# OBJECT-ORIENTED IMPLEMENTATION OF OPTION PRICING VIA MATLAB: MONTE CARLO APPROACH

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖZGE TEKİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
FINANCIAL MATHEMATICS

JULY 2015

Approval of the thesis:

**OBJECT-ORIENTED IMPLEMENTATION OF OPTION PRICING VIA MATLAB: MONTE CARLO APPROACH**

submitted by **ÖZGE TEKİN** in partial fulfillment of the requirements for the degree of **Master of Science in Department of Financial Mathematics, Middle East Technical University** by,

Prof. Dr. Bülent Karasözen
Director, Graduate School of **Applied Mathematics** _____

Assoc. Prof. Dr. Ali Devin Sezer
Head of Department, **Financial Mathematics** _____

Assoc. Prof. Dr. Ömür Uğur
Supervisor, **Scientific Computing** _____

Assoc. Prof. Dr. Yeliz Yolcu Okur
Co-supervisor, **Financial Mathematics** _____

**Examining Committee Members:**

Assoc. Prof. Dr. Ömür Uğur
Scientific Computing, METU _____

Prof. Dr. Gerhard Wilhelm Weber
Scientific Computing, METU _____

Assoc. Prof. Dr. Ümit Aksoy
Department of Mathematics, Atılım University _____

**Date:** _____

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

Name, Last Name: ÖZGE TEKİN

Signature :

# ABSTRACT

OBJECT-ORIENTED IMPLEMENTATION OF OPTION PRICING VIA MATLAB: MONTE CARLO APPROACH

Tekin, Özge

M.S., Department of Financial Mathematics

Supervisor : Assoc. Prof. Dr. Ömür Uğur

Co-Supervisor : Assoc. Prof. Dr. Yeliz Yolcu Okur

July 2015, 69 pages

There are many applications in finance and investment that require the use of methods, which involve time-consuming and laborious iterative calculations. Although closed-form solutions are available for some specific instruments, the valuation methods used in financial engineering in many other situations require analytical methods, which compute approximate solutions on computing environments.

Option pricing is one of the most important and active topics in financial engineering, and there are many fundamental methods for numerous different type options in literature as well as in the derivative market. Close investigation of options shows, besides the underlying parameters, significant relation and similarities, even inheritance, such as options on options. On the other hand, investigation of the valuation methods reveals the use of similar fundamental algorithms, such as Monte Carlo technique or solving the corresponding partial differential equation.

Therefore, a software environment for pricing financial derivatives should be as flexible as possible to modify and extend the options as well as methods for pricing them. Object-oriented principles and modeling techniques, which contains analysis, design and implementation, have to be utilized for such a goal. After having analyzed options and pricing methods, the classes and subclasses to design a hierarchy that forms the structure of those options and methods are to be organized. As the relation between classes is so tight that each individual unit, objects, must be qualified to sending or

receiving information to other objects while being responsible for their own work. Many modern object-oriented programming languages are transferable from one to another. Hence, MATLAB® is preferred in this study as it provides numerous built-in functions and is a very suitable platform to develop OOP based softwares.

*Keywords* : Object-Oriented Programming, Monte Carlo Methods, Option Pricing

# ÖZ

OPSİYON FİYATLANDIRMASININ NESNE YÖNELİMLİ MATLAB UYGULAMASI: MONTE CARLO YÖNTEMİ

Tekin, Özge

Yüksek Lisans, Finansal Matematik Bölümü

Tez Yöneticisi : Doç. Dr. Ömür Uğur

Ortak Tez Yöneticisi : Doç. Dr. Yeliz Yolcu Okur

Temmuz 2015, 69 sayfa

Finans ve yatırım alanlarında zaman alan ve zorlu yinelemeli hesaplar içeren yöntemlerin kullanımını gerektiren birçok uygulama vardır. Finans mühendisliğinde belirli türev ürünler için kapalı-form çözümler olmasına rağmen, birçok durumda hesaplama yöntemleri yaklaşık çözümleri bilgisayar ortamında hesaplayan analitik yöntemler gerekmektedir.

Opsiyon fiyatlama, finans mühendisliğindeki en önemli ve aktif konulardan birisidir ve literatürde çeşitli opsiyonları fiyatlandırmak için birçok temel yöntem bulunmaktadır. Opsiyonlar ve bağlı oldukları parametreler yakından incelendiğinde, opsiyonlar üzerine yazılan opsiyonlar gibi, oldukça önemli bağıntı ve benzerlikler, hatta kalıtım ilişkisi gözlemlenir. Bununla birlikte, fiyatlama yöntemleri incelendiğinde Monte Carlo tekniği ve karşılık gelen kısmi diferansiyel denklem çözümü gibi benzer temel algoritmaların kullanıldığı görülmüştür.

Bu sebeple, finansal türev araçları fiyatlandırmak için oluşturulan yazılım ortamı, opsiyonları ve fiyatlama tekniklerini yeniden düzenlemek ve genişletmek için mümkün olduğunca esnek olmalıdır. Analiz, tasarım ve uygulama basamaklarını içeren nesne yönelimli ilkeler ve modelleme teknikleri bu amaç için kullanılmaktadır. Opsiyon ve fiyatlama tekniklerini analiz ettikten sonra bu opsiyonların yapısını oluşturan hiyerarşiyi tasarlamak için sınıflar ve alt sınıflar organize edilmiştir. Sınıflar arasındaki ilişki oldukça sıkıdır ve her bağımsız birim yani nesne kendi işleyişinden sorumluyken diğer

nesnelerle bilgi alışverişini gerçekleştirir. Birçok nesne yönelimli programlama dili birbirine dönüştürülebilir. Sağladığı hazır fonksiyonlar ve nesne yönelimli programlama için uygun olması sebebiyle bu çalışmada MATLAB® programlama dili tercih edilmiştir.

*Anahtar Kelimeler*: Nesne Yönelimli Programlama, Monte Carlo Yöntemi, Opsiyon Fiyatlama

*To my parents*
*and*
*my love Ulaş*

# ACKNOWLEDGMENTS

First of all, I would like to express my very great appreciation to my thesis supervisor Assoc. Prof. Dr. Ömür Uğur and co-supervisor Assoc. Prof. Dr Yeliz Yolcu Okur for their patient guidance, encouragement and invaluable advices not only throughout the development and preparation of this thesis but also my graduate education so far.

I would like to thank the members of my thesis committee, Prof. Dr. Gerhard Wilhelm Weber and Assoc. Prof. Dr. Ümit Aksoy for their valuable comments and insight.

I deeply thank all members of the Institute of Applied Mathematics (IAM) for providing a helpful and sincere environment. Additionally, my special thank goes to members of room S206, Cansu Evcin, Sinem Kozpınar, Neşe Öztop, Ayşe Sarıaydın, Ahmet Sınak, Meral Şimşek and Büşra Temoçin for their kindness, friendship and support.

Finally, none of my studies would have been possible without the support of my beloved ones. I would like to express my gratefulness to my parents for their continuous support, encouragement, endless love and understanding in all aspects of my life. Also, I would like to thank my significant other, Ulaş, for his full support, encouragement, patience and love. I have been extremely lucky to have them.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| $C$ | Price of European call |
| $P$ | Price of European put |
| $K$ | Strike (exercise) price |
| $S$ | Price of underlying asset |
| $T$ | Time to expiration (maturity time) |
| $B$ | Barrier |
| $C_{do}$ | Price of European down-and-out call |
| $P_{do}$ | Price of European down-and-out put |
| $C_{di}$ | Price of European down-and-in call |
| $P_{di}$ | Price of European down-and-in put |
| $C_{uo}$ | Price of European up-and-out call |
| $P_{uo}$ | Price of European up-and-out put |
| $C_{ui}$ | Price of European up-and-in call |
| $P_{ui}$ | Price of European up-and-in put |
| $r$ | Risk-free interest rate |
| $W$ | Wiener process (Brownian motion) |
| a.s. | almost surely |
| i.i.d | Independent and identically distributed |
| $\Omega$ | Sample space |
| $\mathcal{F}$ | $\sigma$-algebra |
| $\mathbb{P}$ | Probability measure |
| $\mathbb{Q}$ | Risk-neutral probability measure |
| $\mathbb{E}(X)$ | Expectation of the process $X$ |
| $\mathbb{R}$ | Real numbers |
| $\mathbb{N}$ | Natural numbers |
| $N(x)$ | The cumulative normal distribution function |
| OOP | Object-oriented programming |
| UML | Unified modeling language |
| GUI | Graphical user interface |
| GUIDE | Graphical user interface development environment |
| OOMCOP | Object-oriented Monte Carlo option pricer |

# CHAPTER 1

# INTRODUCTION

## 1.1 Aim of the Thesis

The aim of the thesis is to design and implement an object-oriented software to price different kinds of options by using Monte Carlo approach. Option valuation is a sophisticated area of financial mathematics and it involves financial, mathematical and computational theories. In literature, a wide number of methods are applied to the numerous types of options. In this thesis, Monte Carlo approach is preferred since it is a flexible and well defined method. Moreover, it can be easily implemented especially for some complex options lacking any closed-form solutions.

Object-oriented programing paradigm is chosen in an attempt to provide the reusability of the code and to make the software environment more flexible and robust. The software is divided into five main parts and these parts are designed and organized within themselves. These main parts are also interacting between each other.

The implementation of the Object-Oriented Monte Carlo Option Pricer (OOMCOP) is performed by MATLAB® programming language and a graphical user interface (GUI) is designed by MATLAB®/ GUIDE to make the software user friendly.

## 1.2 Literature Review

Option pricing is one of the most attractive topics in financial mathematics with its broad and still expanding theory. The literature on option pricing dates back to the Ph.D. thesis of Louis Bachelier, The Theory of Speculation (1900), based on the analogy between a Brownian motion without drift (also known as arithmetic Brownian motion) and stock prices [34]. He suggested the following model for stock prices:

$$dX_t = \mu X dt + \sigma dW, \text{ for } X(0) = a, \tag{1.1}$$

where $a$, $\mu$ and $\sigma$ are constant parameters. Because Bachelier's model allowed negative stock prices, the model was not suitable for the real stock movements [38]. In 1965, the geometric Brownian motion (GBM) was introduced by Paul A. Samuelson, who is also the inventor of the terms, "American" and "European" options [44].

The celebrated Black-Scholes model [2] is one of the most important cornerstones of the option pricing theory. The core idea behind the formula is that the option price at any time can be mimicked by a dynamically replicating portfolio. In their original work Fisher Black and Myron Scholes applied it to the option whose underlying follows a geometric Brownian motion. The Black-Scholes formula is also valid under different assumptions as shown in Klebaner [29].

Barrier options are path-dependent exotic options that become activated or extinguished if their underlying hits the certain barrier level [31]. The first close form solution for the price of down-and-out call option is formulated by Robert C. Merton in 1973 [36]. The analytical formula for barrier option was extended for eight different types of barrier by Mark Rubinstein and Eric Reiner [43].

The Monte Carlo approach has known as a flexible and easy to implement computational tool in finance. The Monte Carlo approach for option pricing was first introduced by Phelim Boyle in 1977 [6]. The approach was applied to path dependent Asian option by Mark Broadie and Paul Glasserman in 1996 [8].

Procedural or object-oriented software environments in order to price options by Monte Carlo approach can be constructed by using programming languages, such as Java, C++, Python, Excel, and MATLAB®. Elke and Ralf Korn gave the general algorithm to apply Monte Carlo approach. They considered the simulation of continuous and discontinuous paths [30]. Ömür Uğur gave the algorithm and MATLAB® codes for option pricing by Monte Carlo simulations [51]. Various applications of Monte Carlo approach by MATLAB® for plain-vanilla options, barrier and Asian options and some option strategies were shown by Paolo Brandimarte in his book [7].

In literature, object-oriented paradigm was applied for several types of option pricing methods. Joerg Kienitz and Daniel Wetterau wrote algorithms to price several type of options by using various pricing techniques. They also constructed an object-oriented pricer engine for options and examined fast Fourier option pricing method in detail [28]. Daniel J. Duffy built classes for Binomial, Black-Scholes, Monte-Carlo and for some numerical methods to price options by using C++ [14]. Holger Kammeyer and Joerg Kienitz showed the design of a software for Carr-Madan fast Fourier transform pricing and Monte Carlo pricing. The calibration of the Heston-Hull-White model parameters according to the market data was also performed by utilizing C++ programming language [27]. Umberto Cherubini and Giovanni Della Lunga was applied object-oriented paradigm with Java programming language to price and option by using analytical, binomial and Monte Carlo simulation approaches [10]. Monte Carlo approach to price European, American and some exotic options by object-oriented paradigm was investigated by Nick Webber and he used VBA environment for this purpose [53]. Daniel Duffy and Joerg Kienitz was applied Monte Carlo method to price one-factor and multi-factor equity options by using C++ programming language in their book [15].

## 1.3 Structure of the Thesis

The thesis is organized as follows:

- In this chapter, the aim of thesis and the works in literature related to the study are given.
- In Chapter 2, the required definitions and theoretical framework concerning option pricing are presented to analyze the software with regard to object-oriented paradigm.
- In Chapter 3, the terminology and basic concepts of object-oriented programming and unified modeling language (UML) are given. The terminology is highly crucial for the better understanding of the design of the software.
- Chapter 4 presents the detailed design of the software with UML diagrams of main classes and their subclasses. In addition, some basic examples are given to clarify the concepts and implementation.
- Finally, the conclusion and outlook are presented in Chapter 5.

# CHAPTER 2

# PRELIMINARIES

In this chapter, the essential financial mathematical background for object-oriented analysis is given. This analysis is crucial to understand the every units of the software deeply. Because the design of the classes is based on these core units. This section contains three parts; financial derivatives, stochastic differential equations with and without jumps and Monte Carlo approach. In the first part, options, which are one of the most commonly used financial derivatives, are examined according to their types, namely, plain-vanilla and barrier. Next, some main option strategies with their payoffs are given. In the stochastic differential equations (SDEs) part, the SDEs are examined and the Black-Scholes framework is given. The main pure jump processes are introduced to form a basis for jump-diffusion processes. The general form of the jump-diffusion process and Merton jump diffusion model are given. In the final part, the basics for Monte Carlo approach are given and the implementation of this approach onto the European option pricing is explained in detail.

## 2.1 Options

Derivatives are the financial instruments that are used with the purpose of managing financial risks. Since they are derived from an underlying asset they are named as derivatives. Future contracts, forwards, options and swaps are the most commonly used types of derivative instruments. The examples of the derivative products according to their deliverability status and the underlying type are given in Table 2.1 [18]. Within the scope of this thesis, equity options are examined in detailed.

According to Cox and Rubinstein's definition, an option is a contract giving its owner the right to buy (in the case of a call) or sell (in the case of a put) a fixed number of shares of a specified common stock at a fixed price at any time on or before a given date [13]. The options are classified as European or American style of options according to their exercise time specifications. European options can only be exercised at maturity date while American options can be exercised at any time during the lifetime of the option. As expected from the definition, with their flexible exercise time the American options are more expensive than the corresponding European options. Both European and American options are also classified according to the type as plain-vanilla or exotic options. Exotic options have special properties in addition to

Table 2.1: Derivative products according to underlying types [18]

| Deliverability | Underlying Type | Derivative Type |
|---|---|---|
| Deliverable | Interest Rate | Forward rate agreement (FRA)<br>Interest rate swap<br>Interest rate option |
| | Foreign exchange | Outright forward<br>Foreign exchange swap<br>Currency swap<br>Currency option |
| | Equity & stock index | Equity forward<br>Equity swap<br>Equity option |
| | Commodity | Commodity forward<br>Commodity swap<br>Commodity option |
| Non-deliverable | Weather | Weather future<br>Weather option |
| | Credit | Credit default swaps (CDS)<br>Credit-linked notes<br>CDS option |

plain-vanilla options and they include various kinds of options, namely, barrier, look-back, Asian, chooser, binary, etc.. In this thesis, European type plain vanilla and barrier options are investigated.

### 2.1.1 Plain-Vanilla Options

The standard call and put options are named as plain-vanilla options. The payoff functions are given in Definition 2.1.

**Definition 2.1.** [37] A European plain-vanilla call with the strike price $K$ is an option with the payoff at time $T$ of

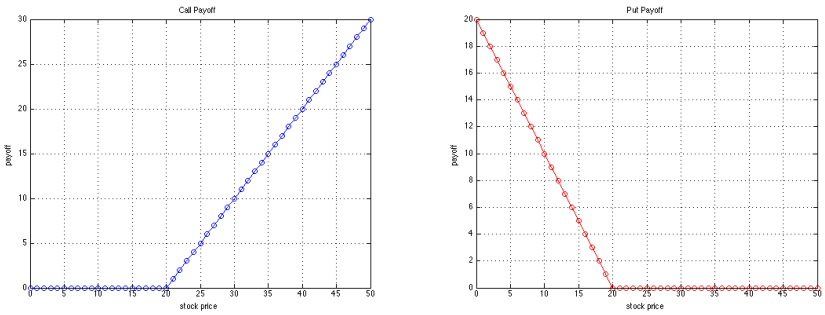
$$V = (S_T - K)^+ = \max(0, S_T - K) = \begin{cases} S_T - K, & \text{if} \quad S_T > K, \\ 0, & \text{if} \quad S_T \leq K. \end{cases} \tag{2.1}$$

A European plain-vanilla put with the strike price $K$ is an option with the payoff at time $T$ of

$$V = (K - S_T)^+ = \max(0, K - S_T) = \begin{cases} K - S_T, & \text{if} \quad S_T < K, \\ 0, & \text{if} \quad S_T \geq K, \end{cases} \tag{2.2}$$

where $S_T$ represents the value of the underlying asset at maturity time $T$.

The payoff graphs of call and put options for $K = 20$ and the stock price range is between $[0, 50]$ is given in Figure 2.1.




(a) Payoff of the call option (b) Payoff of the put option

Figure 2.1: Payoff diagrams for call and put options

### 2.1.2 Barrier Options

Barrier options are one of the most commonly used options in the market. They are more specialized versions of the plain-vanilla options with their additional barrier constraints. It is in the class of path dependent options and these options are cheaper than the plain-vanilla options. Because the payoff not only depends on the final value of the underlying asset and strike price but also depends on the path of the underlying asset. The close-form solutions are available for European type standard barrier options and payoff functions are given in Definition 2.2 [43].

**Definition 2.2.** [20] Let $S_0$ be the price of an underlying asset at time $0$ and $T$ be the maturity time of the option. $B > 0$ is called a barrier.

1. If $B < S_0$ we can define the down-options:

(a) i. A *down-and-out call* with strike price $K$ and out-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (S_T - K)^+, & \text{if} \quad S_t > B \quad \text{for} \quad 0 \leq t \leq T, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

Denote the price of such a down-and-out call at $t = 0$ by $C_{do}^{K,B}(S_0, T)$.

ii. A *down-and-out put* with strike price $K$ and out-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (K - S_T)^+, & \text{if} \quad S_t > B \quad \text{for} \quad 0 \leq t \leq T, \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

Denote the price of such a down-and-out put at $t = 0$ by $P_{do}^{K,B}(S_0, T)$.

(b) i. A *down-and-in call* with strike price $K$ and in-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (S_T - K)^+, & \text{if for some } t \in [0, T] \quad \text{such that} \quad S_t \leq B, \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

Denote the price of such a down-and-in call at $t = 0$ by $C_{di}^{K,B}(S_0, T)$.

ii. A *down-and-in put* with strike price $K$ and in-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (K - S_T)^+, & \text{if for some } t \in [0, T] \quad \text{such that} \quad S_t \leq B, \\ 0, & \text{otherwise.} \end{cases} \tag{2.6}$$

Denote the price of such a down-and-in put at $t = 0$ by $P_{di}^{K,B}(S_0, T)$.

