MSc in High Performance Computing

Risk analysis of foreign exchange and interest rates with Monte-Carlo simulation and AAD

Chanho Eom

Advisor Professor Niall O'Sullivan

School of Mathematics
Trinity College Dublin

Abstract

Risk analysis of foreign exchange and interest rates with Monte-Carlo simulation and AAD

Chanho Eom

School of Mathematics
Trinity College Dublin

A Monte-Carlo simulation serves as an established algorithmic framework, offering robust capabilities in the realms of financial and quantitative analysis—particularly in the pricing of complex derivatives and in navigating intricate market dynamics. This project delves into the conjunction of Monte-Carlo simulation and an advanced computational method known as Algorithmic Adjoint Differentiation (AAD), intending to enhance the performance of the Monte-Carlo simulations. Initially, we examine the foundational principles of both Monte-Carlo simulation and AAD, explaining these concepts through practical examples. Subsequently, we conduct a risk analysis focusing on foreign exchange and interest rates, deploying four distinct simulation methodologies. We check that parallel Monte-Carlo simulation emerges as the most efficient technique. Moreover, we check that Monte-Carlo simulations incorporating AAD require significantly greater computational time and memory resources compared to their classical counterparts.

Keywords : Risk Analysis, Option Sensitivities, Foreign Exchange Rate, Interest Rate, Black-Scholes Formula, Monte-Carlo Simulation, Algorithmic Adjoint Differentiation (AAD), High Performance Computing (HPC)
Student Number : 22304052

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Selecting an appropriate computational model—both in terms of time efficiency and memory management—has long stood as a pivotal area of research. Given the plethora of available methods and techniques, researchers continually experiment with various frameworks to identify the most fitting model for their particular field of inquiry. In the context of this research, we focus on two sophisticated computational techniques: Monte-Carlo simulation and Algorithmic Adjoint Differentiation.
Monte-Carlo simulation is a well-established framework, renowned for its robust capabilities in the domains of financial and quantitative analysis. Utilising this model enables us to tackle challenges related to the pricing of complex derivatives as well as to navigate the intricacies of volatile market dynamics.
Algorithmic Adjoint Differentiation (AAD) has garnered considerable attention, particularly in sectors such as computational finance where sensitivity analysis is of paramount importance. Distinct from traditional methods like Finite Difference or Bump-and-Revalue, AAD offers superior speed and accuracy by allowing for the computation of both function value and its sensitivities in a single pass of a Monte-Carlo simulation.
In this research, we engage in a comprehensive examination of Monte-Carlo simulation and AAD to optimise the performance of Monte-Carlo simulations. Initially, we provide the foundational theories underpinning both Monte-Carlo simulation and AAD, supplementing these discussions with practical examples. Thereafter, we undertake a risk analysis concerning foreign exchange and interest rates, employing four distinct simulation techniques. Our findings reveal that parallel Monte-Carlo simulation without AAD stands as the most efficient method for this analysis. Addition-

ally, it becomes evident that Monte-Carlo simulations incorporating AAD demand substantially more computational time and memory resources compared to their traditional counterparts.

The paper is organised as following:

- In Chapter 2, we will provide an overview of a Monte-Carlo simulation method. Essential financial definitions and formulas relevant to this research will be introduced and explained.
- In Chapter 3, we will offer an in-depth exploration of the Monte-Carlo simulation. We will break down its algorithmic structure, providing a comprehensive understanding of its mechanics and applications.
- In Chapter 4, we will focus on the Algorithmic Adjoint Differentiation method. We will delve into its intricacies, providing readers with a clear understanding of how it functions, its advantages, and its role in optimising certain computational processes in finance.
- In Chapter 5, we will conduct a risk analysis of foreign exchange and interest rates using Monte-Carlo simulation and Algorithmic Adjoint Differentiation.
- In Chapter 6, we will present the findings from the case studies carried out in Chapter 5.
- In Chapter 7, we shall engage in a comprehensive discussion of this project, culminating in a conclusion.

Platform

`Apple` Macbook Pro M1

- CPU: 8 cores Apple Silicon
  - 3.2 GHz quad-core Apple Firestorm
  - 2.06 GHz quad-core Apple Icestorm
- RAM: 16 GB
- Programming Language: C++ 17

GitHub Repository

https://github.com/STEPHAN-EOM/RISK_ANALYSIS

Table 1.1: Table of Notations in the Research

Probability:

Ω	Sample space
X	Random variable
$P(X)$	Probability of the random variable X
$f_X(x)$	Probability density function of the random variable X
$E[X]$	Expectation of the random variable X
μ	Expectation of the random variable
$\hat{\mu}$	Estimated value of the expectation of random variable, μ

Financial Derivatives

S	Current price of the underlying asset
K	Strike price of the financial derivatives
T	Maturity of the financial derivatives
r	Risk-neutral interest rate
σ	Volatility of the underlying asset
S_T	Price of the underlying asset at maturity T
C_T	Payoff of a call option at maturity T
P_T	Payoff of a put option at maturity T

Option Sensitivities

Δ	Rate of change of the option price with respect to S
θ	Rate of change of the value of the portfolio with respect to T
ν	Rate of change of the value of the portfolio with respect to σ
ρ	Rate of change of the value of the portfolio with respect to r

Chapter 2

Preliminaries

2.1 Financial Terminology

Definition 1 (Option Contract)**.** An **option contract** gives the owner the right, but not the obligation, to sell or buy an underlying asset at a fixed time in the future for a specified price. There are two kinds of option contracts. A **call option** gives the owner the right to purchase the underlying asset from the seller of the option. By contrast, a **put option** gives the owner the right to sell the underlying asset to the seller of the option. The specified price in the option contract is called the strike price(K).

Remark 1 (Payoff of a Call Option)**.** At maturity time T, if the price of the underlying asset (S_T) is higher than the strike price (K) (*i.e.* $S_T > K$), then the payoff the call option is $S_T - K$. Therefore, the intrinsic value of the call option at maturity T is $C_T = max\{0, S_T - K\}$.

Remark 2 (Payoff of a Put Option)**.** At maturity time T, if the strike price (K) is higher than the price of the underlying asset (S_T) (*i.e.* $K > S_T$), then the payoff the put option is $K - S_T$. Therefore, the intrinsic value of the put option at maturity T is $P_T = max\{0, K - S_T\}$.

Example 1. Suppose the strike price (K) is €40. At maturity time T, the payoff of each option is $C_T = max\{0, S_T - 40\}$ and $P_T = max\{0, 40 - S_T\}$, respectively.

Definition 2 (Interest Rate)**.** An **interest rate** is a percentage that a borrower promise to pay the lender for the use of assets. It represents the profit earned from lending assets or the cost of borrowing assets.

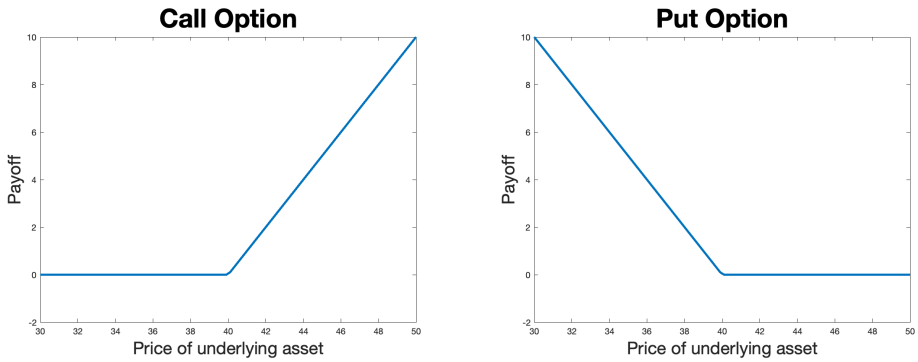


Figure 2.1: Profit from Buying European Options

Example 2. Suppose there is an asset priced at S_0. When an interest rate(r) is compounded annually, the value of the asset after 1 year will be $S_1 = S_0(1+r)$. When an interest rate(r) is compounded semi-annually, the value of the asset after 1 year will be $S_1 = S_0(1+\frac{r}{2})^2$. If it is compounded continuously, we will get $\lim_{n\to\infty} S_0(1+\frac{r}{n})^n = S_0 e^r$.

Remark 3 (Risk-Neutral Interest Rate)**.** The risk-neutral interest rate is the theoretical rate of return of an investment without the risk of financial loss. It represents the interest that an investor would expect from a risk-neutral investment over a specified period.

Definition 3 (Foreign Exchange Rate)**.** The **foreign exchange (FX) rate** provides a valuation between two different currencies, and it is generally structured as a pair. This pair representation captures the relative value of one currency to another.

Example 3. Suppose a foreign exchange rate between U.S. dollars and Korean won (USDKRW) is 1300. This means that the exchange rate from U.S. dollars (USD) to Korean won (KRW) is $1:1300$. In other words, 1 U.S. dollar is equivalent to 1300 Korean won.

