N.

4.6 Conclusion

We conclude that the 1D COS method is very fast and highly accurate. The convergence rate of the error for continuous density functions of the underlying asset is exponential; and its computational complexity is linear. It is important to choose the truncation range carefully.

The 1D COS method has been extended to higher dimensions by Ruijter and Oosterlee [33] and subsequently by Pellegrino and Sabino [32]. In appendix C we present the 2D COS method. In the next chapter we discuss the nD COS method.

Chapter 5

N-dimensional COS method

If an option is based on one underlying asset we call this option one-dimensional. If an option is based on more than one underlying asset it is called multi-dimensional. In this chapter we present the nD COS method. This chapter is organized as follows. Firstly, we show the derivation of the nD COS method, which leads to the nD COS formula. Thereafter, we focus on the payoff coefficients for this formula, analyze the truncation range and the overall error, and pay attention to the computational complexity. Finally, we derive that the overall error converges algebraically and is of second order; and that the calculation complexity is of nth order.

5.1 Derivation

The value of a European option at time $0 \leq t \leq T$ is given by the risk-neutral valuation formula as follows

$$v(\vec{S}(t), t) = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}[g(\vec{S}(T))|\vec{S}(t)], \tag{5.1}$$

where $\vec{S}(t)$ denotes an n-dimensional vector of n stock prices at time t, the function g is the payoff function depending on the stock prices at time T, r is the risk-free interest rate and $\mathbb{E}^{\mathbb{Q}}$ is the expectation under the risk-neutral measure. We write (5.1) as an integral and thereby make use of some convenient shorthand notations:

$$\Delta t := T - t, \quad \vec{x} := \vec{S}(t), \quad \vec{y} := \vec{S}(T).$$

The risk-neutral valuation formula can be written as an integral

$$v(\vec{x}, t) = e^{-r\Delta t} \int_{\mathbb{R}^n} g(\vec{y}) f(\vec{y}|\vec{x}) d\vec{y} \tag{5.2}$$

under the assumption that the risk-free interest rate r is constant and f is the probability density function of $\vec{y}$ given $\vec{x}$.

We distinguish again five successive steps for the derivation of the nD COS method.

1. *We truncate the infinite integration range in the risk-neutral valuation formula*
   Because the probability density function $f(\vec{y}|\vec{x})$ decays to zero rapidly for $||\vec{y}|| \to \infty$, we can approximate (5.2) without losing significant accuracy by a finite integration range $[a_1, b_1] \times \ldots \times [a_n, b_n] \subset \mathbb{R}^n$:

   $$v_1(\vec{x}, t) = e^{-r\Delta t} \int_{a_n}^{b_n} \ldots \int_{a_1}^{b_1} g(\vec{y}) f(\vec{y}|\vec{x}) dy_1 \ldots dy_n$$

   Remark: $v_1(\vec{x}, t)$ is an approximation of $v(\vec{x}, t)$. The subscript 1 means $v_1(\vec{x}, t)$ contains an error based on one approximation.

2. *We replace the probability density function by its cosine expansion in $\vec{y}$*
By means of the n-dimensional Fourier-cosine series expansion we can define a function on a finite domain. The cosine expansion of $f(\vec{y}|\vec{x})$ on $[a_1, b_1] \times \ldots \times [a_n, b_n]$ is given by

$$f(\vec{y}|\vec{x}) = \sum_{k_1=0}^{\infty}{}' \ldots \sum_{k_n=0}^{\infty}{}' A_{k_1,\ldots,k_n}(\vec{x}) \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right), \tag{5.3}$$

where $A_{\vec{k}}(\vec{x})$ is defined as

$$A_{\vec{k}}(\vec{x}) = \prod_{i=1}^{n}\left(\frac{2}{b_i-a_i}\right)\int_{a_n}^{b_n}\ldots\int_{a_1}^{b_1} f(\vec{y}|\vec{x})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right)dy_1\ldots dy_n.$$

Inserting (5.3) into $v_1(\vec{x},t)$ gives

$$\begin{aligned} v_1(\vec{x},t) = e^{-r\Delta t}\int_{a_n}^{b_n}\ldots\int_{a_1}^{b_1} g(\vec{y}) \sum_{k_1=0}^{\infty}{}' \ldots \sum_{k_n=0}^{\infty}{}' A_{\vec{k}}(\vec{x}) \\ \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right)d\vec{y}. \end{aligned} \tag{5.4}$$

Remark: $\sum'$ is a summation where the first term is multiplied by 0.5.

3. *We interchange integration and summation*
Then, (5.4), reads:

$$\begin{aligned} v_1(\vec{x},t) =& e^{-r\Delta t}\prod_{i=1}^{n}\left(\frac{b_i-a_i}{2}\right)\sum_{k_1=0}^{\infty}{}'\ldots\sum_{k_n=0}^{\infty}{}' A_{k_1,\ldots,k_n}(\vec{x})\cdot \\ & \prod_{i=1}^{n}\left(\frac{2}{b_i-a_i}\right)\int_{a_1}^{b_1}\ldots\int_{a_n}^{b_n} g(\vec{y})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right)d\vec{y}. \end{aligned} \tag{5.5}$$

We define $V_{\vec{k}}$ as

$$V_{\vec{k}} = \prod_{i=1}^{n}\left(\frac{2}{b_i-a_i}\right)\int_{a_1}^{b_1}\ldots\int_{a_n}^{b_n} g(\vec{y})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right)d\vec{y}, \tag{5.6}$$

and insert $V_{\vec{k}}$ into (5.5)

$$v_1(\vec{x},t) = e^{-r\Delta t}\prod_{i=1}^{n}\left(\frac{b_i-a_i}{2}\right)\sum_{k_1=0}^{\infty}{}'\ldots\sum_{k_n=0}^{\infty}{}' A_{\vec{k}}(\vec{x})V_{\vec{k}}. \tag{5.7}$$

Remark: now we have written the integral over the product of $f(\vec{y}|\vec{x})$ and $g(\vec{y})$ as a summation of the product of their Fourier-cosine series coefficients.

4. *We truncate the series summation*
The coefficients $A_{\vec{k}}(\vec{x})$ and $V_{\vec{k}}$ decay rapidly. Therefore, we truncate the series summation in (5.7), which gives

$$v_2(\vec{x},t) = e^{-r\Delta t}\prod_{i=1}^{n}\left(\frac{b_i-a_i}{2}\right)\sum_{k_1=0}^{N-1}{}'\ldots\sum_{k_n=0}^{N-1}{}' A_{\vec{k}}(\vec{x})V_{\vec{k}}. \tag{5.8}$$

Remark: the function $v_2(\vec{x},t)$ is an approximation of $v_1(\vec{x},t)$. It contains a overall error consisting of two approximations compared to $v(\vec{x},t)$.

5. *We substitute the series coefficients by the ChF approximation*

We define $F_{\vec{k}}(\vec{x})$ as

$$F_{\vec{k}}(\vec{x}) = \int_{\mathbb{R}^n} f(\vec{y}|\vec{x}) \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right)d\vec{y}. \tag{5.9}$$

We introduce the following trigonometric rule,

$$\prod_{k=1}^{n}\cos(\theta_k) = \frac{1}{2^{n-1}}\sum_{\vec{e}\in\mathbb{G}}\cos(e_1\theta_1+\ldots+e_n\theta_n) \quad \text{for} \quad \vec{\theta}\in\mathbb{R}^n, \tag{5.10}$$

where $\mathbb{G} = \{1\}\{1,-1\}^{n-1}$.
Now, we can write (5.9) as

$$F_{\vec{k}}(\vec{x}) = \frac{1}{2^{n-1}}\sum_{\vec{e}\in\mathbb{G}}F_{\vec{k}}^{\vec{e}}, \tag{5.11}$$

where

$$\begin{aligned}
F_{\vec{k}}^{+}(\vec{x}) =& \int_{\mathbb{R}^n} f(\vec{y}|\vec{x})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}+\ldots+k_n\pi\frac{y_n-a_n}{b_n-a_n}\right)d\vec{y}\\
=&\mathrm{Re}\left\{\int_{\mathbb{R}^n} f(\vec{y}|\vec{x})\exp\left(ik_1\pi\frac{y_1-a_1}{b_1-a_1}+\ldots+ik_n\pi\frac{y_n-a_n}{b_n-a_n}\right)d\vec{y}\right\}\\
=&\mathrm{Re}\Bigg\{\int_{\mathbb{R}^n} f(\vec{y}|\vec{x})\exp\left(ik_1\pi\frac{y_1}{b_1-a_1}+\ldots+ik_n\pi\frac{y_n}{b_n-a_n}\right)d\vec{y}\cdot\\
&\exp\left(ik_1\pi\frac{-a_1}{b_1-a_1}+\ldots+ik_n\pi\frac{-a_n}{b_n-a_n}\right)\Bigg\}.
\end{aligned} \tag{5.12}$$

The integral of (5.12) is the ChF of $f(\vec{y}|\vec{x})$. Writing $F_{\vec{k}}^{+}(\vec{x})$ with its ChF gives

$$F_{\vec{k}}^{+}(\vec{x}) = \mathrm{Re}\left\{\phi\left(\frac{k_1\pi}{b_1-a_1},\ldots,\frac{k_n\pi}{b_n-a_n}|\vec{x}\right)\exp\left(ik_1\pi\frac{-a_1}{b_1-a_1}+\ldots+ik_n\pi\frac{-a_n}{b_n-a_n}\right)\right\}. \tag{5.13}$$

We can derive the formula for $F_{\vec{k}}^{-}(\vec{x})$ in almost the same way.

$$F_{\vec{k}}^{-}(\vec{x}) = \mathrm{Re}\left\{\phi\left(\frac{k_1\pi}{b_1-a_1},\ldots,-\frac{k_n\pi}{b_n-a_n}|\vec{x}\right)\exp\left(ik_1\pi\frac{-a_1}{b_1-a_1}-\ldots-ik_n\pi\frac{-a_n}{b_n-a_n}\right)\right\}. \tag{5.14}$$

Now we can replace $A_{\vec{k}}(\vec{x})$ of (5.8) by $F_{\vec{k}}(\vec{x})$. Together with (5.9), (5.13) and (5.14) gives the nD COS pricing formula

$$\boxed{v_3(\vec{x},t) = e^{-r\Delta t}\frac{1}{2^{n-1}}\sum_{k_1=0}^{N-1}{}'\ldots\sum_{k_n=0}^{N-1}{}'\left(F_{\vec{k}}^{+}(\vec{x})+\ldots+F_{\vec{k}}^{-}(\vec{x})\right)V_{\vec{k}}} \tag{5.15}$$

Remarks:

(a) We explain $\left[F_{\vec{k}}^{+}(\vec{x})+\ldots+F_{\vec{k}}^{-}(\vec{x})\right]$. On the dots in this formula we have to place all other combinations of $F^{\pm}(\vec{x})$. The number of terms depends on the dimension. The first superscript sign is always +. The signs thereafter are either + or -. Some examples:

2D	$2^1=2$	$F^{+}+F^{-}$
3D	$2^2=4$	$F^{++}+F^{+-}+F^{-+}+F^{--}$
4D	$2^3=8$	$F^{+++}+F^{++-}+F^{+-+}+F^{-++}+F^{--+}+F^{+--}+F^{-+-}+F^{---}$
nD	2^{n-1}	It is clear that it is impossible to write all these combinations in the nD COS pricing formula. Therefore we use the dots notation.

(b) We define the characteristic function of $f(\vec{y}|\vec{x})$ on the interval $[a_1, b_1] \times \ldots \times [a_n, b_n]$ by ϕ_A, and the characteristic function of $f(\vec{y}|\vec{x})$ on the domain $\mathbb{R}^n$ as ϕ. If the characteristic function of $f(\vec{y}|\vec{x})$ is known then it will be defined on the whole domain $\mathbb{R}^n$. The function $f(\vec{y}|\vec{x})$ decays to zero very rapidly outside the domain $[\vec{a}, \vec{b}]$. Therefore, ϕ_A will not differ much from ϕ.

(c) The function $v_3(\vec{x}, t)$ is an approximation of $v_2(\vec{x}, t)$. $v_3(\vec{x}, t)$ contains an overall error consisting of three approximations compared to $v(\vec{x}, t)$.

5.2 Payoff coefficients

We defined $V_{\vec{k}}$ (5.6) as

$$V_{\vec{k}} = \prod_{i=1}^{n} \left(\frac{2}{b_i - a_i} \right) \int_{a_1}^{b_1} \ldots \int_{a_n}^{b_n} g(\vec{y}) \cos\left(k_1 \pi \frac{y_1 - a_1}{b_1 - a_1} \right) \ldots \cos\left(k_n \pi \frac{y_n - a_n}{b_n - a_n} \right) d\vec{y},$$

where $g(\vec{y})$ is the payoff function of an option which depends on the asset prices $\vec{S}(T)$ at time T. When we say that the characteristic function of an exponential Lévy processes is known, we mean that the characteristic function of the log asset prices is known. Therefore, the payoff function $g(\vec{y})$ has to be transformed to a payoff function based on log asset prices. We perform a change of variables to achieve this transformation. Let $\hat{x}$ and $\hat{y}$ be defined as

$$\hat{x} := \log\left(\frac{\vec{S}(t)}{K} \right), \quad \hat{y} := \log\left(\frac{\vec{S}(T)}{K} \right),$$

where $\hat{x}$ is the log-asset price at initial time and $\hat{y}$ is the log-asset price at maturity time.

For example, the payoff of an n-dimensional European arithmetic basket call option reads

$$g(\hat{x}) = \max(K(e^{\hat{x}_1 + \ldots + \hat{x}_n} - 1), 0). \tag{5.16}$$

Remark: when $n \geq 2$ it is not possible to calculate the payoff coefficients $V_{\vec{k}}$ analytically. Therefore, for $n \geq 2$ we use the n-dimensional DCT to approximate the values of these coefficients, see chapter 3.

5.3 Truncation range

In an infinite domain the COS method is not defined because of the Fourier cosine series expansions. Therefore, we need to choose a finite domain $[a_1, b_1] \times \ldots \times [a_n, b_n]$ such that the truncated integral approximates the infinite integral very well. The integration range $[a_1, b_1] \times \ldots \times [a_n, b_n]$ we used in step 1 of section (5.1) is taken from [33],

$$a_i := \hat{x}_i + \xi_i^1 - L\sqrt{\xi_i^2 + \sqrt{\xi_i^4}}$$
$$b_i := \hat{x}_i + \xi_i^1 + L\sqrt{\xi_i^2 + \sqrt{\xi_i^4}}$$

where ξ^1, ξ^2 and ξ^4 are the cumulants (see sections 7.2.1 and 7.3.1) and L is a scaling parameter.

Remark: when we increase L, the total error of the COS method (difference between $v_3(\vec{S}(t), t)$ and $v(\vec{S}(t), t)$) is mitigated, but, unfortunately, more terms are needed to reach the same accuracy and this consumes more time. $L = 10$ gives accurate results for option pricing with $\Delta t = 0.1, \ldots, 1$ [17].

5.4 Error analysis

The derivation of the COS formula introduces an error which is composed of three parts. In addition, the DCT approximation gives rise to an extra error. In this section we discuss these errors and their convergence properties.

1. *First part*
   The first error appears at the truncation of the integration range. It can be written as:

$$\epsilon^1 = v(\vec{S}(t),t) - v_1(\vec{S}(t),t) = e^{-r\Delta t} \int_{\mathbb{R}\setminus[a_1,b_1]\times\ldots\times[a_n,b_n]} g(\vec{S}(T)) f(\vec{y}|\vec{x}) d\vec{y}.$$

2. *Second part*
   The second error arises at the truncation of the series summation on $[a_1,b_1]\times\ldots\times[a_n,b_n]$. It can be written as:

$$\epsilon^2 = v_1(\vec{S}(t),t) - v_2(\vec{S}(t),t) = \prod_{i=1}^{n}\left(\frac{b_i-a_i}{2}\right) e^{-r\Delta t} \sum_{k_1=N}^{\infty} \ldots \sum_{k_i=N}^{\infty} A_{\vec{k}}(\vec{x}) V_{\vec{k}}.$$

3. *Third part*
   The third error arises by substituting the series coefficients by the ChF approximation. It can be written as:

$$\begin{aligned}
\epsilon^3 &= v_2(\vec{S}(t),t) - v_3(\vec{S}(t),t) \\
&= \prod_{i=1}^{n}\left(\frac{b_i-a_i}{2}\right) e^{-r\Delta t} {\sum_{k_1=0}^{N_1-1}}' \ldots {\sum_{k_n=0}^{N_n-1}}' (A_{\vec{k}}(\vec{x}) - F_{\vec{k}}(\vec{x})) V_{\vec{k}} \\
&= e^{-r\Delta t} \int\int_{\mathbb{R}^2\setminus[a_1,b_1]\times\ldots\times[a_n,b_n]} \\
&\qquad \left[{\sum_{k_1=0}^{N_1-1}}' \ldots {\sum_{k_n=0}^{N_n-1}}' \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\ldots\cos\left(k_n\pi\frac{y_n-a_n}{b_n-a_n}\right) V_{\vec{k}} \right] f(\vec{y}|\vec{x}) d\vec{y}.
\end{aligned}$$

4. *DCT error*
   This extra error occurs because the elements of $V_{\vec{k}}$ are approximated using a DCT.

$$\epsilon^{DCT} = \prod_{i=1}^{n}\left(\frac{b_i-a_i}{2}\right) e^{-r\Delta t} {\sum_{k_1=0}^{N_1-1}}' {\sum_{k_2=0}^{N_2-1}}' F_{\vec{k}}(\vec{x})[V_{\vec{k}} - V_{\vec{k}}^{DCT}].$$

Let us assume that $V_{\vec{k}}$ terms are exact, then ϵ^{DCT} is zero. Let us also assume that the integration domain $[a_1,b_1]\times\ldots\times[a_n,b_n]$ is sufficiently wide, then the ϵ^2 will dominate the overall error. It follows, that for smooth density functions, i.e. $f(\vec{y}|\vec{x}) \in \mathbb{C}^\infty$, the overall error ϵ converges exponentially in N.

Let us assume that $V_{\vec{k}}$ terms are not exact, then ϵ^{DCT} exists. This error converges algebraically in Q with order two, see section 3.5. Therefore, the overall error converges to zero algebraically with second order in Q.

5.5 Complexity

In this section, we focus on the computational complexity of the calculations for the nD COS method. This complexity can be derived from

$$v_3(\vec{x},t) = e^{-r\Delta t}\frac{1}{2^{n-1}} {\sum_{k_1=0}^{N-1}}' \ldots {\sum_{k_n=0}^{N-1}}' \left(F_{\vec{k}}^{+}(\vec{x}) + \ldots + F_{\vec{k}}^{-}(\vec{x})\right) V_{\vec{k}}.$$

The calculation of this sum consists of $2^{n-1}N^n$ coefficients $F_{\vec{k}}(\vec{x})$ and Q^n coefficients $V_{\vec{k}}$. The time to calculate any of the $F_{\vec{k}}(\vec{x})$ will be the same for every $\vec{k}$. Thus, the calculation time of the $F_{\vec{k}}(\vec{x})$ elements is $\mathcal{O}(N^n)$, where $\mathcal{O}$ denotes the order.
For the elements $V_{\vec{k}}$ a DCT function is needed. A single DCT with the FFT method has calculation complexity $\mathcal{O}(Q\log_2(Q))$. The DCT has to be applied for every dimension. Thus, a total number of nQ^{n-1} DCTs is required. The total calculation complexity for all DCTs is $\mathcal{O}(Q^n\log_2(Q))$.

The summation can be done in $\mathcal{O}(N^n)$ operations. $Q \geq N$: the total complexity of the nD COS method is:

$$\mathcal{O}(Q^n\log_2(Q)). \tag{5.17}$$

From (5.17) it follows that the calculation complexity is of the nth order in the number of terms Q.

Remark: for $n = 2$ the calculation complexity is at least quadratic in the number of terms Q.

5.6 Conclusion

We conclude that the nD ($n \geq 2$) COS method is fast and highly accurate. The convergence rate of the error for continuous density functions of the underlying assets is exponential in N; and has an algebraic convergence in Q with order two. Its computational complexity is of the nth order. It is important to choose the truncation range carefully.

Finally, we note that the dimension n cannot be chosen too large. On the one hand, the computing time grows exponentially with n as a result of which the curse of dimensionality sets in. On the other hand, the memory space needed grows exponentially in n and a computer has a limited storage capacity. An example: if we have 1 GB or 8 GB of memory storage available for the $V_{\vec{k}}$ coefficients, then the number of terms Q per dimension decreases substantially, see table 5.1.

n	1GB	8GB
1	125,000,000	1,000,000,000
2	11,180	31,622
3	500	1000
4	105	177
5	42	63
6	22	31
7	14	19

Table 5.1: number of terms (Q) per dimension with 1GB and 8GB of memory

In this chapter we focused on the mathematics of the nD COS method. Before we perform our numerical experiments it is useful to test the different parts of the COS algorithm in the three aforementioned computer languages. We execute these tests in the next chapter.

Chapter 6

Parallel implementation

In Chapter 7, we will perform our numerical experiments for the nD COS method. Before we start with these experiments, we test our implementation in order to identify the most expensive functions of the nD COS method. In this way, we can get insight into the computational cost of these functions using the different considered programming languages. With this information we will be able to determine the fastest programming language for our purposes and see which functions can benefit from a parallel implementation. In sections 6.3, 6.4, 6.5 and 6.6, we discuss the parallel strategy for the four parts of the method with higher computational cost. We test every part in 1D, 2D and 3D for different numbers of N and Q. We start with an introduction.

6.1 Introduction

For our numerical experiments with the COS method we use three different programming environments, namely MATLAB, C and CUDA. We use MATLAB because the original implementation of COS employed this programming language; MATLAB is our reference. Two important properties of MATLAB are the availability of different libraries and the fast calculations with matrices. Moreover, it is a user-friendly script language. We use C because this programming language has become one of the most widely used languages over the years. Some essential characteristics of C are: C code has to be compiled before it can be run, it is low-level and it is statically typed. Therefore, we expect C to be faster than MATLAB. The programming environments C and MATLAB run on a central processing unit (CPU). A CPU consists of a few cores optimized for serial processing. Serial means that the calculations are performed sequentially. Serial processing is suitable for calculations which are dependent as well as independent. In computational finance, many computations are independent. Thus they are also suitable for parallelization. The use of parallel computing has grown in recent years. In parallel computing multiple computations are performed at the same time. As a consequence, these calculations must be independent of each other. Years ago, mainly supercomputers were able to benefit from parallel computing. These computers with massive numbers of CPUs were able to perform more calculations at the same time.

In 2006 NVIDIA released a compute unified device architecture (CUDA), a general-purpose computing platform and programming model. CUDA is a heterogeneous programming language (combining GPU and CPU code) which eases the development of parallel code for NVIDIA's GPUs. The *host*[1] code is executed directly on the CPU which interfaces to the GPU. The GPU code has to be written as a *kernel* which runs on the GPU. A kernel is a special function that can be defined in the CUDA framework. The instructions within a kernel will be executed in parallel. A kernel specifies the number of parallel items or *threads* by defining a *grid*. One grid can be divided into *thread blocks*. The threads in a grid concurrently execute the same kernel. Threads are grouped together into thread blocks. All the threads in a grid have access to global memory. Every thread block has per-block shared memory and every thread has its own private local memory. Every memory has its own characteristics such as speed and size.

[1]The word host indicates the CPU in CUDA.

A GPU has a parallel architecture consisting of thousands of smaller cores designed for handling multiple computations simultaneously. A GPU consists of several streaming multiprocessors. Each streaming multiprocessor executes a thread block from the grid in a parallel way. The thread blocks are automatically distributed over the streaming multiprocessors. As a consequence, the thread blocks are performed in an unknown order and have to be independent of each other. For more details [22, ch. 1.3],[29, 41].