2. In the case that $B > S_0$ we can define the up-options:

(a) i. An *up-and-out call* with strike price $K$ and out-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (S_T - K)^+, & \text{if} \quad S_t < B \quad \text{for} \quad 0 \leq t \leq T, \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

Denote the price of such a up-and-out call at $t = 0$ by $C_{uo}^{K,B}(S_0, T)$.

ii. An *up-and-out put* with strike price $K$ and out-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (K - S_T)^+, & \text{if} \quad S_t < B \quad \text{for} \quad 0 \leq t \leq T, \\ 0, & \text{otherwise.} \end{cases} \tag{2.8}$$

Denote the price of such a up-and-out put at $t = 0$ by $P_{uo}^{K,B}(S_0, T)$.

(b) i. An *up-and-in call* with strike price $K$ and in-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (S_T - K)^+, & \text{if for some } t \in [0, T] \quad \text{such that} \quad S_t \geq B, \\ 0, & \text{otherwise.} \end{cases} \tag{2.9}$$

Denote the price of such a up-and-in call at $t = 0$ by $C_{ui}^{K,B}(S_0, T)$.

ii. An *up-and-in put* with strike price $K$ and in-barrier $B$ is an option with payoff at time $T$ of

$$V = \begin{cases} (K - S_T)^+, & \text{if for some } t \in [0, T] \quad \text{such that} \quad S_t \geq B, \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

Denote the price of such a up-and-in put at $t = 0$ by $P_{ui}^{K,B}(S_0, T)$.

In Figure 2.2, some possible asset price paths that are assumed to follow a geometric Brownian motion (GBM) with parameters $S_0 = 100$, $\mu = 0.1$, $\sigma = 0.2$, $T = 1$ and $B = 92$ are shown. This figure is demonstrated to give an example of down-and-out call options. The asset price path displayed by purple color becomes worthless because it crosses the barrier and other paths can be considered and priced as standard European call options.

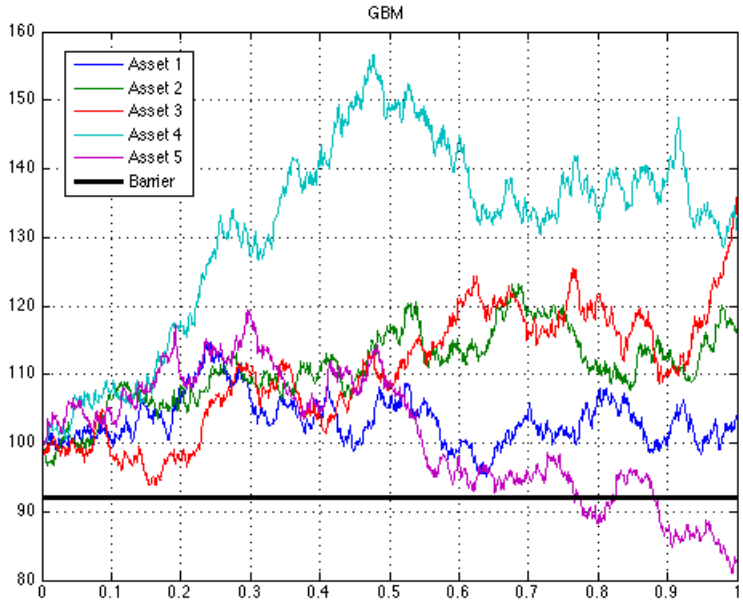



Figure 2.2: Simulation results of asset paths for down-and-out call option

## 2.2 Option Strategies

Depending on the risk preferences of investors and market forecasts, options can be combined with stocks or other options [9]. A basic classification can be done by separating the strategies as bullish, bearish and neutral. Most commonly used strategies with their explanations are given in Table 2.2 [12].

Table 2.2: Option strategies according to the market expectation

| | | |
|---|---|---|
| Bullish | Bull call spread | Buy 1 call, sell 1 call at higher strike |
| | Bull put spread | Sell 1 put, buy 1 put at lower strike with same expiry |
| | Covered call | Buy stock, sell calls on a share-for-share basis |
| | Protective Put | Own 100 shares of stock, buy 1 put |
| Bearish | Bear put spread | Sell 1 put, buy 1 put at higher strike |
| | Bear call spread | Sell 1 call, buy 1 call at higher strike |
| Neutral | Collar | Own stock, protect by purchasing 1 put and selling 1 call with a higher strike |
| | Straddle | Buy 1 call, sell 1 put at same strike |
| | Strangle | Buy 1 call with higher strike, buy 1 put with lower strike |
| | Call butterfly spread | Sell 2 calls, buy 1 call at next lower strike, buy 1 call at next higher strike (the strikes are equidistant) |

In the scope of this thesis, the straddles, strangles and butterfly spreads are considered. Definitions and payoff equations of these strategies are given in Definition 2.3,

Definition 2.4, Definition 2.5, respectively [9].

**Definition 2.3.** A straddle is an option strategy which is a combination of the long position in a call and a put that have a same strike price and maturity with the payoff:

$$V = (S_T - K)^+ + (K - S_T)^+, \tag{2.11}$$

where $K$ is the strike price, $S_T$ is the stock price at maturity.

**Definition 2.4.** A strangle is an option strategy which is a combination of the long position in a call and a put that have a lower strike price and same maturity with the payoff

$$V = (S_T - K_1)^+ + (K_2 - S_T)^+, \tag{2.12}$$

where $K_1$ is the higher strike price, $K_2$ is the lower strike price, $S_T$ is the stock price at maturity.

**Definition 2.5.** A butterfly spread, also called as sandwich spread, is an option strategy which is a combination of the long position in a call with the higher strike, long position in a call with the lower strike and short position in two call with the middle strike price with the payoff

$$V = (S_T - K_1)^+ - 2[(S_T - K_2)^+] + (S_T - K_3)^+, \tag{2.13}$$

where $K_1$ is the higher strike price, $K_2$ is the middle strike price, $K_3$ is the lower strike price, $S_T$ is the stock price at maturity.

In Figure 2.3, the payoff diagrams of the straddle, strangle and butterfly spread are given for $K_1 = 30$, $K_2 = 20$, $K_3 = 40$ and the stock price range is between $[0, 60]$.

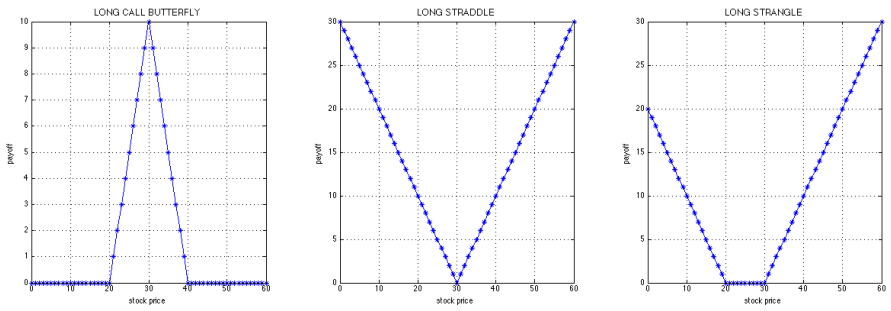



Figure 2.3: Payoff diagrams of straddle, strangle and butterfly spread

Rather than just buying a call or a put as given in Figure 2.1, these option strategies provide protection for different directions of movements of the underlying asset.

## 2.3 Stochastic Differential Equations with Jumps

In this section SDEs are examined and the Black-Scholes framework is given. The main pure jump processes are introduced to a substructure for jump-diffusion processes. The general form of the jump-diffusion process and Merton jump diffusion model are given. The solutions of the particular models and some required theorems are also presented.

### 2.3.1 Stochastic Differential Equations and Itô Process

Stochastic differential equations are the generalized form of ordinary differential equations. Stochastic differential equations differ from ordinary differential equations by the additional Wiener process, which is used to capture the randomness. Stochastic differential equations are used in numerous fields, including biology, chemistry, epidemiology, mechanics, microelectronics, economics and finance [25]. In this subsection, the basic units of stochastic differential equations and necessary theorems for solutions are given.

**Definition 2.6.** [37] A stochastic process $X$ is a collection of random variables

$$(X_t, t \in T) = (X_t(\omega), t \in T, \omega \in \Omega),$$

defined on some space $\Omega$.

**Definition 2.7.** [1] $\{W_t\}_{t\geq 0}$ is a continuous stochastic process which is called Wiener process or Brownian motion under the following statements:

1. Its initial value is zero, i.e. $W(0) = 0$.
2. If $r < s < t < u$, $W_u - W_t$ and $W_s - W_r$ are independent variables, this property is called as independent increments.
3. The process has stationary increments; if $s \leq t$ the stochastic random variables $W_t - W_s$ and $W_{t-s} - W_0$ have the same probability law.
4. $\mathbb{P}$ a.s. the map $u \longmapsto W_u(w)$ is continuous.

In Figure 2.4, a number of paths of a Wiener process satisfying the statements given in Definition 2.7 are illustrated.

**Definition 2.8.** [45] General form of an SDE is represented as follows:

$$dX_t = a(t, X_t)dt + \sigma(t, X_t)dW_t,\ X_0 = x_0, \tag{2.14}$$

or in integral form

$$X_t = \int_0^t a(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \tag{2.15}$$

where $a(t, S_t)$ and $\sigma(t, S_t)$ represent the drift and diffusion terms of this process, respectively.

Eq. 2.14 shows the general form of the Itô stochastic differential equations. The process takes special names according to the specification of drift and diffusion terms, such as Geometric Brownian Motion, Ornstein-Uhlenbeck, Vasicek, etc..

The following theorems are crucial for the existence and uniqueness of solutions of specific stochastic differential equations.

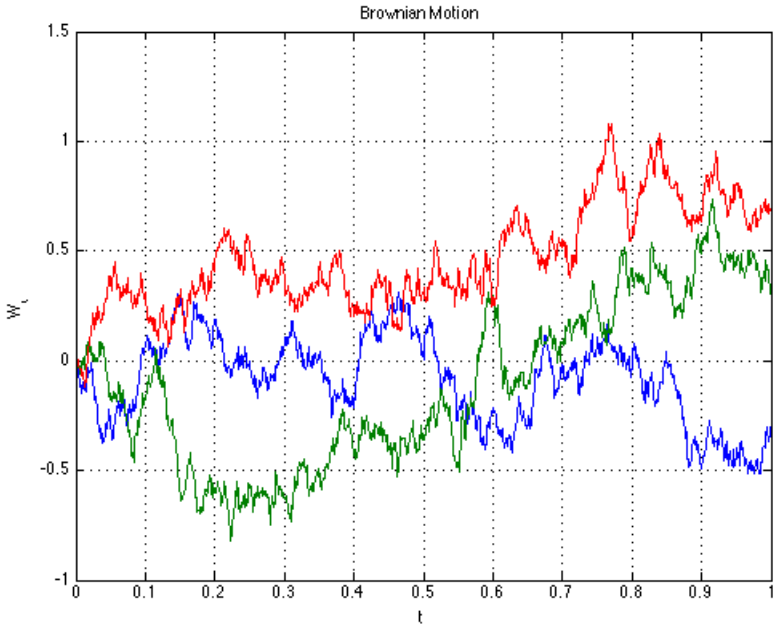



Figure 2.4: Paths of Wiener process

**Theorem 2.1** (**Conditions for Existence and Uniqueness of Solution to SDEs**). *For $0 < t < \infty$, let $t \in [0, T]$ and consider the Itô stochastic differential equation*

$$dX_t = a(t, X_t)dt + b(t, X_t)dW_t. \tag{2.16}$$

*Sufficient condition for existence and uniqueness of solution to this SDE are linear growth*

$$|a(t, x)| + |b(t, x)| \leq C(1 + |x|), \qquad \textit{for all } x \in \mathbb{R} \textit{ and } t \in [0, T], \tag{2.17}$$

*for some finite constant $C > 0$, and Lipschitz continuity*

$$|a(t, x) - a(t, y)| + |b(t, x) - b(t, y)| \leq D|x - y|, \quad \textit{for all } x, y \in \mathbb{R} \textit{ and } t \in [0, T], \tag{2.18}$$

*where $0 < D < \infty$ is the Lipschitz constant.*

**Theorem 2.2.** *Let $(X_t)_{0 \leq t \leq T}$ be an Itô process,*

$$X_t = X_0 + \int_0^t K_s ds + \int_0^t H_s dW_s,$$

*and f be a twice continuously differentiable function. Then*

$$f(X_t) = f(X_0) + \int_0^t f'(X_s)dX_s + \frac{1}{2}\int_0^t f''(X_s)d\langle X, X_s\rangle,$$

*where, by definition*

$$\langle X, X_t\rangle = \int_0^t H_s^2 ds$$

*and*

$$\int_0^t f'(X_s)dX_s = \int_0^t f'(X_s)K_s ds + \int_0^t f'(X_s)H_S dW_s.$$

The geometric Brownian motion provides the existence and uniqueness conditions given in Theorem 2.1. The solution of this process is obtained by applying Itô formula and the solution is given in the Lemma 2.3.

**Lemma 2.3.** *The geometric Brownian motion (GBM) is defined with the specification of drift and diffusion terms in the general form of the stochastic differential equation as $a(t, X_t) = \mu S_t$ and $\sigma(t, X_t) = \sigma S_t$, respectively. The solution of the GBM,*

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{2.19}$$

*where $\mu$ and $\sigma$ are non-negative constants, is given by*

$$S_t = S_0 exp\left\{ \left( \mu - \frac{1}{2}\sigma^2 \right) t + \sigma W_t \right\}. \tag{2.20}$$

The proof can be found in [51].

### 2.3.2 Black Scholes Framework

The celebrated Black-Scholes formula is one of the most important cornerstones in financial mathematics. The following conditions are assumed by Black and Scholes to derive the formula [2]:

- The short-term interest rate is known and constant through time,
- The stock price follows a geometric Brownian motion with constant $\mu$ and $\sigma$,
- The stocks pay no dividends,
- The options are European and can only be exercised at expiration,
- There are no transactions costs or taxes,
- There are no riskless arbitrage opportunities,
- All securities are perfectly divisible and short selling is allowed.

The Black-Scholes theory based on the Efficient Market Hypothesis which states that there is no-arbitrage opportunities in efficient markets. In other words, it is impossible to beat the market. According to Eugene F. Fama's definition "A market in which prices always fully reflect available information is called efficient" [16]. The arbitrage is defined as follows.

**Definition 2.9.** [1] An arbitrage opportunity is the possibility to make a risk-free profit without risk and net initial capital. An arbitrage portfolio $\theta$ satisfies the following conditions:

$$V_0(\theta) \leq 0 \quad \text{and} \quad V_1(\theta) > 0 \quad \text{or} \tag{2.21}$$

$$V_0(\theta) < 0 \quad \text{and} \quad V_1(\theta) \geq 0. \tag{2.22}$$

The Black-Scholes portfolio considers a market consisting of a riskless bond $B_t$ and a stock $S_t$ with the following equations:

$$dB_t = rB_t dt, \tag{2.23}$$

$$dS_t = S_t(\mu dt + \sigma dW_t), \tag{2.24}$$

where $\mu$ and $\sigma$ are constant and $r$ is the risk-free interest rate. The Black-Scholes analytical formulas for the prices of European call and put option for non-dividend paying stocks are as follows:

$$C = S_0 N(d_1) - Ke^{-rT} N(d_2), \tag{2.25}$$

$$P = Ke^{-rT} N(-d_2) - S_0 N(-d_1), \tag{2.26}$$

where $N(\cdot)$ states the cumulative standard normal distribution function

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}y^2} dy \tag{2.27}$$

and

$$d_1 = \frac{\ln(S_0/K) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{(T)}}, \tag{2.28}$$

$$d_2 = \frac{\ln(S_0/K) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{(T)}}, \tag{2.29}$$

remark that

$$d_2 = d_1 - \sigma\sqrt{T}, \tag{2.30}$$

where $S_0$, $K$, $r$, $\sigma$ and $T$ represent the initial price of the stock, the strike price, interest rate, the volatility of the stock and time of maturity, respectively.

The formula can be derived by using different approaches. Straightforward integration, Feynman-Kac theorem, Capital Asset Pricing Model (CAPM) approaches are some severals ways to derive the formula. In the original paper of Black-Scholes, transforming the Black-Scholes PDE into heat equation approach is used. The details about derivation of the formula can be found in [2, 42, 47, 51].

The relation between call and put option prices is indicated by put-call parity given in the following proposition.

**Proposition 2.4.** *[1] Consider a European call and a European put, both with strike price $K$ and time to maturity $T$. Then, the following relation holds.*

$$P = Ke^{-r(T-t)} + C - S. \tag{2.31}$$

*This means that, the put option can be replicated by a portfolio which includes a long position in a zero coupon bond with face value $K$, a long position in a European call option and a short position in a share of the underlying stock.*

The in-out parity, which states a relation between the in and out barrier options, is given in the following:

**Proposition 2.5.** *The summation of the payoff functions of up-and-out call and up-and-in call gives the payoff function of plain vanilla call option as folows:*

$$V_C = V_{C_{uo}} + V_{C_{ui}}. \tag{2.32}$$

*Hence, the price of up-and-out call at time $0$ satisfies the following equation,*

$$C_{uo} = C - C_{ui}. \tag{2.33}$$

*Similar relation holds for down options:*

$$C_{do} = C - C_{di}. \tag{2.34}$$

The closed form solutions obtained by the Black-Scholes formula can be extended for barrier options [26].