Definition 4 (Black-Scholes Model)**.** The classic **Black-Scholes formulas** for a European call option's price:

$$C_T(S,K,\sigma,r,T) = SN(d_1) - Ke^{-rT}N(d_2)$$

$$d_1 = \frac{1}{\sigma\sqrt{T}}[ln(\frac{S}{K}) + T(r+\frac{\sigma^2}{2})]$$

$$d_2 = \frac{1}{\sigma\sqrt{T}}[ln(\frac{S}{K}) + T(r-\frac{\sigma^2}{2})] = d_1 - \sigma\sqrt{T}$$

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

Moreover, a European call option's price is similarly:

$$P_T(S, K, \sigma, r, T) = Ke^{-rT}N(d_2) - SN(d_1)$$

Definition 5 (Option Sensitivities)**.** The **option sensitivities**, or the **Greek letters** are used to measure a different dimension to the risk in an option position. The analysts' goal is to manage the sensitivities in such as a way that all risks are maintained within acceptable bounds[4]. Specifically, for a European call option, the option sensitivities are defined as follows:

$$\Delta = \frac{\partial C}{\partial S} = N(d_1)$$

$$\Gamma = \frac{\partial^2 C}{\partial S^2} = \frac{N'(d_1)}{S\sigma\sqrt{T}}$$

$$\theta = \frac{\partial C}{\partial T} = -\frac{SN'(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}N(d_2)$$

$$\nu = \frac{\partial C}{\partial \sigma} = S\sqrt{T}N'(d_1)$$

$$\rho = \frac{\partial C}{\partial r} = KTe^{-rT}N(-d_2)$$

These equations provide a mathematical means of evaluating different risk factors in an options portfolio, allowing for more precise risk management.

Remark 4 (Bump and Re-value)**.** A technique used to estimate the sensitivity of an option or portfolio to changes in underlying parameters. This methodology involves **bumping** an input variable (such as the underlying asset price, volatility, interest rate, etc.), and then **revaluing** the derivative or portfolio based on this modified input. The change in the value of the portfolio or derivative relative to the change in the input parameter provides an estimate of the sensitivities.

Example 4. Given the function for a European call option's price, denoted as $C_T(S, K, \sigma, r, T)$, the approximate value of the first-order derivative can be obtained through the bump-and-revalue method. Specifically, the Delta(Δ), representing the sensitivity of the option's price to changes in the underlying asset's price, can be approximated as:

$$\Delta = \frac{\partial C_T}{\partial S} \approx \frac{C_T(S + \epsilon, \ldots, T) - C_T(S, \ldots, T)}{\epsilon},$$

where ϵ is a small number.

2.2 Monte-Carlo Simulation

Definition 6 (Random Variable)**.** Consider a random experiment with a sample space Ω. A function X, which assigns to each element $\omega \in \Omega$ one and only one real number $X(\omega) = x$, is called a **random variable**. The **range** of X is the set of real numbers $\{x : x = X(\omega),\ space\ \omega \in \Omega\}$

Definition 7 (Monte-Carlo Simulation)**.** Suppose we are interested in computing

$$\mu = E[h(X)] = \int_{-\infty}^{\infty} h(t) f_X(t) dt,$$

where $h(x)$ is a specified function and $f_X(x)$ is a probability density function of the random vector X. A Monte-Carlo estimate of μ, denoted as $\hat{\mu}$, is given by

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} h(X_i),$$

where $\{X_1, X_2, \ldots, X_N\}$ is a set of independent and identically distributed random samples generated by the distribution $f_X(x)$.

Remark 5. The accuracy of a Monte Carlo simulation can be defined in terms of either absolute or relative error. Specifically, the errors are given by

$$\text{Absolute: } \epsilon = |\hat{\mu} - \mu| \quad \text{or} \quad \text{Relative: } \epsilon = |\tfrac{\hat{\mu_N} - \mu}{\mu}|,$$

where $\hat{\mu}$ is the estimated value from the simulation, and μ is an actual value.

If $E[h(X)] < \infty$, then $\epsilon \to 0$ as $N \to \infty$ meaning that the error tends to zero as the number of simulations grows large. Furthermore, the Central Limit Theorem can be utilised to estimate how large N must be to achieve a certain absolute error within a given confidence level.

Remark 6 (Law of Large Numbers[6])**.** Let $X_1, X_2, \ldots$ be a sequence of independent, identically distributed random variables, and let $E[X_i] = \mu$. Then, with probability 1,

$$\frac{X_1 + X_2 + \cdots + X_N}{N} \to \mu \quad as\ N \to \infty$$

Remark 7 (Central Limit Theorem[6])**.** Let $X_1, X_2, \ldots$ be a sequence of independent, identically distributed random variables with mean μ and variance σ^2, respectively. Then the distribution of $\frac{X_1+X_2+\cdots+X_N-N\mu}{\sigma\sqrt{N}}$ tends to the standard normal distribution as $N \to \infty$. That is,

$$P(\frac{X_1 + X_2 + \cdots + X_N - N\mu}{\sigma\sqrt{N}} \leq k) \to \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{k} e^{-x^2/2} dx$$

as $N \to \infty$.

Chapter 3

Monte-Carlo Simulation

In this chapter, we introduce the Monte-Carlo simulation method and delve into the precise understanding of the algorithm. As a powerful statistical technique that allows for the approximation of complex mathematical problems through repeated random sampling, the Monte-Carlo algorithm consists of a number of independent simulations of a random experiment, each following a known probability distribution. This algorithm is particularly useful when it is challenging to calculate an integral or when an analytical computation is impossible. Moreover, the algorithm relies on the Law of Large Numbers and the Central Limit Theorem. Although the runtime may increase, we can obtain a reliable approximation as the number of simulations grows large.

3.1 Algorithm Sketch

The principal sequential algorithm for the simulation consists of four to five stages:

1) Initialisation stage where defines the number of simulations, N, and initialises a sum variable as zero (lines 1 and 2).

2) Random sampling stage where generates a random sample, X_i, for each iteration from 1 to N, driven from a known probability distribution (line 3).

3) Function evaluation state where the randomly driven sample, X_i is plugged into the function of interest, $h(x)$, and the result is added to the sum (lines 4 and 5).

4) Estimation stage where the sum is divided by N to calculate the estimate of the expected value, $\hat{\mu}$ (line 6).

5) (Optional) Error analysis stage where calculates absolute or relative error, leveraging the Law of Large Numbers and the Central Limit Theorem.

Algorithm 1: Elementary Algorithm for Monte-Carlo Simulation

Data: N, $f_X(x)$, $h(x)$
Result: Estimated expected value, $\hat{\mu}$
1 Initialisation: sum = 0;
2 **for** $i = 1 \rightarrow N$ **do**
3 Generate a random sample X_i from $f_X(x)$;
4 Evaluate $h(X_i)$;
5 sum = sum + $h(X_i)$;
6 $\hat{\mu} = \frac{\text{sum}}{N}$;
7 **return** $\hat{\mu}$;

Figure 3.1: Elementary Algorithm for Monte-Carlo Simulation

3.2 Application in Finance

In financial mathematics, derivatives (such as options or futures) derive their value from underlying assets, like stock prices or interest rates. The actual future payoff of these derivatives can depend on the path and final value of these underlying assets. Given the uncertain future movements of these assets, derivative pricing often requires the evaluation of expected values under certain probabilistic measures. This is where the concept of `filtration` becomes relevant, representing the collection of all available information up to a given point in time. Monte-Carlo simulation serves as an efficient tool in this realm for several reasons:

- Complex payoffs: Some derivatives have payoffs that are complex functions of the underlying asset's path. Analytical solutions may not exist or might be infeasible for these products. Monte-Carlo methods can accommodate virtually any payoff structure by simply simulating the asset's path and then evaluating the payoff directly.

- Market dynamics: Modern finance often models assets using stochastic processes, frequently based on normal or log-normal distributions due to their mathematical tractability and empirical fit to observed market phenomena. Monte-Carlo methods are adept at simulating these processes.
- Adjustments and Extensions: It's relatively straightforward to extend Monte-Carlo methods to account for features like early exercise, dividends, stochastic volatility, or interest rates.

However, it's worth noting that while Monte-Carlo simulations offer flexibility, they also come with challenges like ensuring convergence and reducing estimation errors. Techniques like variance reduction can be employed to enhance their efficiency.
In essence, Monte-Carlo simulations provide a versatile and powerful tool in the arsenal of financial and quantitative analysts, especially when pricing complex derivatives or dealing with intricate market dynamics.

3.3 Simple Example

To help better understand the Monte-Carlo simulation, we provide a simple example. The example is to demonstrate the simulation for estimating the value of π. The idea for the simulation is the following:

1) Initialisation: Define the number of simulations, N.

2) Random sampling: Randomly generate points (x, y) inside a unit square.

3) Function evaluation: Determine the number of points that fall within a quarter circle inscribed within the square (i.e., $x^2 + y^2 \leq 1$).

4) Estimation: The ratio of the points inside the quarter circle to the total number of points generated, multiplied by 4, will approximate the value of π.

$$\pi \approx \frac{4 * \text{number of points inside quarter circle}}{\text{total number of points generated}, N} = \hat{\pi}$$

The Monte-Carlo simulation leverages the Law of Large Numbers, which suggests that the result from a large number of trials should approximate

the expected value. Therefore as we increase N, the number of randomly generated points, our estimate $\hat{\pi}$ converges to the true value π.
The algorithm structure and Python code for the Monte-Carlo simulation estimating the value of π are the following.

