Development Report of R Financial Option Pricing Package `rmcop`

Yuze Zhai

March 6, 2023

Contents

Chapter 1

Introduction

1.1 Abstract

This report discusses the development of the R package `rmcop`, an fully R-based package for pricing financial options (see Section 1.5 for definitions). Pricing financial options is an essensial fragment of financial engineering and quantitative finance. Because options consider future in time, determining their fair price involves modelling uncertainty, so statistical methods are widely applied. Modern option pricing methods can be complex and product-specific, but most works are still built upon the basic frameworks of Black-Scholes formula and Monte Carlo simulations. The package `rmcop` covers the R implementation of some of these fundamental option price models on various option types and styles, and can be a good reference for constructing more complex R opiton pricing algorithms. The report also contains some R programming techniques that one may found useful in reducing computational cost.

1.2 Report Structure

Here in Chapter 1, the literature review in Section 1.5 will introduce necessary definitions of financial options and option pricing methods that are included in `rmcop` package. This is accompanyed by some useful formula deductions that will be seen useful in Chapter 3, where we will discuss the programming implementation of these models.

In Chapter 2, we will introduce three existing option pricing packages in R (`derivmkts`, `fOptions`, `RQuantLib`), and explain their functionalities. We will also discuss their advantages and limitations.

In Chapter 3, we will discuss the package structure of `rmcop`. This includes: 1. a discussion on R's generic Objective Oriented Programming (OOP) environment; 2. an outline of the package's structure; and 3. a detailed elaboration of the R implementation techniques of the financial models introduced in Section 1.5.

Finally in Chapter 4, we will demonstrate examples and interpretations using `rmcop`. The report is then concluded by a discussion on `rmcop`'s limitations and plans for further development.

1.3 Package Description

`rmcop` is an R package used for pricing financial options. The name "rmcop" stands for "R Monte Carlo Option Pricing". The package supports Monte Carlo based pricing for European vanilla options and European exotic options, including Asian, Barrier, Binary, and Lookback options. It also provides some deterministic methods for both European and American vanilla options pricing, including option pricing via Binomial lattice tree and Black-Scholes formula.

Unlike most existing packages in the R Ecosystem, `rmcop` is developed through an object oriented approach. User's argument inputs for pricing functions are encapsulated in corresponding objects, and pricing functions are themselves methods. This introduces ordered arguments control and easier functions application.

1.4 Machine Environment

The majority of the development and testing of the package is completed using my laptop. The laptop' configuration is briefed below.

CPU: AMD Ryzen 7 4800H with Radeon Graphics (base 2.90GHz)
RAM: 16(15.4)GB DDR4 3200MHz
Max TDP: 45W

The setup is typical for majority of personal computer devices, so the benchmarking and profiling presented in the report are reasonable references for ordinary package users.

1.5 Literature Review

1.5.1 Pricing of Financial Options

An option is a common financial derivative[1] in the market. Options can be roughly classified into two types, "call" or "put". A call option, "gives its holder the right (but not the obligation) to purchase from the issuer a prescribed asset[2] for a prescribed price (strike price) at a prescribed time (maturity / expiration) in the future.[1]" Similarly, a put option gives the right to sell.

The two most common styles of financial options are European options and American options. An European option can only be exercised at maturity, whereas an American option can be exercised at anytime prior to maturity (which is a more complex setting).

The most typical "family" of options are the so called "vanilla options", which includes no special or unusual features seen in exotic options (see list below). For vanilla cases, an European call option has a payoff of $(S_T - K)^+ := \max(S_T - K, 0)$, and an European put option has a payoff of $(K - S_T)^+ := \max(K - S_T, 0)$ [2] at the point of exercise T (i.e. maturity). Here, S_t is the underlying asset price at time $t \in [0, T]$, and K is the option's strike price.

Options with "special or unusual features" are called "exotic options". The exotic options whose pricing are supported by `rmcop` package are introduced below based on the definitions given in *An Introduction to Financial Option Valuation* [1].

[1]a security whose value depends on an the value of an underlying (i.e. the related) asset.
[2]Underlying asset/stock, the word "asset" and "stock" may be used interchangably through this report.