The programmer specifies the number of threads in each block and the number of blocks in the grid. This flexibility allows the developer to choose the thread's organization according to the problem at hand. We have chosen our own organization of threads and blocks. For the sake of simplicity, our key decision is to launch one thread per element of the array of numbers. For example, for a 1D case, see Figure 6.1, this means that for a vector of 32 elements we launch 32 threads. We group these threads into 4 thread blocks (thus, each block consists of 8 threads). Thus, we can calculate the 32 elements of the vector simultaneously. Figure 6.1 also shows that each element has its own thread in a specific block. For instance, element 26 of the vector is calculated by Thread 2 of Block 3.

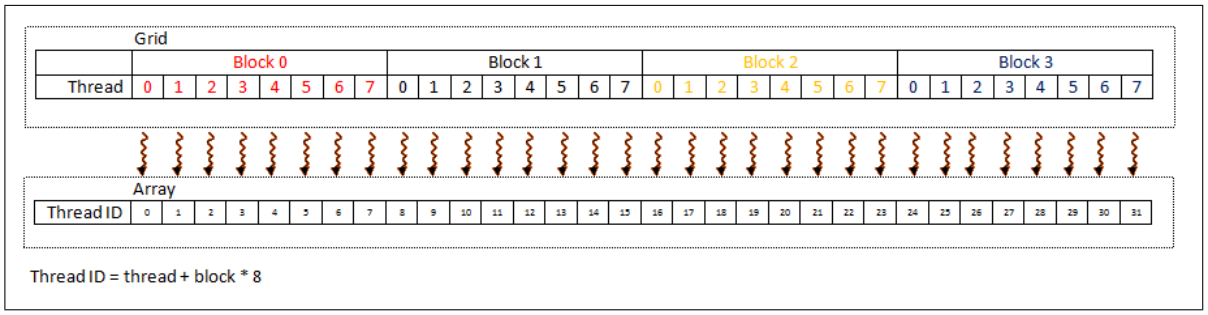

Figure 6.1: GPU calculations for a vector of length 32 with 4 blocks and 8 threads per block

As an example for the 2D case, we give in Figure 6.2 a schematic overview of a 2D grid consisting of four 2D blocks, with in every block 64 threads (8x8) and the corresponding matrix of size 16x16. For purposes of illustration: Thread(5,7) in Block(0,0) calculates element (5,7) of the Matrix, and Thread (6,2) in Block (1,1) performs the calculations of element (14,10) of the Matrix.

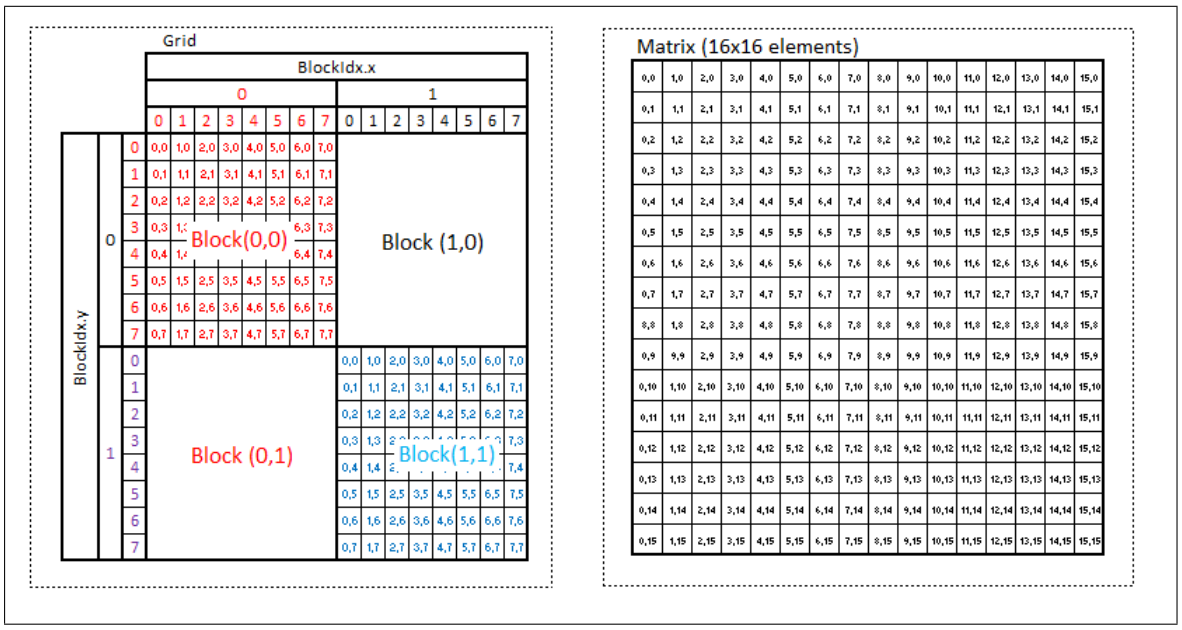

Figure 6.2: GPU calculations for a matrix of size 16x16 with 4 blocks and 64 threads per block

It is also possible to perform this method, one thread per element, for a 3D array. In that case, the grid and the thread blocks have to be three-dimensional.
For the most time-consuming parts of this method we build a parallel implementation whereby every single thread calculates one specific element of the matrix. Then, we test our parallel implementation and the results will show if parallel computing really offers these time advantages.

For MATLAB and C experiments, we use a workstation with an 'Intel(R) Core(TM) i5-4670 (3.40GHz Cache size 6 MB)' CPU, and 8 GB RAM. The workstation runs on Linux SUSE 11 64-bit. For CUDA, we use the same computer supplemented with a 'NVIDIA K600' GPU card. Also for CUDA, we use a Beowulf-cluster ('Little Green Machine') with two 'Intel(R) Xeon(R) E5620 (2.40GHz Cache size 12 MB)' CPUs, 24 GB RAM and two 'NVIDIA GTX480' GPUs. The MATLAB version is 2014b, the C compiler is gcc 4.3.4 where the optimization is set to O3 and the CUDA toolkit version is 6.5.12.

6.2 Algorithm

In this section, we have rewritten the formula in Equation (5.15) according to the different steps involved in the expression.

Algorithm 1 nD COS method

1: **procedure** COS METHOD
2: **Inputs:** $n, N, Q \in \mathbb{N}_{\geq 1}$ and $Q \geq N$
3: option characteristics
4: payoff characteristics
5: model characteristics
6: characteristic function of model
7: **Output:** option value
8: **Calc:** $\vec{a}$, $\vec{b}$
9: **Calc:** covariance matrices
10: $P \leftarrow \text{payoff}(n, \vec{a}, \vec{b}, Q)$
11: $V \leftarrow \text{DCT}(P)$
12: $G \leftarrow \text{Charfunc}(n, \vec{a}, \vec{b}, N)$
13: **for** $i = 0$ **to** N^n **do**
14: $G_i \leftarrow G_i \cdot V_i$
15: $s \leftarrow \sum_i G_i$
16: **return** $\left(\frac{2}{Q}\right)^n \cdot \exp(-r \cdot \tau) \cdot s$

In Algorithm 1 these steps are presented. The most expensive parts are: payoff (line 10), DCT (line 11), Charfunc (line 12) and dot product (lines 13 to 15). We denote these parts by payoff array, DCT, characteristic function array and dot product. These parts consume the most calculation time because the number of calculations depends on Q and N respectively. For the two Calc functions, on the lines 8 and 9, their number of calculations depends on n which is far less than Q or N. The number of calculations of the return function, on line 16, is independent of n, Q and N. Therefore, the last three mentioned functions consume little time. In the next sections, we describe the expensive parts and their GPU parallel implementations.

6.3 Part I: Payoff array

The payoff function returns an array P, containing the payoff coefficients. The size of array P is Q^n and its calculation depends on vectors $\vec{a}$ and $\vec{b}$, the dimension n, the value of Q and the payoff characteristics. We have developed several payoff codes in MATLAB, C and CUDA, for European options, namely a 1D put-, 1D call-, 2D geometric basket-, 2D arithmetic basket-, 2D exchange- and 3D geometric basket option. As an example we present the three codes for the 2D geometric basket payoff function in Table 6.1. It is possible to show the other codes in a similar way, see appendix D.

2D MATLAB payoff function

```
function [ mat ] = payoffgeo2D( a,b,K,Q )
    x = linspace(a(1)+0.5*(b(1)-a(1))/Q,b(1)-0.5*(b(1)-a(1))/Q,Q);
    y = linspace(a(2)+0.5*(b(2)-a(2))/Q,b(2)-0.5*(b(2)-a(2))/Q,Q);
    [x1,y1] = ndgrid(x,y);
    mat = max(K*(1-sqrt(exp(x1+y1))),0);
end
```

2D C payoff function

```
void payoffgeo2D(double *h_A, double *h_a, double *h_b, double K, int Q)
{
    int x,y;
    int offset;
    double x2, y2;
    for (y=0;y<Q;y++)
    {
        for (x=0;x<Q;x++)
        {
            offset = x + y * Q;
            x2 = h_a[0] + (x + 0.5) * (h_b[0]-h_a[0])/Q;
            y2 = h_a[1] + (y + 0.5) * (h_b[1]-h_a[1])/Q;
            h_A[offset] = MAX(K*(1-sqrt(exp(x2+y2))),0.0);
        }
    }
}
```

2D CUDA payoff kernel

```
__global__ void payoff2DGPU(double *d_A, double *d_a, double *d_b, double K, int Q)
{
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    if(x<Q && y<Q)
    {
        int offset = x + y*Q;
        double x2 = d_a[0] + (x + 0.5) * (d_b[0]-d_a[0])/Q;
        double y2 = d_a[1] + (y + 0.5) * (d_b[1]-d_a[1])/Q;
        d_A[offset]=MAX(K*(1-sqrt(exp(x2+y2))),0.0);
    }
}
```

Table 6.1: Codes for 2D geometric payoff function

In our parallel implementation every element of the matrix is calculated independently by one CUDA thread. The kernel function returns the matrix P filled with the payoff coefficients, see Figure 6.2.
In Tables 6.2, 6.3 and 6.4 the execution times (ms) are presented for the 1D put-, 2D geometric put- and 3D geometric put payoff functions, respectively.

In this chapter, we use the following designation for every table:

- Speedup1: MATLAB CPU time divided by C CPU time;
- Speedup2: MATLAB CPU time divided by CUDA K600 GPU time;
- Speedup3: C CPU time divided by CUDA K600 GPU time.
- OOM: Some of the experiments cannot be performed due to memory limitations. This situation is denoted 'OOM' (out of memory).

Q	MATLAB	C	CUDA K600	Speedup1	Speedup2	Speedup3
64	<0.01	<0.01	0.02	1.00	0.50	0.50
128	0.02	<0.01	0.02	2.00	1.00	0.50
256	0.02	0.02	0.02	1.00	1.00	1.00
512	0.03	0.04	0.02	0.75	1.50	2.00
1024	0.04	0.08	0.02	0.50	2.00	4.00
2048	0.11	0.16	0.02	0.69	5.50	8.00
4096	0.13	0.32	0.03	0.41	4.33	10.67
8192	0.22	0.64	0.04	0.34	5.50	16.00

Table 6.2: Time (ms) to create payoff vector 1D put option and speedups

Q^2	MATLAB	C	CUDA K600	Speedup1	Speedup2	Speedup3
64^2	0.20	0.08	0.06	2.50	3.33	1.33
128^2	0.39	0.32	0.17	1.22	2.29	1.88
256^2	1.22	1.27	0.59	0.96	2.07	2.15
512^2	4.8	5.07	2.28	0.95	2.11	2.22
1024^2	16.3	20.3	9.04	0.81	1.80	2.25
2048^2	49.4	81.1	36.0	0.61	1.37	2.25
4096^2	199	324	144	0.62	1.38	2.25
8192^2	780	1297	OOM	0.60	-	-

Table 6.3: Time (ms) to create payoff matrix 2D geometric put option and speedups

Q^3	MATLAB	C	CUDA K600	Speedup1	Speedup2	Speedup3
32^3	1.93	1.5	0.4	1.29	4.83	3.75
64^3	8.86	11.6	2.9	0.76	3.06	4.00
128^3	70.3	92.2	23.4	0.76	3.00	3.94
256^3	560	736	187	0.76	2.99	3.94
512^3	OOM	5889	OOM	-	-	-

Table 6.4: Time (ms) to create payoff array 3D geometric put option and speedups

The test results show that CUDA implementations outperform, in general, MATLAB and C, see Speedup2 and Speedup3. The only exceptions are Speedup2 of 1D put at $Q = 64$ and Speedup3 of 1D put at $Q = 64$ and $Q = 128$. This is because the overhead costs of CUDA K600 are divided over a too small number of Q values.
The maximum obtained speedup is 16.00 when $Q = 8192$ for a 1D put. The speedup increases as Q increases which means that especially for higher Q values the GPU K600 works at its full potential.
We have also tested our developed payoff kernels on a more advanced GPU (GTX480). In Table 6.5 we present our results. These results show that CUDA GTX480 outperforms the CUDA K600 for all cases. Especially when higher Q values are employed, the GPU GTX480 works at its full potential, the obtained acceleration is significant. The GTX480 card has more memory so larger Q values can be employed.

Q (1D)	GTX480	Speedup	Q^2 (2D)	GTX480	Speedup	Q^3 (3D)	GTX480	Speedup
64	<0.01	2.00	64^2	0.03	2.00	32^3	0.08	5.00
128	<0.01	2.00	128^2	0.05	3.40	64^3	0.33	8.79
256	<0.01	2.00	256^2	0.08	7.38	128^3	2.44	9.59
512	<0.01	2.00	512^2	0.25	9.12	256^3	19.22	9.73
1024	<0.01	2.00	1024^2	0.91	9.93	512^3	OOM	-
2048	<0.01	2.00	2048^2	3.54	10.17			
4096	<0.01	3.00	4096^2	14.07	10.23			
8192	0.01	4.00	8192^2	56.16	-			

Table 6.5: Time (ms) to perform the payoff kernel and comparison GTX480 vs K600

6.4 Part II: DCT

The computation of the DCT function returns an array V containing the DCT of array P. The size of V is Q^n and its calculation depends on the payoff array P of size Q^n. For the MATLAB version, Andriy Myronenko has developed the function mirt_dctn. This function is an implementation for the DCT which is appropriate for all dimensions. Because the mirt_dctn is faster than the available 1D DCT and 2D DCT functions in MATLAB, we use Myronenko's function for 1D, 2D and 3D. The mirt_dctn function can be found on the Mathworks website [45].
For the C version, we use the fftw_exec function which can be found in the FFTW library [46]. This function can be used to calculate the DCT for 1D, 2D and 3D. This library was developed by Frigo and Johnson at the Massachusetts Institute of Technology in 2005.
For the CUDA version, we make use of the cuFFT library. The cuFFT library is part of the CUDA Toolkit. This library is the FFTW equivalent for CUDA. However, the cuFFT library does not contain an implementation of the DCT function. Therefore, we have written codes for 1D DCT, 2D DCT and 3D DCT. We follow the mathematics of Makhoul [24], see appendix B. He shows that a DCT can be written as an FFT; the FFT is available in the cuFFT library for all three dimensions.

In our parallel implementation we allocate a new array as large as the input array. Then, we create as many threads as there are elements in the input array. Every thread shuffles (shuffle kernel): read one element from the input array and write it to its new location in the new array. Subsequently, an array for the output of the build-in FFT from the cuFFT library is created. Then, the real to complex FFT kernel is executed; the result is of size $Q/2 + 1$, see Makhoul [24]. Every element of the ouput array is calculated by one thread. Every thread multiplies an input element with exponential functions (multiplication kernel). The CUDA code of the 3D DCT can be found in Table 6.6. The codes for the 1D and 2D versions are less complex but similar.

CUDA DCT (3D)

help functions

```
__device__ int loc(int k, int Q)
{
    if(k<(Q/2)) return 2*k;
    else return 2*Q-2*k-1;
}
__device__ cuDoubleComplex Wab(double a, double b)
{
    cuDoubleComplex wab;
    wab = cuexp(make_cuDoubleComplex(0.0,-2.0*a*M_PI/b));
    return wab;
}
```

shuffle kernel

```
__global__ void shuffle3D(double *output, double *input, int Q)
{
    int k1 = blockIdx.x * blockDim.x + threadIdx.x;
    int k2 = blockIdx.y * blockDim.y + threadIdx.y;
    int k3 = blockIdx.z * blockDim.z + threadIdx.z;
    int offset1;
    int offset2;

    if(k1<Q && k2<Q && k3<Q)
    {
        offset1 = k1 + k2 * Q + k3 * Q * Q;
        offset2 = loc(k1,Q) + loc(k2,Q) * Q + loc(k3,Q) * Q * Q;
        output[offset1] = input[offset2];
    }
}
```

multiplication kernel

```
__global__ void multiply3D(double *output, cufftDoubleComplex *input, int Q, int N)
{
    int k1 = blockIdx.x * blockDim.x + threadIdx.x;
    int k2 = blockIdx.y * blockDim.y + threadIdx.y;
    int k3 = blockIdx.z * blockDim.z + threadIdx.z;

    int offsetN;
    int q2=Q/2+1;           //Q/2+1

    cuDoubleComplex ele;
    cuDoubleComplex wab1, wab2, wab2m, wab3, wab3m;
    if(k1<N && k2<N && k3<N)
    {
        offsetN= k1 + k2*N + k3*N*N;

        wab1 = Wab((double) k1,(double)4*Q);
        wab2 = Wab((double) k2,(double)4*Q);
        wab2m= Wab((double)-k2,(double)4*Q);
        wab3 = Wab((double) k3,(double)4*Q);
        wab3m= Wab((double)-k3,(double)4*Q);

        cuDoubleComplex V23, V23m, V2m3, V2m3m;
```

```
        if(k1<q2)
        {
V23  = cuCmul(wab2, cuCmul(wab3, input[k1+ k2       *q2+     k3   *q2*Q]));
V23m = cuCmul(wab2, cuCmul(wab3m,input[k1+ k2       *q2+((Q-k3)\%Q)*q2*Q]));
V2m3 = cuCmul(wab2m,cuCmul(wab3, input[k1+((Q-k2)\%Q)*q2+     k3   *q2*Q]));
V2m3m= cuCmul(wab2m,cuCmul(wab3m,input[k1+((Q-k2)\%Q)*q2+((Q-k3)\%Q)*q2*Q]));
        }



        else
        {
V23  = cuConj(cuCmul(wab2, cuCmul(wab3, input[(Q-k1)+ k2       *q2+     k3   *q2*Q])));
V23m = cuConj(cuCmul(wab2, cuCmul(wab3m,input[(Q-k1)+ k2       *q2+((Q-k3)%Q)*q2*Q])));
V2m3 = cuConj(cuCmul(wab2m,cuCmul(wab3, input[(Q-k1)+((Q-k2)%Q)*q2+     k3   *q2*Q])));
V2m3m= cuConj(cuCmul(wab2m,cuCmul(wab3m,input[(Q-k1)+((Q-k2)%Q)*q2+((Q-k3)%Q)*q2*Q])));
        }
        ele  = cuCmul(wab1,cuCadd(cuCadd(cuCadd(V23,V23m),V2m3),V2m3m));
        output[offsetN] = 2*ele.x;
    }
}
```

DCT wrapper

```
void dct3DFFTGPU(double *output, double *input, int Q, int N)
{
    // 3D-DCT makes use of the build-in complex to complex FFT.
    // input: double array of length QxQxQ.
    // output: double array of length NxNXN.
    int TpB = 512;
    dim3 TpB1D(TpB);
    dim3 BpG1D((Q*Q*Q+TpB-1)/TpB);

    TpB = 8;
    dim3 TpB3D(TpB,TpB,TpB);
    dim3 BpGQ3D((Q+TpB-1)/TpB,(Q+TpB-1)/TpB,(Q+TpB-1)/TpB);
    dim3 BpGN3D((N+TpB-1)/TpB,(N+TpB-1)/TpB,(N+TpB-1)/TpB);

    double *fftinput;
    cudaMalloc((void **)\&fftinput, Q*Q*Q*sizeof(fftinput[0]));
    cuDoubleComplex *c_out;
    cudaMalloc((void **)&c_out, (Q/2+1)*Q*Q*sizeof(c_out[0]));

    shuffle3D<<<BpGQ3D, TpB3D>>>(fftinput,input,Q);
    FFTd2z3D(c_out, fftinput, Q);
    multiply3D<<<BpGN3D,TpB3D>>>(output,c_out,Q,N);

    cudaFree(c_out);
    cudaFree(fftinput);
}
```

Table 6.6: Parallel CUDA implementation of the 3D DCT

In Table 6.7 we present the results of our experiments with the DCT in the different languages.

dim	Q^n	MATLAB	C	CUDA K600	Speedup1	Speedup2	Speedup3
1	256	0.10	0.09	0.71	1.11	0.14	0.13
1	512	0.11	0.13	0.89	0.85	0.12	0.15
1	1024	0.14	0.19	0.91	0.74	0.15	0.21
1	2048	0.19	0.33	0.91	0.58	0.21	0.36
1	4096	0.29	0.14	0.73	2.07	0.40	0.19
1	8192	0.51	0.30	0.96	1.70	0.53	0.31
1	16384	1.01	0.59	0.89	1.71	1.13	0.66
1	32768	1.97	1.20	1.27	1.64	1.55	0.94
2	64^2	0.23	0.30	0.80	0.77	0.29	0.38
2	128^2	0.61	1.20	1.10	0.51	0.55	1.09
2	256^2	1.92	0.88	2.66	2.18	0.72	0.33
2	512^2	7.24	3.59	8.10	2.02	0.89	0.44
2	1024^2	31.7	20.6	28.5	1.54	1.11	0.72
2	2048^2	142	116	115	1.22	1.23	1.01
2	4096^2	605	536	486	1.13	1.24	1.10
2	8192^2	2493	2582	OOM	0.97	-	-
3	32^3	1.5	0.38	2.28	3.95	0.66	0.17
3	64^3	9.3	4.39	12.4	2.12	0.75	0.35
3	128^3	86	67.7	93.2	1.27	0.92	0.73
3	256^3	785	672	755	1.17	1.04	0.89
3	512^3	OOM	1.36e04	OOM	-	-	-

Table 6.7: Time (ms) to perform the DCT and speedups

Note that for CUDA the P array is already in the memory of the GPU. This fact results in a performance improvement since the transfers between CPU and GPU can be avoided.

In Table 6.7, we firstly observe that MATLAB and C compete for 1D and 2D, and that C conquers its opponent for 3D, see column "Speedup1". Secondly, C outperforms CUDA K600: Speedup3 barely exceeds the ratio of 1, see column "Speedup3". Thirdly, the results show that speedups 2 and 3 increase with Q. This implies that CUDA performs better for large Q values. Lastly, a memory problem occurs for CUDA K600 at $Q^2 = 8192^2$ and at $Q^3 = 512^3$ for MATLAB and CUDA K600.

In Table 6.8 we present the results of the DCT kernel test performed on the GTX480 GPU. These results show that CUDA GTX480 always outperforms the CUDA K600 especially for 2D where $Q = 1024$ and higher and 3D where $Q = 128$ and higher.