Algorithm 2: Monte-Carlo Algorithm for Estimating π

Data: Total number of random points N
Result: Estimated value of π, $\hat{\pi}$
Initialisation: count $= 0$;
for $i = 1 \rightarrow N$ **do**
    Generate random x-coordinate, x in $[0, 1]$;
    Generate random y-coordinate, y in $[0, 1]$;
    if $x_i^2 + y_i^2 \leq 1$ **then**
        count $=$ count $+ 1$;
$\hat{\pi} = 4 \times \frac{\text{count}}{N}$;
return $\hat{\pi}$;

Figure 3.2: Monte-Carlo Algorithm for Estimating π

To execute the subsequent Python code we provide, it is essential to have the **Numpy**[1] library installed.

```
# Number of points to simulate
N = 10000

# Generate random x, y coordinates in the range [0, 1]
x = np.random.uniform(0, 1, N)
y = np.random.uniform(0, 1, N)

# Check if the points are inside the unit circle
inside_circle = x**2 + y**2 <= 1
outside_circle = np.invert(inside_circle)

# Estimate pi
estimated_pi = 4 * np.sum(inside_circle) / N
```

Listing 3.1: Python Code for Simulation Estimating π

[1] https://numpy.org

Chapter 4

Algorithmic Adjoint Differentiation

In this chapter, we delve into Algorithmic Adjoint Differentiation (AAD), providing a comprehensive examination of this advanced differentiation technique. AAD has gained significant traction in fields such as computational finance, where **sensitivity analysis** plays a pivotal role. The technique's primary appeal lies in its ability to efficiently compute derivatives, especially when presented with numerous input variables contrasted with only a handful of output variables. In the financial sector, derivatives have traditionally been determined using FINITE DIFFERENCE METHODS or the BUMP AND RE-VALUE approach. However, these methodologies often become burdensome due to the need to re-run valuations for each distinct market input. Unlike traditional methods, AAD can provide faster and more accurate performances because we can compute the function value and its sensitivities in a single Monte-Carlo simulation pass.

4.1 Introduction

The core idea behind adjoint differentiation is to decompose complex functions into simple elementary operations and then apply the chain rule of calculus to compute adjoints by propagating sensitivities backwards through the computational procedure. Then, we identify the computational procedure for differentials that reuses common components. We call ADJOINT OF y and denote $\bar{y}$ the derivative of the final result to y. We need to calculate the adjoint of all its input parameters for computing all option sensitivities. The following example helps understand all the opera-

tions involved and their dependencies.
Consider the function: $f(x, y) = (x + y) * (x - y)$. If we aim to compute the derivatives with respect to x and y, then the straightforward approach would be to apply the chain rule. However, using AAD, the process becomes more structured and efficient. The calculation graph is the following:

1. $u_1 = x + y$
2. $u_2 = x - y$
3. $f = u_1 * u_2$

Then, we do the back-propagation. The procedure is the following:

1. Set $\bar{f} = 1$
2. $\bar{u_1} = \bar{f} * u_2 = 1 * (x - y)$
3. $\bar{u_2} = \bar{f} * u_1 = 1 * (x + y)$

Then, using the above adjoints, we can determine the derivatives with respect to x and y.

- $\frac{df}{dx} = \bar{u_1} + \bar{u_2}$
- $\frac{df}{dx} = \bar{u_1} - \bar{u_2}$

Thus, AAD provides an efficient mechanism to reuse calculations during the back-propagation, ensuring that computation is both fast and accurate.

4.2 Forward Calculation

When executing the Monte-Carlo simulation utilising five input variables: $S, r, \sigma, K,$ and T, it becomes evident that inter-dependencies exist amongst these parameters. As such, constructing a computational graph for forward calculations provides a lucid visual representation of the interrelationships between input variables and the computational progression. The following figure is a computational graph for forward calculations of a Monte-Carlo simulation.

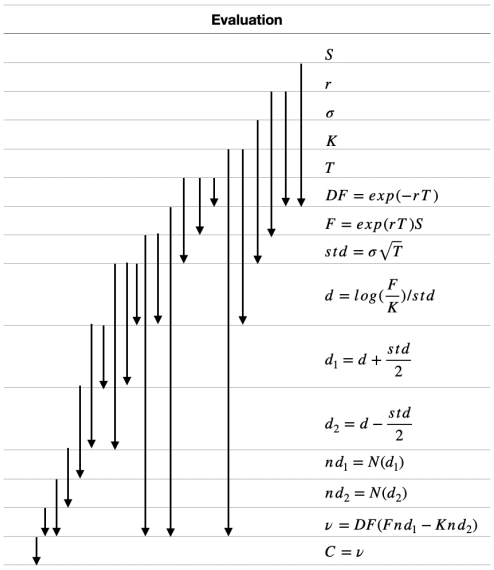


Figure 4.1: Computational Graph for Forward Calculation

4.3 Backward Calculation

Subsequently, when calculating a specific option sensitivity, such as Delta (Δ), the following procedural steps may be explained:

$$\frac{\partial C}{\partial S} = \frac{\partial C}{\partial F}\frac{\partial F}{\partial S} \rightarrow \bar{S} = \frac{\partial F}{\partial S}\bar{F} = \frac{\partial F}{\partial S}\bar{F},$$

where $\bar{S}$ is called the adjoint of S which means the final result of derivative with respect to S. Note that the adjoint relationship is reverse ordered. Therefore, F is a function of S. However, $\bar{S}$ is a function of $\bar{F}$.
Since F is involved into two distinct branches, respectively to G, and ν, we get the following results:

$$d = \frac{log(\frac{F}{K})}{std},\ \nu = DF[Fnd_1 - Knd_2] \rightarrow \bar{F} = \frac{\bar{d}}{F \cdot std} + DF \cdot nd_1 \cdot \bar{\nu}$$
$$= \frac{\bar{d}}{F \cdot std} + DF \cdot nd_1 \cdot 1$$

Subsequently, $\bar{F}$ is a function of $\bar{d}$, where d is associated with two distinct branches, d_1 and d_2. The mathematical relationships between these variables can be formally expressed as:

$$d_1 = d + \frac{std}{2},\ d_2 - \frac{std}{2} \rightarrow \bar{d} = \bar{d_1} + \bar{d_2}\ \ldots$$

Consequently, we get a following results:

$$\begin{aligned}\bar{S} &= \frac{F}{S}\bar{F} = \frac{F}{S}(\frac{\bar{d}}{F \cdot std} + DF \cdot nd_1) \\ &= \frac{F}{S}(\frac{\bar{d_1} + \bar{d_2}}{F \cdot std} + DF \cdot nd_1) = \frac{F}{S}(\frac{n(d_1)n\bar{d}_1 + n(d_2)n\bar{d}_2}{F \cdot std} + DF \cdot nd_1) \\ &= \frac{F}{S}(\frac{n(d_1)DF \cdot F - n(d_2)DF \cdot K}{F \cdot std} + DF \cdot nd_1) = DF\frac{F}{S}[nd_1 + \frac{n(d_1) - \frac{K}{F}n(d_2)}{std}] \\ &= \frac{DF \cdot F}{S}nd_1\end{aligned}$$

The following figure helps understanding the back-propagation process applied to the Monte-Carlo simulation. It comprehensively depicts all the direct adjoint operations, executed in a bottom-to-top sequence.

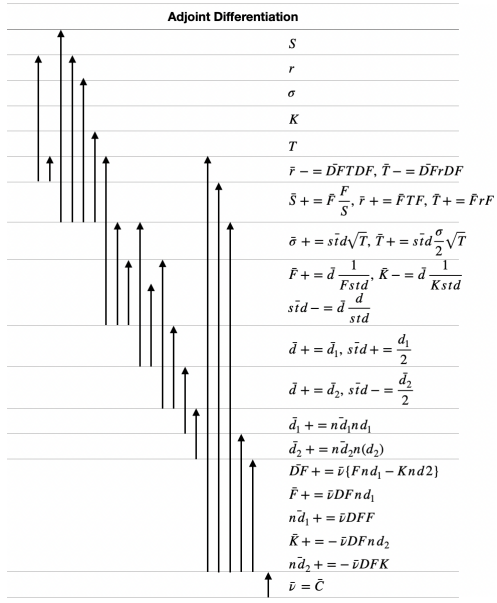


Figure 4.2: Computational Graph for Backward Calculation

Back Propagation[1]

The following rules are what we understand from the previous example:

1) The calculation of adjoints follows a reverse order in comparison to the forward calculation of the variables.

2) For a given operation resulting in y, its adjoint $\bar{y}$ is computed as the sum of the adjoints of subsequent operations that utilise y. These adjoints are weighted by the partial derivatives of those subsequent operations with respect to y.

3) The adjoint corresponding to the final output (or objective) is conventionally set to 1.