- **Asian Options** Asian options' payoff are determined by the average price of the underlying asset throughout its life span.
  - An average price Asian call option has payoff at the expiry T given by $\max(\bar{S} - K)$.
  - An average price Asian put option has payoff at the expiry T given by $\max(K - \bar{S})$.
  - An average strike Asian call option has payoff at the expiry T given by $\max(S_T - \bar{S})$.
  - An average strike Asian put option has payoff at the expiry T given by $\max(\bar{S} - S_T)$.
- **Barrier Options** Barrier options have a payoff that switches on or off depending on whether the asset crosses a pre-defined level (barrier) B.
  - A down-and-out (knock-out) call option has a payoff that is zero if the asset crosses some predefined barrier $B < S_0$ at some time in $[0, T]$. If the barrier is not crossed then the payoff becomes that of a European call, $\max(S_T - K, 0)$.
  - A down-and-in (knock-in) call option has a payoff that is zero unless the asset crosses some predefined barrier $B < S_0$ at some time in $[0, T]$. If the barrier is crossed then the payoff becomes that of a European call, $\max(S(T) - K, 0)$.
- **Binary Options** A binary (a.k.a. cash-or-nothing) option have payoff at expiry being either some specified value A or zero.
  - A binary call option has payoff A if $S_T > K$ and zero otherwise.
  - A binary put option has payoff A if $S_T < K$ and zero otherwise.
- **Lookback Options** The payoff for a lookback option depends upon either the maximum $S^{\max}$ or the minimum value $S^{\min}$ attained by the asset throughout the price trajectory.
  - A fixed strike lookback call option has payoff at expiry T given by $\max(S^{\max} - K, 0)$.
  - A fixed strike lookback put option has payoff at expiry T given by $\max(K - S^{\min}, 0)$.
  - A floating strike lookback call option has payoff at expiry T given by $S_T - S^{\min}$.
  - A floating strike lookback put option has payoff at expiry T given by $S^{\max} - S_T$.

1.5.2 Brownian Motion

A *stochastic process*, by definition [3], is a family of random variables $\{X_\gamma, \gamma \in \Gamma\}$ defined on $\Omega \times \Gamma$ taking values in $\mathbb{R}$. Thus, the random variables of the family (measurable for every $\gamma \in \Gamma$) are functions of the form:

$$X(\gamma, \omega) : \Gamma \times \Omega \mapsto \mathbb{R}$$

Where $(\omega, \mathcal{A}, P)$ is a probability space described by samples ω, action space $\mathcal{A}$, and probability measure P.

For $\Gamma = \mathbb{N}$, we have a *discrete process*, and for $\Gamma \subset \mathbb{R}$, we have a *continuous process*. In our context we will consider Γ to represent the scope in time, and takes values from 0 to maturity T.

A *stochastic process* $\{X(t), t \geq 0\}$ is said to be a *Brownian motion* if [3]:

1. $X(0) = 0$;

2. $\{X(t), t \geq 0\}$ has stationary and independent increments.
3. for every $t > 0$, $X(t) \sim N(0, \sigma^2 t)$.

A *Standard Brownian Motion* (i.e. Wiener Process, abbreviated as SBM) inherits the properties above, and have a volatility measure of zero (i.e. $\sigma = 0$). So for an SBM W, we have $W(t) \sim N(0, t)$.

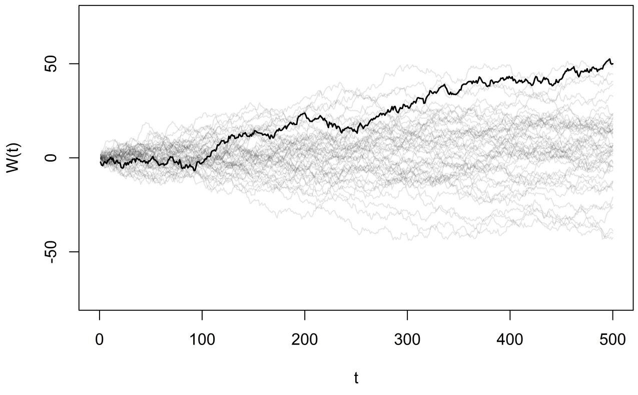


Figure 1.1: Trajectories of Wiener Process

Figure 1.1 demonstrates the trajectories of a one-dimensional SBM, where each gray line represents a simulated path of a Wiener process.