1. For down options, $B < S_0$:

(a) i. A down-and-out call with strike $K$ and barrier $B$ :

$$\begin{aligned} C_{do} = S_0 N(x_1) - Ke^{-rT} N(x_1 - \sigma\sqrt{T}) - S_0 (B/S_0)^{2\lambda} N(y1) \\ + Ke^{-rT} (B/S_0)^{2\lambda - 2} N(y_1 - \sigma\sqrt{T}), \end{aligned} \tag{2.35}$$

where

$$x_1 = \frac{\ln(S_0/B)}{\sigma\sqrt{T}} + \lambda\sigma\sqrt{T}, \tag{2.36}$$

$$y_1 = \frac{\ln(B/S_0)}{\sigma\sqrt{T}} + \lambda\sigma\sqrt{T}, \tag{2.37}$$

$$\lambda = \frac{r + \sigma^2/2}{\sigma^2}. \tag{2.38}$$

ii. A down-and-in call with strike $K$ and barrier $B$ :

$$C_{di} = S_0(B/S_0)^{2\lambda}N(y) - Ke^{-rT}(B/S_0)^{2\lambda-2}N(y-\sigma\sqrt{T}), \tag{2.39}$$

where

$$\lambda = \frac{r+\sigma^2/2}{\sigma^2}, \tag{2.40}$$

$$y = \frac{\ln[B^2/(S_0K)]}{\sigma\sqrt{T}} + \lambda\sigma\sqrt{T}. \tag{2.41}$$

(b) i. A down-and-out put with strike $K$ and barrier $B$:

$$\begin{aligned} P_{do} = Ke^{-rt}N(-d2) - S_0N(-d1) + S_0N(-x_1) \\ -Ke^{-rt}N(-x_1+\sigma\sqrt{T}) - S_0(B/S_0)^{2\lambda}[N(y)-N(y_1)] \\ +Ke^{-rT}(B/S_0)^{2\lambda-2}[N(y-\sigma\sqrt{T}) - N(y_1-\sigma\sqrt{T})]. \end{aligned} \tag{2.42}$$

ii. A down-and-in put with strike $K$ and barrier $B$:

$$\begin{aligned} P_{di} = -S_0N(-x_1) + Ke^{-rt}N(-x_1+\sigma\sqrt{T}) \\ +S_0(B/S_0)^{2\lambda}[N(y)-N(y_1)] \\ -Ke^{-rT}(B/S_0)^{2\lambda-2}[N(y-\sigma\sqrt{T}) - N(y_1-\sigma\sqrt{T})]. \end{aligned} \tag{2.43}$$

By using in-out parity it can be written as:

$$P_{di} = P - P_{do}. \tag{2.44}$$

2. For the up-options, $B > S_0$:

(a) i. An up-and-out call with strike $K$ and barrier $B$:

$$\begin{aligned} C_{ui} = S_0N(d_1) - Ke{-}rTN(d_2) - S_0N(x_1) \\ +Ke^{-rT}N(x_1-\sigma\sqrt{T}) + S_0(B/S_0)^{2\lambda}[N(-y)-N(-y_1)] \\ -Ke^{-rT}(B/S_0)^{2\lambda-2}[N(-y+\sqrt{T}) - N(-y_1)+\sqrt{T}]. \end{aligned} \tag{2.45}$$

ii. An up-and-in call with strike $K$ and barrier $B$:

$$\begin{aligned} C_{ui} = S_0N(x_1) - Ke^{-rT}N(x_1-\sigma\sqrt{T}) \\ -S_0(B/S_0)^{2\lambda}[N(-y)-N(-y_1)] \\ +Ke^{-rT}(B/S_0)^{2\lambda-2}[N(-y+\sqrt{T}) - N(-y_1)+\sqrt{T}]. \end{aligned} \tag{2.46}$$

By using in-out parity it can be stated as:

$$C_{ui} = C - C_{uo}. \tag{2.47}$$

(b) i. An up-and-out put with strike $K$ and barrier $B$:

$$\begin{aligned} P_{uo} = -S_0 N(-x_1) + Ke^{-rT} N(x_1 + \sigma\sqrt{T}) \\ +S_0 (B/S_0)^{2\lambda} N(-y_1) \\ -Ke^{rT} (B/S_0)^{2\lambda-2} N(-y_1 + \sigma\sqrt{T}). \end{aligned} \tag{2.48}$$

ii. An up-and-in put with strike $K$ and barrier $B$:

$$\begin{aligned} P_{ui} = Ke^{-rT} N(-d_2) - S_0 N(-d_1) + S_0 N(-x_1) - \\ Ke^{-rT} N(x_1 + \sigma\sqrt{T}) - S_0 (B/S_0)^{2\lambda} N(-y_1) \\ +Ke^{rT} (B/S_0)^{2\lambda-2} N(-y_1 + \sigma\sqrt{T}). \end{aligned} \tag{2.49}$$

By using in-out parity it can be represented as follows:

$$P_{ui} = P - P_{uo}. \tag{2.50}$$

### 2.3.3 Pure Jump Processes

A pure jump process begins at zero, has finitely many jumps in each finite time interval, and the process is constant between jump times [47]. Poisson, compound Poisson, compensated Poisson, compensated compound Poisson, Variance-Gamma (VG), Normal Inverse Gaussian (NIG) processes are in the class of pure jump processes. In this thesis, only the Poisson family is considered.

#### 2.3.3.1 Poisson Process

Poisson process is one of the core concepts in finance and actuarial sciences. Poisson process is a pure jump process which is also called counting process and point process [22]. The interarrival times or waiting times for this counting process are distributed by exponential distribution with a constant $\lambda$ for homogeneous case. It is used as an elementary unit for other pure jump processes, namely, compound Poisson, compensated Poisson and compensated compound Poisson processes.

**Definition 2.10.** [47] A continuous random variable $\tau$ has the exponential distribution with parameter $\lambda > 0$ if it has a probability distribution function of the form

$$f(t) = \begin{cases} \lambda e^{-\lambda t}, & t \geq 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.51}$$

The cumulative distribution function of $\tau$ is

$$F(t) = 1 - e^{-\lambda t} \quad t \geq 0. \tag{2.52}$$

The expected value of $\tau$ is

$$\mathbb{E}_\tau = \frac{1}{\lambda}. \tag{2.53}$$

**Definition 2.11.** [5] A Poison process $N(t), t \geq 0$ is a counting process which satisfies the following properties:

1. $N(0) = 0$,
2. The process has stationary and independent increments,
3. $\mathbb{P}(N(t) = n) = e^{-\lambda t}\frac{\lambda t^n}{n!}, \quad n = 0, 1, 2, \ldots.$

The mean of a Poisson process is

$$\mathbb{E}(N_t = k) = \sum_{k=0}^{\infty} k e^{-\lambda t}\frac{(\lambda t)^k}{k!} = \lambda t. \tag{2.54}$$

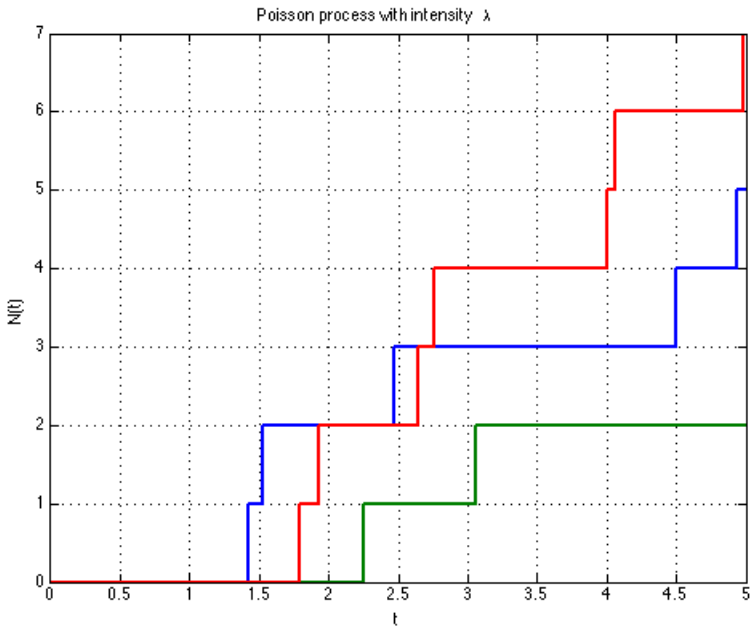



Figure 2.5: Paths of Poisson process with intensity $\lambda = 1$

In Figure 2.5 a number of paths of a Poisson process satisfying the statements given in Definition 2.11 are given.

#### 2.3.3.2 Compound Poisson Process

In financial models, random jump units are necessary in order to capture the randomness in the market. Therefore, different than Poisson process with one unit jump size, the compound Poisson process is highly important for modeling in finance.

**Definition 2.12.** [49] A compound Poisson process $Q(t)$ is a stochastic process with the following representation:

$$Q(t) = \sum_{i=1}^{N_t} Y_i, \tag{2.55}$$

where $N_t$ is a Poisson process with intensity $\lambda$ and $Y_i$'s are i.i.d. with the same distribution and independent from $N_t$. The mean of the compound Poisson process is

$$\mathbb{E}(Q_t) = \sum_{k=0}^{\infty} \mathbb{E}\Big[\sum_{i=1}^{k} Y_i \Big| N(t) = k\Big] \mathbb{P}\{N(t) = k\} = \sum_{k=0}^{\infty} \beta k \frac{(\lambda t)^k}{k!} e^{-\lambda t} = \beta \lambda t. \tag{2.56}$$

Three paths of compound Poisson process are given in the Figure 2.6 as an example.

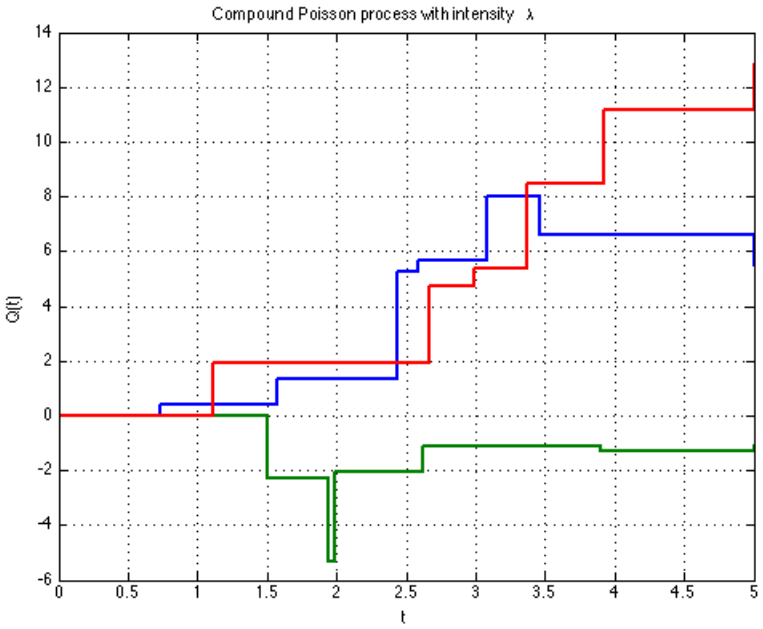



Figure 2.6: Paths of compound Poisson process with intensity $\lambda = 1,\ Y_i \sim N(1, 2)$

#### 2.3.3.3 Compensated Poisson and Compensated Compound Poisson Processes

The compensated versions of Poisson and compound Poisson processes are commonly used in modeling as they provide the martingale property. They are simply obtained by subtracting the mean of the process. In this part, the martingale is initially defined and the definitions of compensated Poisson and compensated compound Poisson processes will follow.

**Definition 2.13.** [1] A stochastic process $X$ is called an $(\mathcal{F}_t)$ martingale if the following conditions are satisfied:

1. $X$ is adapted to the filtration $\{\mathcal{F}_t\}_{t\geq 0}$,

2. $\mathbb{E}[\mid X(t) \mid] < \infty$, for all $t$,

3. $\mathbb{E}[X(t) \mid \mathcal{F}_s] = X(s)$, for all $s \leq t$.

**Definition 2.14.** [47] Compensated Poisson process is defined as follows:

$$M(t) = N(t) - \lambda t,$$

where $N(t)$ is a Poisson process with intensity $\lambda$ and $M(t)$ is a martingale.

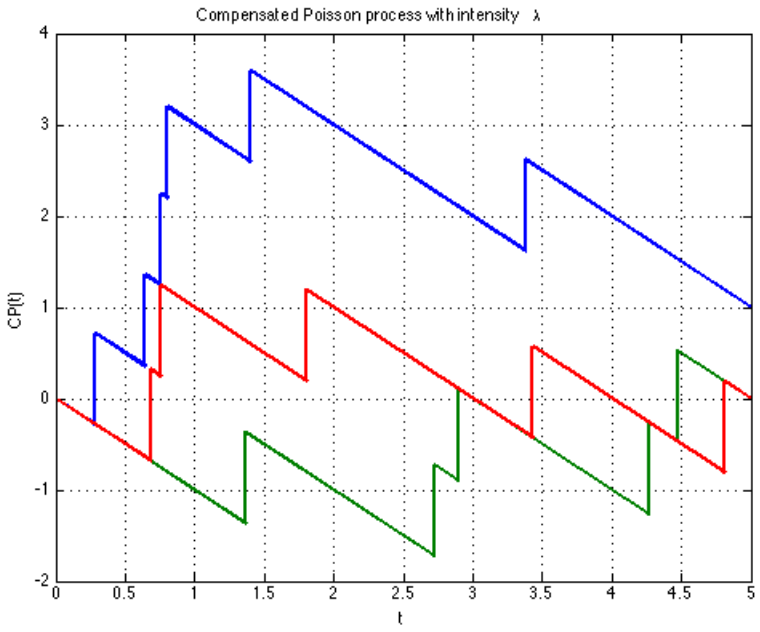



Figure 2.7: Paths of compensated Poisson process with intensity $\lambda = 1$

In Figure 2.7, a number of paths of a compensated Poisson process satisfying the statements in Definition 2.14 are given.

**Definition 2.15.** [47] Compensated compound Poisson process $G(t)$ is a martingale with the following definition:

$$G(t) = Q(t) - \beta\lambda t,$$

where $Q(t)$ is a compound Poisson process with the mean $\beta\lambda t$.

In Figure 2.8, a number of compensated compound Poisson processes satisfying the statements in Definition 2.15 are given.

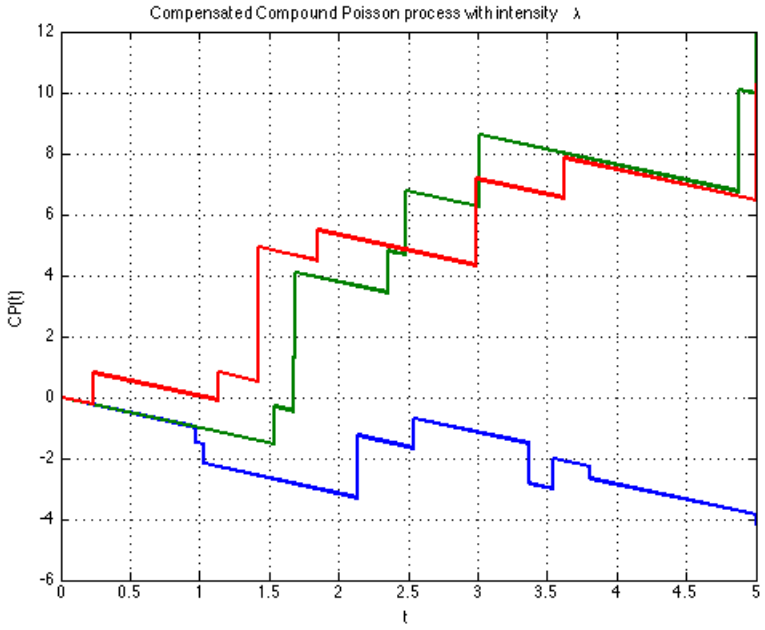



Figure 2.8: Paths of compensated compound Poisson process with intensity $\lambda = 1$, $Y_i \sim N(1, 2)$

### 2.3.4 Jump Diffusion Models

Empirical observations of asset returns show excess kurtosis, also known as fat or heavy tails [4]. Jump diffusion processes are able to capture the heavy tail characteristics of asset returns with encapsuling the jump component to the diffusion model.These processes have been used to capture discontinuous behavior in asset returns caused by extreme events such as financial crisis [32]. Jump diffusion models are the subclasses of Lévy models. They have finite number of jumps. These models can also be considered as the general forms of stochastic differential equations.

**Definition 2.16.** [22] Jump-diffusion stochastic differential equations can be written in differential form with initial condition as

$$dX(t) = f(X(t), t)dt + g(X(t), t)dW(t) + h(X(t), t)dN(t),\ X(0) = x_0, \quad (2.57)$$

or in the integral form as

$$\begin{aligned} X(t) = x_0 + \int_0^t f(X(t), t)ds + \int_0^t g(X(t), t)dW(s) \\ + \int_0^t h(X(t), t)dN(s), \end{aligned} \quad (2.58)$$

where $N(t)$ is a Poisson process and $W(t)$ is a Brownian motion.

**Theorem 2.6.** *[49] Let $X$ be defined as summation of a drift term, a Brownian*

*stochastic integral and a compound Poisson process:*

$$X_t = X_0 + \int_0^t a_s ds + \int_0^t \sigma_s dW_s + \sum_{i=1}^{N_t} \Delta X_i,$$

*where $a_t$ and $\sigma_t$ are continuous, non anticipating processes with*

$$\mathbb{E}\left[\int_0^T \sigma_t^2 dt\right] < \infty.$$

*For any $C^{1,2}$ function $f : [0,T]\times\mathbb{R} \to \mathbb{R}$, the process $Y_t = f(t, X_t)$ can be represented as*

$$\begin{aligned} f(t,X_t) - f(0,X_0) = &\int_0^t \left[\frac{\partial f}{\partial s}(s,X_s) + \frac{\partial f}{\partial s}(s,X_s)a_s\right]ds + \frac{1}{2}\int_0^t \sigma_s^2 \frac{\partial^2 f}{\partial x^2}(s,X_s)ds \\ &+ \int_0^t \frac{\partial f}{\partial x}(s,X_s)\sigma_s dW_s + \sum_{\{i\geq 1, T_i \leq t\}} [f(X_{T_i^-} + \Delta X_i) - f(X_{T_i^-})]. \end{aligned} \tag{2.59}$$

*In the differential notation, it can be represented as*

$$\begin{aligned} dY_t = &\frac{\partial f}{\partial t}(t,X_t)dt + a_t \frac{\partial f}{\partial x}(t,X_t)dt + \frac{\sigma_t^2}{2}\frac{\partial^2 f}{\partial x^2}(t,X_t)dt \\ &+ \frac{\partial f}{\partial x}(t,X_t)\sigma_t dW_t + [f(X_{t^-} + \Delta X_t) - f(X_{t^-})]. \end{aligned} \tag{2.60}$$

Most frequently used jump diffusion models in literature are Merton jump model and Kou's jump model. These models differ from each other by assumption of the distribution of the jump sizes. The distribution of jump sizes are assumed to be log-normal in Merton's model and Kou's model uses double exponentially distributed jump sizes. In this thesis Merton jump model given in Lemma 2.7 is studied.

**Lemma 2.7.** *[30] Merton jump diffusion model represents the underlying differential equation in the form of*

$$dS(t) = S(t^-)[(\mu - \lambda\kappa)dt + \sigma dW(t) + (Y(t) - 1)dN(t)], \tag{2.61}$$

*where $\mu$ and $\sigma$ represent the drift and diffusion terms, respectively; $\lambda$ is the intensity parameter of the Possion process, $N(t)$ and $Y$ shows the independent and log-normally distributed jump variables with $\mathbb{E}[Y(t) - 1] = \kappa$.*

*The solution of Eq. 2.61 is obtained by applying Itô formula for jump diffusion processes given in Theorem 2.6 and it is explicitly given by*

$$S(t) = S_0 e^{(\mu - \lambda\kappa - \frac{1}{2}\sigma^2)t + \sigma W(t)} \prod_{i=1}^{N(t)} Y(t_i). \tag{2.62}$$

The simulation of paths of a Merton jump diffusion process with $\mu = 0.1$, $\sigma = 0.2$, $\lambda = 3$, and the jump size distribution as $LN(0.5, 0.2)$ are given in Figure 2.9.

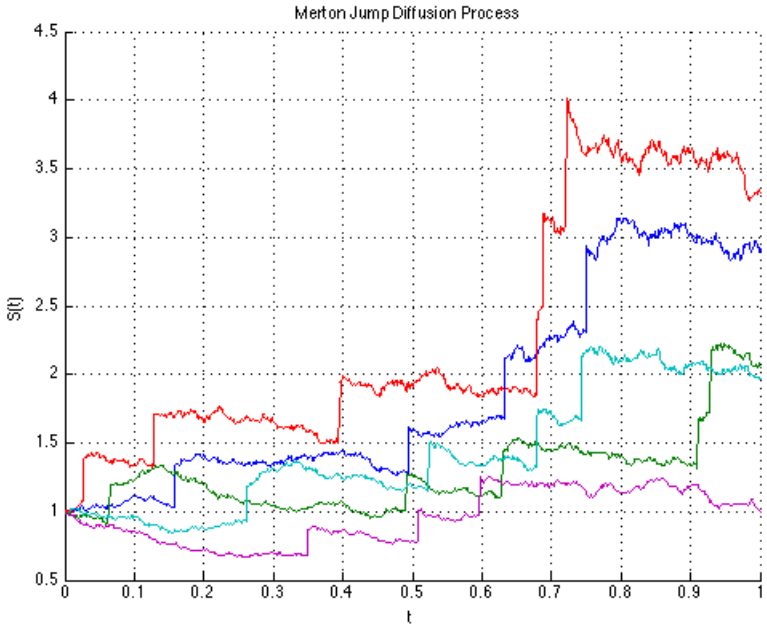



Figure 2.9: Paths of Merton jump diffusion process

## 2.4 Monte Carlo Approach

In literature, there are numerous ways to price options, namely, martingale [23, 24], PDE [2], and simulation-based [6, 8] approaches. In this thesis, Monte Carlo simulation, which is one of the mostly used simulation-based approach, is preferred due to its flexibility and tractability. In this section, the Crude Monte Carlo method is examined generally and then its implementation for pricing European type options is given in the second part of the section.