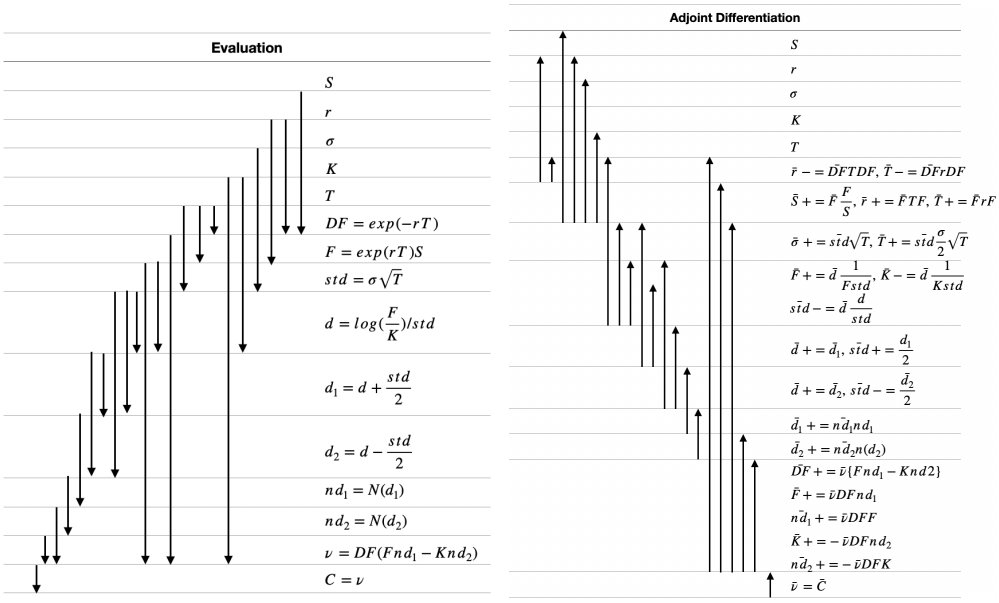


Figure 4.3: Comparison of Forward and Backward Calculation graphs

Chapter 5

Case Studies

In this chapter, we utilise the Monte-Carlo Simulation and an Algorithmic Adjoint Differentiation (AAD) to analyse risk of foreign exchange and interest rates. Foreign exchange (FX) rates let us know how one currency compares to another, and interest rates tell us the cost of borrowing money. Because both these rates can change often and have big effects on businesses and economies, it's crucial to understand the risks tied to them. We explore four different versions of the simulation:

1) Serial Monte-Carlo simulation

2) Serial simulation with AAD

3) Parallel Monte-Carlo simulation

4) Parallel simulation with AAD

By analysing real-life examples, we can understand how each version works and find out which ones are the most helpful. By the end of the chapter, we are able to have a clearer idea of which model is the most appropriate when we manage the risks that come with changing foreign exchange and interest rates.

5.1 Input

Te begin with, we provide the information of input parameters. For our simulation, we choose five input parameters:

- S: Current price of the underlying asset

- K: Strike price of the financial derivatives
- T: Maturity of the financial derivatives
- r: Risk-neutral interest rate ($r_{\text{foreign}} - r_{\text{domestic}}$)
- σ: Volatility of the underlying asset

Risk-Neutral Interest Rate

The risk-neutral interest rate, one of parameter of our simulation, derived from the difference in interest rates of two different countries. For the purpose of our simulation, we have chosen two key central banks[1] to represent these rates. The domestic interest rate, $r_{domestic}$, is based on the rate issued by the Bank of Korea. On the other hand, the foreign interest rate, $r_{foreign}$, is determined by the rate provided by the Federal Reserve System, commonly referred to as the FED. By calculating the difference between these two rates, we can accurately determine the risk-neutral interest rate, ensuring our simulations are grounded in real-world financial dynamics.

5.2 Black-Scholes Formula

The Black-Scholes formula represents a cornerstone in financial mathematics. It's an **analytical method** used to determine the price of European options, derived from assumptions about market behaviors, stock returns, and the specific dynamics of option markets. There are mainly two reasons why this formula important:

- Precision: The formula provides accurate estimates, making it reliable for real-world applications.
- Simplicity: Its straightforward nature means it can be quickly implemented and understood.

Therefore, we can use the results of the Black-Scholes formula to check whether our simulations are working properly or not.

[1] 8-June-2023

Example Code

```
/* Function for the cumulative Normal distribution */
double BSModel::normcdf(double x) const {
    return 0.5 * (1.0 + std::erf(x / std::sqrt(2.0)));
}

/* Function for the option price */
double BSModel::calculateOptionPrice(double d1, double
   d2) const {
    double OptionPrice = spotPrice * normcdf(d1) -
       strikePrice * std::exp(-riskFreeRate *
       timeToMaturity) * normcdf(d2);

    return OptionPrice;
}

/* Function for the European call option price  */
double BSModel::calculateCallOptionPrice() const {
    double d1 = (std::log(spotPrice / strikePrice) + (
       riskFreeRate + 0.5 * volatility * volatility) *
       timeToMaturity) / (volatility * std::sqrt(
       timeToMaturity));
    double d2 = d1 - volatility * std::sqrt(
       timeToMaturity);

    return calculateOptionPrice(d1, d2);
}
```

Listing 5.1: C++ Example Code of Black-Scholes Formula

Outcome

The following results are gotten by the Black-Scholes formula.

```
Call_Option: 85.9907
Delta: 0.565912
Rho: 646.866
Vega: 509.563
Theta: -49.5374
```

5.3 Monte-Carlo Simulation

We provide an algorithm structure and C++ code for the simulation.

5.3.1 Algorithm Structure

The principal sequential algorithm for this simulation consists of the following stages:

1) Initialisation stage: Define the simulation parameters, including the number of simulations, N and the number of steps, M.

2) FX rate evolution stage: For each iteration from 1 to N, the FX rate is evolved over a certain number of steps, M, using random numbers sampled from a known distribution.

3) Payoff calculation stage: The option is calculated for each simulated path as the maximum difference between the simulated FX rate and the strike price K, but never less than zero.
i.e. $max\{\text{payoff} - K, 0\}$

4) Averaging and discounting stage: The simulated payoffs are averaged and then discounted by r_{domestic} to their present value.

Algorithm 3: Algorithm for Monte-Carlo FX Option Pricing

```
   Data: Input parameters, N, M
   Result: Estimated FX rate, Estimated Option Price
 1 Initialisation: sum_op = 0, elementary parameters;
 2 for i = 1 → N do
 3 |   Initialise fx_rate = fx_initial;
 4 |   for j = 1 → M do
 5 |   |   Generate a random sample rand;
 6 |   |   Evolve fx_rate using rand;
 7 |   payoff = max{fx_rate - K, 0};
 8 |   sum_op = sum_op + payoff;
 9 Average and discount sum_op;
10 return Estimated FX rate, Estimated Option Price;
```

Figure 5.1: Algorithm for Monte-Carlo FX Option Pricing

Pseudo Random Number Generator

The Monte-Carlo simulation requires a multitude of random numbers to generate a path. To complete our simulation algorithm, we integrate a programme that produces (pseudo) random numbers, called a pseudo random number generator (PRNG). Additionally, there are methods available to reduce the variance of the Monte-Carlo simulation, allowing for increased precision without needing more samples. These techniques include antithetic variance, low discrepancy sequences, and importance sampling. However, our primary objective is to contrast the performance of the classic Monte-Carlo simulation with that of the simulation incorporating AAD. As a result, we choose the random number generator provided by the C++ STL library to generate random numbers from the Normal distribution.

```
std::default_random_engine generator;
std::normal_distribution<double> distribution;
```

5.3.2 Code for Simulation

The following C++ code is for calculating prices European call option. We use template for the generic parameter types.

```
template <class T>
T MC_Simulation{

    /* Initialisation stage */
    T dt = maturity / (365.0 * num_step);
    T ir_rate = r_foreign - r_dom;
    T first = (ir_rate - (0.5 * fx_vol * fx_vol)) * dt;
    T second = fx_vol * sqrt(dt);

    T sum_op = 0.0;
    T payoff = 0.0;

    /* FX rate evoluation stage */
    for(int i = 0; i < num_sim; ++i){
        T fx_rate = fx_initial;

        for (int j = 0; j < num_step; ++j){
          double rand = distribution(generator);
```

```
19            fx_rate *= exp(first * 365) * exp(second *
                 sqrt(365) * rand);
20          }
21
22          /* Payoff calculation stage */
23          payoff = std::max(fx_rate - K, static_cast<T
                >(0.0));
24          sum_op += payoff;
25      }
26
27      /* Averaging and discounting stage */
28      T average_op = sum_op / num_sim;
29      T discount = exp(-r_dom);
30      T result_op = average_op * discount;
31
32      return result;
33  }
```

Listing 5.2: C++ Code for Serial Monte-Carlo Simulation Estimating C_T

5.4 Algorithmic Adjoint Differentiation

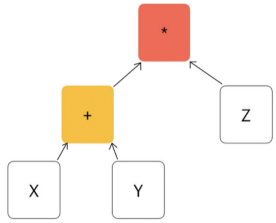
For the implementation and deployment of the Algorithmic Adjoint Differentiation method, we depend on two essential header files: `Node.h` and `Number.h`. More specific, we design the computation code using a custom number type. This type comes with overloaded arithmetic operators and mathematical functions. As a result, not only are the operations evaluated, but they are also systematically logged onto a `Tape`. Once the computation concludes and all operations are recorded in a sequence of nodes, adjoints are back-propagated over the tape, enabling the production of all its differentials.

5.4.1 Node.h

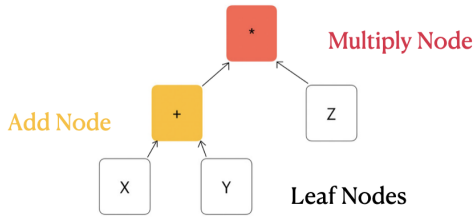
`Node.h` header file is the essence of our automatic differentiation (AD) framework. It provides an abstract class named `Node` that serves as a base class for various operations and leaf nodes. Every mathematical expression or variable is transformed into nodes within a computational graph. This abstract representation is crucial for the back-propagation of derivatives, the heart of AD. The key features are the following:

- Polymorphism and Flexibility
  - Base node: At the center of `Node.h` lies the abstract Node class. This class cannot be instantiated directly but provides a blueprint for all specific node types.
  - Derived node: For each mathematical operation or function, a derived node class is established. This design makes the addition of new operations straightforward, ensuring extensibility.
- Handling Adjoint Computations: The adjoint of a node, essentially the gradient, is of paramount importance in reverse-mode AD. The Node class provides fundamental methods to access and modify these adjoints. By centralising this functionality, the architecture ensures that derivative computations remain consistent across all nodes.

For instance, consider the function $f(x, y, z) = (x + y) * z$.