Q (1D)	GTX480	Speedup	Q^2 (2D)	GTX480	Speedup	Q^3 (3D)	GTX480	Speedup
256	0.18	3.94	64^2	0.24	3.33	32^3	0.65	3.51
512	0.40	2.23	128^2	0.26	4.23	64^3	2.24	5.54
1024	0.18	5.06	256^2	0.82	3.24	128^3	12.0	7.77
2048	0.40	2.28	512^2	1.73	4.68	256^3	93.4	8.08
4096	0.20	3.65	1024^2	4.36	6.54	512^3	OOM	-
8192	0.42	2.29	2048^2	15.3	7.52			
16384	0.21	4.24	4096^2	61.5	7.90			
32768	0.45	2.82	8192^2	OOM	-			

Table 6.8: Time (ms) to perform the DCT kernel on the GTX480 and comparison GTX480 vs K600

6.5 Part III: Characteristic function array

Once the payoff array, P, and DCT computations are performed, the computation of the characteristic array is carried out. The size of the output array G is N^n and its calculation depends on vectors $\vec{a}$ and $\vec{b}$, the option characteristics and the ChF of the model. $G_{\vec{k}}$ reads

$$G_{\vec{k}} = \left(F^+_{\vec{k}}(\vec{x}) + \ldots + F^-_{\vec{k}}(\vec{x}) \right). \tag{6.1}$$

We have developed codes in order to calculate the elements $G_{\vec{k}}$ for 1D GBM, 2D GBM, 3D GBM, 1D MJD, 2D MJD and 3D MJD in the programming languages MATLAB, C and CUDA. Note that the implementation slightly differs from Equation (6.1) because we have to divide some elements of G by the factor two. These halves are linked to the special summations which are used in Equation (5.15).

Our parallel implementation for the characteristic array is similar to the payoff array implementation: we calculate every element in the matrix with one thread. Therefore, we have to calculate the number of threads. Every thread calculates its own unique thread ID. A thread ID can be one-, two- or three-dimensional depending on the number of dimensions of the characteristic function. A thread calculates the value of its characteristic function by means of its own thread ID. When the ID contains one value of zero in its dimensions, the characteristic function is multiplied by 0.5; when it contains two values of zero by 0.25; when it contains three values of zero by 0.125. These halves come from the summations of the nD COS formula, Equation (5.15) of section 5.1.
The calculation complexity of the characteristic function kernel is higher than the calculation complexity of the payoff kernel. The kernel of the characteristic function contains many more operations than the payoff kernel. In Table 6.9, we present our developed CUDA code for the 2D GBM characteristic matrix. The codes for 1D GBM, 3D GBM, 1D MJD, 2D MJD and 3D MJD are similar.

We have only tested 1D GBM, 2D GBM and 3D GBM, because MJD is an extension of GBM and therefore we do not expect substantially different results. The test results in Table 6.10 show that CUDA outperforms MATLAB and C. The maximum obtained speedup is more than 25 for 1D where $Q = 2048$. Also, these results show that MATLAB is faster than C in general.

We have not only tested the characteristic function kernel on the K600 GPU (Table 6.10), but also on the GTX480 GPU. In Table 6.11, we show the results of this test and the ratio between these two GPUs (columns "Speedup"). We observe that the use of the GTX480 GPU gives a major speedup, especially for 2D and 3D cases. For example, $N^2 = 1024^2$ K600 GPU 21.2 milliseconds and GTX480 GPU 3.45 milliseconds: Speedup factor is more than 6.

CUDA characteristic matrix (2D)

help functions

```
__device__ cuDoubleComplex chargbm2D(double u0, double u1, double t, double *logS0,
        double *mu, double *cov)
{
    return  cuCmul(cu_exp(make_cuDoubleComplex(0.0,u0*logS0[0]+u1*logS0[1])),
    cu_exp(make_cuDoubleComplex(-0.5*t*
            (u0*u0*cov[0]+u0*u1*(cov[1]+cov[2])+u1*u1*cov[3]),t*(u0*mu[0]+u1*mu[1]))));
}

__device__ double Fkgbm2D(double u0, double u1, double v0, double v1, double t,
        double *logS0, double *mu, double *cov)
{
    cuDoubleComplex value = cuCmul(chargbm2D(u0,u1, t, logS0,mu,cov),
            cu_exp(make_cuDoubleComplex(0.0,v0+v1)));
    return value.x;
}
```

kernel

```
__global__ void chargbm2DGPU_kernel(double *G, double *precomp, double *precompa,
         double t, double *logS0, double *mu, double *cov, int N)
{
    int k0 = threadIdx.x + blockDim.x * blockIdx.x;
    int k1 = threadIdx.y + blockDim.y * blockIdx.y;
    if(k0<N && k1<N)
    {
        int offset = k0 + k1 * N;
        double u0 = k0*precomp[0];
        double u1 = k1*precomp[1];
        double v0 = k0*precompa[0];
        double v1 = k1*precompa[1];
        G[offset] = Fkgbm2D(u0, u1, v0, v1, t, logS0, mu, cov)
                   +Fkgbm2D(u0,-u1, v0,-v1, t, logS0, mu, cov);

        if(k0==0) G[offset] *= 0.5;
        if(k1==0) G[offset] *= 0.5;
    }
}
```

Table 6.9: CUDA implementation for characteristic matrix (2D)

dim	N^n	MATLAB	C	CUDA K600	Speedup1	Speedup2	Speedup3
1	128	0.02	0.04	0.02	0.50	1.00	2.00
1	256	0.04	0.1	0.01	0.40	4.00	10.00
1	512	0.07	0.18	0.02	0.39	3.50	9.00
1	1024	0.13	0.36	0.02	0.36	6.50	18.00
1	2048	0.20	0.76	0.03	0.26	6.67	25.33
2	64^2	0.72	0.54	0.11	1.33	6.55	4.91
2	128^2	1.75	2.15	0.36	0.81	4.86	5.97
2	256^2	8.60	8.62	1.35	1.00	6.37	6.39
2	512^2	24.3	34.9	5.3	0.70	4.58	6.58
2	1024^2	91.6	140	21.2	0.65	4.32	6.60
3	32^3	8.11	9.47	1.72	0.86	4.72	5.51
3	64^3	50.1	72.8	12.7	0.69	3.94	5.73
3	128^3	382	573	97	0.67	3.94	5.91
3	256^3	3004	4610	763	0.65	3.94	6.04
3	512^3	OOM	37200	OOM	-	-	-

Table 6.10: Time (ms) to create the charasteristic array and speedups (GBM)

N (1D)	GTX480	Speedup	N^2 (2D)	GTX480	Speedup	N^3 (3D)	GTX480	Speedup
128	0.01	2.00	64^2	0.08	1.38	32^3	0.35	4.91
256	0.01	1.00	128^2	0.13	2.77	64^3	1.98	6.41
512	0.01	2.00	256^2	0.29	4.66	128^3	14.8	6.55
1024	0.01	2.00	512^2	0.91	5.82	256^3	117	6.52
2048	0.01	3.00	1024^2	3.45	6.14	512^3	OOM	-

Table 6.11: Time (ms) to create the charasteristic array on GTX480 and comparison GTX480 vs K600 (GBM)

6.6 Part IV: Dot product

The calculation of the dot product depends on array V of size N^n and the N^n sub-array of G. For MATLAB, a dot product is standard available. For C, a dot product is available in the BLAS library (Basic Linear Algebra Subprograms). We use a specific BLAS library. This library is part of the Intel MKL library, which is hand-optimized for our Intel CPU.
For CUDA, we use the dot product from the cuBLAS library. This library is the CUDA equivalent of the BLAS library. By using this library it is not possible to organize grid, blocks or threads in our own way. This organization is done by the built-in dot product itself.
In Table 6.12, we show the results of our experiments. Firstly, these results show that the dot product consumes relatively little time for all languages. Secondly, C outperforms CUDA up to $N^n = 64^3$. The CUDA implementation needs 128^3 or more elements to be faster than the C version. Finally, a memory problem also occurs in this table: for $N^n = 512^3$ the K600 card cannot execute the calculation (OOM).
Table 6.13 contains the results of the test of the dot product, performed on a GTX480. The experiments show that from $N^n = 256^2$ the GTX480 is faster, as expected. However, we also observe that the GTX480 is slower than the K600 up to $N^2 = 128^2$; a phenomenon we did not observe in the previous tests. There probably are differences in overhead costs between the K600 GPU and the GTX480 GPU. We note that because the dot product consumes relatively little time we expect that this slowdown will not substantially influence the overall performance of the GTX480.

N^n	MATLAB	C	CUDA K600	Speedup1	Speedup2	Speedup3
512	<0.01	<0.01	0.05	1.00	0.20	0.20
1024	<0.01	<0.01	0.05	1.00	0.20	0.20
2048	<0.01	<0.01	0.05	1.00	0.20	0.20
4096	<0.01	<0.01	0.05	1.00	0.20	0.20
8192	0.02	<0.01	0.06	2.00	0.33	0.17
64^2	<0.01	<0.01	0.05	1.00	0.20	0.20
128^2	0.03	<0.01	0.06	3.00	0.50	0.17
256^2	0.13	0.02	0.1	6.50	1.30	0.20
512^2	0.33	0.1	0.2	3.30	1.65	0.50
1024^2	2.07	0.83	0.86	2.49	2.41	0.97
32^3	0.06	0.01	0.08	6.00	0.75	0.13
64^3	0.28	0.10	0.20	2.80	1.40	0.50
128^3	4.17	1.70	1.53	2.45	2.73	1.11
256^3	38.6	13.5	11.4	2.86	3.39	1.18
512^3	306	108	OOM	2.83	-	-

Table 6.12: Time (ms) to perform the dot product of N^n elements and speedups

N^n	GTX480	Speedup	N^2	GTX480	Speedup	N^3	GTX480	Speedup
512	0.08	0.63	64^2	0.08	0.63	32^3	0.08	1.00
1024	0.08	0.63	128^2	0.09	0.67	64^3	0.12	1.67
2048	0.08	0.63	256^2	0.10	1.00	128^3	0.41	3.73
4096	0.09	0.56	512^2	0.12	1.67	256^3	1.97	5.79
8192	0.09	0.67	1024^2	0.27	3.19	512^3	OOM	-

Table 6.13: Time (ms) to perform the dot product kernel up N^n elements on the GTX480 and comparison GTX480 vs K600

6.7 Conclusion

In this chapter, we have presented our parallel implementation of the nD COS method. A detailed explanation of the different parallelized parts of the code was given. Regarding the performance, our test results show that the differences in calculation time of MATLAB and C are smaller than expected. The DCT and the dot product can be calculated faster with C than with MATLAB. In contrast, the payoff array and the characteristic function array are faster in MATLAB than in C. We did not expect this result. We guess that our C implementation of these parts is not fully optimized. Furthermore, the results show that the parts payoff array and characteristic function array perform excellent when calculated in parallel on the GPU. For large Q values the payoff array speeds up to a factor of 16 (Table 6.2 1D, $Q = 8192$) and for large N values the characteristic function array to a factor of 25 (Table 6.10 1D, $N = 2048$).

We also perform the parallel kernels on a faster GPU (GTX480) which results in an additional speedup between 2 and 10 compared to the GPU (K600). Combining the maximum speedup 25.33 in Table 6.10 (1D, $N = 2048$) and the speedup 3 in Table 6.11 (1D, $N = 2048$), the maximum speedup with CUDA GTX480 reaches to a factor of $(3 \cdot 25.33 =)$ 76. Note that although the experiments with the GPU (GTX480) were performed on a different machine (Little Green Machine), we expect a similar speedup on any machine since the time interacting with the CPU is negligible.

The GPUs suffer from a limited amount of available memory. MATLAB too has its memory limitations; therefore some large problems only can be calculated with C. In general, for larger problems the GPUs work at their full potential. Therefore, we expect that calculating option values with the nD COS method in high dimensions will be very suitable for parallelization.

Chapter 7

Numerical results

7.1 Introduction

In this chapter, we use the COS method for performing numerical calculations on two models in several dimensions. These models are a correlated GBM and a correlated MJD. We perform these calculations in three programming languages, namely MATLAB, C and CUDA. For these languages we have written specific codes. The code of MATLAB is recorded in Appendix E and the relevant parts of the codes of C and CUDA are recorded in the appendices F and G.
We mainly use the Black-Scholes method and a Monte Carlo simulation to obtain reference values. We test various European payoffs, namely a call option, a geometric basket put option, an arithmetic basket call option and an exchange option.

The Black-Scholes values are considered to be exact. Therefore, we calculate the reference values of the different European options with the Black-Scholes formula. We demonstrate that the COS method is (very) fast and (highly) accurate for one and medium-sized dimensions (1D, 2D and 3D). For these cases the programming language CUDA (with video card GTX480), as expected, is by far the fastest way to price these options.

7.2 GBM model

7.2.1 Parameter set and GBM cumulants

For our numerical calculations we use the following three parameter sets.

Set 1	$S(t) = 100$, $r = 0.02$, $q = 0$, $\Delta t = 0.1$, $\sigma = 0.4$.
Set 2	$\vec{S}(t) = [90, 110]'$, $r = 0.04$, $q = [0, 0]'$, $\Delta t = 1$, $\sigma = [0.2, 0.3]'$, $\rho = \begin{bmatrix} 1 & 0.25 \\ 0.25 & 1 \end{bmatrix}$.
Set 3	$\vec{S}(t) = [90, 100, 110]'$, $r = 0.04$, $q = [0, 0, 0]'$, $\Delta t = 1$, $\sigma = [0.2, 0.3, 0.25]'$ $\rho = \begin{bmatrix} 1 & 0.25 & 0.1 \\ 0.25 & 1 & 0.2 \\ 0.1 & 0.2 & 1 \end{bmatrix}$.
where $S(t)$ is the value of the asset at time t, and $\vec{S}(t)$ is a vector of the asset prices at time t, r is the risk-free rate, q is the continuous dividend, $\Delta t = T - t$ is the time to maturity, σ is the volatility and ρ is the correlation matrix.	
The cumulants for the GBM model are an adaptation of [26]. $\xi^1 = \Delta t \left(r - \frac{\sigma^2}{2} \right)$, $\xi^2 = \Delta t(\sigma^2)$, $\xi^4 = 0$.	

7.2.2 Tests

7.2.2.1 One-dimensional tests

In this subsection we price a 1D European call option under GBM and use parameter set 1.

We will take three different strike prices K, 80, 100 and 120. The test results are given in Tables 7.1, 7.2 and 7.3. Table 7.1 shows the reference values, Table 7.2 the results of the Monte Carlo simulation and Table 7.3 the results of the COS method. The time differences in MATLAB between the three methods show that the Monte Carlo simulation consumes substantially more time than the other methods. For example $K = 80$: the Black-Scholes model consumes 0.06 milliseconds CPU time. With $N = 64$ the COS method takes 0.08 milliseconds CPU time; and with $\hat{M} = 10^6$ the Monte Carlo simulation takes 22 milliseconds CPU time. Moreover, the Monte Carlo technique has a substantially larger absolute error. We assume that these findings do not alter for C and CUDA. Therefore, we do not use this simulation technique for 2D and 3D dimensions if an analytical solution is available.

Black-Scholes values of call option			
K	reference value	error	MATLAB time (ms)
80	20.32926862429101	<1e-15	0.06
100	5.138393968748168	<1e-15	0.06
120	0.478986995446254	<1e-15	0.06

Table 7.1: Option value calculation with Black-Scholes formula

	time(ms)	$K = 80$		$K = 100$		$K = 120$	
$\hat{M}$	MATLAB	abs error	std error	abs error	std error	abs error	std error
1000	6.4e-02	2.0e-01	1.5e-01	1.3e-01	9.7e-02	1.3e-02	1.0e-02
10000	2.2e-01	6.5e-02	4.8e-02	4.2e-02	3.1e-02	4.2e-03	3.2e-03
100000	1.6	2.0e-02	1.5e-02	1.3e-02	1.0e-02	1.3e-03	9.9e-04
1000000	22	6.1e-03	4.8e-03	4.2e-03	3.3e-03	4.5e-04	3.3e-04

Table 7.2: Time and error when pricing a European call option with Monte Carlo simulation

	abs error			calculation time (ms)			
N	K=80	K=100	K=120	MATLAB	C	CUDA K600	CUDA GTX480
16	3.77e-03	1.07e-02	1.12e-02	0.02	<0.01	0.09	0.14
32	1.53e-07	1.51e-07	1.94e-08	0.05	<0.01	0.09	0.14
64	1.42e-14	7.11e-15	1.95e-14	0.08	0.01	0.09	0.12
128	1.42e-14	7.11e-15	1.95e-14	0.10	0.11	0.26	0.35
256	1.42e-14	7.11e-15	1.95e-14	0.21	0.24	0.25	0.35
512	1.42e-14	7.11e-15	1.95e-14	0.47	0.09	0.26	0.36
1024	1.42e-14	7.11e-15	1.95e-14	0.91	0.17	0.12	0.12
2048	1.42e-14	7.11e-15	1.95e-14	1.78	0.34	0.32	0.35
4096	1.42e-14	7.11e-15	1.95e-14	3.39	0.75	0.41	0.35

Table 7.3: Time and error when pricing a European call option with the COS method, $L = 10$

In Table 7.2, $\hat{M}$ is the number of simulations of the value of the stocks paths under GBM. For the 1D case, it is not necessary to simulate paths, because the analytic solution of the GBM SDE is known, see Formula (7.1). We can suffice by simulating the points at maturity time directly (the end points of the paths), by simulating $W(\Delta t)$.

$$S(T) = S(t) \exp\left(\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma W(\Delta t)\right) \tag{7.1}$$

We perform one thousand runs for every $\hat{M}$, so we can accurately calculate the mean of the absolute error and the standard deviation of the absolute error. We note that the CPU times of MATLAB is timed for one run. The test results show that that the standard deviation of the absolute error decays with $1/\sqrt{\hat{M}}$.

The test results as shown in Table 7.3 make clear that the 1D COS method is very fast and can highly accurately reproduce the exact values of the Black-Scholes formula, under the GBM process. We note that Fang and Oosterlee [17] and Fang [16] have already shown these excellent properties of the 1D COS method under GBM. But these authors limited themselves to tests with MATLAB. However, our numerical experiments also include tests with C and different CUDAs. We show that the computation time of the COS method can be mitigated substantially by using C and CUDA (for high N). For example, for low $N = 16$: MATLAB 0.02ms, C <0.01ms and CUDA K600 0.09ms. And $N = 512$: MATLAB 0.47ms, C 0.09ms (the speedup factor exceeds 5) and CUDA 0.26ms (the speedup factor is 1.8). For $N = 4096$: MATLAB 3.39ms, C 0.75ms (speedup factor 4.5) and CUDA K600 0.41ms (speedup factor 8.3). CUDA shows its potential at very high N ($N = 4096$), more than 8 times faster as MATLAB.

The test results of MATLAB as shown in Table 7.3 confirm that the calculation complexity is linear. In Table 7.4, we show this linear complexity in MATLAB (by doubling N the time differences also double).

N	MATLAB	differences between subsequent times	ratio
128	0.10	-	-
256	0.21	0.11	-
512	0.47	0.26	2.36
1024	0.91	0.44	1.69
2048	1.78	0.87	1.98
4096	3.39	1.61	1.85

Table 7.4: Time, difference and ratio for 1D COS

Figure 7.1 confirms that the convergence of the absolute errors from Table 7.3 is exponential. With Fang and Oosterlee [17], we observe that with $N = 2^6 = 64$ the 1D COS results correspond with the reference price in double precision. We also observe that the error convergence rate is similar for the different strike prices.

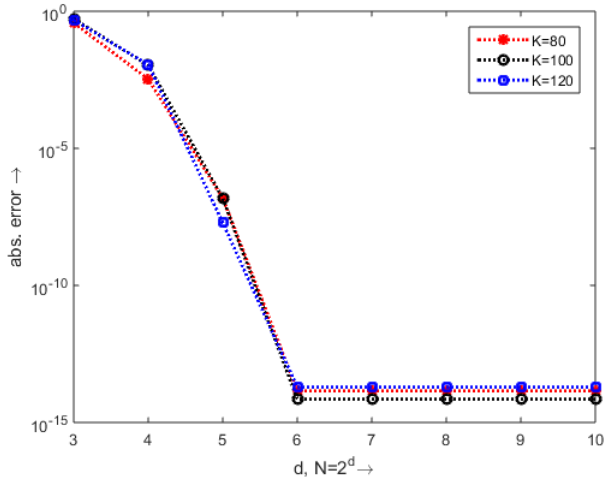

Figure 7.1: Error convergence for pricing European call options (1D COS)

7.2.2.2 Two-dimensional tests

In this subsection we price a 2D European geometric basket put option, a European arithmetic call option and a European exchange option, under GBM. We use parameter set 2. We compare the COS method with Black-Scholes or Monte Carlo simulation. The results of this experiment show that the 2D COS method is also fast and highly accurate, for GBM.

7.2.2.2.1 Geometric basket put option A geometric basket put option is an option whose payoff depends on the geometric average of several underlying stocks. This basket option has payoff function

$$g(\vec{S}(T)) = \left(K - \sqrt{S_1(T)S_2(T)}\right)^+,$$
$$g(\hat{y}) = \left(K(1 - \sqrt{\exp(\hat{y}_1)\exp(\hat{y}_2)}\right)^+.$$

If the underlying assets are GBM and the payoff is a geometric basket payoff, then an analytic formula, an extended Black-Scholes formula with continuous yield dividends, can be derived for the option value, which reads

$$v(\vec{S}(t),t) = \exp(-\hat{q}\Delta t)\hat{S}N(\hat{d}_1) - e^{-r\Delta t}KN(\hat{d}_2) \qquad \text{(call)} \qquad (7.2)$$
$$v(\vec{S}(t),t) = e^{-r\Delta t}KN(-\hat{d}_2) - \exp(-\hat{q}\Delta t)\hat{S}N(-\hat{d}_1) \qquad \text{(put)} \qquad (7.3)$$

where

$$\hat{d}_1 = \frac{\ln(\hat{S}/K) + \left(r - \hat{q} + \frac{\hat{\sigma}^2}{2}\right)\Delta t}{\hat{\sigma}\sqrt{\Delta t}}, \qquad \hat{d}_2 = \hat{d}_1 - \hat{\sigma}\sqrt{\Delta t}, \qquad \hat{S} = \prod_{j=1}^{n} S_j(t)^{1/n},$$
$$\hat{\sigma} = \frac{\sqrt{\sum_{i,j}\sigma_i\sigma_j\rho_{ij}}}{n}, \qquad \hat{q} = \frac{\sum_{i=1}^{n}\left(q_i + \frac{1}{2}\sigma_i^2\right)}{n} - \frac{\hat{\sigma}^2}{2}.$$

We refer to [23, 33]. The value of the call option $v(\vec{S}(t),t)$ is given by stock price $\hat{S}$, volatility $\hat{\sigma}$ and dividend rate $\hat{q}$. For the initial conditions above

$$\hat{S}_0 = \sqrt{90}\sqrt{110} = 30\sqrt{11}, \quad \hat{\sigma} = \sqrt{0.16}/2 = 0.2, \quad \hat{q} = (0.065 - 0.04)/2 = 0.0125.$$

We test two different strike prices, $K = 100$ and 200. The test results are given in Tables 7.5 and 7.6.

Black-Scholes values of 2D geometric basket put			
K	reference value	error	MATLAB time (ms)
100	6.696961159991261	<1e-15	11.2
200	93.897977093048610	<1e-15	10.2

Table 7.5: Reference values for 2D European geometric basket put options with Black-Scholes formula

Monte Carlo		$K = 100$		$K = 200$	
$\hat{M}$	time (ms)	abs error	std error	abs error	std error
1000	20	0.11	0.42	5.3e-04	5.6e-03
10000	193	0.03	0.13	1.8e-04	2.6e-03
100000	1925	6.7e-05	4.1e-02	6.0e-05	7.8e-04
1000000	19292	1.3e-03	1.2e-02	4.1e-05	2.1e-04

Table 7.6: Time and error when pricing a European geometric basket put option with a Monte Carlo simulation (100 runs)

For the 2D COS method we take three different N values, $16, 32, 64$, $N = N_1 = N_2$ and five different Q values, $64, 128$, 256, 512 and 1024. The test results are given in Tables 7.7 and 7.8.