### 2.4.1 Crude Monte Carlo Approach

The Monte Carlo method is based on the idea of computing an expectation by using the strong law of large numbers which states that the average of a large number of i.i.d. random variables converges to their expected value almost surely. Its mathematical representation is given in the following theorem.

**Theorem 2.8.** *[30, 51] Let $(X_n)_{\{n\in\mathbb{N}\}}$ be a sequence of integrable, real-valued, i.i.d. random variables which are defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Suppose that the mean exists:*

$$\mu = \mathbb{E}(X_1).$$

*Then, we have for $\mathbb{P}$-almost all $\omega \in \Omega$*

$$\frac{1}{n}\sum_{i=1}^{n} X_i(w) \xrightarrow{n\to\infty} \mu.$$

An algorithm to compute this expectation is given in Algorithm 1, where we assume that $X$ is a real-valued random variable with a finite expectation $\mathbb{E}(X)$ [30].

**Algorithm 1** The Crude Monte Carlo method

Approximate $\mathbb{E}[X]$ by the arithmetic mean, $\frac{1}{N}\sum_{i=1}^{N} X_i(\omega)$ for some $N \in \mathbb{N}$.

Here, the $X_i(\omega)$ are the results of $N$ independent experiments that have the same probability distribution as $X$.

The central limit theorem is used to explain error estimation in Monte Carlo framework, and it is given in the following.

**Theorem 2.9.** *[30, 51] Let $(X_n)_{\{n\in\mathbb{N}\}}$ be a sequence of integrable, real-valued, i.i.d. random variables which are defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ with finite mean $\mu$ and finite variance $\sigma^2 > 0$. Then, the normalized and centralized sum of these random variables converges in distribution towards the standard normal distribution:*

$$\frac{\sum_{i=1}^{N} X_i - N_\mu}{\sqrt{N}\sigma} \xrightarrow{D} \mathcal{N}(0,1) \quad as \quad N \to \infty.$$

### 2.4.2 Implementation of Monte Carlo Approach to European Option Pricing

The no-arbitrage theory allows to find option prices by calculating the expectation of discounted payoffs with respect to the risk-neutral measure by setting $\mu = r$ in the corresponding stochastic differential equation [15, 51].

In order to find the approximate value of the option, a large number of paths of the underlying assets are first simulated and the terminal payoff of the option on each path is calculated. Expectation of the terminal payoff is then obtained by averaging the payoff values. Next, it is discounted at risk-free rate to find the approximate value of the option. The mathematical representation of this process is given as

$$\hat{V} = e^{-rT}\hat{\mathbb{E}}(V) \approx V(S_0, 0) = e^{-rT}\mathbb{E}_{\mathbb{Q}}[V(S_T, T)], \tag{2.63}$$

where $\hat{V}$ is the approximate value of the option, $e^{-rT}$ is the discount factor, $V(S_0, 0)$ is the value of the option at $t = 0$ and $\mathbb{Q}$ represents the risk-neutral measure.

The law of large numbers guarantees that the approximate value converges to the correct value as number of simulated paths increases, which can be shown as

$$\mathbb{P}(\hat{V} \to V) = 1 \quad \text{as } N \to \infty. \tag{2.64}$$

For a finite and sufficiently large $N$, the range of $\hat{V}$ with its confidence interval is given as follows:

$$\hat{V} \pm z_{\frac{1-\alpha}{2}} \frac{1}{\sqrt{N}} \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(V_i - \hat{V}_i)^2}, \tag{2.65}$$

where $z_{\frac{1-\alpha}{2}}$ states the standard normally distributed random variable for the $(1-\alpha)$th level of confidence and $N$ is the number of replication.

#### 2.4.2.1 Numerical Solution

Finding an explicit solution for some stochastic differential equations is not possible. In order to approximate the solution of stochastic differential equations, numerical methods are used. Numerical solutions are also used to get some possible sample paths, which form the basis for Monte Carlo approach. This sample paths are also called as scenarios [37].

There are a number of approaches to find numerical approximations for stochastic differential equations. The most commonly preferred ones are Euler-Maruyama, Milstein, and possibly, Runge-Kutta methods. In the scope of this thesis, Euler-Maruyama discretization is investigated.

In this approach, the approximate solution of the general form of the SDE given in Eq. 2.14 turns out to be

$$X_{t_{j+1}} = X_{t_j} + a(t_j, X_{t_j})\Delta t_j + \sigma(t_j, X_{t_j})\Delta W_{t_j}. \tag{2.66}$$

The approximate solution of general jump-diffusion process given in Eq. 2.57 is obtained by taking the increments $dW_t$, $dt$ and $dJ_t$ terms as follows:

$$X_{t_{j+1}} = X_{t_j} + f(t_j, X_{t_j})\Delta t_j + f(t_j, X_{t_j})\Delta W_{t_j} + h(t_j, X_{t_j})\Delta J_{t_j}. \tag{2.67}$$

The increments of $\Delta t_j$, $\Delta W_{t_j}$ and $\Delta J_{t_j}$ over the interval $[0, T]$ are defined by

$$\Delta t_j = t_{j+1} - t_j, \tag{2.68}$$

$$\Delta W_{t_j} = W_{t_{j+1}} - W_{tj} \quad \text{and} \tag{2.69}$$

$$\Delta J_{t_j} = J_{t_j} - J_{t_{j^-}}. \tag{2.70}$$

When there is no jump $\Delta J_{t_j} = 0$ [30].

The algorithm of Euler-Maruyama scheme for the general form of SDEs as in Eq. 2.14 is given in Algorithm 2. The algorithm of Euler-Maruyama scheme for general form of the jump diffusion processes given in Eq. 2.57 is defined in Algorithm 3 [30].

To illustrate, for geometric Brownian motion model, it is possible to simulate paths by using closed-form solution, as well as by Euler-Maruyama discretization. The closed-form solution of GBM given in Eq. 2.20 is discretized for simulation as follows:

$$S_{t_k} = S_{t_{k-1}} \exp\left(\left(r - \frac{\sigma^2}{2}\right)\Delta_k + \sigma\sqrt{\Delta k}Z\right),\ Z \sim N(0,1). \tag{2.71}$$

The simulation result of a path of GBM by using closed-form solution and Euler-Maruyama discretization is given in Figure 2.10 part (a) and the discretization error is presented in part (b).

**Algorithm 2** The Euler-Maruyama Scheme for jump-diffusion processes

Let $\Delta t := T/N$ for a given N.

1. Set $Y_0 = X_0 = x_0$.
2. For $j = 0$ to $N - 1$ do
   a) Simulate a standard normally distributed random number $Z_j$.
   d) Set $\Delta W_j = \sqrt{\Delta t} Z_j$ and

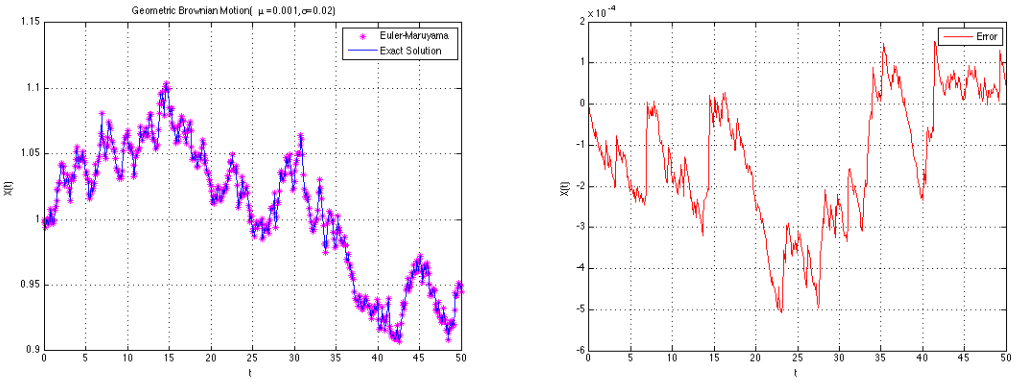
$$Y_{(j+1)} = Y_j + a(j, Y_j)\Delta t + \sigma(j, Y_j)\Delta W_j.$$

**Algorithm 3** The Euler-Maruyama Scheme

Let $\Delta t := T/N$ for a given N.

1. Set $Y_0 = X_0 = x_0$.
2. For $j = 0$ to $N - 1$ do
   a) Simulate a standard normally distributed random number $Z_j$.
   b) Simulate a random variable $N_j \sim \text{Poisson}(\lambda \Delta t)$.
   c) Simulate a random variable $\Lambda_J$ from the given distribution for jump sizes.
   d) Set $\Delta W_j = \sqrt{\Delta t} Z_j$ and

$$Y_{(j+1)} = Y_j + f(j, Y_j)\Delta t + g(j, Y_j)\Delta W_j + h(j, Y_j) N_j \Lambda_J.$$




(a) Closed-form and Euler-Maruyama simulations

(b) Discretization error

Figure 2.10: Simulation results of GBM

# CHAPTER 3

# OBJECT-ORIENTED PROGRAMMING

The required background information in order to understand the constructed software in this study is explained in this chapter. First, the object-oriented programming (OOP) paradigm is introduced with its main terminology and concepts. The terminology and concepts of OOP are highly related with each other. To represent these main units and relations, the unified modeling language (UML) is used as a standard language. The details of this language is illustrated by some basic examples.

## 3.1 What is the OOP?

A programming paradigm can generally be regarded as an approach, style or way of programming. Each programming language performs one or more than one paradigm and each paradigm includes a set of concepts [52]. This close relation is explained in Figure 3.1.

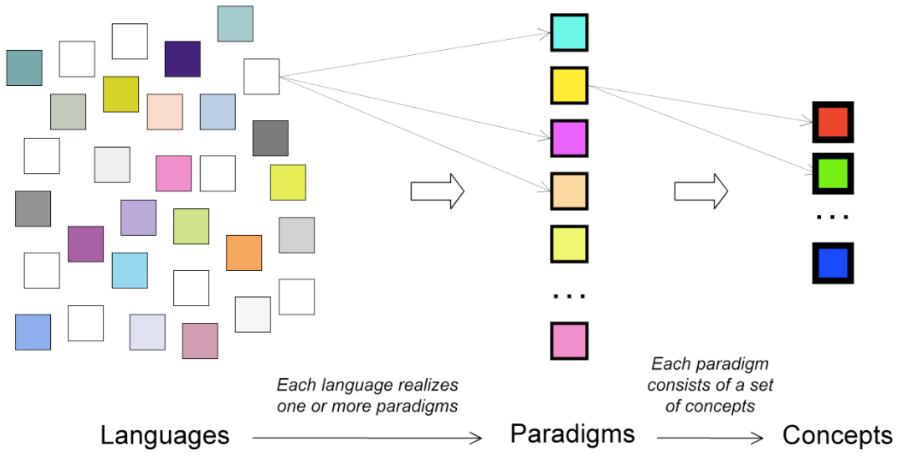



Figure 3.1: Languages, paradigms and concepts [52]

The spectrum of programming paradigms are quite extensive. A number of major programming paradigms are classified with the kinds of abstraction that they employ

and they are shown in Table 3.1 [19]:

Table 3.1: Classification of programming paradigms

| | |
|---|---|
| Procedure-oriented | Algorithms |
| Object-oriented | Classes and objects |
| Logic-oriented | Goals,often expressed in a predicate calculus |
| Rule-oriented | If-then rules |
| Constraint-oriented | Invariant relationships |

The most commonly preferred paradigms are procedural and object-oriented programming paradigms. Procedural programming is constructed as a list of instructions which issue a command to the computer step by step. In procedural programming, the data and the functions are created separately and there is no connection between them. On the other hand, OOP is constructed around objects which hold the data and functions together. OOP divides the overall process into basic units. Each of these units are responsible for its own duty and also these units can interact between each other.

There is no superiority of any programming paradigm over others, each has specialized concepts for different kinds of problems. For example, object-oriented programming is suitable while working with a huge number of related data abstraction organized in a hierarchy [52].

Some of the programming languages are given in Table 3.2 according to their provided programming paradigm types. Some programming languages follow only one paradigm whereas the others provide an environment for more than one paradigm and these programming languages are called as hybrid or multiparadigm languages.

Table 3.2: Programming languages according to their supported paradigm

| **Procedural** | **Pure OO** | **Hybrid** |
|---|---|---|
| Fortran | Simula | C++ |
| Pascal | Smalltalk | MATLAB |
| C | Eiffel | Java |
| Modula | Ruby | Objective-C |

### 3.1.1 Class and Object

In OOP, a class is defined as a set of objects which share common behaviors and operations [19]. In order to categorize objects according to their types, hierarchies by using generalization and specialization ideas and classes are used [41]. Each object is an instance of a class [19]. The structure and behavior of similar objects are defined in their common class. The terms instance and object can be used interchangeable [19]. Objects are the instances of classes that include both information (property or sometimes called data) and its operations (methods). For example, financial derivatives

can be considered as a parent class for forwards, futures, options and swaps. Since it shares the common behaviors and operation of their subclasses. This hierarchy relation is illustrated in Figure 3.2. Moreover, the subclasses share the common behaviors and operations of the objects which are created from these subclasses. For instance, from option class many option objects can be created with different maturities, exercise prices or payoff types.

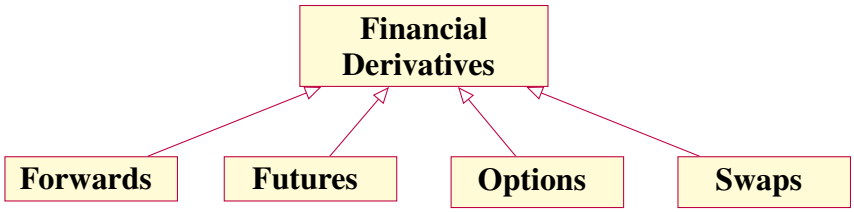



Figure 3.2: Financial derivatives class and its subclasses

## 3.2 Concepts of the OOP

An object-oriented program is expected to satisfy the characteristic features of OOP paradigm, namely, abstraction, encapsulation, inheritance and polymorphism. These concepts have important meanings and specialized functionalities. Also, they are tightly interrelated between each other. These main concepts are explained in detail with examples in the following subsections.

### 3.2.1 Abstraction

The abstraction concept is used to cope with the complex systems. Abstraction allows to see the big picture by selecting the crucial characteristics and ignoring the redundant details. It also provides organization among classes. Abstract classes are written to form a basis for other classes which share the common characteristics. These classes cannot produce an object and their methods are also abstract so that their child classes override the methods which are inherited from an abstract class.

For example, the financial derivative class and its subclasses are illustrated in Figure 3.2. Financial derivatives class is an abstract class as seen in the figure. It is used only for defining the subclasses and there is no instance associated with it. It gives a general idea about its subclasses and it reduces the complexity by defining a hierarchy.

### 3.2.2 Encapsulation

Encapsulation is the concept that properties (data) and methods (operations) of objects are packaged, in other words, capsulated. It is also called as information or data hiding. By this property, the working details of the class is hided from the outside classes so

that outside classes have no permission to access or modify the data or methods of the class. Only the class itself has that permission. This feature provides the coder with a convenient working platform in which any modification can be made privately on the properties and methods of the class. Therefore, the integrity of the code cannot be damaged by the outside classes. If the permission is given to the outside classes, it can only be done by using the so-called set/get methods. Encapsulation reduces the complexity and increases the robustness by limiting the access from outside world [11, 17, 48].

Some programming languages, generally pure object oriented languages like Smalltalk, automatically make the properties and methods private, but for other programming languages, the decision-maker about privateness is the coder [40]. Also, the coder determines which properties and methods are to be set as private or public. The implementation of the encapsulation concept in MATLAB® programming language is illustrated by defining private and public properties as seen in Figure 3.3.

newClass.m

```
classdef newClass
    %   Detailed explanation about newClass

    properties (Access = private)
        property1;
        property2;
    end

    properties (Access = public)
        property3;
        property4;
    end

    methods
        some methods
    end

end
```

Figure 3.3: Encapsulation concept in MATLAB®

### 3.2.3 Inheritance

Inheritance notion simply means that subclasses automatically inherit properties and methods from their superclasses. Superclasses are also known as based classes or parent classes since they are used as a basis for inheritance. Subclasses, which are also named as derived, extended or child classes, are the more specific cases of parent classes with their additional properties and methods. The relation between subclass and superclass is also known as *is-a* relation. It is a one-way relation to indicate that each subclass contains all data and operations of its superclass, but not conversely.

In OOP, subclasses can also have their own subclasses and this feature is known as multilevel inheritance.

Inheritance provides many advantages. First of all, it reduces the development time since the designer does not have to re-create the common properties and methods for similar classes. In addition, the use of inheritance reduces the risk of errors because the designer can utilize the previously developed and already tested properties and methods to create new classes. As a result of these benefits of inheritance, the code becomes reusable, which is one of the most essential benefits of OOP.

Many object oriented design contains multiple inheritance. Multiple inheritance can be defined as the usability of more than one class as a parent class. Although single inheritance is a more straightforward approach, some systems need to combine attributes and operations from different classes [17, 33].

Schematics representing the single, multilevel and multiple inheritance concepts are given in Figure 3.4.

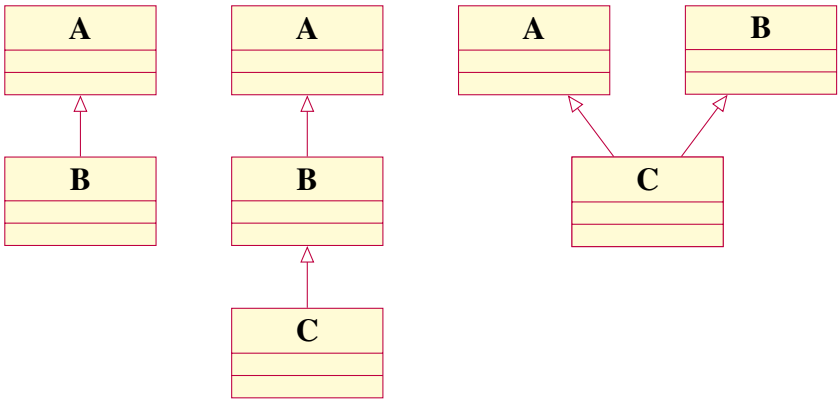



Figure 3.4: Single, multilevel and multiple inheritance

### 3.2.4 Polymorphism

The roots of the word polymorphism, poly (many) and morph (shape), come from the Greek and the word means that "having or passing through many forms". In OOP, more than one method can share the same name although they belong to different objects and have different internal structures [39]. The program differentiates between the objects and applies the related procedure according to the class of the object. Hence, code sharing and reusing are achieved with the help of the polymorphism [48].

Polymorphism concept includes both overloading and overriding concepts. Overloading (ad hoc polymorphism) allows methods to have the same names. However, the methods having the same name may use different parameters. Overriding (inclusion polymorphism) notion means that the name and parameters of the inherited method of the subclass are the same, but the internal structure of the method is different [41, 50].

As a simple example of polymorphism, "+" operator can be considered. In MAT-

LAB®, it is used to sum the integers as follows:

```
>> 3+5

ans =
8
```

Also, it can be defined to sum the payoff functions. For example, Payoff1 and Payoff2 object are created as follows:

```
>> Payoff1 = Payoff(@(x) x^2, 'long',1)

Payoff1 =
Payoff with properties:
payoff: {[@(x)x^2]  'long'  [1]}

>> Payoff2 = Payoff(@(x) x^3, 'long',1)

Payoff2 =
Payoff with properties:
payoff: {[@(x)x^3]  'long'  [1]}
```

Then, Payoff1 and Payoff2 objects are summed by + operator.

```
>> Payoff3 = Payoff1 + Payoff2

Payoff3 =
Payoff with properties:
payoff: {2x3 cell}
```

The main terminologies and concepts of the OOP make it a powerful paradigm and they certainly provide some important advantages. OOP provides code reusability and it is a huge advantage in terms of saving time. In addition, it is suitable for code expansion, since in OOP, the overall structure can be divided into small parts. Thereby, adding or subtracting new classes do not ruin the existing classes. Also, due to this feature, debugging is easy. Classes can be tested independently and the objects which are already tested can be reused. Last but not least, by using OOP, robust softwares are formed. This feature of the OOP makes the software suitable for teamworks.