Subsequently, each input parameter as well as each mathematical operation is transmuted into discrete nodes, categorised as leaf nodes, addition nodes, or multiply nodes.




```
class Node {
protected:
    std::vector<Node*> arguments;

```

```
    double result;
    double adjoint = 0.0;

public:
    Node(){}
    virtual ~Node(){};

    double Get_result(){
        return result;
    }

    double & Get_adjoint(){
        return adjoint;
    }

    void Set_result(double value) {
        result = value;
    }

    void Reset_adjoint(){
        for (auto i : arguments) i -> Reset_adjoint();
        adjoint = 0.0;
    }

    virtual void Propagate_adj() = 0;
};
```

Listing 5.3: Snapshot Code of Node.h

More precisely, every mathematical results or variable is transformed into derived nodes. Then, we provide an example of derived node, called as AddNode.

```
class AddNode : public Node{
public:
    AddNode(Node* lhs, Node* rhs){
        arguments.resize(2);
        arguments[0] = lhs;
        arguments[1] = rhs;

        result = lhs->Get_result() + rhs->Get_result();
    };
```

```
10
11      ~AddNode(){};
12
13      void Propagate_adj() override{
14          arguments[0] -> Get_adjoint() += adjoint;
15          arguments[1] -> Get_adjoint() += adjoint;
16      }
17  };
18  /* Similarly, other derived nodes are followed */
```

5.4.2 Number.h

While `Node.h` deals with the low-level representation and operations of the computational graph, `Number.h` is the user-facing interface, providing a seamless way to define and manipulate variables that support automatic differentiation. The key features are the following:

- Memory management: The `Number` class maintains a `tape`, a dynamic record of operations. We use a smart pointer, *unique_ptr*, to `Node` objects, ensuring proper memory management.
- Overloaded operators: Arithmetic operations (like addition, subtraction) and functions (like log, exp) are overloaded, allowing users to write mathematical expressions in a natural way. Internally, these operations create appropriate nodes in the computational graph.
- Adjoint propagation: The *Propagate_adj* method facilitates the backpropagation for computing gradients. It utilises the tape to move through the computation in reverse, propagating adjoints accordingly.
- Mark and Rewind: Two static methods, *Mark_tape* and *Rewind_Mark*, allow us to control the size of the tape. This is especially useful when we want to discard parts of the computational graph, optimising memory use.

```
1  class Number{
2      Node* mynode;
3
4  public:
5      static std::vector<std::unique_ptr<Node>> tape;
```

```
6
7       Number(double value) : mynode(new Leaf(value)) {
8           tape.push_back(std::unique_ptr<Node>(mynode));
9       };
10
11      Number(Node* node) : mynode(node) {};
12      ~Number(){};
13
14      Node* node(){
15           return mynode;
16      }
17
18      void Set_value(double value){
19          dynamic_cast<Leaf*>(mynode) -> Set_value(value);
20      }
21
22      double Get_value() {
23          return dynamic_cast<Leaf*>(mynode) -> Get_value
                ();
24      }
25
26      double& Get_adjoint(){
27          return mynode -> Get_adjoint();
28      }
29
30      void Propagate_adj(){
31          mynode -> Reset_adjoint();
32          mynode -> Get_adjoint() = 1.0;
33
34          auto it = tape.rbegin();
35          while (it-> get() != mynode){
36              ++it;
37          }
38
39          while (it != tape.rend()){
40              (*it) -> Propagate_adj();
41              ++it;
42          }
43      }
44
45      static void Mark_tape(){
```

```
46          tapeMark = tape.size();
47      }
48
49      static void Rewind_Mark(){
50          if(tapeMark.has_value()){
51              tape.resize(tapeMark.value());
52              tapeMark.reset();
53          }
54      }
55
56  private:
57      static std::optional<size_t> tapeMark;
58  };
```

Listing 5.4: Snapshot Code of Number.h

For each mathematical operation, we have constructed overloaded operators tailored to our custom numerical types. Listing 5.6 illustrates the functionality of each overloaded operator. A corresponding node is generated and appended to the computational Tape..

```
1  Number operator+(Number lhs, Number rhs){
2      Node* n = new AddNode(lhs.node(), rhs.node());
3      Number::tape.push_back(std::unique_ptr<Node>(n));
4
5      return n;
6  }
7  /* Similarly, other overloaded operators are followed */
```

Tape

The Tape is a sequence of nodes, so it is natural to store the nodes in a vector. Consequently, the lifespan of individual nodes is commensurate with that of the tape itself.

```
1  std::vector<std::unique_ptr<Node>> Number::tape;
```

As an illustrative example to facilitate understanding of the tape mechanism, we execute the following template function, f, and enumerate the constituent elements recorded on the tape.

Example for Tape

```
template <class T>
T f(T spot_p, T strike_p, T risk_neutral, T vol, T
   maturity){

auto d1 = (log(spot_p / strike_p) + (risk_neutral + 0.5
   *   vol * vol) * maturity) / (vol * sqrt(maturity));
auto d2 = d1 -vol * sqrt(maturity);

auto y = spot_p * N(d1) -strike_p * exp((-risk_neutral)
   * maturity) * N(d2);

return y;
}
```

Listing 5.5: C++ Example Code for Black-Scholes Formula with AAD

Upon executing the function with the initial parameters, a computational graph is generated. Each node in this graph is recorded on the tape.

```
tape(0) = 1295          tape(14) = 0.15
tape(1) = 1300          tape(15) = 0.165976
tape(2) = 0.0175        tape(16) = 1
tape(3) = 0.15          tape(17) = 0.15
tape(4) = 1             tape(18) = 0.0159762
tape(5) = 0.996154      tape(19) = 0.565912
tape(6) = -0.00385357   tape(20) = 732.856
tape(7) = 0.5           tape(21) = -0.0175
tape(8) = 0.075         tape(22) = -0.0175
tape(9) = 0.01125       tape(23) = 0.982652
tape(10) = 0.02875      tape(24) = 1277.45
tape(11) = 0.02875      tape(25) = 0.506373
tape(12) = 0.0248964    tape(26) = 646.866
tape(13) = 1            tape(27) = 85.9907
```

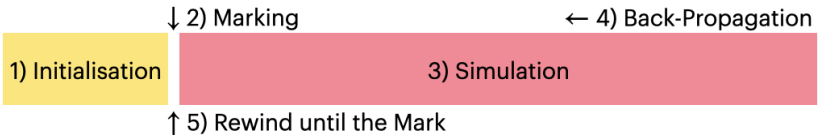
Tape Rewind

The primary rationale for employing tape in the context of AAD is to facilitate the process of tape rewinding. On our computational platform[2], the upper limit for simulations using AAD is approximately 135,000. To

[2]Apple, MacBook Pro M1

synchronise the number of simulations with those of the traditional simulation model, we employ tape rewinding as a mechanism for memory management. The procedural steps are as follows:

- 1) Initialisation of input parameters for the simulations
- 2) Marking a point on the tape
- 3) Executing the simulation
- 4) Implementing back-propagation
- 5) Rewinding the tape to the previously marked point
- 6) Re-executing the simulation



When the number of simulations is configured at one hundred thousand and the tape is rewound nine times, it becomes feasible to achieve an effective number of simulations equivalent to one million. This thus harmonises the number of simulations between the two models

Active / Inactive Number Type

For the purposes of memory management, especially to reduce unnecessary AAD overhead, the number types are categorised as either ACTIVE or INACTIVE. ACTIVE numbers are transformed into nodes and subsequently recorded on the tape. By contrast, numbers designated as INACTIVE are used in simulations but are not converted into nodes. Specifically, these INACTIVE numbers remain unrecorded on the tape.

- Active: Spot price, Strike price, Interest rate, Maturity, Volatility, and Computation result of active numbers
- Inactive: Random Number, Number of simulations, Number of steps, and other unrelated numbers (int, double, float type)

All input parameters are categorised as ACTIVE. Conversely, random numbers are designated as INACTIVE owing to their irrelevance in the back-propagation process. Consequently, random numbers, along with other parameters such as the number of simulations (N), the number of steps (M), and other unrelated INT, DOUBLE, and FLOAT-type numbers, fall under the INACTIVE category and are thus not recorded on the computational tape. Note that the computation results of ACTIVE and INACTIVE numbers is ACTIVE.

5.5 Parallel Simulation

In this section, we explain option sensitivities through the parallel Monte-Carlo simulations, employing two distinct methodologies. The first approach involves a classic parallel Monte-Carlo simulation, implemented using `Pthreads`. The other approach utilises a Monte-Carlo simulation with Algorithmic Adjoint Differentiation (AAD). The second approach is implemented using different multi-threading method, `Future`.

- Classic parallel Monte-Carlo simulation
- Parallel Monte-Carlo simulation with AAD

5.5.1 Concurrency

Parallel processing is a critical component in Monte Carlo simulations, especially when dealing with large datasets or a significant number of simulations. While `OpenMP` offers a way to achieve this parallelism, for this project, we have chosen the `Pthreads` and `Futures`frameworks.

Data Structure

Before executing a thread function, we built `ThreadData` by using struct. This data structure is going to be convenient to bundle multiple related pieces of parameters into a single object. This is helpful because the thread function *pthread_create* uses can only accept one argument of type *void**. By encapsulating all the parameters we need in a single struct and passing a pointer to that struct, we can work around this limitation and provide multiple pieces of data to each thread.

```
1 template <class T>
```

```
struct ThreadData {
    T fx_initial, fx_vol, r_dom, r_foreign, maturity, K;
    int num_step, num_sim_per_thread;
    T sum_op;
    std::default_random_engine generator;
    std::normal_distribution<double> distribution;
};
```

Listing 5.6: Data Structure for Parallel Simulations

Pthreads

The `Pthreads` is a library that provides C++ with the capability to create multithreaded applications. While `OpenMP` offers a higher-level, directive-based approach to parallelism, `Pthreads` provides more explicit control over thread creation, synchronisation, and management.