$K = 100$	$N^2 = 16^2$				
Q^2	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^2	1.07e-03	0.6	0.20	1.27	0.98
128^2	4.48e-04	1.2	0.62	1.35	0.98
256^2	4.38e-04	3.4	2.45	2.30	1.48
512^2	3.63e-04	11.2	9.89	5.45	1.94
1024^2	3.81e-04	47.7	44.6	19.0	3.38
$N^2 = 32^2$					
64^2	1.37e-03	0.9	0.33	1.15	0.78
128^2	9.88e-05	1.4	0.76	1.48	0.78
256^2	7.46e-05	3.3	2.58	1.95	1.26
512^2	4.34e-06	11.7	10.0	5.33	1.72
1024^2	1.23e-05	46.9	44.4	18.9	3.17
$N^2 = 64^2$					
64^2	1.37e-03	1.3	0.90	1.32	0.78
128^2	9.87e-05	1.8	1.34	1.56	0.98
256^2	7.46e-05	4.1	3.15	2.25	1.47
512^2	4.39e-06	12.0	10.6	5.64	1.93
1024^2	1.22e-05	45.4	44.8	19.2	3.44

Table 7.7: Error and time 2D European geometric basket put option ($K = 100$)

$K = 200$	$N^2 = 16^2$				
Q^2	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^2	5.86e-04	0.8	0.21	1.27	0.96
128^2	6.52e-04	1.1	0.66	1.35	0.98
256^2	6.67e-04	3.1	2.61	2.30	1.51
512^2	6.71e-04	10.2	10.5	5.44	1.97
1024^2	6.71e-04	45.1	47.0	19.0	3.50
$N^2 = 32^2$					
64^2	5.84e-06	0.8	0.35	1.15	0.79
128^2	2.45e-06	1.3	0.81	1.47	0.79
256^2	2.53e-07	3.3	2.75	1.95	1.24
512^2	2.06e-07	11.1	10.7	5.32	1.77
1024^2	8.48e-08	47.0	47.0	18.9	3.26
$N^2 = 64^2$					
64^2	5.92e-06	1.4	0.95	1.32	0.80
128^2	2.36e-06	1.9	1.40	1.56	0.97
256^2	3.48e-07	4.2	3.35	2.25	1.45
512^2	1.10e-07	10.8	11.3	5.64	1.92
1024^2	1.04e-08	47.4	47.4	19.2	3.43

Table 7.8: Error and time 2D European geometric basket put option ($K = 200$)

Also, for the 2D case we observe that the COS method can quickly and accurately reproduce the exact values of Black-Scholes. Errors of size e-06, e-07 and e-08 are no exception. A few examples:

- the absolute error is 4.34e-06, which is reached by $K = 100$, $N^2 = 32^2$ and $Q^2 = 512^2$. The MATLAB CPU time for this accuracy is 11.7 milliseconds; the C CPU time is 10.0 milliseconds; the CUDA K600 time is 5.33 milliseconds.
- the absolute error is 8.48e-08, which is reached by $K = 200$, $N^2 = 32^2$ and $Q^2 = 1024^2$. The MATLAB CPU time for this accuracy is 47.0 milliseconds; the C CPU time is 47.0 milliseconds; the CUDA K600 time is 18.9 milliseconds.

We also observe that CUDA GTX480 outperforms CUDA K600. For $N^2 = 32^2$, $Q^2 = 1024^2$ and $K = 100$ the speedup factor is almost 6.

Appendix C shows that, under the assumption that the V_{k_1,k_2} terms are exact and the integration domain $[a_1, b_1] \times [a_2, b_2]$ is sufficiently wide, the ϵ^2 will dominate the overall error. In that case, for smooth density functions, the overall error ϵ converges exponentially in N. In Figure 7.2, left side, we plot the absolute errors for various N with a fixed value of Q, $Q = 2048$, and observe that the overall error convergences exponentially in N, as expected.
This appendix also shows that, under the assumption that the V_{k_1,k_2} terms are not exact, ϵ^{DCT} is present and dominates the absolute error. This error converges algebraically in the number of terms Q with order two. In Figure 7.2, right side, we plotted the absolute errors for various Q with a fixed value of N, $N = 40$. We observe that the overall error convergences algebraically in Q, as expected.

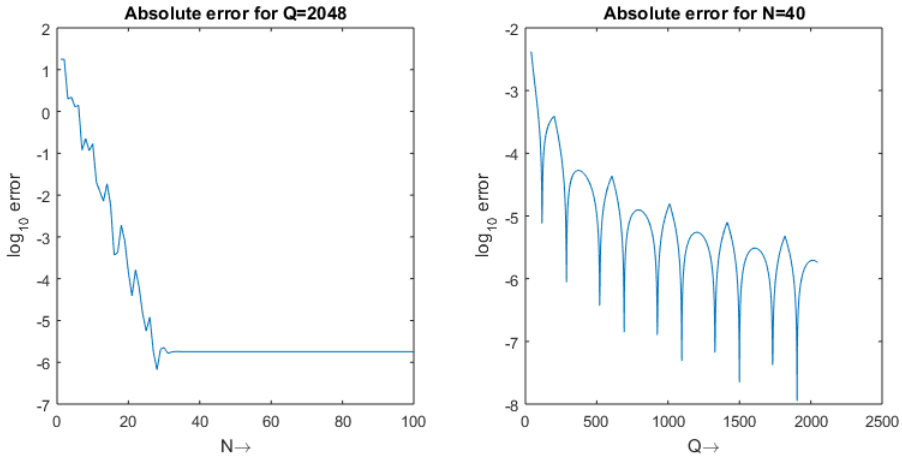

Figure 7.2: Error convergence for pricing a 2D European geometric put option (K=100)

The results in Table 7.9 confirm that the calculation complexity of the 2D COS method is quadratic in C. By doubling the Q the ratio is four.

$K = 200$	$N^2 = 16^2$		
Q^2	C	difference between subsequent timings	ratio
64^2	0.21	-	-
128^2	0.66	0.45	-
256^2	2.61	1.95	4.33
512^2	10.5	7.89	4.05
1024^2	47.0	36.5	4.63
$N^2 = 32^2$			
64^2	0.35	-	-
128^2	0.81	0.46	-
256^2	2.75	1.94	4.22
512^2	10.7	7.95	4.10
1024^2	47.0	36.3	4.57
$N^2 = 64^2$			
64^2	0.95	-	-
128^2	1.40	0.45	-
256^2	3.35	1.95	4.33
512^2	11.3	7.95	4.08
1024^2	47.4	36.1	4.54

Table 7.9: Time analysis of 2D COS geometric basket European put option

7.2.2.2.2 Arithmetic basket call option An arithmetic basket call option is an option whose payoff depends on the arithmetic average of several underlying stocks. This basket call option has payoff function

$$g(\vec{S}(T)) = \left(\frac{S_1(T) + S_2(T)}{2} - K\right)^+,$$
$$g(\hat{y}) = \left(K\left(\frac{\exp(\hat{y}_1) + \exp(\hat{y}_2)}{2} - 1\right)\right)^+.$$

It is not possible to calculate the value of an arithmetic basket call option analytically. We prefer to compare the results of the 2D COS method to a reference value which is determined by another method,

namely Monte Carlo simulation. With this simulation we have calculated a reference value of 10.1732612. We note that this value is the mean of 1,000 runs consisting of $\hat{M} = 100,000,000$ paths.

Monte Carlo simulation value of arithmetic basket call option			
reference value	standard deviation	error	MATLAB time (ms)
10.1732612	0.00148	1e-05	6844000

Table 7.10: Reference value for arithmetic basket call option (GBM)

$K = 100$	$N^2 = 16^2$				
Q^2	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^2	1.80e-01	0.6	0.24	1.26	0.94
128^2	1.83e-01	1.1	0.81	1.33	0.94
256^2	1.84e-01	3.4	3.18	2.05	1.45
512^2	1.84e-01	11.3	12.8	5.78	1.84
1024^2	1.84e-01	47.0	55.8	20.3	3.59
$N^2 = 32^2$					
64^2	5.47e-04	0.8	0.38	1.39	0.75
128^2	4.96e-04	1.3	0.94	1.45	0.76
256^2	5.08e-05	3.6	3.31	2.14	1.24
512^2	5.05e-05	11.8	13.0	5.66	1.70
1024^2	3.46e-05	47.3	56.0	20.3	3.24
$N^2 = 64^2$					
64^2	5.44e-04	1.3	0.94	1.57	0.76
128^2	4.92e-04	1.8	1.50	1.55	0.97
256^2	5.49e-05	4.0	3.88	2.50	1.45
512^2	4.64e-05	13.9	13.6	5.97	1.90
1024^2	3.05e-05	47.5	56.6	20.6	3.48

Table 7.11: Error and time 2D European arithmetic basket call option

For this 2D arithmetic option, we observe that the COS method reproduces the value of the Monte Carlo simulation quick and accurate. For example, the absolute error is 5.08e-05, which is reached by $N^2 = 32^2$ and $Q^2 = 256^2$. The MATLAB CPU time for this accuracy is 3.6 milliseconds; the C CPU time is 3.31 milliseconds; the CUDA K600 time is 2.14 milliseconds. Note that CUDA GTX480 outperforms CUDA K600 especially for high Q values. For example, when $N^2 = 64^2$ and $Q^2 = 1024^2$ the speedup is almost 6.

$K = 100$	N^2		
Q^2	16^2	32^2	64^2
64^2	1.80e-01	5.16e-04	5.12e-04
128^2	1.83e-01	4.65e-04	4.61e-04
256^2	1.84e-01	8.20e-05	8.61e-05
512^2	1.84e-01	1.93e-05	1.52e-05
1024^2	1.84e-01	3.37e-06	7.38e-07

Table 7.12: Error 2D European arithmetic basket call option

We note that Ruijter and Oosterlee [33] also tested the 2D European arithmetic basket call option. They used the reference value 10.173230, obtained by using the 2D COS method for $Q^2 = 5000^2$ and $N^2 = 100^2$. When we use this value instead of the value of our Monte Carlo simulation the test results shown in Table 7.12 give a higher accuracy.

7.2.2.2.3 Exchange option An European exchange option between two stocks is the right given to someone to exchange stock two for stock one at maturity time. This basket option has payoff function:

$$g(\vec{S}(T)) = (S_1(T) - S_2(T))^+,$$
$$g(\hat{y}) = (K(\exp(\hat{y}_1) - \exp(\hat{y}_2)))^+.$$

An analytic solution, based on the Black-Scholes formula, is available for an exchange option, namely Margrabe's formula[21, 25].

$$v(\vec{S}(t), t) = e^{-q_1 \Delta t} S_1(t) N(d_1) - e^{-q_2 \Delta t} S_2(t) N(d_2),$$
$$d_1 = \frac{\ln\left(\frac{S_1(t)}{S_2(t)}\right) + \Delta t\left(q_2 - q_1 + \frac{\hat{\sigma}^2}{2}\right)}{\hat{\sigma}\sqrt{\Delta t}},$$
$$d_2 = d_1 - \hat{\sigma}\sqrt{\Delta t}, \quad \hat{\sigma} = \sqrt{\sigma_1^2 + \sigma_2^2 - 2\sigma_1\sigma_2\rho}.$$

Exact values of exchange option		
reference value	error	time (ms) MATLAB
4.980814613075189	<1e-15	0.02

Table 7.13: Results for exchange option (GBM) value calculation with Margrabe's formula

$N^2 = 16^2$					
Q^2	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^2	6.18e-02	0.6	0.23	1.27	0.96
128^2	5.97e-02	1.1	0.74	1.35	0.96
256^2	6.07e-02	3.1	2.93	2.05	1.45
512^2	6.03e-02	11.0	11.8	5.73	1.85
1024^2	6.03e-02	49.5	51.8	20.2	3.48
$N^2 = 32^2$					
64^2	3.05e-03	0.8	0.36	1.38	0.75
128^2	3.11e-04	1.3	0.88	1.45	0.75
256^2	4.74e-04	4.0	3.09	1.93	1.24
512^2	6.23e-05	11.3	12.0	5.63	1.64
1024^2	1.40e-05	49.5	52.0	20.1	3.20
$N^2 = 64^2$					
64^2	3.05e-03	1.3	0.93	1.52	0.76
128^2	3.13e-04	1.9	1.44	1.52	0.96
256^2	4.72e-04	5.4	3.62	2.27	1.45
512^2	6.38e-05	15.1	12.5	5.95	1.86
1024^2	1.55e-05	57.0	52.4	20.4	3.47

Table 7.14: Error and time 2D European exchange option

In Table 7.14 we observe for this exchange option that the 2D COS method can quickly and accurately reproduce the exact values of Margrabe's formula. For example, the absolute error is 6.38e-05, which is reached by $N = 64$ and $Q = 512$. The MATLAB CPU time for this accuracy is 15.1 milliseconds; the C CPU time is 12.5 milliseconds; the CUDA K600 time is 5.95 milliseconds. Note again that CUDA GTX480 outperforms CUDA K600.

7.2.2.3 Three-dimensional tests

In this subsection we price a 3D European geometric basket put option, under GBM. We use parameter set 3. We compare the COS method with the Black-Scholes formula. The results of our experiment show that the 3D COS method is also highly accurate and especially for low Q values relatively fast, for GBM.

Geometric basket put option The payoff function of a three-dimensional geometric basket put option reads

$$g(\vec{S}(T)) = \left(K - \sqrt[3]{S_1(T)S_2(T)S_3(T)}\right)^+,$$
$$g(\hat{y}) = \left(K\left(1 - \sqrt[3]{e^{\hat{y}_1}e^{\hat{y}_2}e^{\hat{y}_3}}\right)\right)^+.$$

With formula 7.2 we calculate the reference values, which are shown in Table 7.15.

Black-Scholes values of a 3D European geometric basket put			
K	reference value	error	MATLAB time (ms)
75	0.205984525354658	<1e-15	0.90
150	46.264576516644198	<1e-15	0.92

Table 7.15: Value for 3D European geometric basket put option (GBM) calculated with the altered Black-Scholes formula

$K = 75$	$N^3 = 16^3$				
Q^3	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^3	4.61e-03	19.7	19.9	6.60	1.96
128^3	4.74e-03	161	179	46.1	6.79
160^3	4.75e-03	295	320	99.1	12.5
256^3	4.77e-03	1338	1535	376	47.1
$N^3 = 32^3$					
64^3	3.78e-05	25.8	30.5	9.77	2.53
128^3	4.41e-06	167	189	49.3	7.18
160^3	4.03e-07	303	330	102	12.9
256^3	3.94e-06	1346	1549	379	47.8
$N^3 = 64^3$					
64^3	3.79e-05	68.0	117	33.9	5.28
128^3	4.52e-06	210	276	73.3	9.94
160^3	5.07e-07	350	418	126	15.7
256^3	3.84e-06	1381	1630	403	50.5

Table 7.16: Error and time 3D European geometric basket put option

In Tables 7.16 and 7.17 we observe that the 3D COS method can accurately reproduce the exact values of Black-Scholes. But, the calculation in the higher dimension consumes more time. In this higher dimension the CUDA K600 outperforms MATLAB and C and CUDA GTX480 is much faster than CUDA K600. For example, the absolute error is 5.07e-07, which is reached by $K = 75$, $N = 64$ and $Q = 160$. The MATLAB CPU time for this accuracy is 350 milliseconds; the C CPU time is 418 milliseconds; the CUDA K600 time is 126 milliseconds. Note, CUDA GTX480 outperforms CUDA K600 by a factor 8.0.
The results in Table 7.18 confirm that the calculation complexity of the 3D COS method is cubic in C. By doubling the Q the ratio becomes eight.
Under the assumptions in section 5.4 the left-side plot of Figure 7.3 shows that the absolute error for various N with a fixed value of $Q = 256$ converges exponentially in N, as expected. The right-side plot of this figure shows that the absolute error for various Q with $N = 40$ convergences algebraically in Q, as expected.

$K = 150$	$N^3 = 16^3$				
Q^3	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^3	5.35e-03	19.9	22.7	6.60	1.95
128^3	5.46e-03	164	201	46.1	6.92
160^3	5.48e-03	296	359	99.2	13.1
256^3	5.49e-03	1337	1709	376	49.7
$N^3 = 32^3$					
64^3	2.38e-05	26.4	33.5	9.78	2.53
128^3	5.70e-06	167	211	49.4	7.50
160^3	4.86e-06	307	370	102	13.6
256^3	1.18e-06	1343	1718	379	50.4
$N^3 = 64^3$					
64^3	2.39e-05	65.2	122	34.0	5.27
128^3	5.81e-06	206	299	73.4	10.3
160^3	4.75e-06	342	458	126	16.4
256^3	1.29e-06	1386	1809	405	53.2

Table 7.17: Error and time 3D European geometric basket put option

$K = 75$	$N^3 = 16^3$		
Q^3	C	difference between subsequent times	ratio
64^3	19.9		
128^3	179	159.1	
256^3	1535	1356	8.5
$N^3 = 32^3$			
64^3	30.5		
128^3	189	158.5	
256^3	1549	1360	8.6
$N^3 = 64^3$			
64^3	117		
128^3	276	159	
256^3	1630	1354	8.5

Table 7.18: Error and time 3D European geometric basket put option

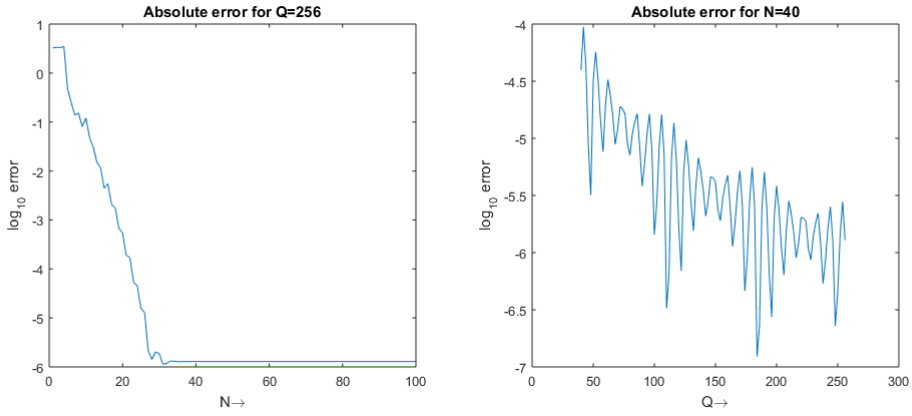

Figure 7.3: Error convergence for pricing a 3D European geometric put option (K=150)

7.3 MJD model

7.3.1 Parameter set and MJD cumulants

For our numerical calculations we use the following three parameter sets.

Set 1	$S(t)$=100, r=0.02, $q=0$, Δt=1, σ=0.4, α=0, δ=0.15, λ=5.
Set 2	$\vec{S}(t)=[90,110]'$, $r=0.04$, $q=[0,0]'$, $\Delta t=1$, $\sigma=[0.2,0.3]'$, $\alpha=[0,0.05]'$, $\delta=[0.1,0.2]'$, $\lambda=4$, $\rho=\begin{bmatrix}1 & 0.25\\ 0.25 & 1\end{bmatrix}$, $\rho^J=\begin{bmatrix}1 & -0.2\\ -0.2 & 1\end{bmatrix}$.
Set 3	$\vec{S}(t)=[90,100,110]'$, $r=0.04$, $q=[0,0,0]'$, $\Delta t=1$, $\sigma=[0.2,0.3,0.25]'$, $\alpha=[0,0.05,-0.03]'$, $\delta=[0.1,0.2,0.15]'$, $\lambda=4$, $\rho=\begin{bmatrix}1 & 0.25 & 0.1\\ 0.25 & 1 & 0.2\\ 0.1 & 0.2 & 1\end{bmatrix}$, $\rho^J=\begin{bmatrix}1 & 0.5 & 0.5\\ 0.5 & 1 & 0.5\\ 0.5 & 0.5 & 1\end{bmatrix}$.
where $S(t)$ is the value of the asset at time t, and $\vec{S}(t)$ is a vector of the asset prices at time t, r is the risk-free rate, q is the continuous dividend, $\Delta t=T-t$ is the time to maturity, σ is the volatility, α is the mean of the jump, δ is the volatility of the jump, λ is the jump intensity, ρ is the correlation of the assets and ρ^J is the correlation of the jumps.	
The cumulants for the MJD model are an adaptation of [26]. $\xi^1=\Delta t\left(r-\frac{\sigma^2}{2}-\lambda k+\lambda\alpha\right)$, $\xi^2=\Delta t\left(\sigma^2+\lambda\delta^2+\lambda\alpha^2\right)$, $\xi^3=\Delta t\lambda\left(3\delta^2\alpha+\alpha^3\right)$, $\xi^4=\Delta t\lambda\left(3\delta^4+6\alpha^2\delta^2+\alpha^4\right)$.	
Skewness $=\frac{\xi^3}{\xi^2\sqrt{\xi^2}}=\frac{\Delta t\lambda\left(3\delta^2\alpha+\alpha^3\right)}{\Delta t\sqrt{\Delta t}\left(\sigma^2+\lambda\delta^2+\lambda\alpha^2\right)^{3/2}}$; excess kurtosis $=\frac{\xi^4}{\xi^2\xi^2}=\frac{\Delta t\lambda\left(3\delta^4+6\alpha^2\delta^2+\alpha^4\right)}{\Delta t^2\left(\sigma^2+\lambda\delta^2+\lambda\alpha^2\right)^2}$. We note that in the case $\lambda=0$ (GBM) then there is no skewness nor excess kurtosis.	

7.3.2 Tests

7.3.2.1 One-dimensional tests

In this subsection we price a 1D European call option under MJD and use parameter set 1.

For this experiment we use three different strike prices K, 50, 100 and 200. The test results are given in Tables 7.19 and 7.20. Table 7.19 also shows the reference values. The test results as shown in Table 7.20 make clear that the 1D COS method reproduces very quickly and precisely the exact values of the MJD formula. The time differences between the MJD formula in Table 7.19 and the COS method in Table 7.20 measured in MATLAB CPU time show that the COS method is substantially faster than the MJD formula. Although the error of the MJD formula is smaller than the error of the COS method, the difference between these errors is negligible.
We observe that the computation time of the COS method can be mitigated substantially by using C and CUDA (for high N). For example, for $N=32$: MATLAB 0.06 ms, C 0.03ms (speedup factor 2). And for $N=2048$: MATLAB 1.76 ms, C 0.40ms (speedup factor 4.4) and CUDA K600 0.33ms (speedup factor 5.3).

MJD values of call option			
K	reference value	error	MATLAB time (ms)
50	52.397318140114308	<1e-15	9.5
100	21.345306404263468	<1e-15	9.5
200	3.451431536462926	<1e-15	9.6

Table 7.19: Option value calculation with MJD formula (120 terms)

The test results as shown in Table 7.21 confirm that the calculation complexity is linear. By doubling N we observe a factor 2 in the ratio column.
Figure 7.4 shows that the convergence of the error is exponential for 1D. We also observe that the error convergence for the different strike prices is similar.

	abs. errors			calculation times			
N	K=50	K=100	K=200	MATLAB	C	CUDA K600	CUDA GTX480
16	5.61e-02	1.17e-01	2.13e-01	0.02	0.02	0.10	0.13
32	5.93e-06	3.56e-05	1.06e-04	0.06	0.03	0.10	0.13
64	6.39e-14	4.26e-14	1.22e-13	0.09	0.06	0.10	0.12
128	7.11e-15	<1e-15	3.69e-14	0.15	0.02	0.25	0.35
256	7.11e-15	<1e-15	3.69e-14	0.27	0.25	0.25	0.34
512	7.11e-15	<1e-15	3.69e-14	0.50	0.10	0.27	0.36
1024	7.11e-15	<1e-15	3.69e-14	0.93	0.20	0.13	0.12
2048	7.11e-15	<1e-15	3.69e-14	1.76	0.40	0.33	0.35
4096	7.11e-15	<1e-15	3.69e-14	3.42	0.79	0.44	0.35

Table 7.20: Error and time when pricing a European call option with the 1D COS method under MJD

N	MATLAB	difference	ratio
16	0.02	-	-
32	0.06	0.04	-
64	0.09	0.03	0.75
128	0.15	0.06	2.00
256	0.27	0.12	2.00
512	0.50	0.23	1.92
1024	0.93	0.43	1.87
2048	1.76	0.83	1.93
4096	3.42	1.66	2.00

Table 7.21: Time analysis of 1D COS method under MJD

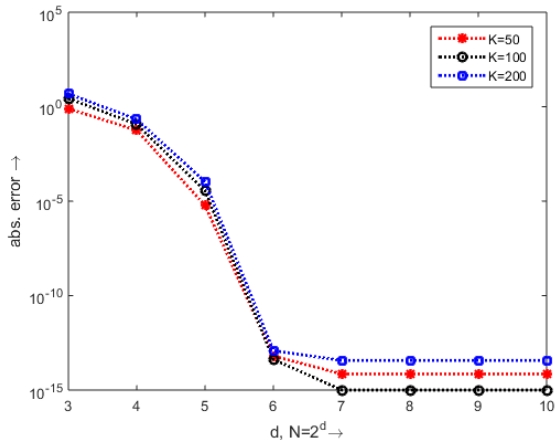

Figure 7.4: Error convergence for pricing 1D European call options (COS)

7.3.2.2 Two-dimensional tests

In this subsection we price a 2D European geometric basket put option, under MJD. We use parameter set 2. We compare the COS method results with the altered MJD formula. We test the COS method for three different N values, $16, 32, 64$, $N = N_1 = N_2$ and five different Q values, $64, 128$, 256, 512 and 1024. The results in Tables 7.23, 7.24 and 7.25 show that the 2D COS method is also fast and highly accurate, for MJD.