### 3.3 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is an international standard language for documenting object-oriented systems. The UML was published by the Object Management Group in 1997 [3]. The current version is UML 2.5. The UML diagrams can

be categorized as structure diagrams and behavior diagrams. The general overview of UML diagrams are given in Figure 3.5 [17].

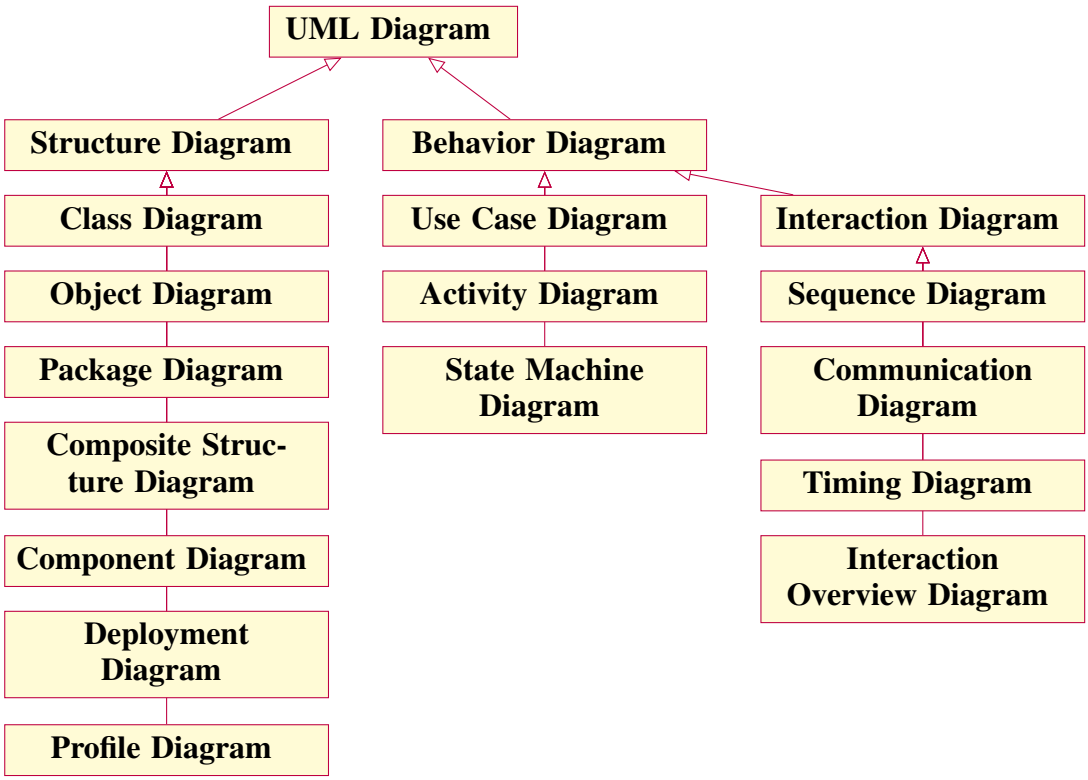



Figure 3.5: UML diagrams overview

In the scope of thesis, the class and object diagrams are used in order to explain the structure of the software. Hence, only the details of class and object diagrams are examined.

In UML, each class is represented by a box. This box composed of three sections. The name of the class, properties and methods are written to the upper, middle and bottom sections, respectively. An illustration for a class is shown in Figure 3.6

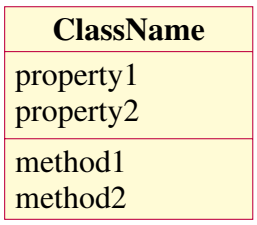



Figure 3.6: An example of UML class diagram

The signs in front of the names of the properties and methods indicate their visibility status as seen in Table 3.3.

In UML, the relationships between classes are expressed by the different notations. Generalization/specialization (is-a) relation is indicated by an arrow and the head of

Table 3.3: Visibility types of UML class diagram

| Symbol | Visibility Type |
|---|---|
| + | public |
| - | private |
| # | protected |

the arrow shows the parent class as illustrated in Figure 3.7, in which the child class inherits from its parent class as indicated by an arrow.


Figure 3.7: Generalization relation

Association relation is used to demonstrate the connection between classes and it is shown by a line between classes. Multiplicity information is written at the both ends of the line. An example is given in Figure 3.8 and it shows that each option can be associated with one or more than one asset. The commonly used multiplicity symbols are given in Table 3.4.

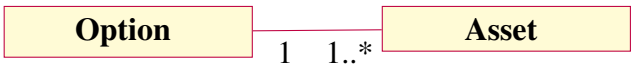



Figure 3.8: Association relation

Table 3.4: Multiplicity types

| | |
|---|---|
| 1 | no more than one |
| 0..1 | zero or one |
| * | any number |
| 1..* | one or more |
| 0..* | zero or more |

In addition to the generalization and association relationships, the aggregation and composition relations are commonly used in UML diagrams. Aggregation is used to show *has-a* relationship, which is also called whole-part relationship between classes. In *whole-part* relationship, one class represents the whole and consists of other classes. Aggregation relation is shown by an arrow with a diamond shape head. In Figure 3.9, this relationship between two classes is illustrated.


Figure 3.9: Aggregation relation

Another whole-part relationship is the composition. It is used to indicate strong form of the association relation. In Figure 3.10, it is given that the whole part composed of

existentially dependent part(s). The existentially dependent parts cannot exist without the whole part in composition relation.

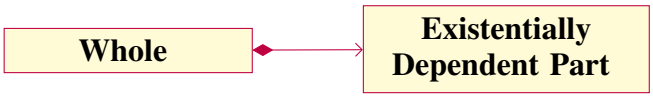



Figure 3.10: Composition relation

Dependency relationship is used to show that one class requires other class for its implementation or operation. It is indicated by a direct dashed line as given in Figure 3.11.


Figure 3.11: Dependency relation

Packages are used to organize a system by grouping its classes. An example for package representation of European options is given in Figure 3.12.

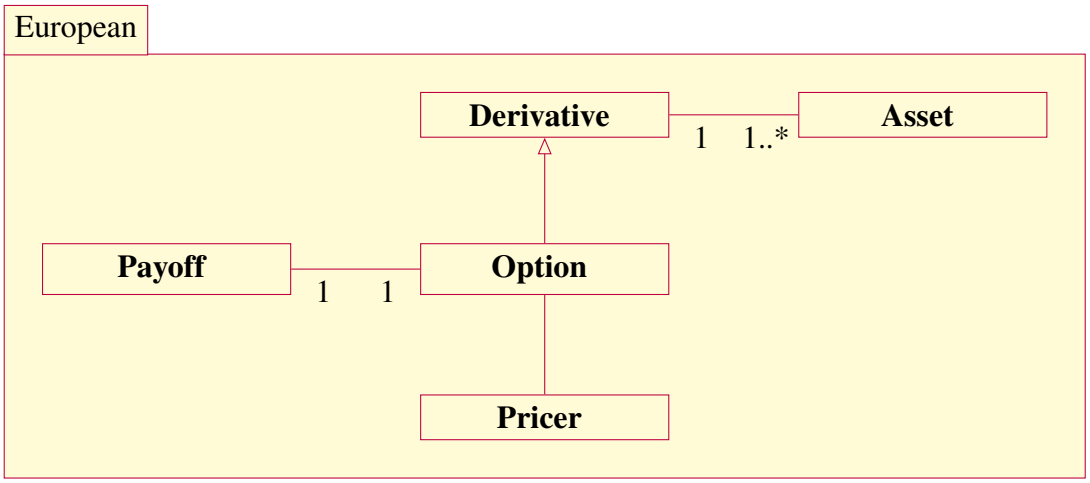



Figure 3.12: Package representation

# CHAPTER 4

# DESIGN AND IMPLEMENTATION OF OOMCOP

In this chapter, the design and implementation of an object-oriented software to price various type of options by using Monte Carlo approach are presented. First, one of the the built-in classes available in the Financial Toolbox™ of MATLAB®, SDE class, is explained with its subclasses, since it is used in the Object-Oriented Monte Carlo Option Pricer (OOMCOP) as one of the main classes. The structure of the software with its main classes and subclasses is given. The properties and methods of these classes are illustrated by using unified modeling language (UML) diagrams in order to show the structure of the design. The relations between classes are investigated to understand the structure of the program. Finally, the design and implementation of a graphical user interface (GUI) by using MATLAB® / GUIDE are explained.

## 4.1 OOP in MATLAB®

MATLAB®, whose name comes from Matrix Laboratory, is one of the mostly used programming languages in academia and industry with its over one million user around the world [35].

MATLAB® provides an environment for both procedural and object-oriented programming paradigms. In addition, it has lots of built-in functions and classes so that there is no need to recreate those functions and classes. This feature of MATLAB® saves time and gives an opportunity to the programmer to only focus on the algorithms that are necessary. For instance, the built-in functions for random number generators can directly be used instead of writing these algorithms from scratch.

MATLAB®'s Financial Toolbox™ includes data preprocessing, financial time series, financial data analytics, portfolio optimization and asset allocation, credit risk, price and analyze financial instruments and stochastic differential equation (SDE) models functionalities. Especially, the built-in stochastic differential equation models in the toolbox provide SDE specification and simulation for different SDE models [35].

MATLAB®'s built-in SDE class which comes with the Financial Toolbox™ is the starting point of this thesis and it is used in the software as a parent class with its properties and methods. The hierarchy of the built-in SDE class and the subclasses is given in

Figure 4.1.

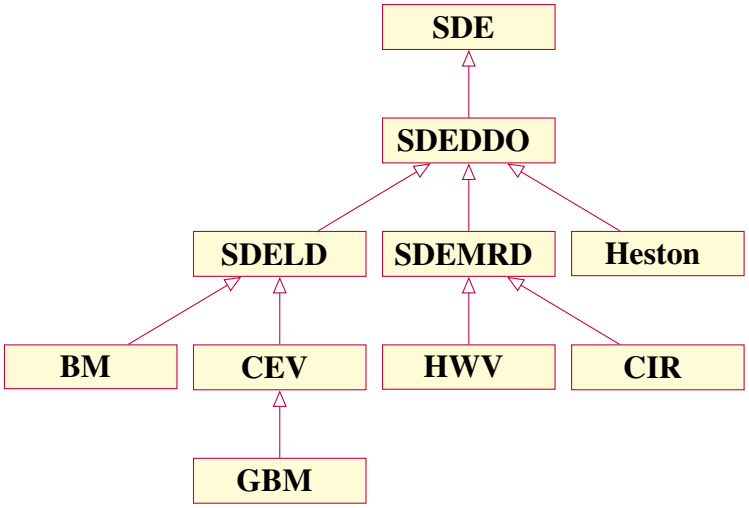



Figure 4.1: MATLAB® SDE class hierarchy

Table 4.1 gives the SDE class and subclasses with their operations. Left column includes the name of the class and the right column indicated the objects the corresponding class creates.

Table 4.1: MATLAB® SDE class and subclasses with their operations

| | |
|---|---|
| sde | sde model object by using user-specified(defined)functions |
| sddedo | sdeddo model object with Drift and Diffusion objects |
| sdeld | sde model from linear drift-rate models |
| sdemrd | sde model from mean-reverting drift-rate models |
| bm | Brownian motion object |
| cev | constant elasticity of variance (CEV) objects |
| cir | Cox-Ingersoll-Ross (CIR) mean-reverting square root diffusion objects |
| gbm | geometric Brownian motion(GBM) objects |
| heston | Heston model objects |
| hwv | HWV model objects |
| drift | drift-rate model components (objects) |
| diffusion | diffusion-rate model components |

The SDE class and its subclasses are used after defining the drift and diffusion components. For instance, consider creating an SDE object which has the following equation:

$$dX(t) = X(t)[0.3dt + 0.1dW(t)]. \tag{4.1}$$

In MATLAB®, the drift and diffusion functions are initially created by using the following commands:

```
F = @(t,X) 0.1 * X; %drift function
G = @(t,X) 0.3 * X; %diffusion function
```

```
A = sde(F,G) %sde object

A =

Class SDE: Stochastic Differential Equation
-------------------------------------------
Dimensions: State = 1, Brownian = 1
-------------------------------------------
StartTime: 0
StartState: 1
Correlation: 1
Drift: drift rate function F(t,X(t))
Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
```

Having created the SDE object, the simulation methods can now be applied easily. By default, simByEuler (simulation by Euler) method is assigned as a Simulation property for every SDE object. The simBySolution (simulation by Solution) method can only be used for GBM and HWV (Hull-White-Vasicek) objects and it should be specified by calling the simBySolution method with required parameters.

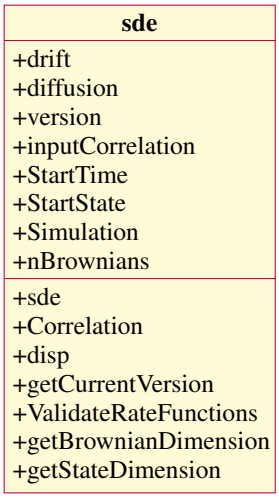



Figure 4.2: UML diagram of sde class with its properties and methods

## 4.2 Structure of the Program

The UML diagram of the OOMCOP is given in Figure A.1. The software is structured in five main parts, namely, Payoff, PureJumpProcess, Asset, Derivative and Pricer. Also, MATLAB®'s SDE class is used as a main class with their subclasses. The details of the newly developed classes are given in the following subsections.

### 4.2.1 Payoff Class

Payoff class is one of the main abstract classes of the structure. The required data for Payoff class is payoff function(s) and it is stored as a payoff property. Payoff function of Payoff class can be determined as any user-specified functions. The position and the multiplier should also be indicated within the payoff function. Position states whether the options are bought or sold. As for, multiplier indicates how many option is bought or sold. In order to create an object from Payoff class, the constructor method is initially applied. Then, other methods can be used by the created payoff object.

In order to create a payoff object, payoff function, position and multiplier should be specified. An example to create a payoff object is given below.

```
>> CallPayoff = Payoff(@(x) max(0, x - K), 'long', 1)

CallPayoff =
Payoff with properties:
payoff: {[@(x)max(0,x-K)]  'long'  [1]}
```

The payoff method, which is the constructor method for Payoff class, is used by setting the payoff function, position and multiplier as `@(x)max(0,x-K)`, `'long'`, `1`, respectively. The above mentioned payoff function is written for call option payoff where $x$ and $K$ represent the stock prices and the strike price, respectively.

Beside Payoff (constructor) method, Payoff class has important methods as plus, minus and times. The "+" operator is redefined for plus method to sum two payoff objects. The minus method uses "−" operator and it reverses the position and multiplier. The times method, which is refined by "∗" operator, multiplies the multiplier by given integer.

In order to show the examples of plus, minus and times methods beside the CallPayoff object, PutPayoff object is created as follows:

```
>> PutPayoff = Payoff(@(x) max(0, K-x), 'long', 1)

PutPayoff =
Payoff with properties:
payoff: {[@(x)max(0,K-x)]  'long'  [1]}
```

The plus, minus and times methods can be applied to CallPayoff and PutPayoff objects as noted below:

```
>> CallPutPayoff = CallPayoff + PutPayoff

CallPutPayoff =
Payoff with properties:
```

```
payoff: {2x3 cell}

>> CallPutPayoff2 = CallPayoff - PutPayoff

CallPutPayoff2 =
Payoff with properties:
payoff: {2x3 cell}

>> CallPayoff3 = 3 * CallPayoff

CallPayoff3 =
Payoff with properties:
payoff: {[@(x)max(0,x-K)]  'long'  [3]}
```

The plots of created payoff objects are given in Figure 4.3 and the test scrip is given in Listing 1.

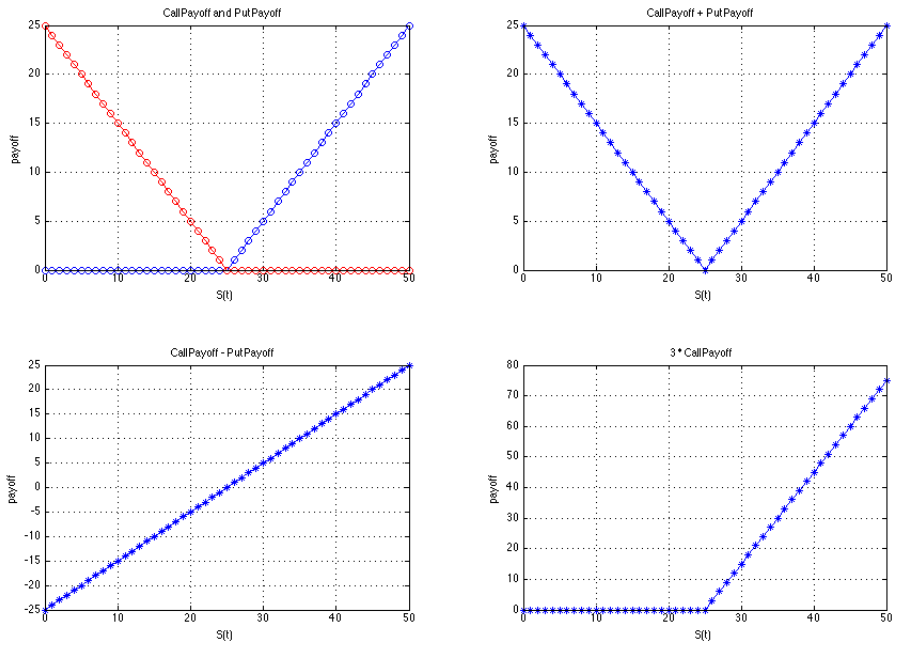



Figure 4.3: Plots of the objects generated from Payoff class

The evaluate method is used to calculate the values of payoff function at given points. Apart from the above examples, the payoff function can be defined as any function. For example, the payoff function can be defined for different alpha values as follows:

```
myF = @(x) thisNewPayoffFunction(x, alpha);
```

where the thisNewPayoffFunction is defined as

thisNewPayoffFunction.m

```
function values = thisNewPayoffFunction(x, alpha)

  values = exp(-alpha.*(abs(x)).^2);

end
```

After creating the payoff object by using myF function, the values of the payoff can be calculated by using evaluate method. The corresponding script file is given in Listing 2 and the output of the script for the values, $x = [-5 : 0.1 : 5]$, is given in Figure 4.4.

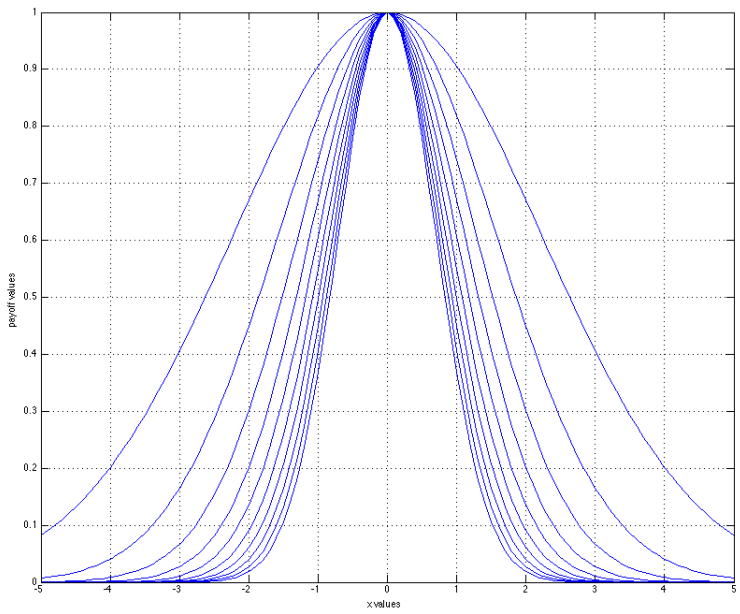



Figure 4.4: Evaluated payoff values for myPay object

The VanillaPayoff class is a specialized version of Payoff class. It has the additional type and strike properties additionally. The European style vanilla options are categorized according to their payoff type as call or put. Strike is required to define the payoff function. After constructing the VanillaPayoff object, other methods can be applied.