Notice that the expectation of a SBM at any time $t \geq 0$ is zero, this makes a SBM a *martingale*, which is a stochastic process that "does not tend to rise or fall[3] [4]". If S_t is the value of a SBM at time t, assume $S_0 = 0$, its SBM dynamics can be defined by the Stochastic Differential Equation (SDE):

$$dS_t = S_t dW_t$$

We can generalised the scenarios by considering drifts through time (measured by μ) and scaling the amount of diffusion with respect to time (measured by σ). For S_t that follows a *Geometric Brownian Motion* (GBM), its dynamics can be described by the SDE [4]:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

Where $\mu S_t dt$ is called the *drift term* and $\sigma S_t dW_t$ the *diffusion term*.

Expanding on S_t, for a function $f : \mathbb{R} \mapsto \mathbb{R}$, the GBM $f(S_t)$ is described as, according to Itô's rule [4]:

$$df(S_t) = \left[\mu_t \frac{\partial f}{\partial S}(S_t) + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}(S_t)\right] dt + \sigma_t \frac{\partial f}{\partial S} dW_t$$

[3]That is, $E[W(t+k)] = W(t)$ for any k.

Using Itô-Doeblin rule for $f : \mathbb{R} \mapsto \mathbb{R}$ [4], we can derive the following explicit solution to the SDE:

$$f(S_t) = f(S_0) + \int_0^t \mu_t \frac{\partial f}{\partial S}(S_t)dt + \frac{1}{2}\int_0^t \sigma_t^2 \frac{\partial^2 f}{\partial S^2}(S_t)dt + \int_0^t \sigma_t \frac{\partial f}{\partial S}(S_t)dW_t \tag{1.1}$$

Equation 1.1 will be shown useful in the Black-Scholes model introduced in the following section.

1.5.3 Black-Scholes Model

The very first attempt of applying quantitative method in option pricing (perhaps in all finance) is by the French mathematician Louis Bachelier in 1900. In his work *The Theory of Speculation* [5], he deducted deterministic formulas for pricing European (vanilla) call and put options as follows:

$$C(S,T) = SN(\frac{S-X}{\sigma\sqrt{t}}) - XN(\frac{S-X}{\sigma\sqrt{t}}) + \sigma\sqrt{t}N(\frac{S-X}{\sigma\sqrt{t}})$$
$$P(S,T) = XN(\frac{S-X}{\sigma\sqrt{t}}) - SN(\frac{S-X}{\sigma\sqrt{t}}) + \sigma\sqrt{t}N(\frac{S-X}{\sigma\sqrt{t}})$$

Being the earlist approach, Bachelier's formula had already outlined the relationship between option price and asset price S, strike price X, and volatility measure σ, which are essential fragments in modern formulas. However, based on limited data, Bachelier's solution was built under some unrealistic assumptions. The normality assumption violates the non-negativity of the stock price, and the formula's discrete measure in time omitted the effect of continuous movements in the stock price. Also, the formula did not discount the effect of interest rate. These errors cause Bachelier's model fails to price options accurately.

It wasn't until 1960s had further improvement been made to quantitative option pricing. In 1961, Case Sprenkle [6] introduced the Sprenkle formula. The formula addressed the above issues by describing the stock price by the more suitable log-normal distribution and discounting for the effect of interest rate, which successfully explained the time value of an option. In the following decade, improvements have been made by scholars such as Boness and Samuelson [7], who introduced emperical constants to increase the effectiveness of Sprenkle's model.

The model was finalised by Black and Scholes in 1973 [7], who explained the stock price movement by GBM. The GBM was described in the form of a Stochastic Differential Equation (SDE), which effectly model the continuity of price movement. The solution (derived through Itô's lemma) of Black-Scholes formula under an risk-neutral approach[4] eliminates the emperical measure of Sprenkle et al's model, which resulted in an objective and deterministic estimation of option price, as is used by most contemporary pricing methods.