We use the dimension reduction technique of Cong and Oosterlee [8] to calculate the reference value. For 3D calculations we also use this technique.

MJD values of geometric put option (2D)			
K	reference value	error	MATLAB time (ms)
50	0.054429494761151	<1e-15	9.6
100	11.561224484806646	<1e-15	9.8
200	97.210213877504671	<1e-15	9.7

Table 7.22: Reference values for 2D European geometric basket put options with MJD formula (120 terms)

$K = 50$	$N^2 = 16^2$				
Q^2	abs error	MATLAB	C	CUDA K600	CUDAGTX480
64^2	7.77e-03	0.7	0.33	1.30	1.02
128^2	8.10e-03	1.2	0.62	1.38	1.02
256^2	8.13e-03	3.1	2.41	2.08	1.52
512^2	8.12e-03	11.2	9.57	5.48	1.98
1024^2	8.12e-03	50.3	42.9	19.1	3.40
$N^2 = 32^2$					
64^2	2.94e-04	0.9	0.40	1.20	0.82
128^2	5.07e-05	1.4	0.81	1.28	0.81
256^2	3.74e-05	3.8	2.57	1.96	1.30
512^2	3.73e-05	11.5	9.77	5.37	1.77
1024^2	3.73e-05	49.1	42.9	19.0	3.20
$N^2 = 64^2$					
64^2	2.61e-04	1.3	1.17	1.38	0.83
128^2	1.32e-05	2.1	1.58	1.88	1.03
256^2	9.67e-05	3.7	3.36	2.32	1.53
512^2	8.88e-09	13.8	10.6	5.71	2.00
1024^2	1.99e-09	48.8	43.6	19.3	3.42

Table 7.23: Error and time 2D European geometric basket put option ($K = 50$)

$K = 100$	$N^2 = 16^2$				
Q^2	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^2	5.82e-02	0.6	0.33	1.30	0.98
128^2	4.89e-02	1.1	0.63	1.38	0.99
256^2	4.83e-02	3.0	2.49	2.08	1.47
512^2	4.83e-02	12.4	9.89	5.49	1.89
1024^2	4.83e-02	47.2	44.1	19.1	3.27
$N^2 = 32^2$					
64^2	1.04e-02	0.9	0.39	1.20	0.79
128^2	8.78e-04	1.4	0.82	1.28	0.80
256^2	5.95e-07	3.7	2.64	1.97	1.27
512^2	1.65e-06	13.5	10.1	5.37	1.67
1024^2	1.93e-06	49.9	44.3	19.0	3.12
$N^2 = 64^2$					
64^2	1.04e-02	1.5	1.15	1.39	0.81
128^2	8.79e-04	2.0	1.57	1.87	1.00
256^2	1.51e-06	3.9	3.40	2.32	1.49
512^2	2.00e-07	11.8	10.9	5.71	1.98
1024^2	5.91e-08	50.1	45.0	19.3	3.32

Table 7.24: Error and time 2D European geometric basket put option ($K = 100$)

$K = 200$	$N^2 = 16^2$				
Q^2	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^2	9.75e-02	0.6	0.33	1.30	0.99
128^2	9.76e-02	1.1	0.65	1.37	0.99
256^2	9.79e-02	3.3	2.57	2.08	1.48
512^2	9.79e-02	11.4	10.2	5.48	1.89
1024^2	9.79e-02	52.2	45.4	19.1	3.39
$N^2 = 32^2$					
64^2	3.69e-05	0.9	0.41	1.20	0.79
128^2	8.30e-05	1.4	0.85	1.28	0.79
256^2	3.22e-05	3.3	2.73	1.99	1.27
512^2	3.28e-05	13.2	10.4	5.37	1.67
1024^2	3.28e-05	47.4	45.9	19.0	3.22
$N^2 = 64^2$					
64^2	6.80e-05	1.6	1.18	1.38	0.80
128^2	5.04e-05	1.8	1.62	1.87	1.01
256^2	4.38e-07	3.8	3.52	2.32	1.48
512^2	5.47e-09	13.6	11.2	5.71	1.89
1024^2	5.43e-09	48.8	46.0	19.3	3.46

Table 7.25: Error and time 2D European geometric basket put option ($K = 200$)

For example, in Table 7.23 the absolute error is 3.74e-05, which is reached by $K = 50$, $N = 32$ and $Q = 256$. The MATLAB CPU time for this accuracy is 3.8 milliseconds; the C CPU time is 2.57 milliseconds; the CUDA K600 time is 1.96 milliseconds. Note that CUDA GTX480 outperforms CUDA K600.

The calculation times of MJD are almost the same as GBM. Therefore, the calculation complexity under MJD is also quadratic in Q. In Figure 7.5 we observe exponential convergence in N and algebraic convergence in Q, as expected.

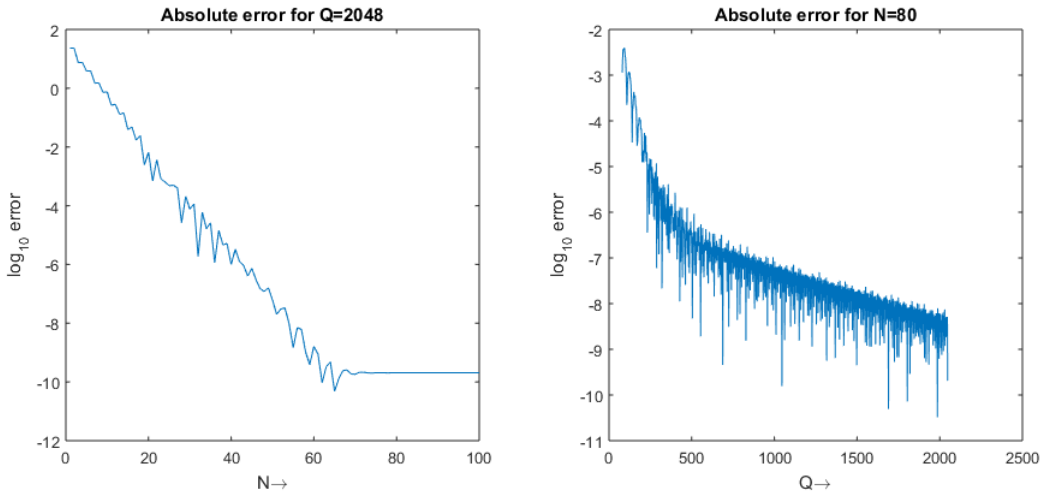

Figure 7.5: Error convergence for pricing a 2D European geometric put option (K=100)

7.3.2.3 Three-dimensional tests

In this subsection we perform an experiment for a 3D European geometric basket put option, under MJD. We use the third parameter set mentioned in section 7.3.1. We compare the test results of the COS method with the exact values of the altered MJD formula.

Black-Scholes values of 3D geometric basket call			
K	value	error	MATLAB time (ms)
75	2.204256395101063	<1e-15	0.99
150	49.847557528086831	<1e-15	0.98

Table 7.26: Results for 3D European geometric basket call option (MJD) value calculation with Black-Scholes formula

$K = 75$	$N^3 = 16^3$				
Q^3	abs error	MATLAB	C	CUDA K600	GTX480
64^3	2.13e-02	23.6	21.0	6.74	2.01
128^3	2.15e-02	168	183	46.3	6.76
160^3	2.15e-02	307	328	101	12.8
256^3	2.15e-02	1342	1580	378	47.9
$N^3 = 32^3$					
64^3	2.38e-05	29.9	35.3	10.7	2.69
128^3	1.97e-05	176	200	50.2	7.44
160^3	2.52e-05	314	343	103	13.3
256^3	2.28e-05	1353	1593	380	48.5
$N^3 = 64^3$					
64^3	4.31e-05	89.0	153	40.8	6.19
128^3	3.32e-06	237	316	80.1	10.9
160^3	2.07e-06	375	461	133	16.7
256^3	5.89e-07	1413	1713	410	52.0

Table 7.27: Error and time 3D European geometric basket put option ($K = 75$)

$K = 150$	$N^3 = 16^3$				
Q^3	abs error	MATLAB	C	CUDA K600	CUDA GTX480
64^3	2.14e-03	20.9	22.5	6.74	2.02
128^3	2.42e-03	166	195	46.4	6.91
160^3	2.44e-03	307	348	101	12.9
256^3	2.47e-03	1349	1670	377	49.2
$N^3 = 32^3$					
64^3	6.81e-06	26.4	37.1	10.7	2.68
128^3	5.28e-05	178	211	50.2	7.58
160^3	5.02e-05	313	363	103	13.6
256^3	4.81e-05	1352	1684	381	49.9
$N^3 = 64^3$					
64^3	3.77e-05	93.0	157	40.8	6.20
128^3	5.98e-06	243	330	80.2	11.0
160^3	2.74e-06	376	484	133	17.1
256^3	3.64e-07	1434	1805	411	53.4

Table 7.28: Error and time 3D European geometric basket put option ($K = 150$)

The results in Tables 7.27 and 7.28 show that the 3D COS method reproduces quickly and precisely the exact values of the 3D MJD formula. According to the 3D GBM case, we also observe that for this high dimension the calculation time increases significantly: the curse of dimensionality sets in. A few examples:

- the absolute error is 5.89e-07, which is reached by $K = 75$, $N^3 = 64^3$ and $Q^3 = 256^3$. The MATLAB CPU time for this accuracy is 1413 milliseconds; the C CPU time is 1713 milliseconds; the CUDA K600 time is 410 milliseconds.
- the absolute error is 3.64e-06, which is reached by $K = 150$, $N^3 = 64^3$ and $Q^3 = 256^3$. The MATLAB CPU time for this accuracy is 1434 milliseconds; the C CPU time is 1805 milliseconds; the CUDA K600 time is 411 milliseconds.

The data in these tables demonstrate that the CUDA GTX480 is substantially faster than the CUDA K600. The speedup rises to a factor of almost 8.

The calculation times of MJD are similar to GBM. Therefore, the calculation complexity under MJD is also cubic in Q. In Figure 7.6 we observe exponential convergence in N and algebraic convergence in Q, as expected.

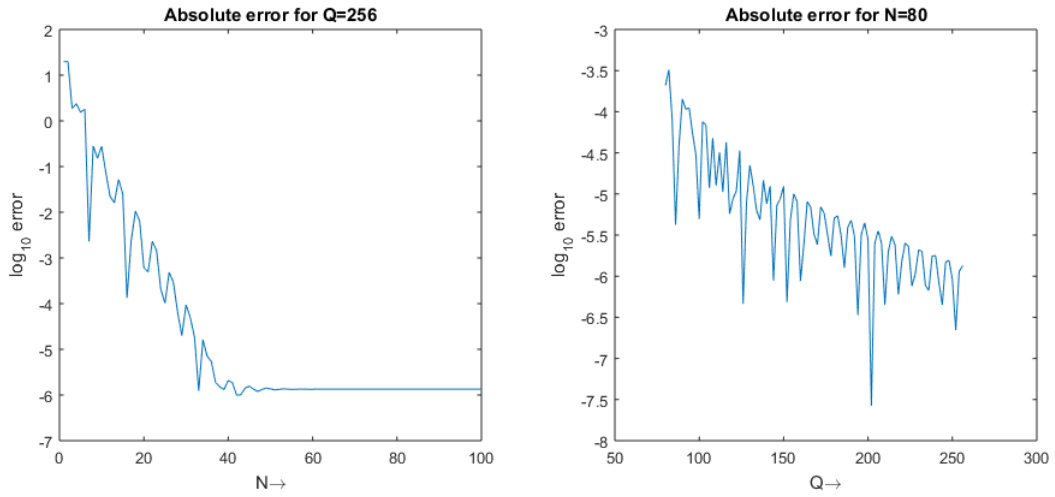

Figure 7.6: Error convergence for pricing a 3D European geometric put option (K=100)

7.4 Conclusion

Under the GBM model and the MJD model we have performed different numerical experiments in 1D, 2D and 3D. The test results show that the COS method is very accurate, robust and fast to very fast in all these dimensions. We observe time differences between the three programming languages MATLAB, C and CUDA K600. In general, CUDA consumes less time than the other two languages. Moreover, CUDA GTX480 outperforms CUDA K600. We also observe the expected rate of error convergence and calculation complexity.

Chapter 8

Conclusions

In this chapter we present our conclusion and give some recommendations for further research.

8.1 Conclusions

In 2008, Fang and Oosterlee introduced the 1D COS method [17], a Fourier-based option pricing technique for European options. This numerical method is based on Fourier cosine series expansions of the discounted expected payoff. The characteristic function of the underlying asset is used to approximate the Fourier coefficients. In 2012, this method was extended to higher dimensions by Ruijter and Oosterlee [33] and subsequently by Pellegrino and Sabino [32].
We discussed the 1D and 2D COS method and then, extended and discussed it for n-dimensions. Under certain assumptions the error of the nD COS method convergences exponentially in N for smooth density functions. But when the payoff coefficients are approximated with the DCT, the overall error converges algebraically with second order. The calculation complexity of the nD COS method is of n-th order in the number of terms Q. We have noted that n must not be chosen too large. On the one hand, the computing time grows exponentially with n as a result of which the curse of dimensionality sets in. On the other hand, the memory space needed grows exponentially in n and a computer has a limited storage capacity.

The aforementioned authors have observed that the COS method is an accurate, fast and robust option pricing technique. However, they have only performed their numerical experiments in the MATLAB environment. We studied how we can speed up computations without losing accuracy and robustness by using other programming environments, such as C (serial) and CUDA (parallel GPU computing), especially in the case of multi-asset options. In the case of CUDA, a study about the parallelization strategy has been carried out.

We have performed several numerical experiments under the GBM model and the MJD model. Under these models, we have tested 1D call options, 2D geometric put and arithmic call options, exchange options and 3D geometric put options. We observed that the (1D, 2D, 3D) COS methods perform very well. In the tested dimensions, it is very fast, very precise and robust. We also observed exponential convergence in N and algebraic convergence in Q, as expected. The calculation complexity is of n-th order. For CUDA, we have presented our parallel implementation of the nD COS formula. The tests of our developed codes show that in particular the payoff function array and the characteristic function array computations benefit from parallelization on the GPU. For example, the characteristic function array performs excellent with a speedup by a factor of 76.

We conclude that the nD COS method for pricing European multi-dimensional stock options with parallel GPU computing is fast, accurate and robust and outperforms the serial computing languages MATLAB and C.

8.2 Outlook

In this section we present some suggestions for future research.

In this thesis we have focused on the pricing of some path-independent European stock options. Surkov's [38, 39] experiments by means of his Fourier Space Time-stepping method on GPUs (FST-GPU) show that parallel option pricing for an American 2D "double-trigger stop-loss" option is much more efficient than a serial computing with his FST-CPU. Therefore, our suggestion is to investigate path-dependent American options with the nD COS method under the CUDA architecture. Moreover, while the trading volumes of American options exceed the volumes of European options, the benefits of parallel option pricing on GPU can spread out widely.

Since our parallel version of nD COS is very efficient in terms of computational cost, it can be employed when nested simulations are required. Nested simulations are often required in the financial markets. A recent example is the evaluation of counterparty credit risk, commonly referred to as Credit Valuation Adjustment (CVA).

In this thesis we have used two classic models: GBM and MJD. Later, different exponential Lévy models were developed, such as Variance Gamma and CGMY. These modern models are better able to replicate the log-returns of market data. However, for option pricing in higher dimensions, ($\geq$2D) the characteristic functions of these models are not available and have to be derived. This derivation is a challenging research direction.

In this thesis we have done research in different dimensions up to and including 3D. In order to explore higher dimensions, several limiting factors need to be handled. For the parallel GPU implemetations, the amount of memory is typically less than in a CPU-based system. To overcome this drawback, we suggest the use of the later generation GPUs or the extension to multi-GPU approach. Another point to be considered is that the FFT in the cuFFT library is developed for calculations up to and including 3D. The development of the CUDA implementation for the FFT in n-dimensions can be an interesting and challenging topic for future research.

Appendices

Appendix A

Derivation of ChF of a standard normal distribution

The characteristic function of a random variable Z with a $\mathcal{N}(0,1)$ distribution reads

$$\begin{aligned}\phi(\omega) &= \mathbb{E}[\exp^{i\omega Z}] = \mathbb{E}[\cos(\omega Z) + i\sin(\omega Z)] \\ &= \int_{-\infty}^{\infty} \cos(\omega z) f(z) dz + i \int_{-\infty}^{\infty} \sin(\omega z) f(z) dz. \end{aligned} \tag{A.1}$$

The distribution function $f(z)$ of Z is an even function around $z = 0$ and $\sin(i\omega z)$ is an odd function around $z = 0$. Therefore, the second integral is zero. The differential of the ChF reads

$$\begin{aligned}\frac{d}{d\omega}\phi(\omega) &= \frac{d}{d\omega}\mathbb{E}[\exp(i\omega Z)] = \mathbb{E}[\frac{d}{d\omega}\exp(i\omega Z)] = \mathbb{E}[iZ\exp(i\omega Z)] = i\mathbb{E}[Z\cos(\omega Z)] - \mathbb{E}[Z\sin(\omega Z)] \\ &= i\int_{-\infty}^{\infty} z\cos(\omega z) f(z) dz - \int_{-\infty}^{\infty} z\sin(\omega z) f(z) dz. \end{aligned} \tag{A.2}$$

In Equation (A.2) the first integral is equal to zero because of the odd integrand. Inserting into Equation (A.2) gives

$$\begin{aligned}\frac{d}{d\omega}\phi(\omega) &= -\int_{-\infty}^{\infty} z\sin(\omega z)\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{z^2}{2}\right) dz = \int_{-\infty}^{\infty} \sin(\omega z)\frac{d}{dz} f(z) dz \\ &= [\sin(\omega z) f(z)]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} \omega\cos(\omega z) f(z) dz = -\omega\int_{-\infty}^{\infty} \cos(\omega z) f(z) dz. \end{aligned} \tag{A.3}$$

Equation (A.1) and (A.3) form an ODE

$$\frac{d\phi(\omega)}{d\omega} = -\omega\phi(\omega) \quad \text{with initial condition} \quad \phi(0) = 1.$$

The solution of this ODE reads

$$\phi(\omega) = \exp\left(-\frac{\omega^2}{2}\right).$$

Appendix B

Derivation of FCTs

In Chapter 3, we show that the DCT can be accelerated by means of a FFT. We call this accelerated DCT the FCT. It is also possible to speed up the FFT itself because the input vector consists of real elements only. In this way the FCT accelerates even more.
Firstly, in this appendix we discuss the derivation of the FCTs, for 1D and 2D, given by Makhoul[24]. Secondly, we give the formula of the 3D FCT. Lastly, we show the speed up of the FFT by using a real sequence.

B.1 Definitions

Define $H_{\hat{b}}^{\hat{a}}$ as follows

$$H_{\hat{b}}^{\hat{a}} = \exp\left(\frac{-2i\pi\hat{a}}{\hat{b}}\right).$$

Then the following identities hold

$$H_{2M}^{kM} = 1, \quad H_{2M}^{2k} = H_M^k, \quad H_b^a + H_b^{-a} = 2\cos\left(\frac{2\pi a}{b}\right) = 2\mathrm{Re}\left\{H_b^a\right\}.$$

Define the function DFT: $\mathbb{C}^M \to \mathbb{C}^M$ of complex vector $\vec{c}$ of length M as

$$\mathrm{DFT}(\vec{c})_k = \sum_{n=0}^{M-1} c_n H_M^{nk}, \quad 0 \le k \le M-1. \tag{B.1}$$

Define the function DCT: $\mathbb{R}^M \to \mathbb{R}^M$ of real vector $\vec{c}$ of length M as

$$\mathrm{DCT}(\vec{c})_k = 2\sum_{n=0}^{M-1} c_n \cos\left(\frac{\pi(2n+1)k}{2M}\right), \quad 0 \le k \le M-1. \tag{B.2}$$

Remark: although k is between 0 and $M-1$ we sometimes use

$$\mathrm{DCT}(\vec{c})_M = 2\sum_{n=0}^{M-1} x(n)\cos\left(\pi n + \frac{\pi}{2}\right) = 0. \tag{B.3}$$

B.2 DCT of length M with the use of a DFT of length $2M$

Define a real vector $\mathring{c}$ of length $2M$ which consists of a real times series $\vec{c}$ and its reverse

$$\mathring{c}_n = \begin{cases} c_n & 0 \le n \le M-1 \\ c_{2M-n-1} & M \le n \le 2M-1. \end{cases} \tag{B.4}$$

Then the DFT of vector $\mathring{c}$ is given by

$$\mathrm{DFT}(\mathring{c})_k = \sum_{n=0}^{2M-1} \mathring{c}_n H_{2M}^{nk}, \quad 0 \le k \le 2M-1. \tag{B.5}$$

Inserting (B.4) into (B.5) gives

$$\begin{aligned}
\mathrm{DFT}(\mathring{c}) &= \sum_{n=0}^{M-1} c_n H_{2M}^{nk} + \sum_{n=M}^{2M-1} c_{2M-n-1} H_{2M}^{nk} = \sum_{n=0}^{M-1} c_n H_{2M}^{nk} + \sum_{n=0}^{M-1} c_n H_{2M}^{(2M-n-1)k} \\
&= \sum_{n=0}^{M-1} c_n H_{2M}^{nk} + \sum_{n=0}^{M-1} c_n H_{2M}^{2kM} H_{2M}^{(-n-1)k} = \sum_{n=0}^{M-1} c_n \left(H_{2M}^{nk} + H_{2M}^{-(n+1)k} \right) \\
&= H_{2M}^{-k/2} \sum_{n=0}^{M-1} c_n \left(H_{2M}^{nk} H_{2M}^{k/2} + H_{2M}^{-nk} H_{2M}^{-k/2} \right) = H_{2M}^{-k/2} \sum_{n=0}^{M-1} c_n \left(H_{2M}^{(n+0.5)k} + H_{2M}^{-(n+0.5)k} \right) \\
&= H_{2M}^{-k/2} 2 \sum_{n=0}^{M-1} c_n \cos\left(\frac{\pi(2n+1)k}{2M} \right) \qquad \text{(B.6)} \\
&= H_{2M}^{-k/2} 2\mathrm{Re} \left\{ \sum_{n=0}^{M-1} c_n \exp\left(-i \frac{\pi(2n+1)k}{2M} \right) \right\} \\
&= H_{2M}^{-k/2} 2\mathrm{Re} \left\{ H_{2M}^{k/2} \sum_{n=0}^{M-1} c_n H_{2M}^{nk} \right\} \quad \text{for} \quad 0 \le k \le 2M-1. \qquad \text{(B.7)}
\end{aligned}$$