Call and put payoff objects can be generated directly from Payoff class as stated in the above examples. Also, more practically they can be generated from VanillaPayoff class as follows:

```
>> myPay = VanillaPayoff(3, 'call')

myPay =
```

```
VanillaPayoff with properties:
type: 'call'
strike: 3
payoff: {[@(x)max(0, x-obj.strike)]  'long'  [1]}

>> myPayPut = VanillaPayoff(3, 'put')

myPayPut =

VanillaPayoff with properties:
type: 'put'
strike: 3
payoff: {[@(x)max(0, obj.strike-x)]  'long'  [1]}
```

The payoff functions of this objects are illustrated in Figure 4.5 and the test script to create this output is given in Listing 3.

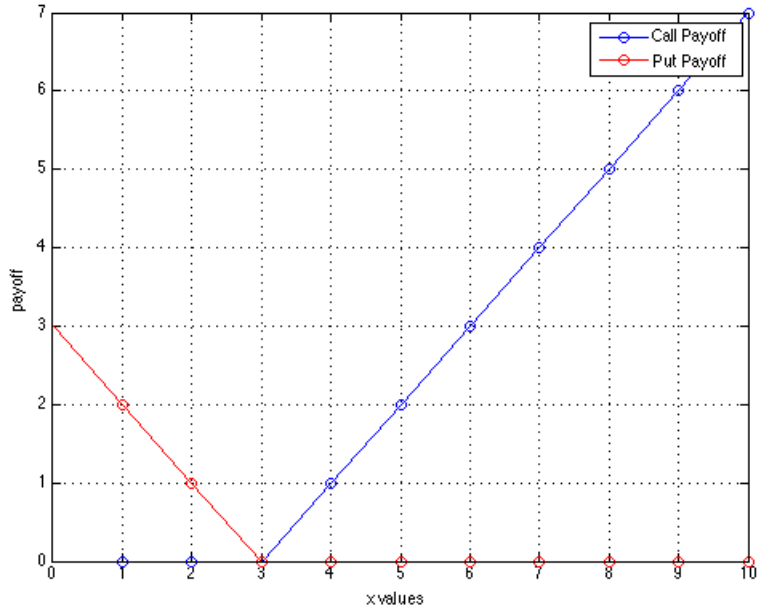



Figure 4.5: Plot of Call Payoff, Put Payoff objects generated from Vanilla Payoff class

A more specialized version of Vanilla Payoff class is the Barrier Payoff class. It has the characteristics of Vanilla Payoff and, in addition, it has name and barrier properties. Barrier property is used to determine the barrier as its name implies; name property is used to specify the type of the barrier option. The type property is inherited from VanillaPayoff class, but it only determines whether it is a call or a put. Furthermore, specific type of the payoff has to be determined: barrier payoff can have eight different type, namely, down-and-out call, down-and-out put, down-and-in call, down-

and-in put, up-and-down call, up-and-down put, up-and-in call, up-and-in put. These eight kinds of barrier payoff can be determined by the collocation of type and name properties. Construction of barrier payoff object is given below.

```
>> BarrierEx = BarrierPayoff(50,'call',60,'down-and-out');

BarrierEx =
BarrierPayoff with properties:
name: 'down-and-out'
barrier: 60
type: 'call'
strike: 50
payoff: {[@(x)max(0,x-obj.strike)]  'long'  [1]}

Type  : call
Strike: 50
```

Strategy is another subclass of Payoff class with its additional property, StrategyName, and its constructor method. The StraddlePayoff, StranglePayoff and ButterflyPayoff classes are subclasses of Strategy class. Although the Strategy class is not an abstract class, it can be thought of as an abstract class for its subclasses in terms of hierarchical structure. The construction of StraddlePayoff, StranglePayoff and ButterflyPayoff objects are given in Listing 4 and the output of the test script is given in Figure 4.6.

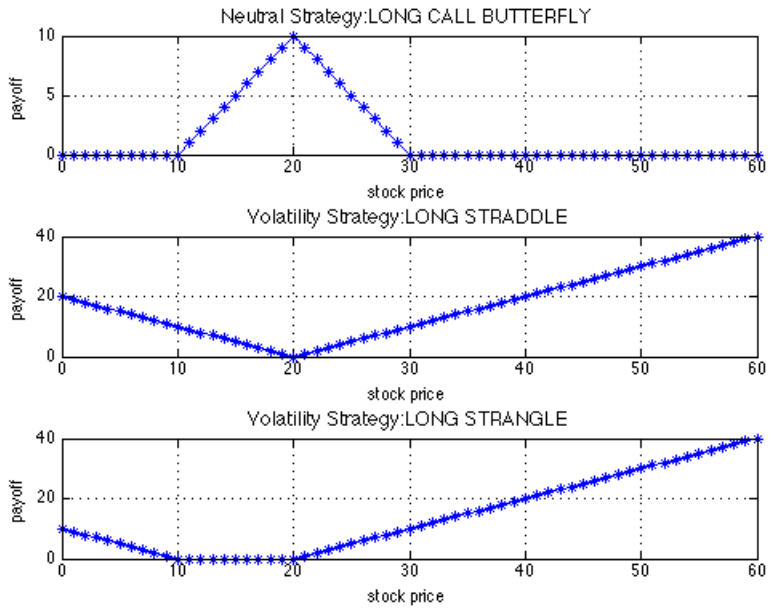



Figure 4.6: Output of the test script for option strategies

The subclasses of the Strategy class are created to provide ready-to-use classes for the

user. This subclasses can further be expanded by adding particular option strategies. Besides, the strategies can be created directly from Payoff class without creating new subclasses. Since the Payoff class allows to use any user defined function as a payoff function.

For example, the iron condor strategy, can be constructed with four strike prices $K_1 < K_2 < K_3 < K_4$ and the following positions [46]:

- Buying a put at $K_1$
- Selling a put at $K_2$
- Selling a call at $K_3$
- Selling a call at $K_4$

This strategy can be formed by using the four payoff object which are created from Payoff class as given in Listing 5. The payoff diagram of the iron object with the strike prices, $K_1 = 55$, $K_2 = 60$, $K_3 = 70$, $K_4 = 75$ for the the values, $x = [45 : 85]$ is illustrated in the following figure.

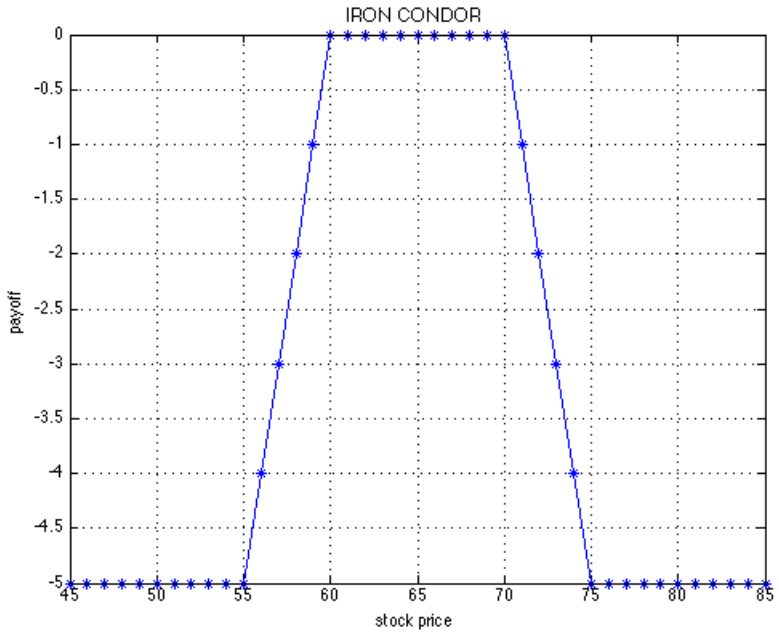



Figure 4.7: Payoff diagram for the iron condor object

It can be seen from the graph that the iron condor strategy limits both the gain and loss. The strategy provides maximum gain for a significant range which is between $x = [60, 70]$ for the given example.

The UML diagram of the Payoff class with its subclasses is given in Figure 4.8.

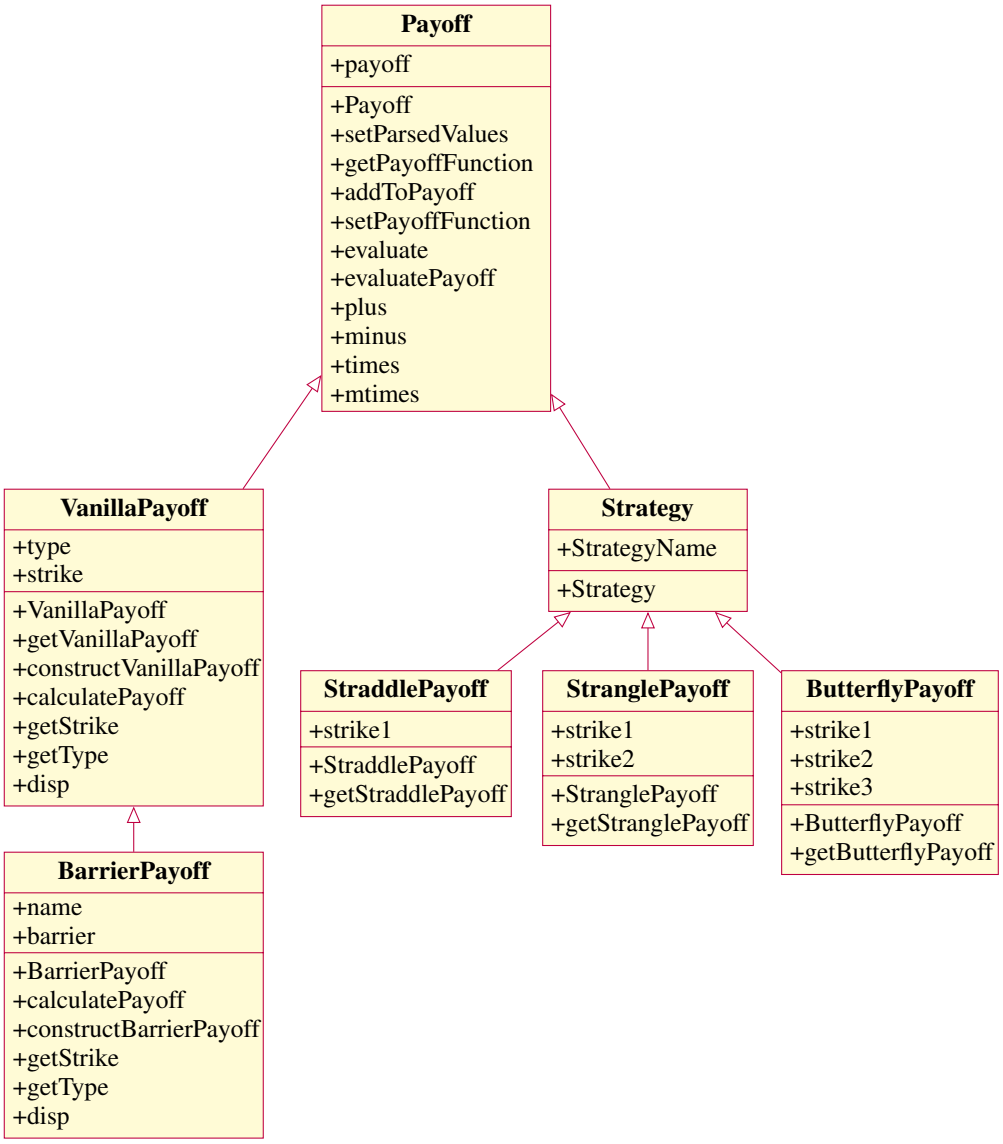



Figure 4.8: UML diagram of Payoff class & generalization relation between its subclasses

### 4.2.2 PureJumpProcess Class

The PureJumpProcess class is created on the purpose of making the software suitable for the assets which follows jump diffusion processes. The PureJumpProcess class is formed as an abstract class. Poisson class is the subclass of PureJumpProcess class and it serves as a main class for this part of the software. It has only lambda ($\lambda$) property. From the Definition 2.11, the knowledge about $\lambda$ is enough in order to construct a Poisson process. The methods of this class are the generatePoissonRandn, simulate and plot. The generatePoissonRandn is used to create random numbers from Poisson distribution with $\lambda$ intensity. The simulate and plot methods are created to simulate and plot Poisson object with $\lambda$ property. The CompensatedP, which refers to compensated

Poisson, and CompoundPoisson classes are the subclasses of Poisson class. These subclasses inherit properties and methods of Poisson class automatically.

The CompensatedP class has one property which comes from its superclass Poisson and it is not in need of additional property according to Definition 2.14. Beside its constructor method, CompensatedP, it has methods named as simulate and plot as its superclass. However, inside operations of simulate and plot methods of CompensatedP class are different from the methods of its superclass. This situation is one of the polymorphism applications of this software.

The additional property of the CompoundPoisson class is its distribution property. In order to define a compound Poisson object, beside lambda property, distribution of random variables, say $Y_i$, are also required. The methods of the CompoundPoisson class, which are the constructor method, CompoundPoisson, simulate and plot methods, are similar to the CompensatedP class. The structure of CompensatedCP is also very similar to CompoundPoisson class.

The code in order to create and plot Poisson, compound Poisson, compensated Poisson and compensated compound Poisson objects is given in Listing 6 and its output is shown in Figure 4.9.

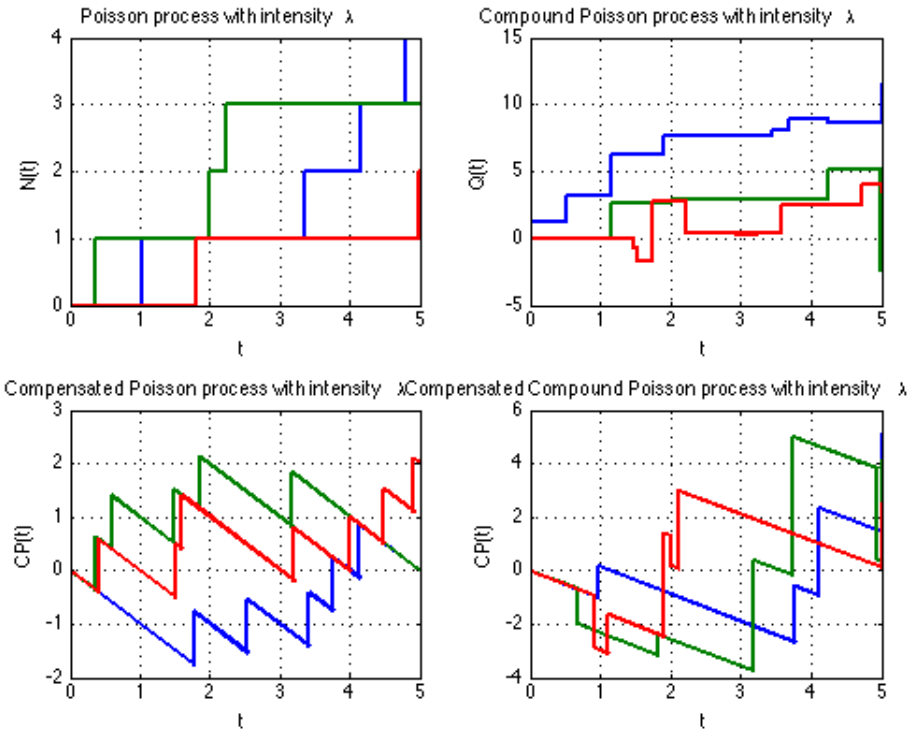



Figure 4.9: Output of the test script of Poisson class and its subclasses

The UML diagram of the PureJumpProcess part of the software is given in Figure 4.10.

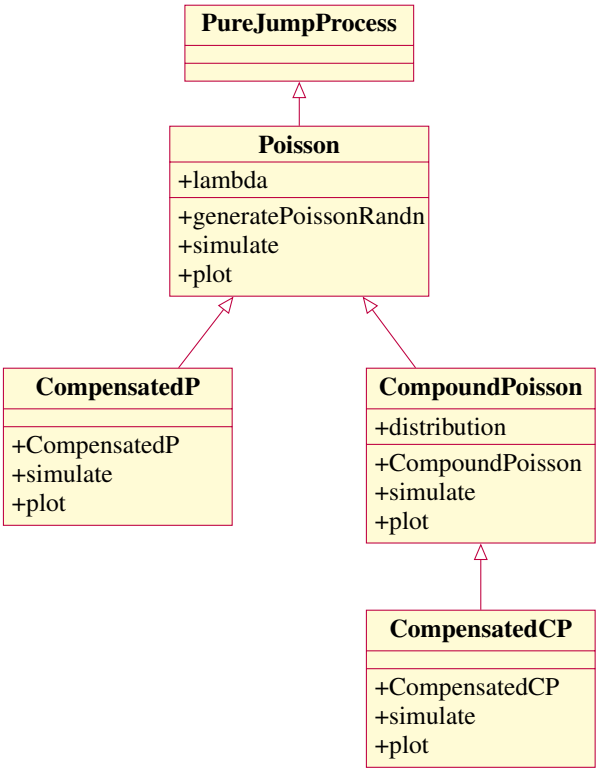



Figure 4.10: UML diagram of PureJumpProcess class with its subclasses

### 4.2.3 Asset Class

Another important main class for the structure is the Asset class with assetName and currentPrice properties. The SDE_Asset, GBM_Asset, JD_Asset, MertonJD_Asset classes are inherited the properties assetName and currentPrice from Asset class. Besides, SDE_Asset and GBM_Asset classes are inherited properties and methods from MATLAB®'s built-in class SDE.

Hence, the SDE class only covers pure diffusion and diffusion with drift processes, JumpDiffusion and MertonJD (Merton jump diffusion) classes are created with the help of the built-in SDE class and newly developed PureJumpProcess class as depicted in Figure 4.11.

The SDE_Asset and GBM_Asset classes are the examples of multiple inheritance concept. This classes inherits from the Asset class as well as the SDE class. In the same manner, JD_Asset and MertonJD_Asset classes take over the properties and methods from both the Asset class and the JumpDiffusion and MertonJD classes, respectively. The SDE_Asset, GBM_Asset, JD_Asset, MertonJD_Asset classes have no additional properties except for their inherited properties. These four subclasses possess the simulatePaths and disp methods, in addition to their constructor methods. For instance, after creating a GBM_Asset object by using display method (disp), the following information about the object, is obtained.