Thread Function

The thread function for the parallel Monte-Carlo simulation has close structure to classic serial simulation, with the exception of the estimation stage. Upon completion of individual simulations by each thread, the results are combined into a global sum, which is subsequently averaged and discounted.

```
template <class T>
void* MCSimulation<T>::threadFunction(void* arg) {
    ThreadData* data = static_cast<ThreadData*>(arg);

    T dt = data->maturity / (365.0 * data->num_step);
    T ir_rate = data->r_foreign - data->r_dom;
    T first = (ir_rate - (0.5 * data->fx_vol * data->
       fx_vol)) * dt;
    T second = data->fx_vol * sqrt(dt);

    for(int i = 0; i < data->num_sim_per_thread; ++i){
        T fx_rate = data->fx_initial;
        for (int j = 0; j < data->num_step; ++j){
            double rand = data->distribution(data->
               generator);
```

```
14              fx_rate *= exp(first * 365) * exp(second *
                    sqrt(365) * rand);
15          }
16
17          T payoff = std::max(fx_rate - data->K,
                static_cast<T>(0.0));
18          data->sum_op += payoff;
19      }
20      return nullptr;
21  }
```

Listing 5.7: Thread Function for Parallel Monte-Carlo Simulation

5.5.2 Monte-Carlo Simulation

The number of threads is explicitly defined and set to 4 (line 3). For the generation of random numbers, two alternatives are available: Global random number generator or Individual random number generator for each thread. To maintain consistency with the serial simulation, we opt for the `global random number generator` (lines 7, 9).

```
1   template <class T>
2   T MCSimulation<T>::MC_Simulation(T K, int num_step) {
3       const int NUM_THREADS = 4;
4       pthread_t threads[NUM_THREADS];
5       ThreadData threadData[NUM_THREADS];
6
7       std::default_random_engine master_generator;
8       for (int i = 0; i < NUM_THREADS; ++i) {
9           std::default_random_engine thread_generator(
                master_generator());
10          threadData[i] = {fx_initial, fx_vol, r_dom,
11          r_foreign, maturity, K, num_step, num_sim /
12          NUM_THREADS, 0.0, 0.0, thread_generator,
13          std::normal_distribution<double>(0.0, 1.0)
14          };
15          pthread_create(&threads[i], nullptr,
                threadFunction, &threadData[i]);
16      }
17
18      T total_sum_op = 0.0;
19      for (int i = 0; i < NUM_THREADS; ++i) {
```

```
        pthread_join(threads[i], nullptr);
        total_sum_op += threadData[i].sum_op;
    }

    T average_op = total_sum_op / num_sim;
    T discount = exp(-r_dom * maturity);
    T result_op = average_op * discount;

    return result_op;
}
```

Listing 5.8: Monte-Carlo Simulation for Parallel Monte-Carlo Simulation

5.5.3 Algorithmic Adjoint Differentiation

In order to implement a parallel simulation with Algorithmic Adjoint Differentiation (AAD), modifications are necessitated within the `Number.h` header file, specifically in the `Tape`, `Constructor`, and `Overloading Operator` sections. This is due to the employment of two distinct tape types: Thread-local tape allocated for each separate thread and Global tape. Each thread records its calculation graph onto its designated thread-local tape. Upon the completion of the individual simulations executed by each thread, the data accumulated in these local tapes is then transferred to the global tape and synchronised.

Tape

```
std::vector<std::unique_ptr<Node>> global;
thread_local std::vector<std::unique_ptr<Node>> tape;
std::mutex tape_mutex;
```

Constructor

```
Number(double value) : mynode(new Leaf(value)) {
    std::lock_guard<std::mutex> lock(tape_mutex);
    tape.push_back(std::unique_ptr<Node>(mynode));
    global.push_back(std::unique_ptr<Node>(mynode));
};
```

Overloading Operator

```
Number operator+(Number lhs, Number rhs){
    Node* n = new AddNode(lhs.node(), rhs.node());
    std::lock_guard<std::mutex>lock(Number::tape_mutex);
    Number::tape.push_back(std::unique_ptr<Node>(n));
    Number::global.push_back(std::unique_ptr<Node>(n));

    return n;
}
/*Similarly, other overloading operators are followed*/
```

Futures

We opt to employ `Futures` for parallel simulation with AAD instead of using `Pthread`. This decision is predicated on the superior stability that `Futures` offers in the synchronisation of local tapes to the global tape.

```
for(int i = 0; i < num_threads; ++i) {
    futures.push_back(std::async(std::launch::async,
       Simulation<T>, spot_p, strike_p, risk_neutral,
       vol, maturity, num_step, sim_thread));
    }

T total_sum = 0.0;
for(auto& fut : futures) {
    total_sum += fut.get();
}
```

Synchronisation

For the synchronisation of local tapes to the global tape, we added the following code to the bottom part of the function evaluation stage. Subsequently, the aggregated results are reflected during the estimation stage.

```
std::lock_guard<std::mutex> lock(Number::tape_mutex);

// Move data from the local tape to the global tape
for (auto& node : Number::tape) {
Number::global_tape.push_back(std::move(node));
}

```

```
// Clear the thread-local tape
Number::tape.clear();
```

5.6 Simulation Results

The input parameters we choose for the simulations are the following:

- S (Current price of the underlying asset): 1295.0
- K (Strike price of the financial derivatives): 1300.0
- T (Maturity of the financial derivatives): 1.0
- r (Risk-neutral interest rate): 0.0175
- r_{foreign} (Foreign interest rate): 0.0525
- r_{domestic} (Domestic interest rate): 0.035
- σ (Volatility of the underlying asset): 0.15

Subsequently, we proceed to execute a multi-step Monte-Carlo simulations. The number of steps and simulations is five and one million, respectively.

- M (Number of steps): 5
- N (Number of simulations): 1,000,000

For the simulation incorporating AAD, the number of simulations is set at one hundred thousand, with the tape rewinding count established at 9. This configuration allows us to equalise the number of simulations between the two different models.

Bump and Revalue

```
Call_Option: 84.5121
Delta: 0.556997
Rho: 723.363
Vega: 500.621
Theta: 551.031
```

Algorithmic Adjoint Differentiation

```
Call_Option: 84.1883
Delta: 0.555803
Rho: 719.765
Vega: 498.557
Theta: 4.49853
```

Model	Call Option	Delta	Rho	Vega	Theta
Black-Scholes	85.9907	0.565912	646.866	509.563	-49.5374
Classic Model	84.6121	0.556997	723.363	500.621	551.031
Model with AAD	84.1883	0.555803	719.765	498.557	4.49853

Table 5.1: Summary Table for the Case Studies

Chapter 6

Experimental Evaluation

6.1 Serial Simulation

Time Consumption

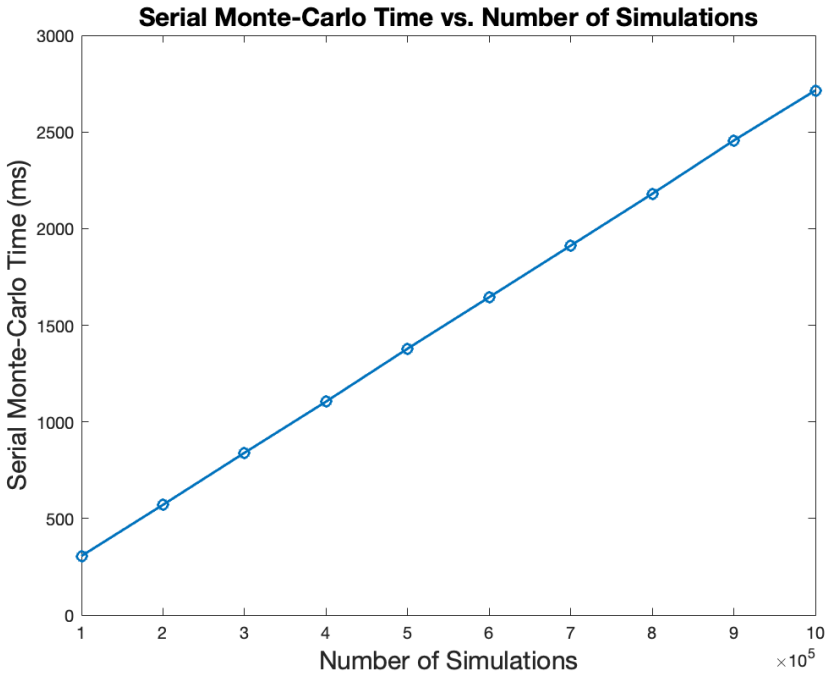


Figure 6.1: Performance of Serial Monte-Carl Simulation

Figure 6.1 illustrates how the number of simulations affects the time taken to run the classic serial simulation. As we scale up the number of simulations, there's a corresponding rise in the duration it takes to complete. In addition, the line graph displays a consistent slope, indicating that as the number of simulations grows, the time increment remains relatively steady.