From (B.6), (B.7) and the definition of the DCT it follows that

$$\mathrm{DCT}(\vec{c})_k = H_{2M}^{k/2} \mathrm{DFT}(\mathring{c})_k = 2\mathrm{Re} \left\{ H_{2M}^{k/2} \sum_{n=0}^{M-1} c_n H_{2M}^{nk} \right\} \quad \text{for} \quad 0 \le k \le M-1.$$

B.3 Derivation 1D FCT

Define vectors $\breve{c}$ and $\overline{c}$ from $\mathring{c}$ as

$$\breve{c}_n := \mathring{c}_{2n}, \quad \overline{c}_n := \mathring{c}_{2n+1}, \quad 0 \le n \le M-1.$$

Remark: both timeseries $\breve{c}$ and $\overline{c}$ are of length M and contain all the elements of the vector c. Formula (B.5) can be written with $\breve{c}$ and $\overline{c}$ as

$$\mathrm{DFT}(\mathring{c})_k = \sum_{n=0}^{M-1} \breve{c}_n H_{2M}^{2nk} + \sum_{n=0}^{M-1} \overline{c}_n H_{2M}^{(2n+1)k}, \quad 0 \le k \le 2M-1. \tag{B.8}$$

Note that

$$\overline{c}_n = \breve{c}_{M-n-1}, \quad 0 \le n \le M-1. \tag{B.9}$$

Inserting (B.9) into (B.8) gives us

$$\begin{aligned}
\mathrm{DFT}(\mathring{c})_k &= \sum_{n=0}^{M-1} \breve{c}_n H_{2M}^{2nk} + \sum_{n=0}^{M-1} \breve{c}_{M-n-1} H_{2M}^{(2n+1)k} = \sum_{n=0}^{M-1} \breve{c}_n H_{2M}^{2nk} + \sum_{n=0}^{M-1} \breve{c}_n H_{2M}^{(2M-2n-1)k} \\
&= \sum_{n=0}^{M-1} \breve{c}_n \left(H_{2M}^{2nk} + H_{2M}^{(-2n-1)k} \right) = \sum_{n=0}^{M-1} \breve{c}_n \left(H_{M}^{nk} + H_{M}^{(-n-0.5)k} \right) \\
&= H_M^{-k/4} \sum_{n=0}^{M-1} \breve{c}_n \left(H_M^{nk} H_M^{k/4} + H_M^{(-n-0.5)k} H_M^{k/4} \right) = H_M^{-k/4} \sum_{n=0}^{M-1} \breve{c}_n \left(H_M^{(n+0.25)k} + H_M^{-(n+0.25)k} \right) \\
&= H_{4M}^{-k} 2 \sum_{n=0}^{M-1} \breve{c}_n \cos\left(\frac{\pi(4n+1)k}{2M} \right) = H_{4M}^{-k} 2\mathrm{Re} \left\{ \sum_{n=0}^{M-1} \breve{c}_n \exp\left(-i \frac{\pi(4n+1)k}{2M} \right) \right\} \\
&= H_{4M}^{-k} 2\mathrm{Re} \left\{ H_{4M}^{k} \sum_{n=0}^{M-1} \breve{c}_n H_M^{nk} \right\}. \qquad \text{(B.10)}
\end{aligned}$$

Therefore, the 1D FCT formula reads

$$\mathrm{FCT}(\vec{c})_k = 2\mathrm{Re}\left\{ H_{4M}^k \mathrm{FFT}(\breve{c})_k \right\},$$

where

$$\breve{c}_n = \begin{cases} c_{2n} & 0 \le n \le \left\lfloor \frac{M-1}{2} \right\rfloor, \\ c_{2M-2n-1} & \left\lfloor \frac{M+1}{2} \right\rfloor \le n \le M-1. \end{cases}$$

B.4 Derivation 2D FCT

In a similar way, we discuss Makhoul's proof of the 2D FCT[24]. We use C, a 2D real matrix of size $M_1 \times M_2$, and define the 2D DCT as

$$\begin{aligned}
\mathrm{DCT}(C)_{k_1,k_2} := 4 \textstyle\sum_{n_1=0}^{M_1-1} \sum_{n_2=0}^{M_2-1} C_{n_1,n_2} \cos\left(k_1 \frac{(2n_1+1)\pi}{2M_1} \right) \cos\left(k_2 \frac{(2n_2+1)\pi}{2M_2} \right), \\
\text{for} \quad 0 \le k_1, k_2 \le M-1. \qquad \text{(B.11)}
\end{aligned}$$

We create an extension $\mathring{C}$ of size $2M_1 \times 2M_2$ such that

$$\mathring{C}_{n_1,n_2} = \begin{cases} C_{n_1,n_2}, & 0 \le n_1 \le M_1-1, & 0 \le n_2 \le M_2-1, \\ C_{2M_1-n_1-1,n_2}, & M_1 \le n_1 \le 2M_1-1, & 0 \le n_2 \le M_2-1, \\ C_{n_1,2M_2-n_2-1}, & 0 \le n_1 \le M_1-1, & M_2 \le n_2 \le 2M_2-1, \\ C_{2M_1-n_1-1,2M_2-n_2-1}, & M_1 \le n_1 \le 2M_1-1, & M_2 \le n_2 \le 2M_2-1. \end{cases} \qquad \text{(B.12)}$$

The 2D DFT of $\mathring{C}$ reads

$$\mathrm{DFT}(\mathring{C})_{k_1,k_2} = \sum_{n_1=0}^{2M_1-1} \sum_{n_2=0}^{2M_2-1} \mathring{C}_{n_1,n_2} H_{2M_1}^{n_1k_1} H_{2M_2}^{n_2k_2}. \qquad \text{(B.13)}$$

We define 2D real matrix $\breve{C}$, $\overline{C}^1$, $\overline{C}^2$ and $\overline{C}^3$ as

$$\begin{aligned}
\breve{C}_{n_1,n_2} &= \mathring{C}_{2n_1,2n_2}, & \overline{C}^1_{n_1,n_2} &= \mathring{C}_{2n_1+1,2n_2}, \\
\overline{C}^2_{n_1,n_2} &= \mathring{C}_{2n_1,2n_2+1}, & \overline{C}^3_{n_1,n_2} &= \mathring{C}_{2n_1+1,2n_2+1}. \qquad \text{(B.14)}
\end{aligned}$$

We write times series $\overline{C}^1$, $\overline{C}^2$ and $\overline{C}^3$ as functions of $\breve{C}$ as

$$\begin{aligned}
\overline{C}^1_{n_1,n_2} &= \breve{C}_{M_1-n_1-1,n_2}, \\
\overline{C}^2_{n_1,n_2} &= \breve{C}_{n_1,M_2-n_2-1}, \\
\overline{C}^3_{n_1,n_2} &= \breve{C}_{M_1-n_1-1,M_2-n_2-1}. \qquad \text{(B.15)}
\end{aligned}$$

The 2D DFT of $\mathring{C}$ (B.13) reads

$$
\begin{aligned}
\mathrm{DFT}(\mathring{C})_{k_1,k_2} =& \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{2M_1}^{2k_1n_1} H_{2M_2}^{2k_1n_1} + \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \overline{C}^1_{n_1,n_2} H_{2M_1}^{(2n_1+1)k_1} H_{2M_2}^{2k_2n_2} \\
&+ \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \overline{C}^2_{n_1,n_2} H_{2M_1}^{2k_1n_1} H_{2M_2}^{(2n_2+1)k_2} + \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \overline{C}^3_{n_1,n_2} H_{2M_1}^{(2n_1+1)k_1} H_{2M_2}^{(2n_2+1)k_2} \\
=& \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{2M_1}^{2k_1n_1} H_{2M_2}^{2k_1n_1} + \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{M_1-n_1-1,n_2} H_{2M_1}^{(2n_1+1)k_1} H_{2M_2}^{2k_2n_2} \\
&+ \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{(n_1,M_2-n_2-1)} H_{2M_1}^{2k_1n_1} H_{2M_2}^{(2n_2+1)k_2} + \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{M_1-n_1-1,M_2-n_2-1} H_{2M_1}^{(2n_1+1)k_1} H_{2M_2}^{(2n_2+1)k_2} \\
=& \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{2M_1}^{2k_1n_1} H_{2M_2}^{2k_1n_1} + \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{2M_1}^{(2M_1-2n_1-1)k_1} H_{2M_2}^{2k_2n_2} \\
&+ \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{2M_1}^{2k_1n_1} H_{2M_2}^{(2M_2-2n_2-1)k_2} + \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{2M_1}^{(2M_1-2n_1-1)k_1} H_{2M_2}^{(2M_2-2n_2-1)k_2} \\
=& \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} \Big(H_{2M_1}^{2k_1n_1} H_{2M_2}^{2k_1n_1} + H_{2M_1}^{(2M_1-2n_1-1)k_1} H_{2M_2}^{2k_2n_2} \\
&+ H_{2M_1}^{2k_1n_1} H_{2M_2}^{(2M_2-2n_2-1)k_2} + H_{2M_1}^{(2M_1-2n_1-1)k_1} H_{2M_2}^{(2M_2-2n_2-1)k_2} \Big) \\
=& \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} \Big(H_{M_1}^{k_1n_1} H_{M_2}^{k_1n_1} + H_{M_1}^{(-n_1-0.5)k_1} H_{M_2}^{k_2n_2} \\
&+ H_{M_1}^{k_1n_1} H_{M_2}^{(-n_2-0.5)k_2} + H_{M_1}^{(-n_1-0.5)k_1} H_{M_2}^{(-n_2-0.5)k_2} \Big) \\
=& H_{M_1}^{-k_1/4} H_{M_2}^{-k_2/4} \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} \Big(H_{M_1}^{(n_1+0.25)k_1} H_{M_2}^{(n_2+0.25)k_2} + H_{M_1}^{-(n_1+0.25)k_1} H_{M_2}^{(n_2+0.25)k_2} \\
&+ H_{M_1}^{(n_1+0.25)k_1} H_{M_2}^{-(n_2+0.25)k_2} + H_{M_1}^{-(n_1+0.25)k_1} H_{M_2}^{-(n_2+0.25)k_2} \Big).
\end{aligned} \tag{B.16}
$$

We write formula (B.16) as

$$
\mathrm{DFT}(\mathring{C})_{k_1,k_2} = H_{M_1}^{-k_1/4} H_{M_2}^{-k_2/4} 4 \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} \cos\left(\frac{\pi(4n_1+1)k_1}{2M_1}\right) \cos\left(\frac{\pi(4n_2+1)k_2}{2M_2}\right). \tag{B.17}
$$

Also, formula (B.16) can be written as

$$
\begin{aligned}
\mathrm{DFT}(\mathring{C})_{k_1,k_2} =& H_{4M_1}^{-k_1} H_{4M_2}^{-k_2} 2 \Big(\mathrm{Re}\left\{ H_{4M_1}^{k_1} H_{4M_2}^{k_2} \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{M_1}^{n_1k_1} H_{M_2}^{n_2k_2} \right\} \\
&+ \mathrm{Re}\left\{ H_{4M_1}^{k_1} H_{4M_2}^{-k_2} \sum_{n_1=0}^{M_1-1}\sum_{n_2=0}^{M_2-1} \breve{C}_{n_1,n_2} H_{M_1}^{n_1k_1} H_{M_2}^{-n_2k_2} \right\} \Big) \\
=& H_{4M_1}^{-k_1} H_{4M_2}^{-k_2} 2\mathrm{Re}\left\{ H_{4M_1}^{k_1} H_{4M_2}^{k_2} \mathrm{FFT}(\breve{C})_{k_1,k_2} + H_{4M_1}^{k_1} H_{4M_2}^{-k_2} \mathrm{FFT}(\breve{C})_{k_1,M-k_2} \right\}.
\end{aligned}
$$

Therefore, the 2D FCT formula reads

$$
\mathrm{FCT}(C)_{k_1,k_2} = 2\mathrm{Re}\left\{ H_{4M_1}^{k_1} H_{4M_2}^{k_2} \mathrm{FFT}(\breve{C})_{k_1,k_2} + H_{4M_1}^{k_1} H_{4M_2}^{-k_2} \mathrm{FFT}(\breve{C})_{k_1,M-k_2} \right\}, \tag{B.18}
$$

where

$$\check{C}_{n_1,n_2} = \begin{cases} C_{2n_1,2n_2}, & 0 \le n_1 \le \left\lfloor \frac{M_1-1}{2} \right\rfloor, \quad 0 \le n_2 \le \left\lfloor \frac{M_2-1}{2} \right\rfloor, \\ C_{2n_1,2M_2-2n_2-1}, & 0 \le n_1 \le \left\lfloor \frac{M_1-1}{2} \right\rfloor, \quad \left\lfloor \frac{M_2+1}{2} \right\rfloor \le n_2 \le M_2-1, \\ C_{2M_1-2n_1-1,2n_2}, & \left\lfloor \frac{M_1+1}{2} \right\rfloor \le n_1 \le M_1-1, \quad 0 \le n_2 \le \left\lfloor \frac{M_2-1}{2} \right\rfloor, \\ C_{2M_1-2n_1-1,2M_2-2n_2-1}, & \left\lfloor \frac{M_1+1}{2} \right\rfloor \le n_1 \le M_1-1, \quad \left\lfloor \frac{M_2+1}{2} \right\rfloor \le n_2 \le M_2-1. \end{cases}$$

B.5 3D FCT

The derivation of the 3D FCT formula is similar to the derivation of the 2D FCT formula. C is a 3D array of size $M_1 \times M_2 \times M_3$. The FCT of C is defined as

$$\begin{aligned}\mathrm{FCT}(C)_{k_1,k_2,k_3} =& 2\mathrm{Re}\Big\{H_{4M_1}^{k_1}\Big(H_{4M_2}^{k_2}H_{4M_3}^{k_3}\mathrm{FFT}(\breve{C})_{k_1,k_2,k_3} + H_{4M_2}^{k_2}H_{4M_3}^{-k_3}\mathrm{FFT}(\breve{C})_{k_1,k_2,M_3-k_3} \\ &+ H_{4M_2}^{-k_2}H_{4M_3}^{k_3}\mathrm{FFT}(\breve{C})_{k_1,M_2-k_2,k_3} + H_{4M_2}^{-k_2}H_{4M_3}^{-k_3}\mathrm{FFT}(\breve{C})_{k_1,M-k_2,M-k_3}\Big)\Big\},\end{aligned}$$

where

$$\breve{C}_{n_1,n_2,n_3} =$$

$$\begin{cases}
C_{2n_1,2n_2,2n_3}, & 0 \le n_1 \le \left\lfloor \frac{M_1-1}{2} \right\rfloor, & 0 \le n_2 \le \left\lfloor \frac{M_2-1}{2} \right\rfloor & 0 \le n_3 \le \left\lfloor \frac{M_3-1}{2} \right\rfloor \\
C_{2n_1,2n_2,2M_3-2n_3-1}, & 0 \le n_1 \le \left\lfloor \frac{M_1-1}{2} \right\rfloor, & 0 \le n_2 \le \left\lfloor \frac{M_2-1}{2} \right\rfloor & \left\lfloor \frac{M_3+1}{2} \right\rfloor \le n_3 \le M_3-1 \\
C_{2n_1,2M_2-2n_2-1,2n_3}, & 0 \le n_1 \le \left\lfloor \frac{M_1-1}{2} \right\rfloor, & \left\lfloor \frac{M_2+1}{2} \right\rfloor \le n_2 \le M_2-1 & 0 \le n_3 \le \left\lfloor \frac{M_3-1}{2} \right\rfloor \\
C_{2n_1,2M_2-2n_2-1,2M_3-2n_3-1}, & 0 \le n_1 \le \left\lfloor \frac{M_1-1}{2} \right\rfloor, & \left\lfloor \frac{M_2+1}{2} \right\rfloor \le n_2 \le M_2-1 & \left\lfloor \frac{M_3+1}{2} \right\rfloor \le n_3 \le M_3-1 \\
C_{2M_1-2n_1-1,2n_2,2n_3}, & \left\lfloor \frac{M_1+1}{2} \right\rfloor \le n_1 \le M_1-1, & 0 \le n_2 \le \left\lfloor \frac{M_2-1}{2} \right\rfloor & 0 \le n_3 \le \left\lfloor \frac{M_3-1}{2} \right\rfloor \\
C_{2M_1-2n_1-1,2n_2,2M_3-2n_3-1}, & \left\lfloor \frac{M_1+1}{2} \right\rfloor \le n_1 \le M_1-1, & 0 \le n_2 \le \left\lfloor \frac{M_2-1}{2} \right\rfloor & \left\lfloor \frac{M_3+1}{2} \right\rfloor \le n_3 \le M_3-1 \\
C_{2M_1-2n_1-1,2M_2-2n_2-1,2n_3}, & \left\lfloor \frac{M_1+1}{2} \right\rfloor \le n_1 \le M_1-1, & \left\lfloor \frac{M_2+1}{2} \right\rfloor \le n_2 \le M_2-1 & 0 \le n_3 \le \left\lfloor \frac{M_3-1}{2} \right\rfloor \\
C_{2M_1-2n_1-1,2M_2-2n_2-1,2M_3-2n_3-1}, & \left\lfloor \frac{M_1+1}{2} \right\rfloor \le n_1 \le M_1-1, & \left\lfloor \frac{M_2+1}{2} \right\rfloor \le n_2 \le M_2-1 & \left\lfloor \frac{M_3+1}{2} \right\rfloor \le n_3 \le M_3-1.
\end{cases}$$

B.6 DFT of a real sequence

It is possible to accelerate the FFT itself. By eliminating the complex input part and only using a real input it is possible to decrease the computing time of the FFT. The DFT of a real vector $\vec{c}$ of length M can be calculated using an $M/2$ point DFT. We define vectors $\grave{c}$ and $\acute{c}$ as

$$\begin{cases} \grave{c}_n = c_{2n} & 0 \le n \le M/2-1 \\ \acute{c}_n = c_{2n+1} & 0 \le n \le M/2-1. \end{cases}$$

We write the DFT of $\vec{c}$ as

$$\begin{aligned} \mathrm{DFT}(\vec{c})_k &= \sum_{n=0}^{M-1} c_n H_M^{kn} \\ &= \sum_{n=0}^{M/2-1} \left[\grave{c}_n H_M^{2nk} + \acute{c}_n H_M^{(2n+1)k} \right] \\ &= \sum_{n=0}^{M/2-1} \grave{c}_n H_{M/2}^{nk} + H_M^k \sum_{n=0}^{M/2-1} \acute{c}_n H_{M/2}^{nk} \\ &= \mathrm{DFT}(\grave{c})_k + H_M^k \mathrm{DFT}(\acute{c})_k, \quad k = 0, \ldots, M-1. \end{aligned}$$

We define the vector $\dot{c}$ as

$$\dot{c}_n = \grave{c}_n + i\acute{c}_n \quad n = 0, 1, \ldots, M/2-1.$$

Then, we can write $\mathrm{DFT}(\dot{c})_k$, the $M/2$ point DFT of $\dot{c}$, as

$$\mathrm{DFT}(\dot{c})_k = \mathrm{DFT}(\grave{c})_k + i\mathrm{DFT}(\acute{c})_k \quad k = 0, \ldots, M/2-1. \tag{B.19}$$

Since $\grave{c}$ and $\acute{c}$ are real vectors, their DFTs are Hermitian symmetric

$$\mathrm{DFT}(\grave{c})_k = \mathrm{conj}\left(\mathrm{DFT}(\grave{c})_{M/2-k}\right) \quad \mathrm{DFT}(\acute{c})_k = \mathrm{conj}\left(\mathrm{DFT}(\acute{c})_{M/2-k}\right).$$

Therefore, we write

$$\mathrm{conj}\left(\mathrm{DFT}(\dot{c})_{M/2-k}\right) = \mathrm{DFT}(\grave{c})_k - i\mathrm{DFT}(\acute{c})_k. \tag{B.20}$$

Then, (B.19) and (B.20) gives

$$\begin{cases} \mathrm{DFT}(c)_0 = \mathrm{Re}\left\{\mathrm{DFT}(\dot{c})_0\right\} + \mathrm{Imag}\left\{\mathrm{DFT}(\dot{c})_0\right\}, & k = 0, \\ \mathrm{DFT}(c)_k = \frac{1}{2}\left[\mathrm{DFT}(\dot{c})_k + \mathrm{conj}(\mathrm{DFT}(\dot{c})_{M/2-k}) - iH_M^k\left(\mathrm{conj}(\mathrm{DFT}(\dot{c})_{M/2-k})\right)\right], & 1 \le k \le M/2-1, \\ \mathrm{DFT}(c)_{M/2} = \mathrm{Re}\left\{\mathrm{DFT}(\dot{c})_{M/2}\right\} - \mathrm{Imag}\left\{\mathrm{DFT}(\dot{c})_{M/2}\right\}, & k = M/2. \end{cases} \tag{B.21}$$

Appendix C

Two-dimensional COS method

In this appendix we present the two-dimensional COS method. It serves as a special example of the contents in Chapter 5. This appendix is organized as follows: firstly, we show the derivation of the 2D COS method which leads to the 2D COS formula. Thereafter, we focus on the payoff coefficients, the truncation range, the overall error and the computational complexity. Finally, we show that the overall error converges algebraically and is of the second order; and that the calculation complexity is also of the second order. We conclude that the 2D COS method is a very fast and highly precise option pricing technique for European options.

C.1 Derivation

Just as the derivation of the 1D COS method (see chapter 4) we start from the risk-neutral valuation formula:

$$v(\vec{S}(t),t) = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}\left[g\left(\vec{S}(T)\right)|\vec{S}(t)\right] = e^{-r\Delta t}\iint_{\mathbb{R}^2} g(\vec{y})f(\vec{y}|\vec{x})d\vec{y}, \tag{C.1}$$

where $v(\vec{S}(t),t)$ is the value of the option at time t, $\vec{S}(t)$ are the values of the underlying assets at time t, r is the constant risk-free interest rate, $\mathbb{E}^{\mathbb{Q}}$ is the expectation operator, the function f is the probability density function of $\vec{S}(T)$ given $\vec{S}(t)$ and the function g is the payoff function.

The derivation of the 2D COS method consists of five steps. We follow [33].

First Step *We will truncate the integration range*[1]

The density function $f(\vec{y}|\vec{x})$ decays to zero quickly for $||\vec{y}|| \to \infty$. Therefore, $v(\vec{S}(t),t)$ can be well approximated by a finite integration range $[a_1,b_1]\times[a_2,b_2] \subset \mathbb{R}^2$:

$$v_1(\vec{x},t) = e^{-r\Delta t}\int_{a_2}^{b_2}\int_{a_1}^{b_1} g(\vec{y})f(\vec{y}|\vec{x})dy_1dy_2. \tag{C.2}$$

Second Step *We will replace the probability density function by its cosine expansion*[2]

By means of the two-dimensional Fourier-cosine series expansion we can define a function on a finite domain. The cosine expansion of $f(\vec{y}|\vec{x})$ on $[a_1,b_1]\times[a_2,b_2]$ is given by

$$f(\vec{y}|\vec{x}) = \sum_{k_1=0}^{\infty}{}' \sum_{k_2=0}^{\infty}{}' A_{k_1,k_2}(\vec{x})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right), \tag{C.3}$$

where A_{k_1,k_2} is defined as

$$A_{k_1,k_2}(\vec{x}) = \frac{2}{b_1-a_1}\frac{2}{b_2-a_2}\int_{a_2}^{b_2}\int_{a_1}^{b_1} f(\vec{y}|\vec{x})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right)dy_1dy_2.$$

[1] $v_1(\vec{x},t)$ is an approximation of $v(\vec{x},t)$.