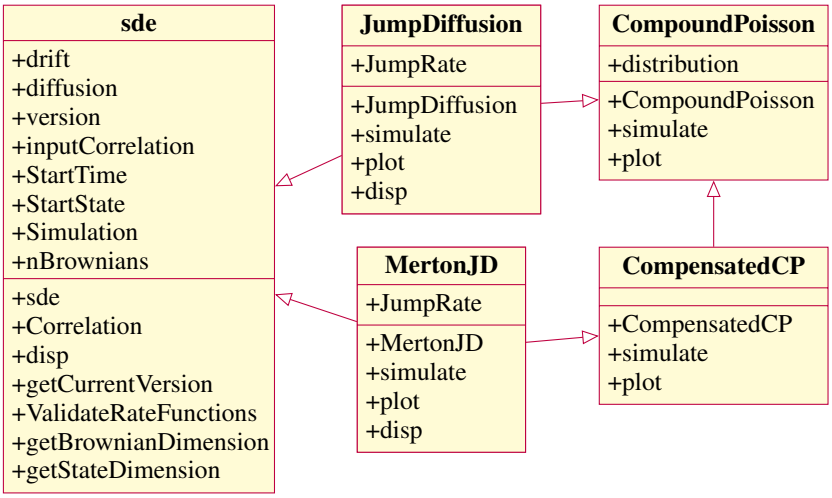



Figure 4.11: UML diagram JumpDiffusion and MertonJD clases

```
F = @(t,X) 0.1 * X; %drift object
G = @(t,X) 0.3 * X; %diffusion object
myAsset = GBM_Asset(F,G); %Asset object
myAsset = myAsset.setAssetName('GBM');
myAsset.disp();

Class GBM_Asset:
-----------------------------------------
GBM :Asset Name   Current Price: 1
-----------------------------------------
StartTime: 0
StartState: 1
Drift: drift rate function F(t,X(t))
Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
-----------------------------------------
```

The simulatePaths method is an example of the overriding (inclusion polymorphism) concept. That is, although the name and parameters of the method are the same, the internal structure of the method works in a different way. For the assets, which follows SDE or GBM process, MATLAB®'s built-in functions, simByEuler (simulation by Euler method) or simBySolution (simulation by solution), are used in order to simulate the asset paths. A plot of several simulated asset price paths, which follow GBM process, using simByEuler method is illustrated in Figure 4.12 and the test is given in Listing 7.

In the scope of this software, the types of the asset prices and the supported simulation methods are given in the Table 4.2. Simulation by Euler method is provided for all types of asset models whereas the simulation by solution method can only be used for geometric Brownian motion and Merton jump diffusion models. The UML diagram of the Asset class and its subclasses is given in Figure 4.13.

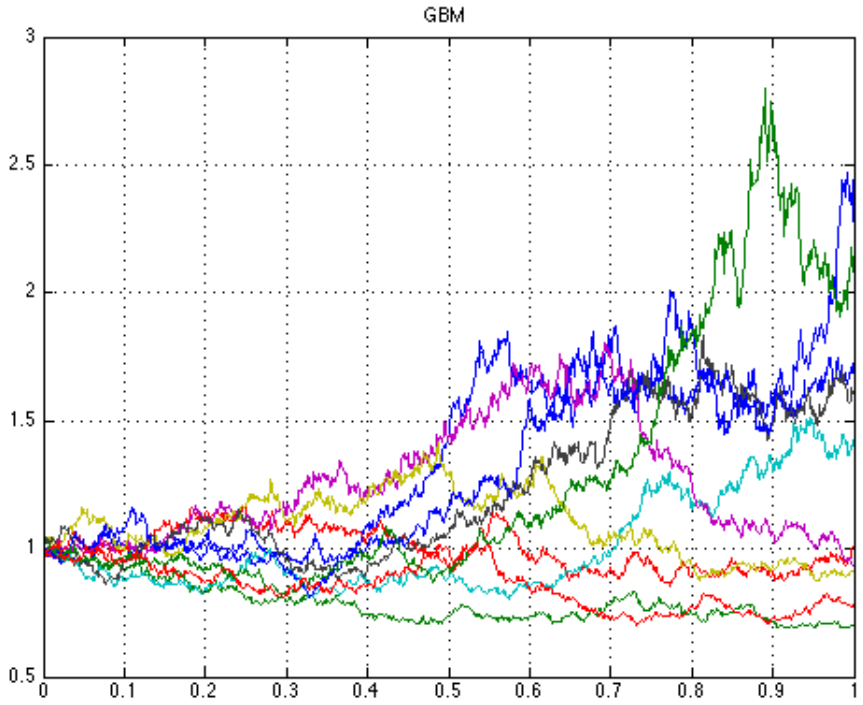



Figure 4.12: Paths of Geometric Brownian Motion

Table 4.2: Supported simulation methods for the types of asset prices

| Type of the asset price | Simulation by Euler | Simulation by Solution |
|---|---|---|
| SDE | ✓ | |
| GBM | ✓ | ✓ |
| JD | ✓ | |
| MertonJD | ✓ | ✓ |

### 4.2.4 Derivative and Option Classes

Derivative class is an abstract class and Option class is the subclass of Derivative class in this structure. The Derivative class has only one property which is the underlying, since every derivative instrument has at least one underlying asset in real life. The underlying asset can be stock, commodity, index, currency, etc.. In the scope of this thesis, only stocks as an underlying asset are considered.

In this structure, one subclass of the Derivative class is examined. In addition to Option subclass, Future, Forward and Swap subclasses can be included.

The definition of an option is as follows [21]:

"An option financial derivative that represents a contract sold by one

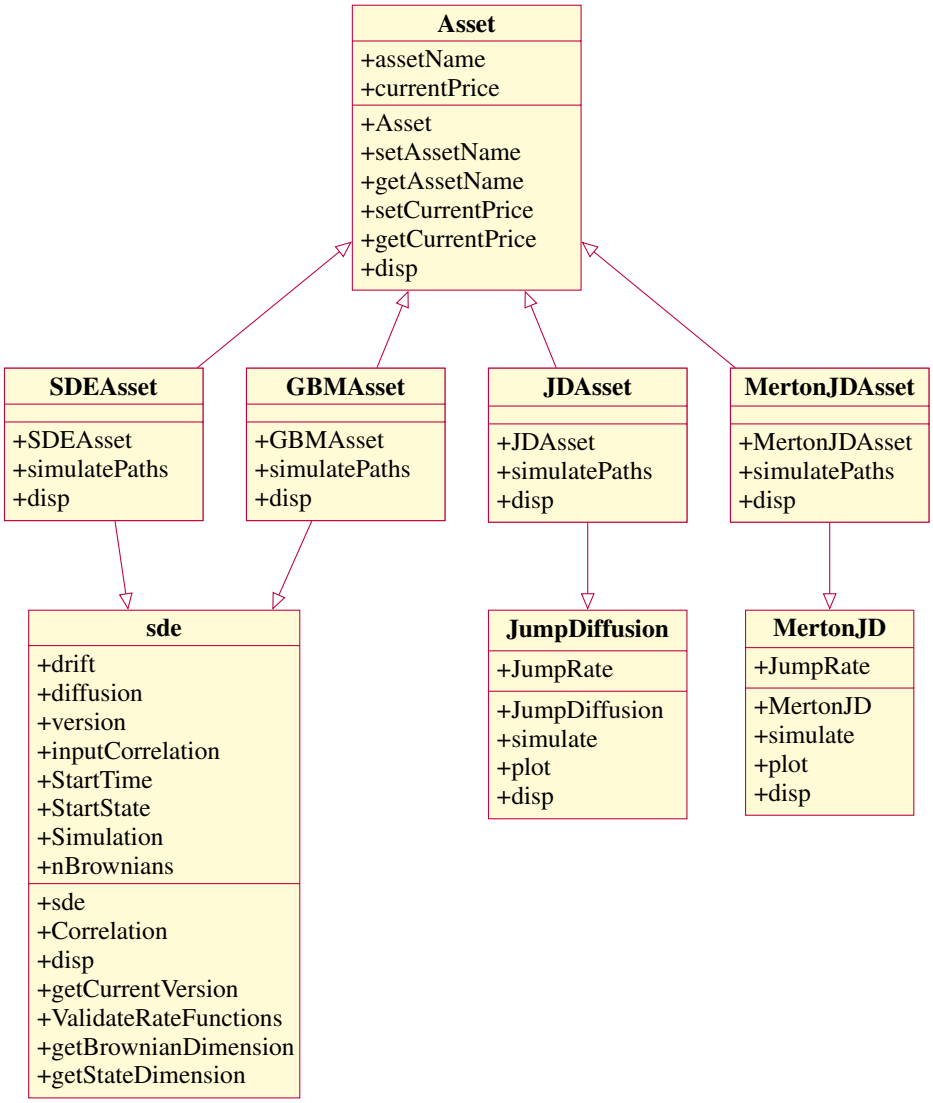



Figure 4.13: UML diagram of Asset class and subclasses of Asset class

party (option writer) to another party (option holder). The contract offers the buyer the right, but not the obligation, to buy (call) or sell (put) a security or another financial asset at an agreed-on price (the strike price) during a certain period or on a specific date (exercise date)."

Since our interest is pricing options, the information about the option writer and holder is unimportant. However, the type of the option contract, whether it is a call or a put, the strike price and exercise date (maturity) are crucial for pricing. Hence, the properties of the Option class are specified according to the required data, namely, strike, maturity and payoff. Also, the underlying property is inherited from the Derivative class.

Before creating the option object, the asset object must be created in order to define the underlying property. The test script to create an option object is given in Listing 8 as an example. The information about created option object is given below.

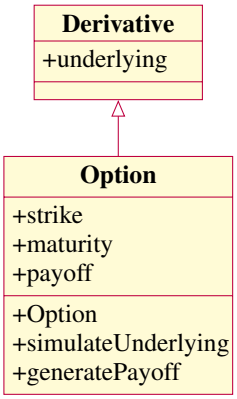



Figure 4.14: UML diagram of Derivative and Option class with generalization relation

```
myOption =

Option with properties:
strike: 125
maturity: 0.0959
payoff: [1x1 BarrierPayoff]
underlying: [1x1 GBM_Asset]
```

The detailed information about the payoff of the myOption object is

```
>> myOption.payoff

BarrierPayoff with properties:

name: 'down-and-out'
barrier: 120
type: 'call'
strike: 125
payoff: {1x3 cell}
```

In the same manner, the information about the underlying asset of the myOption object can be obtained.

```
>> myOption.underlying

Class GBM_Asset:
---------------------------------------
GBM-Example :Asset Name   Current Price: 125.94
---------------------------------------
StartTime: 0
StartState: 125.94
Drift: drift rate function F(t,X(t))
```

```
Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
----------------------------------------
```

### 4.2.5 Pricer Class

The Pricer class can be considered as the final stage of the software. This class is created to price European type plain vanilla and barrier options by analytic solution of the Black-Scholes equation (closed-form), as well as by Monte Carlo method. The supported solution methods according to the payoffs and underlying asset types to calculate the option price are given in Table 4.3.

Table 4.3: Supported pricing methods based on the payoffs and asset model types

| **Payoff** | **Underlying Asset** | **Closed-Form** | **Monte Carlo** |
|---|---|---|---|
| Plain Vanilla | SDE | ✓ | ✓ |
| | GBM | ✓ | ✓ |
| | JD | ✓ | ✓ |
| | MertonJD | ✓ | ✓ |
| Barrier | SDE | | ✓ |
| | GBM | ✓ | ✓ |
| | JD | | ✓ |
| | MertonJD | | ✓ |

After an option object is created with the help of the payoff and asset objects, it is used in defining the pricer object.

The Pricer class is an abstract class and it has only one property which is the price. MonteCarlo class is created to price options by Monte Carlo approaches. MonteCarloCrude subclass is created on the purpose of pricing options with crude Monte Carlo approach. On the other hand, the ClosedForm subclass is created with the purpose of benchmarking the price.

In order to price the created option object, Monte Carlo object is created by using option object as a MCdata property of Monte Carlo object. The test script to show how to create a Monte Carlo and closed-form solution objects is given in Listing 9. The output of the script is as follows:

```
closedForm with properties:

CFdata: [1x1 Option]
price: 14.0153

MonteCarloCrude with properties:

MCdata: [1x1 Option]
```

```
approxsigma: 19.3660
CI: [2x1 double]
NRepl: 10000
price: 14.8368
```

As seen in the output, for a barrier option whose underlying asset follows a GBM with the parameters given in the test script in Listing 9, the closed-form solution and Monte Carlo results for 10000 trials are close to each other. The data, used for both closed-form and Monte Carlo objects, is the same option object.

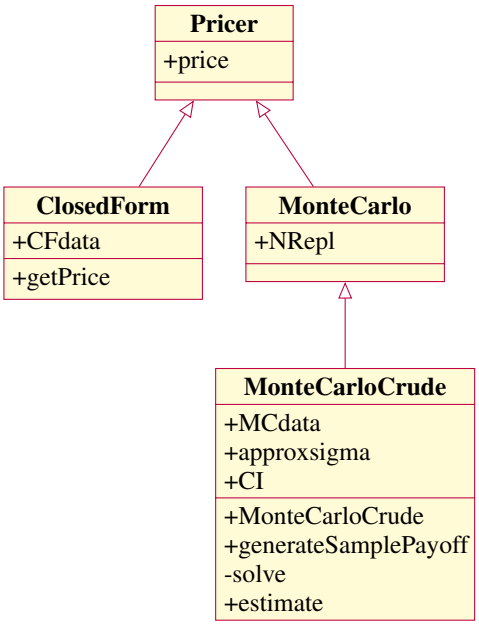



Figure 4.15: UML diagram of Pricer class with its subclasses

### 4.2.6 Relation Between Classes

The inheritance relations of the main parts of the software is given in the previous sections. In this structure beside the inheritance relation, the association relation between the main parts of the software plays an important role. To define the underlying property of the Derivative class, the object generated from the Asset class is used. In the same way, the object created from the Payoff class is utilized as a payoff property for Option class. Pricer class is associated with Option class. Because the Pricer class cannot be used without creating the object from an Option class. A simple illustration to show the association relations is given in Figure 4.16.

## 4.3 Graphical User Interface (GUI)

A user interface (UI) is a graphical tool consisting of a number of windows with some interactive elements so that the user can perform interactive tasks. As a result, the user

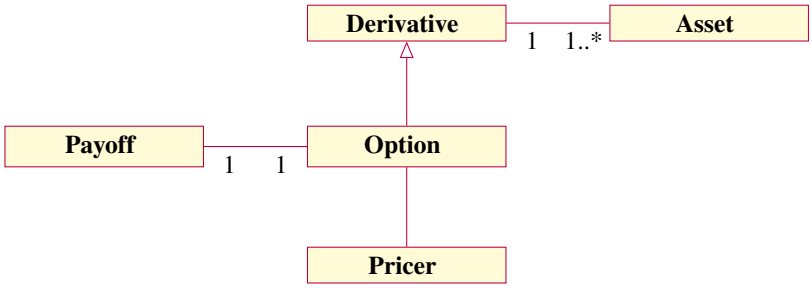



Figure 4.16: Association relations between main classes

does not have to spent time on writing a new code to complete the tasks. For that purpose, a graphical user interface (GUI) is designed and implemented as a by-product of the current thesis. A general view of the Object-Oriented Monte Carlo Option Pricer (OOMCOP) GUI is given in Figure 4.17.

The created GUI consists of three main parts for gathering inputs, providing outputs and displaying the results with a figure. The input region is divided into three parts. In select option part, the user should select the option type and payoff type by using scroll-down buttons. Then, the inputs for the selected option type should be specified and the necessary simulation inputs to run the Monte Carlo simulation should be written into the suitable text boxes. After all the inputs are entered, the Monte Carlo price and the Black-Scholes price outputs can be seen by pushing the Price button. The Visualize button is used to demonstrate the simulated asset paths in the main plot area of the OOMCOP GUI.

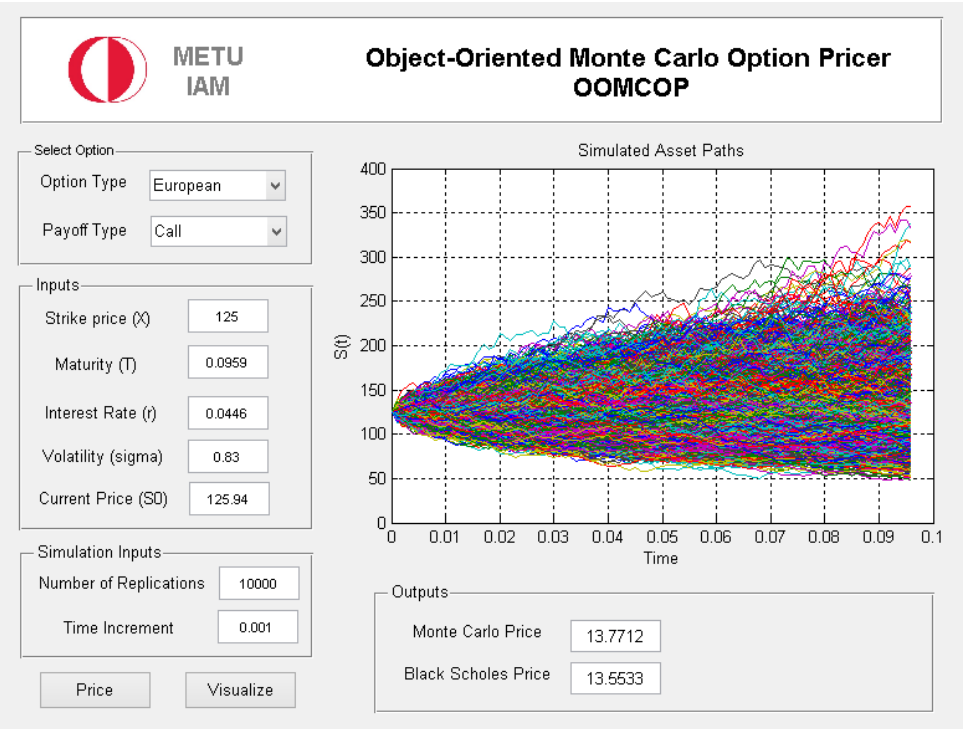



Figure 4.17: GUI for OOMCOP

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

In this study, a software is developed to price European plain-vanilla and barrier options by Monte Carlo approach. In this way the program provides a convenient environment to study on various kinds of underlying assets, option strategies and option themselves.

Object-oriented software development is investigated in three parts: analysis, design and implementation. In object-oriented analysis phase, the process of pricing the European type of plain-vanilla and barrier option by Monte Carlo approach is examined and the process is divided into main parts to reduce the complexity. First, the derivative and the pricer parts are considered as separate classes. The underlying process of the derivative part is designed as another main part, namely, Asset class. MATLAB®'s built-in SDE class is considered as the parent class for the subclasses of Asset class. Because MATLAB®'s SDE class only covers the drift and the diffusion processes, to include the jump diffusion processes for underlying assets the PureJumpProcess class is created as one of the main classes. In the same way, Payoff class is separated from Option class and it is considered as another main class of the software.

Afterwards, object-oriented design of the main classes and hierarchy relations are constructed by determining the common properties and methods of the classes. The design of these main parts and relation between them is given with the UML diagrams in Figure A.1.

Finally, the implementation is put into practice by MATLAB® programing language due to its simplicity and a variety of built-in functions and classes. Also, MATLAB® is a hybrid object-oriented programming language. In addition to its procedural programming features it is suitable for developing object-oriented programming concepts. During the study, it is observed that combining procedural and object-oriented programming can reduce the complexity and provides the reuse of the code. By the help of the created graphical user interface (GUI), it is aimed that the program might well be used by practitioners quite easily.

It can be concluded that the software is flexible and expandable. Because it is divided into small main parts, addition and subtraction of new classes under main classes is convenient and do not ruin the integrity of the existing code.

As a further study, option types, underlying asset types and pricing methods can be

developed. Some expansions that are planned to be done are as follows. Compound and Basket options can be added as a subclass of the Derivative class. Other exotic option payoffs, namely, Asian, lookback, Bermuda, chooser, exchange, shout can be included. Also, PureJumpProcess class can be extended by Variance-Gamma (VG) and Normal Inverse Gaussian (NIG) processes. In addition, Asset price models can be diversified by double exponential jump diffusion (Kou), Bates, Bates-Hull-White models. In Pricer part, Monte Carlo class can be enhanced by variance reduction techniques as antithetic and control variates to increase the efficiency of simulations. Also, new pricing methods, namely, lattice, finite difference, finite element and fast Fourier transform methods can be included. Finally, the current version and the expansions can be considered as European package, in addition to that American package to price option can be developed. The UML diagram of the planned future version of the software is given in Figure C.1. In the light of these expansions and improvements the current GUI can also be upgraded. Thanks to OOP, these modifications will not disarrange the substantial structure and the capabilities of the OOMCOP software can further be improved.