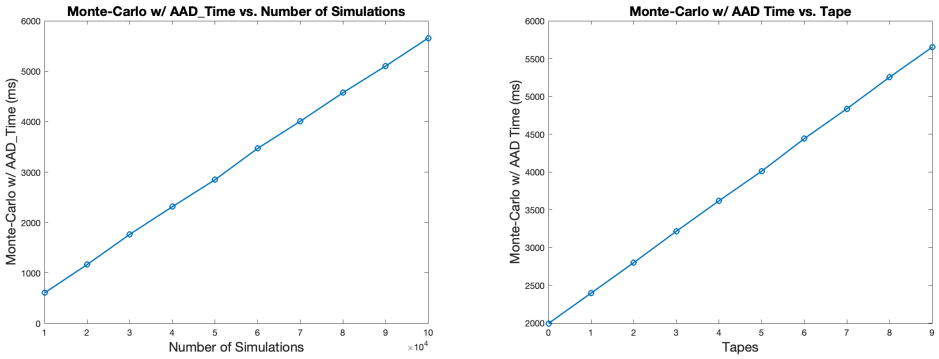


Figure 6.2: Performance of Serial Monte-Carl Simulations with AAD

Figure 6.2 explores the relationship between the number of simulations and the frequency of tape rewinding with respect to the execution time in simulations using AAD. It is evident that an increase in either the number of simulations or the frequency of tape rewinding makes the simulation spend more time until completion. Intriguingly, both line graphs demonstrate a relatively consistent gradient, indicating that the increments in execution time are proportionate as the number of simulations and tape rewinds escalates. It should be noted that the frequency of tape rewinding is fixed at 9 in the left panel of Figure 6.2, while the number of simulations is set at one hundred thousand in the right panel.

Memory Usage

Figure 6.3 displays how the maximum memory usage relates to both the number of simulations and the frequency of tape rewinding in the context of serial simulation employing AAD. It's evident that if we increase the number of simulations, the peak memory usage goes up too. On the other hand, the link between the maximum memory usage and the frequency of tape rewinds is not apparent. The memory consumption exhibits a notably inconsistent pattern as the number of tape rewinds increases. This irregularity is not mitigated even when considering the average values derived

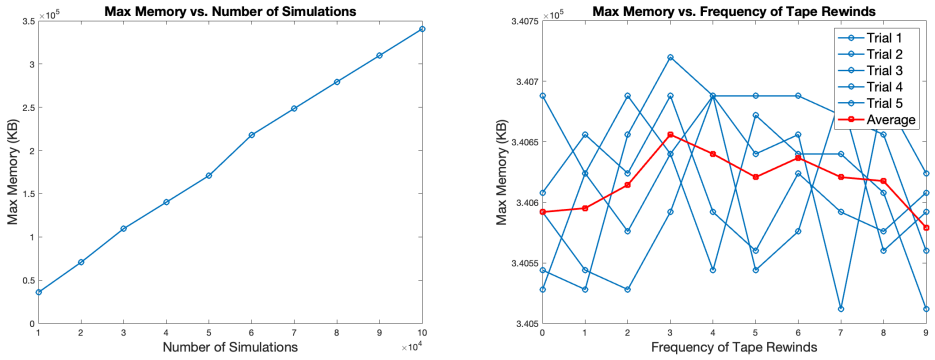


Figure 6.3: Memory Usage of Serial Monte-Carl Simulations with AAD

from five independent trials, further complicating any straightforward interpretation of the relationship between memory usage and tape rewinding frequency. Moreover, the graph between the number of simulations and the maximum memory usage is close to the line graph. However, the graph that maps the number of tape rewinds to the maximum memory usage exhibits irregular fluctuations.

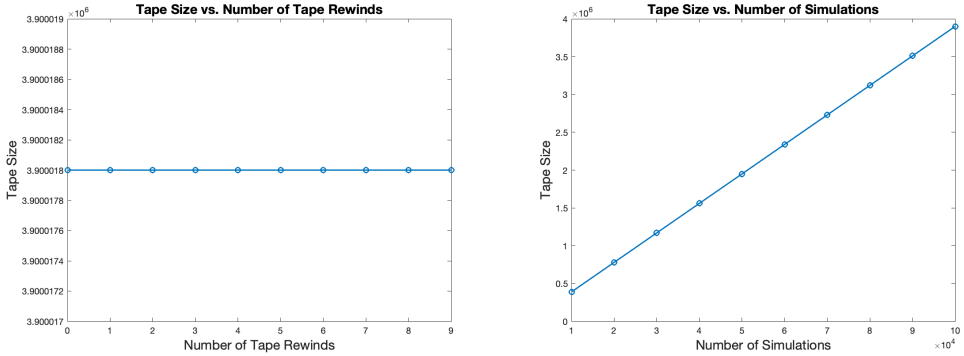


Figure 6.4: Tape Size of Serial Monte-Carl Simulations with AAD

Figure 6.4 shows how the size of the tape relates to both the number of simulations and the frequency of tape rewinds in the context of serial simulation employing AAD. After the first simulation, we rewind the tape back to the initialisation and reuse it. This means the tape size stays the same no matter how many times it's rewound. By contrast, if we keep the frequency of tape rewinds constant and increase the number of the simulations, the tape size does grow. Moreover, the graph plotting the number of simulations and the size of tapes is close to a line graph.

6.2 Parallel Simulation

Time Consumption

Figure 6.5 illustrate the performance of the parallel Monte-Carlo simulation with respect to thread counts. As we use more threads, the time taken to run the simulation drops. The most pronounced time reduction is observed when changing from one to two threads. After that, the rate of decrease slows down. When the number of threads exceed four, the gains in performance start to sluggish. Interestingly, using five threads even results in a slight increase in time compared to using just four.

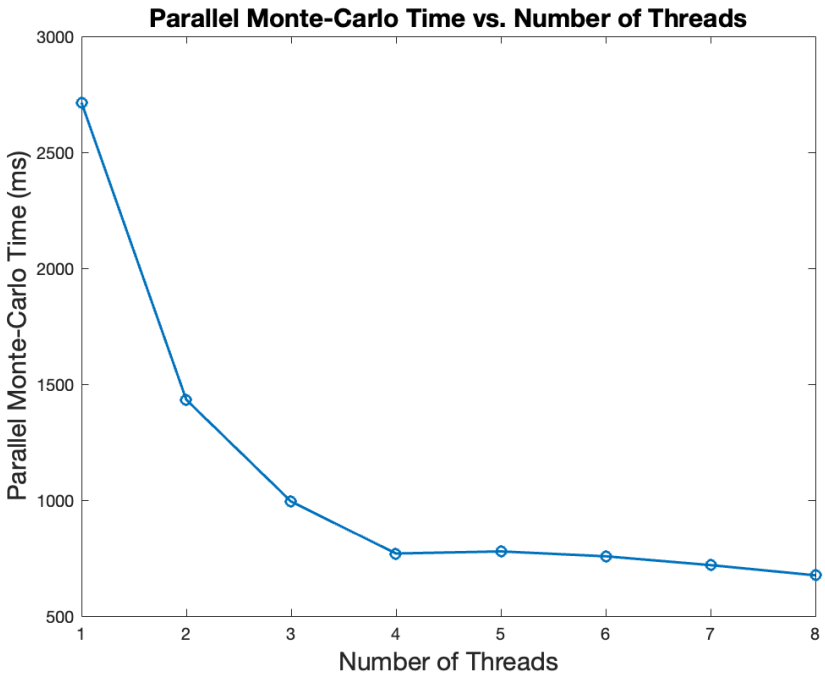


Figure 6.5: Performance of Parallel Monte-Carl Simulation

Figure 6.6 illustrates the performance of the parallel Monte-Carlo simulation incorporating AAD in relation to the number of threads utilised. Initially, as more threads are employed, the duration required to complete the simulation diminishes. This reduction is due to the distribution of the workload across a greater number of threads, facilitating parallel computations and thereby reducing the total time required. Nevertheless, beyond the usage of four threads, there is a notable increase and fluctuation in the

time expended. This phenomenon could be attributed to a variety of factors, including:

- Increased Computational Complexity
- Memory Consumption
- Parallel Efficiency
- Load Balancing

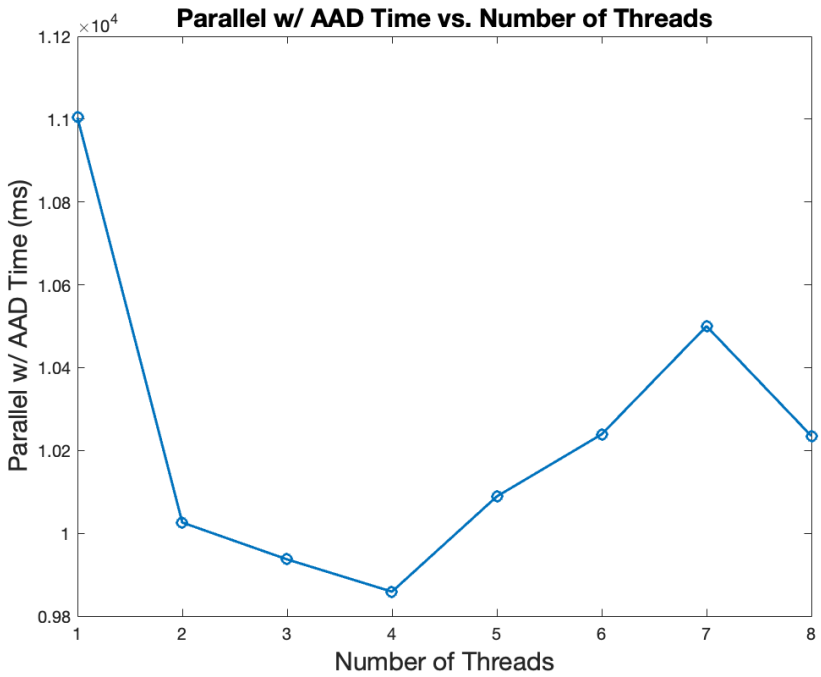


Figure 6.6: Performance of Parallel Monte-Carl Simulation with AAD

Memory Usage

Figure 6.7 illustrates the relationship between the maximum memory usage and the number of threads utilised in the parallel simulation. As the number of threads increases, there is a concomitant growth in the memory required to run the simulation. Moreover, the line graph demonstrates a steady ascent, indicating a uniform rate of increase in memory usage with

the addition of more threads. This pattern is evident irrespective of the use of AAD.