[2] The $\sum'$-summation is a summation where the first term is multiplied by 0.5.

Inserting (C.3) into $v_1(\vec{x},t)$ gives

$$v_1(\vec{x},t) = e^{-r\Delta t}\int_{a_2}^{b_2}\int_{a_1}^{b_1} g(\vec{y}) \sum_{k_1=0}^{\infty}{}' \sum_{k_2=0}^{\infty}{}' A_{k_1,k_2}(\vec{x}) \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2. \tag{C.4}$$

Third Step *We will interchange the summation and the integration*
Then, (C.4), reads

$$\begin{aligned} v_1(\vec{x},t) =& e^{-r\Delta t} \sum_{k_1=0}^{\infty}{}' \sum_{k_2=0}^{\infty}{}' \frac{b_1-a_1}{2}\frac{b_2-a_2}{2} A_{k_1,k_2}(\vec{x}) \cdot \\ & \frac{2}{b_1-a_1}\frac{2}{b_2-a_2}\int_{a_2}^{b_2}\int_{a_1}^{b_1} g(\vec{y})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2. \end{aligned} \tag{C.5}$$

We define V_{k_1,k_2} as

$$V_{k_1,k_2} := \frac{2}{b_1-a_1}\frac{2}{b_2-a_2}\int_{a_2}^{b_2}\int_{a_1}^{b_1} g(\vec{y})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2. \tag{C.6}$$

Insert V_{k_1,k_2} into (C.5), gives

$$v_1(\vec{x},t) = e^{-r\Delta t} \sum_{k_1=0}^{\infty}{}' \sum_{k_2=0}^{\infty}{}' \frac{b_1-a_1}{2}\frac{b_2-a_2}{2} A_{k_1,k_2}(\vec{x}) V_{k_1,k_2}. \tag{C.7}$$

Now, the integral over the product of $f(\vec{y}|\vec{x})$ and $g(\vec{y})$ is written as a summation of the product of their Fourier-cosine coefficients.

Fourth Step *We will truncate the series summation*[3]

Then (C.7) reads

$$v_2(\vec{x},t) = e^{-r\Delta t} \sum_{k_1=0}^{N_1-1}{}' \sum_{k_2=0}^{N_2-1}{}' \frac{b_1-a_1}{2}\frac{b_2-a_2}{2} A_{k_1,k_2}(\vec{x}) V_{k_1,k_2}. \tag{C.8}$$

Fifth Step *We will insert the characteristic function*
We define the characteristic function of $f(\vec{y}|\vec{x})$ on the interval $[a_1,b_1]\times[a_2,b_2]$ by ϕ_A. Take ϕ as the characteristic function of $f(\vec{y}|\vec{x})$ on the domain $\mathbb{R}^2$. If the characteristic function of $f(\vec{y}|\vec{x})$ is known, then it will be defined on the whole domain $\mathbb{R}^2$. The function $f(\vec{y}|\vec{x})$ decays to zero very rapidly outside the domain $[\vec{a},\vec{b}]$. Therefore, ϕ_A will not differ much from ϕ.

$$A_{k_1,k_2}(\vec{x}) = \frac{2}{b_1-a_1}\frac{2}{b_2-a_2}\int_{a_2}^{b_2}\int_{a_1}^{b_1} f(\vec{y}|\vec{x})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2.$$

We make use of a trigonometric rule,

$$\cos(\theta_1)\cos(\theta_2) = \frac{1}{2}\left[\cos(\theta_1+\theta_2)+\cos(\theta_1-\theta_2)\right], \tag{C.9}$$

and denote $F_{k_1,k_2}(\vec{x})$ as

$$F_{k_1,k_2}(\vec{x}) = \iint_{\mathbb{R}^2} f(\vec{y}|\vec{x})\cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2.$$

[3]The function $v_2(\vec{x},t)$ is an approximation of $v_1(\vec{x},t)$. It contains a overall error consisting of two approximations compared to $v(t_0,\vec{S}(t_0))$.

From (C.9) it follows

$$F_{k_1,k_2}(\vec{x}) = \frac{1}{2}\left(F^{+}_{k_1,k_2}(\vec{x}) + F^{-}_{k_1,k_2}(\vec{x})\right),$$

where

$$F^{+}_{k_1,k_2}(\vec{x}) = \iint_{\mathbb{R}^2} f(\vec{y}|\vec{x}) \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1} + k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2,$$
$$F^{-}_{k_1,k_2}(\vec{x}) = \iint_{\mathbb{R}^2} f(\vec{y}|\vec{x}) \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1} - k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2.$$

$$F^{+}_{k_1,k_2}(\vec{x}) = \mathrm{Re}\left\{\iint_{\mathbb{R}^2} f(\vec{y}|\vec{x}) \exp\left(ik_1\pi\frac{y_1-a_1}{b_1-a_1} + ik_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2\right\} \tag{C.10}$$

$$= \mathrm{Re}\Bigg\{\iint_{\mathbb{R}^2} f(\vec{y}|\vec{x}) \exp\left(ik_1\pi\frac{y_1}{b_1-a_1} + ik_2\pi\frac{y_2}{b_2-a_2}\right) dy_1 dy_2 \exp\left(ik_1\pi\frac{-a_1}{b_1-a_1} + ik_2\pi\frac{-a_2}{b_2-a_2}\right)\Bigg\}. \tag{C.11}$$

The integral of (C.11) is the characteristic function of $f(\vec{y}|\vec{x})$. Writing $F^{+}_{k_1,k_2}(\vec{x})$ with its characteristic function gives

$$F^{+}_{k_1,k_2}(\vec{x}) = \mathrm{Re}\left\{\phi\left(\frac{k_1\pi}{b_1-a_1}, \frac{k_2\pi}{b_2-a_2}|\vec{x}\right) \exp\left(ik_1\pi\frac{-a_1}{b_1-a_1} + ik_2\pi\frac{-a_2}{b_2-a_2}\right)\right\}. \tag{C.12}$$

We can derive the formula for $F^{-}_{k_1,k_2}(\vec{x})$ in almost the same way.

$$F^{-}_{k_1,k_2}(\vec{x}) = \mathrm{Re}\left\{\phi\left(\frac{k_1\pi}{b_1-a_1}, -\frac{k_2\pi}{b_2-a_2}|\vec{x}\right) \exp\left(ik_1\pi\frac{-a_1}{b_1-a_1} - ik_2\pi\frac{-a_2}{b_2-a_2}\right)\right\}. \tag{C.13}$$

We combine (C.8), (C.12) and (C.13). This gives us the 2D COS pricing formula[4]:

$$\boxed{v_3(\vec{x},t) = e^{-r\Delta t}\frac{1}{2}{\sum_{k_1=0}^{N-1}}'{\sum_{k_2=0}^{N-1}}'\left(F^{+}_{k_1,k_2}(\vec{x}) + F^{-}_{k_1,k_2}(\vec{x})\right)V_{k_1,k_2}.} \tag{C.14}$$

C.2 Payoff coefficients

We defined V_{k_1,k_2} (C.6) by the equation

$$V_{k_1,k_2} = \frac{2}{b_1-a_1}\frac{2}{b_2-a_2}\int_{a_1}^{b_1}\int_{a_2}^{b_2} g(\vec{y}) \cos\left(k_1\pi\frac{y_1-a_1}{b_1-a_1}\right)\cos\left(k_2\pi\frac{y_2-a_2}{b_2-a_2}\right) dy_1 dy_2,$$

where $g(\vec{y})$ is the payoff function of the option which depends on the asset prices $\vec{S}(T)$ at time T.
In practice, when the characteristic function is known it will be the characteristic function of the log-asset prices, which is known. Therefore, the payoff function has to be related to the log-asset prices. We perform a change of variables to achieve this transformation. Let $\hat{x}$ and $\hat{y}$ be defined as

$$\hat{x} := \log\left(\frac{\vec{S}(t)}{K}\right) \quad \hat{y} := \log\left(\frac{\vec{S}(T)}{K}\right),$$

then $\hat{x}$ is a log-asset process. The payoff of a two-dimensional European arithmetic basket call option becomes

$$g(\hat{y}) = \max(K(0.5(e^{\hat{y}_1} + e^{\hat{y}_2}) - 1), 0). \tag{C.15}$$

It is generally not possible to calculate the payoff coefficients analytically for a two-dimensional European option. Therefore, we will use the two-dimensional DCT (see section 3.5).

[4] $v_3(\vec{x},t)$ contains a overall error consisting of three approximations compared to $v(\vec{x},t)$.

C.3 Truncation range

Also in 2D, we need to choose a finite domain $[a_1, b_1]\times[a_2, b_2]$, such that the truncated integral approximates the infinite integral closely. The integration range $[a_1, b_1] \times [a_2, b_2]$ we used in step 1 of section (C.1) is directly taken from [33],

$$a_i := \hat{x}_i + \xi_i^1 - L\sqrt{\xi_i^2 + \sqrt{\xi_i^4}},$$
$$b_i := \hat{x}_i + \xi_i^1 + L\sqrt{\xi_i^2 + \sqrt{\xi_i^4}},$$

where ξ^1, ξ^2 and ξ^4 are the cumulants (see section 4.3), L is a scaling parameter. Tests by [33] show that $L = 10$ will give good results for option pricing with $\Delta t = 1$.

C.4 Error analysis

In this section we discuss the errors in the numerical approximation. We will show that the overall error has an algebraic convergence. Therefore, we perform four steps. We follow Ruijter and Oosterlee [33].

1. *First error*
   The first error appears at the truncation of the integration range (see section C.1). The error can be written as:

$$\epsilon^1 = v(\vec{x}, t) - v_1(\vec{x}, t) = e^{-r\Delta t} \iint_{\mathbb{R}^2\setminus[a_1,b_1]\times[a_2,b_2]} g(\vec{S}(T)) f(\vec{y}|\vec{x}) d\vec{y}.$$

2. *Second error*
   The second error arises at the truncation of the series summation on $[a_1, b_1] \times [a_2, b_2]$. The error can be written as:

$$\epsilon^2 = v_1(\vec{x}, t) - v_2(\vec{x}, t) = \frac{b_1 - a_1}{2}\frac{b_2 - a_2}{2} e^{-r\Delta t} \sum_{k_1=N}^{\infty} \sum_{k_2=N}^{\infty} A_{\vec{k}}(\vec{x}) V_{\vec{k}}.$$

3. *Third error*
   The third error arises by inserting the Fourier-cosine transform and can be written as:

$$\begin{aligned}
\epsilon^3 &= v_2(\vec{x}, t) - v_3(\vec{x}, t) \\
&= \frac{b_1 - a_1}{2}\frac{b_2 - a_2}{2} e^{-r\Delta t} {\sum_{k_1=0}^{N_1-1}}' {\sum_{k_2=0}^{N_2-1}}' (A_{\vec{k}}(\vec{x}) - F_{\vec{k}}(\vec{x})) V_{\vec{k}} \\
&= e^{-r\Delta t} \iint_{\mathbb{R}^2\setminus[a_1,b_1]\times[a_2,b_2]} \\
&\left[{\sum_{k_1=0}^{N_1-1}}' {\sum_{k_2=0}^{N_2-1}}' \cos\left(k_1\pi\frac{y_1 - a_1}{b_1 - a_1}\right) \cos\left(k_2\pi\frac{y_2 - a_2}{b_2 - a_2}\right) V_{\vec{k}} \right] f(\vec{y}|\vec{x}) d\vec{y}.
\end{aligned}$$

4. *Fourth error*
   This extra error occurs because the elements of V_{k_1,k_2} are approximated by a DCT.

$$\epsilon^{DCT} = \frac{b_1 - a_1}{2}\frac{b_2 - a_2}{2} e^{-r\Delta t} {\sum_{k_1=0}^{N_1-1}}' {\sum_{k_2=0}^{N_2-1}}' F_{\vec{k}}(\vec{x}) [V_{k_1,k_2}(T) - V_{k_1,k_2}^{DCT}(T)].$$

If the V_{k_1,k_2}s are known, the fourth error does not occur. In that situation, the overall error will be dominated by the second error when the integration range is chosen large enough. That means that smooth probability density functions of class $\mathbb{C}^\infty([a_1,b_1]\times[a_2,b_2])$ will have an overall error ϵ which converges exponentially in N.

However, if the V_{k_1,k_2}s are not known, we have to approximate the V_{k_1,k_2}s by the DCT. And consequently, the fourth error will occur. In this situation, this error will dominate the overall error. Then, for smooth density functions of the aforementioned class, the overall error converges algebraically in Q of second order.

C.5 Complexity

In the previous section we showed that the decay of the convergence rate of the overall error is algebraic. Therefore, the question arises by how much the computational time increases. In this section, we discuss this question. We focus on the complexity of the calculations of the 2D COS method. The calculation complexity can be derived from the 2D COS pricing formula

$$v_3(\vec{x},t) = e^{-r\Delta t}\frac{1}{2}{\sum_{k_1=0}^{N-1}}'{\sum_{k_2=0}^{N-1}}'\left(F^+_{k_1,k_2}(\vec{x}) + F^-_{k_1,k_2}(\vec{x})\right)V_{k_1,k_2}.$$

The calculation of this sum consists of N^2 elements for $F^+_{k_1,k_2}(\vec{x})$, $F^-_{k_1,k_2}(\vec{x})$ and V_{k_1,k_2}. The time to calculate any of the F_{k_1,k_2} elements will be the same for every k_1 and k_2. Thus, the calculation time of all F_{k_1,k_2}s is $\mathcal{O}(N^2)$, where $\mathcal{O}$ denotes the order.
We need a DCT to calculate the elements of V_{k_1,k_2}. From section 3.4 we know that a single DCT with the FFT method has calculation complexity $\mathcal{O}(Q\log_2(Q))$. The DCT has to be carried out for both dimensions for each vector; hence, there are $2Q$ DCTs. Thus, the complexity to calculate all the elements of V_{k_1,k_2} is $\mathcal{O}(Q^2\log_2(Q))$.

The summation also has an $\mathcal{O}(N^2)$ complexity. Q has to be at least the same value as N. Even when taking Q equal to N, the total complexity of the 2D COS method is:

$$\mathcal{O}(Q^2\log_2(Q)).$$

From the notion $\mathcal{O}(Q^2\log_2(Q))$ it follows that the calculation complexity is at least quadratic in the number of terms Q.

C.6 Conclusion

We conclude that the 2D COS method is accurate and fast. Accurate because the error analysis shows that the convergence rate of the error is algebraically and of second order; the V_{k_1,k_2} terms are not exact. Fast because its computational complexity is at least quadratic in the number of terms Q. It is important to choose the truncation range carefully.

Appendix D

Payoff kernel 1D and 3D

In chapter 6, we have performed calculations of the different payoff kernels by means of CUDA. And, we have presented the 2D kernel. In this appendix we show the other two kernels.

D.1 1D exact put payoff

```
__device__ double psi(int k, double a, double b, double c, double d)
{
    if(k==0) return d-c;
    else
    {
        double kapi = (k*M_PI)/(b-a);
        return (sin((d-a)*kapi)-sin((c-a)*kapi))/kapi;
    }
}

__device__ double xi(int k, double a, double b, double c, double d)
{
    double kapi = (k*M_PI)/(b-a);
    double termd = (d-a)*kapi;
    double termc = (c-a)*kapi;

    double term1 = cos(termd)*exp(d);
    double term2 = cos(termc)*exp(c);
    double term3 = sin(termd)*exp(d)*kapi;
    double term4 = sin(termc)*exp(c)*kapi;

    return (1/(1+kapi*kapi))*(term1-term2+term3-term4);
}

__global__ void payoffput1DGPU_exact(double *d_A, double a, double b, double K, int N)
{
    int ii = threadIdx.x + blockDim.x * blockIdx.x;
    d_A[ii] = (2/(b-a))*K*(psi(ii,a,b,a,0.0)-xi(ii,a,b,a,0.0));
}
```

D.2 3D geometric put payoff

```
__global__ void payoffgeo3DGPU(double *d_A, double *d_a, double *d_b, double K, int Q){
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    int z = threadIdx.z + blockDim.z * blockIdx.z;
    int offset = x + y*Q + z*Q*Q;

    if(x < Q && y < Q && z < Q)
    {
        // Geometric payoff
        double x2 = d_a[0] + (x + 0.5) * (d_b[0]-d_a[0])/Q;
        double y2 = d_a[1] + (y + 0.5) * (d_b[1]-d_a[1])/Q;
        double z2 = d_a[2] + (z + 0.5) * (d_b[2]-d_a[2])/Q;
        d_A[offset]=MAX(K*(1-cbrt(exp(x2+y2+z2))),0.0);
    }
}
```

Appendix E

MATLAB Code

In chapter 7 we have performed our numerical experiments. For these experiments we have developed specific codes for MATLAB, C and CUDA. In this appendix we show the MATLAB codes for the 1D, 2D and 3D options under GBM and MJD.

E.1 1D COS method GBM

```
function [ calloption ] = COS1DGBMCALL(S0,K,tau,r,sig,N)
    %COS   Value of 1D option with the 1D COS method.
    %   Input of the COS function is S0,K,tau,r,sig,N. Where
    %   S0   [1x1]   Start prices of assets
    %   K    [1x1]   Strike price
    %   tau  [1x1]   Time to maturity
    %   r    [1x1]   Risk free rate
    %   sig  [1x1]   Variance of assets
    %   N    [1x1]   Number of terms in each dimension
    %
    %   COS1DGBMCALL(S0,K,tau,r,sig,N) is a single element.

    [a,b] = abgbm(log(S0/K), r, sig, tau);

    Vk = payoffput1D_dctexact(a,b,K,N);
    Gk = chargbm1D(a,b,tau,S0,K,r,sig,N);

    tot = totsum(Vk,Gk);

    putoption = exp(-r*tau)*tot;
    calloption = S0 + putoption - K * exp(-r*tau);
end
```

E.2 1D COS method MJD

```
function [ calloption ] = COS1DMJDCALL(S0,K,tau,r,sig,alpha,delta,lambda,N)
%COS   Value of 1D option with the 1D COS method.
%   Input of the COS function is S0,K,tau,r,sig,alpha,delta,lambda,N. Where
%   S0   [1x1]   Start prices of assets
%   K    [1x1]   Strike price
%   tau  [1x1]   Time to maturity
%   r    [1x1]   Risk free rate
```

```
%   sig  [1x1]   Variance of assets
%   alpha[1x1]   Mean of the jumps
%   delta[1x1]   Variance of the jumps
%   lambda[1x1]  Intensity of the jumps
%   N    [1x1]   Number of terms in each dimension
%
%   COS1DMJDCALL(S0,K,tau,r,sig,alpha,delta,lambda,N) is a single element.


[a,b] = abmjd(log(S0/K), r, sig, alpha, delta, lambda, tau);

Vk = payoffput1D_dctexact(a,b,K,N);
Gk = charmjd1D(a,b,tau,S0,K,r,sig,alpha,delta,lambda,N);

tot = totsum(Vk,Gk);

putoption = exp(-r*tau)*tot;
calloption = S0 + putoption - K * exp(-r*tau);
end
```

E.3 2D COS method GBM

```
function [ putoption ] = COS2DGBMGEO(S0,K,tau,r,sig,rho,N,Q)
    %COS   Value of 2D option with the 2D COS method.
    %   Input of the 2D COS function is S0,K,tau,r,sig,rho,N,Q. Where
    %   S0   [2x1]   Start prices of assets
    %   K    [1x1]   Strike price
    %   tau  [1x1]   Time to maturity
    %   r    [1x1]   Risk free rate
    %   sig  [2x1]   Variance of assets
    %   rho  [2x2]   Covariance matrix
    %   N    [1x1]   Number of terms in each dimension
    %   Q    [1x1]   Number of terms in the approximation of the payoff
    %
    %   COS2DGBMGEO(S0,K,tau,r,sig,rho,N,Q) is a single element.

    [a,b] = abgbm(log(S0/K), r, sig, tau);

    P = payoffgeo2D(a,b,K,Q);
    Vk = dctn(P);
    Gk = chargbm2D(a,b,tau,S0,K,r,sig,rho,N);

    tot = totsum(Vk(1:N,1:N),Gk);

    putoption = exp(-r*tau)*0.5*tot;
end
```

E.4 2D COS method MJD

```
function [ putoption ] = COS2DMJDGEO(S0,K,tau,r,sig,alpha,delta,lambda,rho,rhoJ,N,Q)
%COS   Value of 2D option with the 2D COS method.
```

```
%   Input of the 2D COS function is S0,K,tau,r,sig,alpha,delta,lambda,rho,rhoJ,N,Q. Where
%   S0    [2x1]   Start prices of assets
%   K     [1x1]   Strike price
%   tau   [1x1]   Time to maturity
%   r     [1x1]   Risk free rate
%   sig   [2x1]   Variance of assets
%   alpha[2x1]    Mean of the jumps
%   delta[2x1]    Variance of the jumps
%   lambda[1x1]   Intensity of the jumps
%   rho   [2x2]   Covariance matrix of BS part
%   rhoJ  [2x2]   Covariance matrix of Jump part
%   N     [1x1]   Number of terms in each dimension
%   Q     [1x1]   Number of terms in the approximation of the payoff
%
%   COS2DMJDGEO(S0,K,tau,r,sig,alpha,delta,lambda,rho,rhoJ,N,Q) is a single element.

[a,b] = abmjd(log(S0/K), r, sig, alpha, delta, lambda, tau);

P = payoffgeo2D(a,b,K,Q);
Vk = dctn(P);
Gk = charmjd2D(a,b,tau,S0,K,r,sig,alpha,delta,lambda,rho,rhoJ,N);

tot = totsum(Vk(1:N,1:N),Gk);

putoption = exp(-r*tau)*0.5*tot;
end
```

E.5 3D COS method GBM

```
function [ putoption ] = COS3DGBMGEO(S0,K,tau,r,sig,rho,N,Q)
    %COS   Value of 3D option with the 3D COS method.
    %   Input of the 3D COS function is S0,K,tau,r,sig,rho,N,Q. Where
    %   S0    [3x1]   Start prices of assets
    %   K     [1x1]   Strike price
    %   tau   [1x1]   Time to maturity
    %   r     [1x1]   Risk free rate
    %   sig   [3x1]   Variance of assets
    %   rho   [3x3]   Covariance matrix
    %   N     [1x1]   Number of terms in each dimension
    %   Q     [1x1]   Number of terms in the approximation of the payoff
    %
    %   COS3DGBMGEO(S0,K,tau,r,sig,rho,N,Q) is a single element.

    [a,b] = abgbm(log(S0/K), r, sig, tau);

    P = payoffgeo3D(a,b,K,Q);
    Vk = dctn(P);
    Gk = chargbm3D(a,b,tau,S0,K,r,sig,rho,N);

    tot = totsum(Vk(1:N,1:N,1:N),Gk);

    putoption = exp(-r*tau)*0.25*tot;
end
```

E.6 3D COS method MJD

```
function [ putoption ] = COS3DMJDGEO(S0,K,tau,r,sig,alpha,delta,lambda,rho,rhoJ,N,Q)
    %COS   Value of 3D option with the 3D COS method.
    %   Input of the 3D COS function is S0,K,tau,r,sig,alpha,delta,lambda,rho,rhoJ,N,Q. Where
    %   S0    [3x1]   Start prices of assets
    %   K     [1x1]   Strike price
    %   tau   [1x1]   Time to maturity
    %   r     [1x1]   Risk free rate
    %   sig   [3x1]   Variance of assets
    %   alpha[3x1]    Mean of the jumps
    %   delta[3x1]    Variance of the jumps
    %   lambda[1x1]   Intensity of the jumps
    %   rho   [3x3]   Covariance matrix of BS part
    %   rhoJ  [3x3]   Covariance matrix of Jump part
    %   N     [1x1]   Number of terms in each dimension
    %   Q     [1x1]   Number of terms in the approximation of the payoff
    %
    %   COS3DMJDGEO(S0,K,tau,r,sig,alpha,delta,lambda,rho,rhoJ,N,Q) is a single element.

    [a,b] = abmjd(log(S0/K), r, sig, alpha, delta, lambda, tau);

    P = payoffgeo3D(a,b,K,Q);
    Vk = dctn(P);
    Gk = charmjd3D(a,b,tau,S0,K,r,sig,alpha,delta,lambda,rho,rhojump,N);

    tot = totsum(Vk(1:N,1:N,1:N),Gk);

    putoption = exp(-r*tau)*0.25*tot;
end
```

Appendix F

C code

In chapter 7 we have performed our numerical experiments. For these experiments we have developed specific codes for MATLAB, C and CUDA. In this appendix we show the C codes for the 1D, 2D and 3D options under GBM and MJD.