# REFERENCES


[1] T. Björk, *Arbitrage theory in continuous time*, Oxford university press, 2004.

[2] F. Black and M. Scholes, The pricing of options and corporate liabilities, The journal of political economy, pp. 637–654, 1973.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, The unified modeling language user guide, Reading, UK: Addison Wesley, 1999.

[4] S. Borak, A. Misiorek, and R. Weron, Models for heavy-tailed asset returns, in *Statistical tools for finance and insurance*, pp. 21–55, Springer, 2011.

[5] O. J. Boxma and U. Yechiali, Poisson processes, ordinary and compound.

[6] P. P. Boyle, Options: A Monte Carlo approach, Journal of financial economics, 4(3), pp. 323–338, 1977.

[7] P. Brandimarte, *Numerical methods in finance and economics: a MATLAB-based introduction*, John Wiley & Sons, 2013.

[8] M. Broadie and P. Glasserman, Estimating security price derivatives using simulation, Management science, 42(2), pp. 269–285, 1996.

[9] D. Chance and R. Brooks, *Introduction to derivatives and risk management*, Cengage Learning, 2015.

[10] U. Cherubini and G. Della Lunga, *Structured Finance: The Object Oriented Approach*, volume 419, John Wiley & Sons, 2007.

[11] R. Clair, *Learning Objective-C 2.0: A hands-on guide to Objective-C for Mac and iOS developers*, Addison-Wesley, 2012.

[12] T. O. I. Council, Option strategies quick guide, `http://www.optionseducation.org/documents/literature/files/options-strategies-quick-guide.pdf`, accessed: 2015-07-01.

[13] J. C. Cox and M. Rubinstein, *Options markets*, Prentice Hall, 1985.

[14] D. J. Duffy, *Introduction to C++ for financial engineers: an object-oriented approach*, John Wiley & Sons, 2013.

[15] D. J. Duffy and J. Kienitz, *Monte Carlo Frameworks: Building Customisable High-Performance C++ Applications*, volume 406, John Wiley & Sons, 2009.

[16] E. F. Fama, Efficient capital markets: A review of theory and empirical work*, The journal of Finance, 25(2), pp. 383–417, 1970.

[17] J. Farrell, *An object-oriented approach to programming logic and design*, Cengage Learning, 2012.

[18] B. for International Settlements, Otc derivatives statistics at end-june 2014, `http://http://www.bis.org/publ/otc_hy1411.pdf`, accessed: 2015-07-01.

[19] B. Grady, Object-oriented analysis and design with applications, 1994.

[20] U. Gruber, Pricing barrier and Asian options with an emphasis on Monte-Carlo methods, M. Sc. paper, Oregon State University, Corvallis, Oregon, USA, 1997.

[21] J. Guinan, Investopedia's guide to wall speak, 2009.

[22] F. B. Hanson, *Applied stochastic processes and control for jump-diffusions: modeling, analysis, and computation*, volume 13, Siam, 2007.

[23] J. M. Harrison and D. M. Kreps, Martingales and arbitrage in multiperiod securities markets, Journal of Economic theory, 20(3), pp. 381–408, 1979.

[24] J. M. Harrison and S. R. Pliska, Martingales and stochastic integrals in the theory of continuous trading, Stochastic processes and their applications, 11(3), pp. 215–260, 1981.

[25] D. J. Higham, An algorithmic introduction to numerical simulation of stochastic differential equations, SIAM review, 43(3), pp. 525–546, 2001.

[26] J. C. Hull, *Options, futures, and other derivatives*, Pearson Education India, 2006.

[27] H. Kammeyer and J. Kienitz, The Heston-Hull-White model part III: Design and implementation, Wilmott, 2012(59), pp. 44–49, 2012.

[28] J. Kienitz and D. Wetterau, *Financial modelling: Theory, implementation and practice with MATLAB source*, John Wiley & Sons, 2012.

[29] F. C. Klebaner et al., *Introduction to stochastic calculus with applications*, volume 57, World Scientific, 2005.

[30] R. Korn, E. Korn, and G. Kroisandt, *Monte Carlo methods and models in finance and insurance*, CRC press, 2010.

[31] S. Kou, On pricing of discrete barrier options, Statistica Sinica, 13(4), pp. 955–964, 2003.

[32] S. Kou, Jump-diffusion models for asset pricing in financial engineering, Handbooks in operations research and management science, 15, pp. 73–116, 2007.

[33] P. Lavin, *Object-Oriented PHP: Concepts, techniques, and code*, No Starch Press, 2006.

[34] B. Louis, Theory of speculation, Paul H. Cootner, The Random Character of Stock Market Prices (Cambridge, Mass.: MIT Press, 1964), pp. 17–78, 1900.

[35] MATLAB, *version 8.3.0.532 (R2014a)*, The MathWorks Inc., Natick, Massachusetts, 2014.

[36] R. C. Merton, Theory of rational option pricing, The Bell Journal of Economics and Management Science, pp. 141–183, 1973.

[37] T. Mikosch, *Elementary stochastic calculus with finance in view*, volume 6, World scientific, 1998.

[38] S. Mitra, A review of volatility and option pricing, International Journal of Financial Markets and Derivatives, 2(3), pp. 149–179, 2011.

[39] L. Mollamustafaoğlu, *Object-Oriented Programming with Examples in Borland Pascal*, Boğaziçi University Publications, 1995.

[40] M. O'docherty, *Object-Oriented Analysis & Design*, John Wiley & Sons, 2005.

[41] S. Ramnath and B. Dathan, *Object-Oriented Analysis and Design*, Springer Science & Business Media, 2010.

[42] F. D. Rouah, Four derivations of the Black-Scholes formula.

[43] M. Rubinstein and E. Reiner, Breaking down the barriers, Risk, 4(8), pp. 28–35, 1991.

[44] P. A. Samuelson, Proof that properly anticipated prices fluctuate randomly, Industrial management review, 6(2), pp. 41–49, 1965.

[45] R. Seydel, *Tools for computational finance*, volume 4, Springer, 2002.

[46] R. W. Shonkwiler, *Finance with Monte Carlo*, Springer, 2013.

[47] S. E. Shreve, *Stochastic Calculus for Finance II: Continuous-time models*, volume 11, Springer Science & Business Media, 2004.

[48] D. N. Smith, *Concepts of Object-Oriented Programming*, McGraw-Hill, Inc., 1991.

[49] P. Tankov and R. Cont, *Financial modelling with jump processes*, volume 2, CRC press, 2004.

[50] B. Timothy, *An Introduction to Object-Oriented Programming*, Pearson Education, 2002.

[51] Ö. Uğur, *An Introduction to Computational Finance*, volume 1, Imperial College Press, 2009.

[52] P. Van Roy et al., Programming paradigms for dummies: What every programmer should know, New computational paradigms for computer music, 104, 2009.

[53] N. Webber, *Implementing models of financial derivatives: Object oriented applications with VBA*, John Wiley & Sons, 2011.

# APPENDIX A

## UML Diagram of the Software

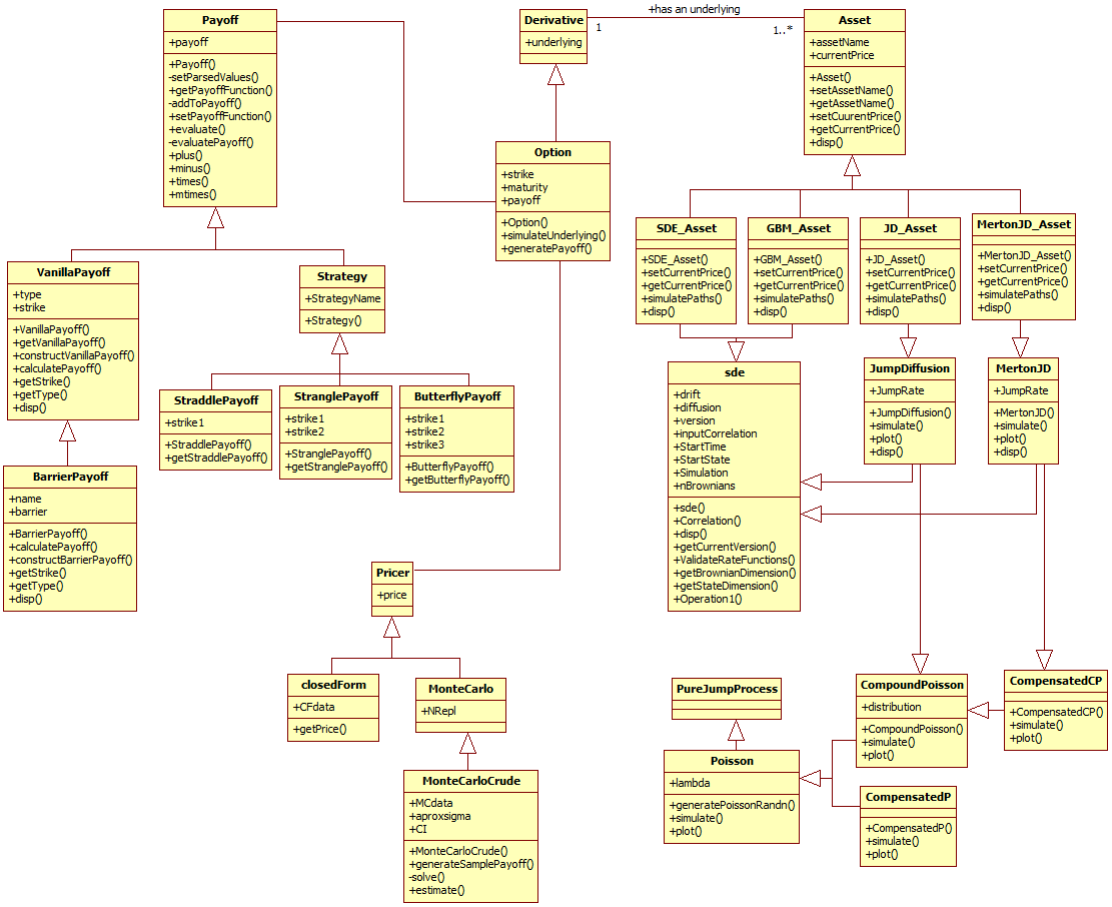



Figure A.1: UML diagram of the OOMCOP

# APPENDIX B

## Test Scripts

These MATLAB® scripts are used to create objects, which are given in Chapter 4.

Listing B.1: An example for plus, minus, times method of Payoff class

```
clear all; close all; clear classes;
x = 0:50; K = 25;

CallPayoff = Payoff(@(x) max(0, x - K), 'long', 1)  % Payoff object
PutPayoff = Payoff(@(x) max(0, K - x), 'long', 1)   % Payoff object
CallPutPayoff = CallPayoff + PutPayoff
CallPutPayoff2 = CallPayoff - PutPayoff
CallPayoff3 = 3 * CallPayoff

subplot(2,2,1)
plot(x, CallPayoff.evaluatePayoff(x), 'b-o'); hold on;
plot(x, PutPayoff.evaluatePayoff(x), 'r-o');hold on; grid on;
subplot(2,2,2)
plot(x, CallPutPayoff.evaluatePayoff(x), 'b-*'); grid on;
subplot(2,2,3)
plot(x, CallPutPayoff2.evaluatePayoff(x), 'b-*'); grid on;
subplot(2,2,4)
plot(x, CallPayoff3.evaluatePayoff(x), 'b-*');grid on;
```

Listing B.2: Test script for Payoff class

```
clear all; close all; clear classes;
x = [-5:0.1:5]';

for alpha = 0:0.1:1
    myF = @(x) thisNewPayoffFunction(x, alpha);
    myPay = Payoff(myF);  % Payoff object
    plot(x, myPay.evaluate(x), 'b-'); hold on; grid on;
end
```

Listing B.3: Test script for VanillaPayoff class

```
clear all; close all; clear classes;

x = [0:10]';
myPay = VanillaPayoff(3,'call');  % VanillaPayoff object
myPayPut = VanillaPayoff(3,'put');  % VanillaPayoff object
plot(x, myPay.evaluate(x), 'b-o'), hold on;
plot(x, myPayPut.evaluate(x), 'r-o'), grid on;
xlabel('x values'); ylabel('payoff');legend('Call Payoff','Put Payoff')
```

Listing B.4: Test script for Option Strategies

```
clear all; close all; clear classes;
x = [0:60]'; strike = 20; strike2 = 10; strike3 = 30; strike4 = 15;

butterfly = ButterflyPayoff(strike2,strike,strike3);
subplot(3,1,1)
plot(x,butterfly.evaluate(x),'b-*')
title('Neutral Strategy:LONG CALL BUTTERFLY','FontSize',12);
xlabel('stock price'); ylabel('payoff'); grid on;

straddle = StraddlePayoff(strike); %StraddlePayoff object
subplot(3,1,2)
plot(x,straddle.evaluate(x),'b-*')
title('Volatility Strategy:LONG STRADDLE','FontSize',12);
xlabel('stock price'); ylabel('payoff'); grid on;

strangle = StranglePayoff(strike, strike2);  %StranglePayoff object
subplot(3,1,3)
plot(x,strangle.evaluate(x),'b-*')
title('Volatility Strategy:LONG STRANGLE','FontSize',12);
xlabel('stock price'); ylabel('payoff'); grid on;
```

Listing B.5: Test script for Iron Condor object

```
x = [45:85]'; strike = 55; strike2 = 60; strike3 = 70; strike4 = 75;

Put1 = Payoff(@(x) max(0, strike-x), 'long',1) % Payoff object
Put2 = Payoff(@(x) max(0, strike2-x), 'short',1) % Payoff object
Call1 = Payoff(@(x) max(0, x-strike3), 'short',1) % Payoff object
Call2 = Payoff(@(x) max(0, x-strike4), 'long',1) % Payoff object
iron = Put1 + Put2 + Call1 + Call2;

plot(x,iron.evaluate(x),'b-*');title('IRON CONDOR','FontSize',12);
xlabel('stock price'); ylabel('payoff'); grid on;
```

Listing B.6: Test script for Poisson class

```
a = Poisson(1) %Poisson object
subplot(2,2,1)
a.plot(3,5)

b = CompoundPoisson(1,'Normal',1,2) %Compound Poisson object
subplot(2,2,2)
b.plot(3,5)

c = CompensatedP(1) %Compensated Poisson object
subplot(2,2,3)
c.plot(3,5)

d = CompensatedCP(1,'Normal',1,2) %Compensated compound Poisson object
subplot(2,2,4)
d.plot(3,5)
```

Listing B.7: Test script for GBM_Asset class

```
F = @(t,X) 0.1 * X; %drift object
G = @(t,X) 0.3 * X; %diffusion object
myAsset = GBM_Asset(F,G); %Asset object
myAsset = myAsset.setAssetName('GBM');
myAsset.disp();
```

Listing B.8: Test script for Option class

```
clear classes; clear all; close all; clc;
%inputs
drift = 0.0446; diffusion = 0.83; CurrentPrice = 125.94;
DeltaTime = 0.01; NTrials = 10000; Maturity = 0.0959;
nPeriods = ceil(Maturity/DeltaTime); Strike = 125; Barrier = 120;
%%
F = @(t,X) drift;
G = @(t,X) diffusion;
myAsset = GBM_Asset(F,G); % Asset object
myAsset = myAsset.setAssetName('GBM-Example');
myAsset = myAsset.setCurrentPrice(CurrentPrice);
```

```
myOption = Option(myAsset,Maturity,Strike); %Option object
myOption.payoff = BarrierPayoff(Strike,'call',Barrier,'down-and-out');
```

Listing B.9: Test script for Pricer class

```
clear classes; clear all; close all; clc;
%inputs
drift = 0.0446; diffusion = 0.83; CurrentPrice = 125.94;
DeltaTime = 0.01; NTrials = 10000; Maturity = 0.0959;
nPeriods = ceil(Maturity/DeltaTime); Strike = 125; Barrier = 120;
%%
F = @(t,X) drift;
G = @(t,X) diffusion;
myAsset = GBM_Asset(F,G); % Asset object
myAsset = myAsset.setAssetName('GBM-Example');
myAsset = myAsset.setCurrentPrice(CurrentPrice);

myOption = Option(myAsset,Maturity,Strike); %Option object
myOption.payoff = BarrierPayoff(Strike,'call',Barrier,'down-and-out');

cf = closedForm(myOption2); %closedForm object
cf.getPrice()
monte = MonteCarloCrude(NTrials,myOption2); %MonteCarloCrude object
monte.solve(nPeriods,'DeltaTime',DeltaTime,'ntrials',NTrials)
```

# APPENDIX C

## Possible Extension to the Software

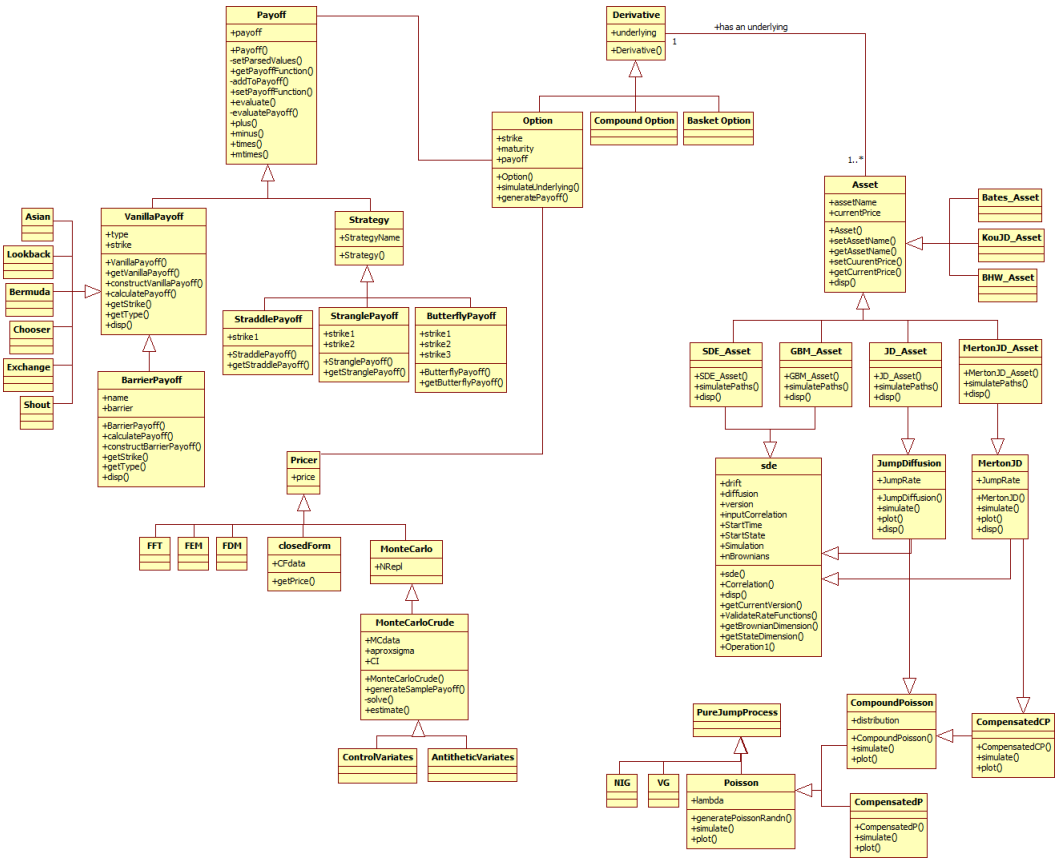



Figure C.1: UML diagram of the possible extension to the software