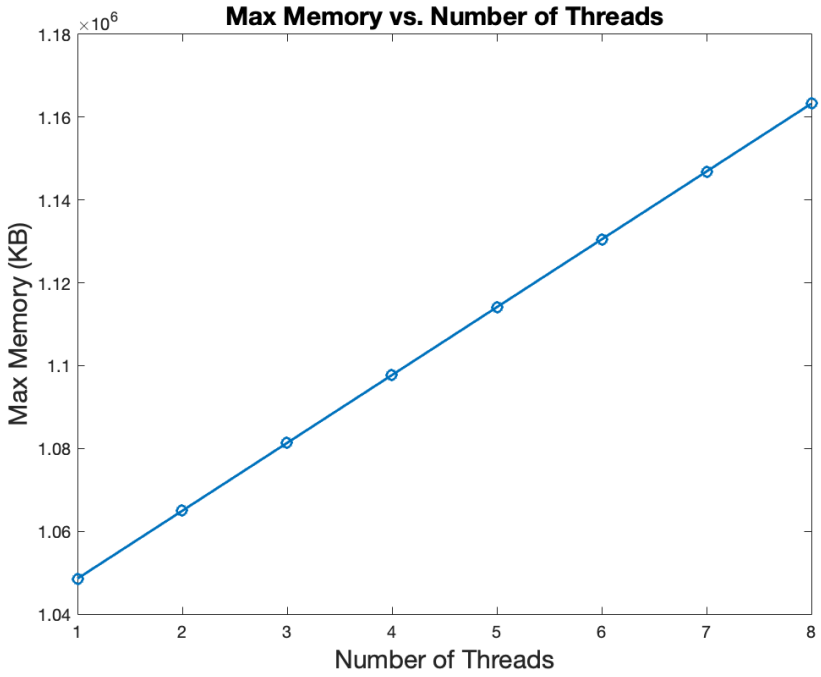


Figure 6.7: Memory Usage of Parallel Monte-Carl Simulation

6.3 Comparison

- Number of Simulations = 1,000,000 (1 million)
- Number of Steps = 5

Figure 6.8 shows the comparisons of the four district models with respect to time consumption and maximum memory usage, respectively. The parallel implementation without AAD demonstrates a significant improvement in terms of time efficiency. It operates 3.5x and 7.4x faster than the classic serial simulation, and the serial simulation with AAD, respectively. Conversely, the parallel simulation incorporating AAD exhibits the least efficient performance in terms of time, requiring 19.1x times more time upon completion relative to the fastest models. Moreover, this inefficiency

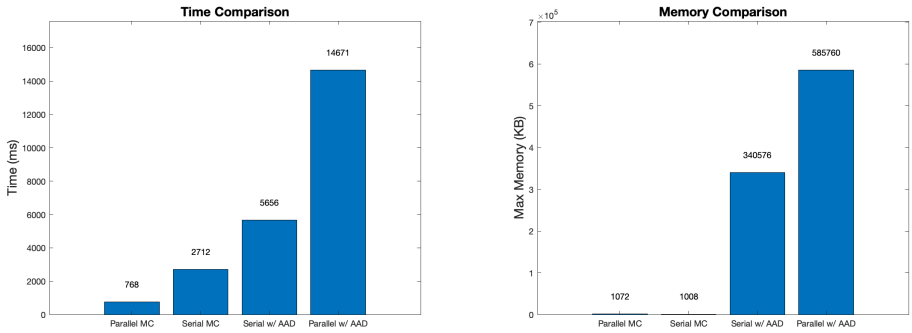


Figure 6.8: Summary of Results

is accompanied by a striking escalation in memory usage; the Parallel AAD model consumes memory resources approximately 550 times greater than both its serial and parallel Monte-Carlo counterparts. In essence, while the parallel simulation with AAD demands roughly 550 times more memory compared to the parallel simulation without AAD, it incurs a 19.1x times increase in time consumption relative to the latter.

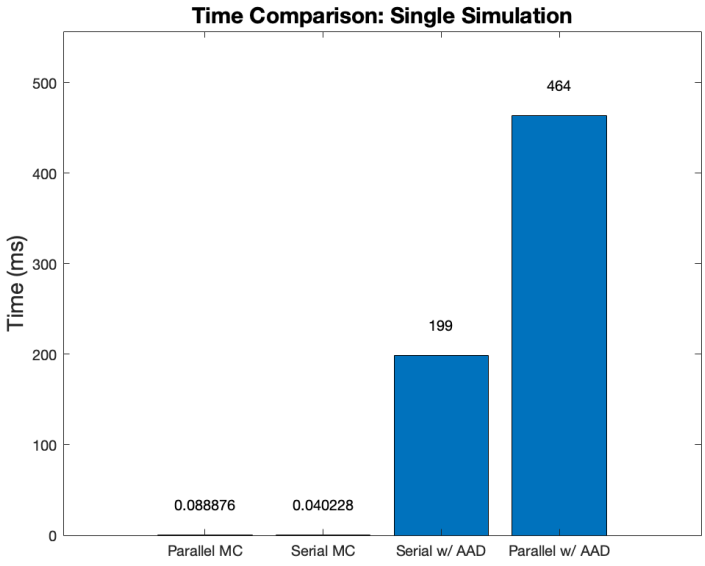


Figure 6.9: Performance of Each Model's Single Simulation

Figure 6.9 illustrates the performance of each model's single Monte-Carlo

simulation. In this analysis, one hundred thousand simulations are conducted, with the number of steps and threads set at 5 and 4, respectively. For a single Monte-Carlo simulation, the classic serial simulation proves to be the fastest, while the parallel simulation incorporating AAD is the slowest.

Serial vs. Parallel Models

The parallel model spends slightly more time than the serial version, which might be due to the overhead associated with parallelisation.

Serial Models: With and Without AAD

The time consumption for the serial version incorporating AAD is significantly higher compared to the model without AAD. This is expected since AAD involves a more complex structure, increasing the computational time considerably.

Parallel Models: With and Without AAD

Similar to the serial version, incorporating AAD to the parallel model increases the time consumption significantly. However, it seems the time increase is more substantial than in the serial model, possibly suggesting that the parallelisation might be less efficient with AAD or that the parallel overhead is compounded by the additional computations required by AAD.

Serial Model with AAD vs. Parallel Model with AAD

The parallelisation of the Monte-Carlo simulation with AAD does not appear to reduce time consumption; instead, it increases it. This trend might be attributable to various factors, including increased complexity, higher parallelisation overhead, and potential contention for resources, among other elements.

Chapter 7

Conclusion

Our project sought to explore various simulation methods to identify the fastest model. Through our work, we found some interesting results that were both expected and unexpected.
The parallel simulation method without AAD emerged as the top performer in our tests, displaying remarkable speed. Specifically, it completed tasks 3.5x and 7.4x times faster than the classic serial simulation and the serial simulation with AAD, respectively. Even more striking was its performance relative to the parallel simulation with AAD; it was 19.1x times faster.
Contrary to our initial expectations, the advanced methods incorporating AAD did not fare well. We had initially thought that these models would be among the fastest, if not the fastest. However, the actual results told a different story: These AAD models took significantly more time to complete the simulations than we had anticipated.
Additionally, we identified a major drawback with the AAD methods: their memory consumption. To be precise, the serial simulation with AAD consumed around 340 times more memory than the serial method. Even more concerning was that the parallel method with AAD consumed approximately 550 times more memory than both the serial and parallel methods. Such high memory usage could be a limiting factor for many systems. This means that even if we were willing to wait for the AAD simulations to complete, we might not have enough memory to run them in the first place.
In summary, whilst the parallel method proved to be our best option, the advanced AAD models forced us to reconsider our assumptions. We learnt that complexity or advanced features don't necessarily make a method the

best choice. Comprehensive testing is crucial before making a final decision. While speed is important, other factors like memory consumption must also be taken into account.

Bibliography

[1] A. Savine. *Modern computational finance: AAD and parallel simulations with professional implementation in C++.* Wiley, New Jersey, 2019.

[2] Dale Varberg, Edwin J. Purcell, Steve E. Rigdon. *Calculus Early Transcendentals.* Pearson, Essex, first edition, 2014.

[3] James Stewart. *Calculus Early Transcendentals.* Thomson Brooks/-Cole, Belmont, sixth edition, 2008.

[4] John C. Hull. *Options, futures, and other derivatives.* Prentice Hall, Boston, tenth edition, 2012.

[5] John C. Hull. *Risk management and financial institutions.* Wiley, New Jersey, fourth edition, 2015.

[6] Sheldon M. Ross. *Introduction to probability model.* Elsevier, Oxford, tenth edition, 2010.

[7] Chanho Eom. *On the cases of Pareto and related distributions.* Ajou University, 2022.