F.1 1D COS method GBM

```
double COS1DGBMCALL(double S0, double K, double tau, double r, double sig, int N)
{
    double *V, *G;

    double L = 10.0;
    double a = log(S0/K) + tau*(r-0.5*sig*sig) - L*sqrt(tau*(sig*sig));
    double b = log(S0/K) + tau*(r-0.5*sig*sig) + L*sqrt(tau*(sig*sig));

    V = (double *) malloc(N*sizeof(V[0]));
    G = (double *) malloc(N*sizeof(G[0]));

    payoffput1D_exact(V,a,b,K,N);
    chargbm1D(G,a,b,tau,S0,K,r,sig,N);

    double put = exp(-r*tau)*dotCPUblas(V,G,N);
    free(V);
    free(G);

    return S0 + put - K * exp(-r*tau);
}
```

F.2 1D COS method MJD

```
double COS1DMJDCALL(double S0, double K, double tau, double r, double sig, double alpha,
   double delta, double lambda, int N)
{
    double *V, *G;

    double k = exp(alpha+0.5*delta*delta)-1.0;
    double mu = r - lambda * k - 0.5*sig*sig;
    double c1 = tau*(mu+lambda*alpha);
    double c2 = tau*(sig*sig+lambda*alpha*alpha+lambda*delta*delta);
```

```
    double c4 = tau*lambda*(alpha*alpha*alpha*alpha+6*delta*delta*alpha*alpha
        +3*delta*delta*delta*delta);
    double L = 10.0;
    double a = log(S0/K) +c1 - L * sqrt(c2 + sqrt(c4));
    double b = log(S0/K) +c1 + L * sqrt(c2 + sqrt(c4));

    V = (double *) malloc(N*sizeof(V[0]));
    G = (double *) malloc(N*sizeof(G[0]));

    payoffput1D_exact(V,a,b,K,N);
    charmjd1D(G,a,b,tau,S0,K,r,sig,alpha,delta,lambda,N);

    double put = exp(-r*tau)*dotCPUblas(V,G,N);
    free(V);
    free(G);

    return S0 + put - K * exp(-r*tau);
}
```

F.3 2D COS method GBM

```
double COS2DGBMGEO(double *S0, double K, double tau, double r, double *sig, double *rho,
   int N, int Q)
{
    double *P, *V, *G, *VN;
    double a[2];
    double b[2];
    abgbm(a,b,S0,K,r,sig,tau,2);

    P = (double *) malloc(Q*Q*sizeof(P[0]));
    V = (double *) malloc(Q*Q*sizeof(V[0]));
    VN= (double *) malloc(N*N*sizeof(VN[0]));
    G = (double *) malloc(N*N*sizeof(G[0]));

    payoffgeo2D(P,a,b,K,Q);
    dct2DFFT(V,P,Q);
    QtoN2D(VN,V,N,Q);
    chargbm2D(G,a,b,tau,S0,K,r,sig,rho,N);

    double dot = dotCPUblas(G,VN,N*N);

    free(G);
    free(VN);
    free(V);
    free(P);

    return exp(-r*tau)*0.5*(1.0/(Q*Q))*dot;
}
```

F.4 2D COS method MJD

```
double COS2DMJDGEO(double *S0, double K, double tau, double r, double *sig, double *alpha,
   double *delta, double lambda, double *rho, double *rhoJ, int N, int Q)
{
    double *P, *V, *G, *VN;
    double a[2];
    double b[2];
    double logS0[2] = {log(S0[0]/K), log(S0[1]/K)};
    abmjd(a, b, logS0, r, sig, tau, alpha, delta, lambda, 2);

    P = (double *) malloc(Q*Q*sizeof(P[0]));
    V = (double *) malloc(Q*Q*sizeof(V[0]));
    VN= (double *) malloc(N*N*sizeof(VN[0]));
    G = (double *) malloc(N*N*sizeof(G[0]));

    payoffgeo2D(P,a,b,K,Q);
    dct2DFFT(V,P,Q);
    QtoN2D(VN,V,N,Q);
    charmjd2D(G,a,b,tau,S0,K,r,sig,alpha,delta,lambda,rho,rhoJ,N);

    double dot = dotCPUblas(G,VN,N*N);

    free(G);
    free(VN);
    free(V);
    free(P);

    return exp(-r*tau)*0.5*(1.0/(Q*Q))*dot;
}
```

F.5 3D COS method GBM

```
double COS3DGBMGEO(double *S0, double K, double tau, double r, double *sig, double *rho,
   int N, int Q)
{
    double *P, *V, *G, *VN;
    double a[3];
    double b[3];

    abgbm(a,b,S0,K,r,sig,tau,3);

    P = (double *) malloc(Q*Q*Q*sizeof(P[0]));
    V = (double *) malloc(Q*Q*Q*sizeof(V[0]));
    VN= (double *) malloc(N*N*N*sizeof(VN[0]));
    G = (double *) malloc(N*N*N*sizeof(G[0]));

    payoffgeo3D(P,a,b,K,Q);
    dct3DFFT(V,P,Q);
    QtoN3D(VN,V,N,Q);
    chargbm3D(G,a,b,tau,S0,K,r,sig,rho,N);

    double dot = dotCPUblas(G,VN,N*N*N);
```

```
    free(G);
    free(VN);
    free(V);
    free(P);

    return exp(-r*tau)*0.25*(1.0/(Q*Q*Q))*dot;
}
```

F.6 3D COS method MJD

```
double COS3DMJDGEO(double *S0, double K, double tau, double r, double *sig, double *alpha,
   double *delta, double lambda, double *rho, double *rhoJ, int N, int Q)
{
    double *P, *V, *G, *VN;
    double a[3];
    double b[3];
    double logS0[3] = {log(S0[0]/K),log(S0[1]/K),log(S0[2]/K)};

    abmjd(a,b,logS0,r,sig,tau,alpha,delta,lambda,3);

    P = (double *) malloc(Q*Q*Q*sizeof(P[0]));
    V = (double *) malloc(Q*Q*Q*sizeof(V[0]));
    VN= (double *) malloc(N*N*N*sizeof(VN[0]));
    G = (double *) malloc(N*N*N*sizeof(G[0]));

    payoffgeo3D(P,a,b,K,Q);
    dct3DFFT(V,P,Q);
    QtoN3D(VN,V,N,Q);
    charmjd3D(G,a,b,tau,S0,K,r,sig,alpha,delta,lambda,rho,rhoJ,N);

    double dot = dotCPUblas(G,VN,N*N*N);

    free(G);
    free(VN);
    free(V);
    free(P);

    return exp(-r*tau)*0.25*(1.0/(Q*Q*Q))*dot;
}
```

Appendix G

CUDA code

In chapter 7 we have performed our numerical experiments. For these experiments we have developed specific codes for MATLAB, C and CUDA. In this appendix we show the CUDA codes for the 1D, 2D and 3D options under GBM and MJD.

G.1 1D COS method GBM

```
double GPU_COS1DGBMCALL(double S0, double K, double tau, double r, double sig, int N)
{
    double *d_V, *d_G;

    double L = 10.0;
    double a = log(S0/K) + tau*(r-0.5*sig*sig) - L*sqrt(tau*(sig*sig));
    double b = log(S0/K) + tau*(r-0.5*sig*sig) + L*sqrt(tau*(sig*sig));

    cudaMalloc((void**)&d_V, N*sizeof(d_V[0]));
    cudaMalloc((void**)&d_G, (N+1)*sizeof(d_G[0]));

    int thr = 256;
    int tpb = (N+thr-1)/thr;
    payoffput1DGPU_exact<<<tpb,thr>>>(d_V,a,b,K,N);
    chargbm1DGPU(d_G,a,b,tau,S0,K,r,sig,N);

    double put = exp(-r*tau)*dotGPU_wrapper(d_V,d_G,&d_G[N],N);
    cudaFree(d_V);
    cudaFree(d_G);

    return S0 + put - K * exp(-r*tau);
}
```

G.2 1D COS method MJD

```
double GPU_COS1DMJDCALL(double S0, double K, double tau, double r, double sig, double
    alpha, double delta, double lambda, int N)
{
    double *d_V, *d_G;

    double k = exp(alpha+0.5*delta*delta)-1.0;
    double mu = r - lambda * k - 0.5*sig*sig;
```

```
    double c1 = tau*(mu+lambda*alpha);
    double c2 = tau*(sig*sig+lambda*alpha*alpha+lambda*delta*delta);
    double c4 = tau*lambda*(alpha*alpha*alpha*alpha+6*delta*delta*alpha*alpha
        +3*delta*delta*delta*delta);
    double L = 10.0;
    double a = log(S0/K) +c1 - L * sqrt(c2 + sqrt(c4));
    double b = log(S0/K) +c1 + L * sqrt(c2 + sqrt(c4));

    cudaMalloc((void**)&d_V,N*sizeof(d_V[0]));
    cudaMalloc((void**)&d_G,(N+1)*sizeof(d_G[0]));

    int thr = 256;
    int tpb = (N+thr-1)/thr;
    payoffput1DGPU_exact<<<tpb,thr>>>(d_V,a,b,K,N);
    charmjd1DGPU(d_G,a,b,tau,S0,K,r,sig,alpha,delta,lambda,N);

    double put = exp(-r*tau)*dotGPU_wrapper(d_V,d_G,&d_G[N],N);
    cudaFree(d_V);
    cudaFree(d_G);

    return S0 + put - K * exp(-r*tau);
}
```

G.3 2D COS method GBM

```
double GPU_COS2DGBMGEO(double *S0, double K, double tau, double r, double *sig,
    double *rho, int N, int Q)
{
    double *d_P, *d_V, *d_G;
    double a[2];
    double b[2];
    abgbm(a,b,S0,K,r,sig,tau,2);

    cudaMalloc((void**)&d_P,Q*Q*sizeof(d_P[0]));
    cudaMalloc((void**)&d_V,N*N*sizeof(d_V[0]));
    cudaMalloc((void**)&d_G,(N*N+1)*sizeof(d_G[0]));

    double *d_a, *d_b;
    cudaMalloc((void**)&d_a,2*sizeof(d_a[0]));
    cudaMalloc((void**)&d_b,2*sizeof(d_b[0]));
    cudaMemcpy(d_a,a,2*sizeof(d_a[0]),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b,2*sizeof(d_b[0]),cudaMemcpyHostToDevice);

    int thr = 16;
    dim3 TpB(thr,thr);
    dim3 grid((Q+thr-1)/thr,(Q+thr-1)/thr);


    payoffgeo2DGPU<<<grid,TpB>>>(d_P,d_a,d_b,K,Q);
    dct2DFFTGPU(d_V,d_P,Q,N);
    chargbm2DGPU(d_G,a,b,tau,S0,K,r,sig,rho,N);

    double dot = dotGPU_wrapper(d_V,d_G,&d_G[N*N],N*N);
```

```
    cudaFree(d_G);
    cudaFree(d_V);
    cudaFree(d_P);
    cudaFree(d_a);
    cudaFree(d_b);

   return exp(-r*tau)*0.5*(1.0/(Q*Q))*dot;
}
```

G.4 2D COS method MJD

```
double GPU_COS2DMJDGEO(double *S0, double K, double tau, double r, double *sig,
    double *alpha, double *delta, double lambda, double *rho, double *rhoJ, int N, int Q)
{
    double *d_P, *d_V, *d_G;
    double a[2];
    double b[2];
    double logS0[2] = {log(S0[0]/K), log(S0[1]/K)};
    abmjd(a, b, logS0, r, sig, tau, alpha, delta, lambda, 2);

    cudaMalloc((void**)&d_P,Q*Q*sizeof(d_P[0]));
    cudaMalloc((void**)&d_V,N*N*sizeof(d_V[0]));
    cudaMalloc((void**)&d_G,(N*N+1)*sizeof(d_G[0]));

    double *d_a, *d_b;
    cudaMalloc((void**)&d_a,2*sizeof(d_a[0]));
    cudaMalloc((void**)&d_b,2*sizeof(d_b[0]));
    cudaMemcpy(d_a,a,2*sizeof(d_a[0]),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b,2*sizeof(d_b[0]),cudaMemcpyHostToDevice);

    int thr = 16;
    dim3 TpB(thr,thr);
    dim3 grid((Q+thr-1)/thr,(Q+thr-1)/thr);
    payoffgeo2DGPU<<<grid,TpB>>>(d_P,d_a,d_b,K,Q);
    dct2DFFTGPU(d_V,d_P,Q,N);
    charmjd2DGPU(d_G,a,b,tau,S0,K,r,sig,alpha,delta,lambda,rho,rhoJ,N);

    double dot = dotGPU_wrapper(d_V,d_G,&d_G[N*N],N*N);

    cudaFree(d_G);
    cudaFree(d_V);
    cudaFree(d_P);
    cudaFree(d_a);
    cudaFree(d_b);

    return exp(-r*tau)*0.5*(1.0/(Q*Q))*dot;
}
```

G.5 3D COS method GBM

```
double GPU_COS3DGBMGEO(double *S0, double K, double tau, double r, double *sig,
    double *rho, int N, int Q)
{
    double *d_P, *d_V, *d_G;
    double a[3];
    double b[3];

    abgbm(a,b,S0,K,r,sig,tau,3);

    cudaMalloc((void**)&d_P,Q*Q*Q*sizeof(d_P[0]));
    cudaMalloc((void**)&d_V,N*N*N*sizeof(d_V[0]));
    cudaMalloc((void**)&d_G,(N*N*N+1)*sizeof(d_G[0]));

    double *d_a, *d_b;
    cudaMalloc((void**)&d_a,3*sizeof(d_a[0]));
    cudaMalloc((void**)&d_b,3*sizeof(d_b[0]));
    cudaMemcpy(d_a,a,3*sizeof(d_a[0]),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b,3*sizeof(d_b[0]),cudaMemcpyHostToDevice);

    int thr = 8;
    dim3 TpB(thr,thr,thr);
    dim3 grid((Q+thr-1)/thr,(Q+thr-1)/thr,(Q+thr-1)/thr);
    payoffgeo3DGPU<<<grid,TpB>>>(d_P,d_a,d_b,K,Q);
    dct3DFFTGPU(d_V,d_P,Q,N);
    chargbm3DGPU(d_G,a,b,tau,S0,K,r,sig,rho,N);

    double dot = dotGPU_wrapper(d_V,d_G,&d_G[N*N*N],N*N*N);

    cudaFree(d_G);
    cudaFree(d_V);
    cudaFree(d_P);
    cudaFree(d_a);
    cudaFree(d_b);

    return exp(-r*tau)*0.25*(1.0/(Q*Q*Q))*dot;
}
```

G.6 3D COS method MJD

```
double GPU_COS3DMJDGEO(double *S0, double K, double tau, double r, double *sig, double *alpha,
     double *delta, double lambda, double *rho, double *rhoJ, int N, int Q)
{
    double *d_P, *d_V, *d_G;
    double a[3];
    double b[3];
    double logS0[3] = {log(S0[0]/K),log(S0[1]/K),log(S0[2]/K)};

    abmjd(a,b,logS0,r,sig,tau,alpha,delta,lambda,3);

    cudaMalloc((void**)&d_P,Q*Q*Q*sizeof(d_P[0]));
    cudaMalloc((void**)&d_V,N*N*N*sizeof(d_V[0]));
```

```
    cudaMalloc((void**)&d_G,(N*N*N+1)*sizeof(d_G[0]));

    double *d_a, *d_b;
    cudaMalloc((void**)&d_a,3*sizeof(d_a[0]));
    cudaMalloc((void**)&d_b,3*sizeof(d_b[0]));
    cudaMemcpy(d_a,a,3*sizeof(d_a[0]),cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b,3*sizeof(d_b[0]),cudaMemcpyHostToDevice);

    int thr = 8;
    dim3 TpB(thr,thr,thr);
    dim3 grid((Q+thr-1)/thr,(Q+thr-1)/thr,(Q+thr-1)/thr);
    payoffgeo3DGPU<<<grid,TpB>>>(d_P,d_a,d_b,K,Q);
    dct3DFFTGPU(d_V,d_P,Q,N);
    charmjd3DGPU(d_G,a,b,tau,S0,K,r,sig,alpha,delta,lambda,rho,rhoJ,N);

    double dot = dotGPU_wrapper(d_V,d_G,&d_G[N*N*N],N*N*N);

    cudaFree(d_G);
    cudaFree(d_V);
    cudaFree(d_P);
    cudaFree(d_a);
    cudaFree(d_b);

    return exp(-r*tau)*0.25*(1.0/(Q*Q*Q))*dot;
}
```

Bibliography

[1] D. Applebaum. Lévy Processes - From Probability to Finance and Quantum Groups. *Notices of the AMS*, Vol. 51(11), 2004.

[2] D. Applebaum. *Lévy Processes and Stochastic Calculus*. Cambridge University Press, 2009.

[3] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, Vol. 81(3):637–654, 1973.

[4] M. Bouzoubaa and A. Osseiran. *Exotic Options and Hybrids: A Guide to Structuring, Pricing and Trading*. John Wiley & Sons, 2010.

[5] J.P. Boyd. *Chebyshev and Fourier Spectral Methods*. Springer-Verlag, Berlin, 2000.

[6] P.P. Carr and D.B. Madan. Option Valuation using the Fast Fourier Transform. *The Journal of Computational Finance*, Vol. 2(4):61–73, 1999.

[7] U. Cherubini, G. Della Lunga, S. Mulinacci, and P. Rossi. *Fourier transform methods in finance*. John Wiley & Sons, 2010.

[8] F. Cong and C.W. Oosterlee. Pricing Bermudan Options under Merton Jump-Diffusion Asset Dynamics. *International Journal of Computer Mathematics, Forthcoming. Available at SSRN:* `http://ssrn.com/abstract=2557958`, 2015.

[9] R. Cont and P. Tankov. *Financial Modelling with Jump Processes*. Chapman & Hall, 2004.

[10] J.W. Cooley and J.W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, pages 297–301, 1965.

[11] J.M. Courtault et al. Louis Bachelier on the centenary of Théorie de la Spéculation. *Mathematical Finance*, Vol. 10(3):341–350, 2000.

[12] P. Davis and P. Rabinowitz. *Methods of Numerical Integration*. Academic Press Inc., 1984.

[13] M.A.H. Dempster and S.S.G. Hong. Pricing spread options with the fast Fourier transform. `http://www.cfr.statslab.cam.ac.uk/publications/content/presentations/bachelier.pdf`, 2000.

[14] D. Ding. Fourier Transform Methods for Option Pricing. *Fourier transform applications, editor S.M. Salih, InTech*, 2012.

[15] D. Ding and S.C. U. Efficient Option Pricing Methods Based on Fourier Series Expansions. *Journal of Mathematical & Research Exposition*, 2011.

[16] F. Fang. *The COS Method: An Efficient Fourier Method for Pricing Financial Derivatives*. Delft University of Technology, 2010.

[17] F. Fang and C.W. Oosterlee. A novel pricing method for European options based on Fourier-cosine series expansions. *SIAM Journal on Scientific Computing*, Vol. 31(2):826–848, 2008.

[18] F. Fang and C.W. Oosterlee. Pricing Eearly-Exercise and Discrete Barrier Options by Fourier-Cosine Series Expansions. *Numerische Mathematik, Springer*, Vol. 114:27–62, 2009.

[19] N. Finizio and G. Ladas. *An Introduction to Differential Equations.* Wadsworth Publishing Company, 1982.

[20] J.M. Harrison and S.R. Pliska. Martingales and stochastic integrals in the theory of continuous trading. *Stochastic Processes and Their Applications North-Holland Publishing Company*, Vol. 11(1):215–260, 1980.

[21] J.C. Hull. *Options, Futures and Other Derivatives.* Prentice Hall, (8th ed.), 2012.

[22] H.P. Knibbe. *Acceleration of computing time for seismic applications based on the Helmholtz equation by Graphics Processing Units.* Delft University of Technology, 2015.

[23] C.C.W. Leentvaar and C.W. Oosterlee. Multi-asset option pricing using a parallel Fourier-based technique. *The Journal of Computational Finance*, Vol. 12(1):1–26, 2008.

[24] J. Makhoul. A fast cosine transform in one and two dimensions. *IEEE Transactions on acoustics, speech, and signal processing*, Vol. ASSP-28(1), 1980.

[25] W. Margrabe. The Value of an Option to Exchange One Asset for Another. *The Journal of Finance*, Vol. 33(1):177–186, 1978.

[26] K. Matsuda. Introduction to Merton Jump Diffusion Model. *Department of Economics, The Graduate Center, The City University of New York*, 2004.

[27] R.C. Merton. Theory of rational option pricing. *The Bell Journal of Economics and Management Science*, Vol. 4(1):141–183, 1973.

[28] R.C. Merton. Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics, North-Holland Publishing Company*, Vol. 3:125–144, 1976.

[29] NVIDIA. CUDA C Programming Guide. *https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf*, 2015.

[30] C.W. Oosterlee, L.A. Grzelak, and F. Fang. *Efficient Valuation and Computation in Finance.* manuscript, 2014.

[31] A. Papapantoleon. An introduction to Lévy processes with applications in finance. *Lecture notes, available at http://arxiv.org/abs/0804.0482*, 2005.

[32] T. Pellegrino and P. Sabino. Pricing and hedging multi-asset spread options by a three-dimensional Fourier cosine series expansion model. *Available at SSRN: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2410176*, 2014.

[33] M.J. Ruijter and C.W. Oosterlee. Two-dimensional Fourier cosine series expansion method for pricing financial options. *SIAM Journal on Scientific Computing*, Vol. 34(5):642–671, 2012.

[34] U. Schaede. Forwards and Futures in Tokugawa-period Japan: A New Perspective on the Dōjima Rice Market. *Jounal of Banking and Finance 13*, pages 487–513, 1989.

[35] W. Schoutens. *Lévy Processes in Finance: Pricing Financial Derivatives.* John Wiley & Sons, 2003.

[36] S.E. Shreve. *Stochastic Calculus for Finance II: Continuous-Time Models.* Springer Finance, 2008.

[37] I. Stewart. *In Pursuit of the Unknown.* Basic Books, 2012.

[38] V. Surkov. Parallel Option Pricing with Fourier Space Time-stepping Method on Graphics Processing Units. *University of Toronto*, 2007.

[39] V. Surkov. *Option Pricing using Fourier Space Time-stepping Framework.* University of Toronto, 2009.

[40] P. Tankov and E. Voltchkova. Jump-diffusion models: a practitioner's guide. *Banques et Marches*, 2009.

[41] C. Vuik and C.W.J. Lemmens. *Programming on the GPU with CUDA*. Delft University of Technology, 2013.

[42] P. Wilmott. *Derivatives: The theory and practice of financial engineering*. John Wiley & Sons, 2000.

[43] P. Wilmott, S. Howison, and J. Dewynne. *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge University Press, 1995.

[44] S. Wörner. Fast Fourier Transform. *Swiss Federal Institute of Technology Zurich*, 2008.

[45] `http://www.mathworks.com/matlabcentral/fileexchange/24050-multidimensional-discrete-cosine-transform-dct/content/mirt_dctn.m`.

[46] `http://www.fftw.org`.