Recall the explicit form of an SDE $f(S_t)$ given by Itô-Doeblin rule in Equation 1.1. We assign $f(S_t) = \log(S_t)$ where log is the natural logarithm. The solution is:

$$\log(S_t) = \log(S_t) + (\mu - \frac{\sigma^2}{2})t + \sigma W_t \tag{1.2}$$

[4]The risk-neutral approach, in simple terms, is constructing a portfolio at a moment in time such that the portfolio value will be identical at the next moment in time regardless of the price movement, so the portfolio will be riskless to the price movement.

Because $W_t \sim N(0,t)$, we can see that Equation 1.2 shows that:

$$\log(S_t) \sim N(\log(S_0) + (\mu - \frac{\sigma^2}{2})t, \sigma^2 t)$$

This implies that S_t follows a *log-normal* distribution. A log-normal distribution takes only positive value, which addressed the negativity issue caused by Gaussian stock price models, like the one developed by Bachelier [5].

By taking the exponential on the two sides of the equation, one can obtain the following expression:

$$S_t = S_0 \exp\left\{(\mu - \frac{\sigma^2}{2})t + \sigma W_t\right\} \tag{1.3}$$

Which describes the stock price S_t at anytime $t \in [0,T]$. When simulating real market conditions, one may replace the factor μ with r and q, which stands for the (fixed) interest rate and dividend yield rate, respectively.

$$S_t = S_0 \exp\left\{(r - q - \frac{\sigma^2}{2})t + \sigma W_t\right\}$$

Under the risk-neutral assumption, the payoff of a (European vanilla) call option with strike price K and expiration T is given by the expectation $E[e^{-rT}(S(T)-K)^+]$, and the corresponding payoff of a (European vanilla) put option at the same strike price and expiration is given by $E[e^{-rT}(K - S(T))^+]$. Using the solution of the Black-Scholes SDE we can deterministically eavluate the payoffs as follows [1]:

$$\begin{aligned} C(S(0),T) &= S(0)\Phi(d_1) - e^{-rT}K\Phi(d_2) \\ P(S(0),T) &= e^{-rT}K\Phi(-d_2) - S(0)\Phi(-d_1) \end{aligned}$$

Where Φ is the cumulative normal distribution, $d_1 := \frac{\log(S(0)/K)+(r+\frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}$, and $d_2 = d_1 - \sigma\sqrt{T}$. The Equations 1.4 and 1.5 are the so-called Black-Scholes formula.

A genearlised solution which addressed the impact of the presence of dividends (assuming the option of interest has an annual dividend yield rate of δ) specified as [2]:

$$C(S(0),T) = e^{-\delta t}S(0)\Phi(d_1) - e^{-rT}K\Phi(d_2) \tag{1.4}$$

$$P(S(0),T) = e^{-rT}K\Phi(-d_2) - e^{-\delta t}S(0)S(0)\Phi(-d_1) \tag{1.5}$$

Where now $d_1 := \frac{\log(S(0)/K)+(r-\delta+\frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}$.

A further result to the American option cases is such that an American call option is never optimal (i.e. the estimated value would be the same as the European call option under same conditions), and that an American put option's value does not have an analytical form and required numerical methods to calculate[5].

[5]This is related to the studies on Monte Carlo pricing for American Options, which is beyond the current scope of `rmcop`.

1.5.4 Binomial Lattice Model

In 1979, based on the risk-neutral approach used by the Black-Scholes model, Cox, Ross and Rubinstein (CRR) [8] introduced a Binomial lattice tree model for modelling stock prices.

To construct a Binomial tree of stock prices, one breaks down the option's life $[0, T]$ into n time steps with fixed interval $\Delta t = T/n$. For each time steps, the stock price can move either up by a factor u with probablity $\hat{p}$ or down by a factor d with probability $\hat{q} = 1 - \hat{p}$.

$$\begin{tikzcd}
 & & u^2S \\
 & uS \arrow[ur, "\hat{p}"] \arrow[dr, "1-\hat{p}"] & \\
S \arrow[ur, "\hat{p}"] \arrow[dr, "1-\hat{p}"'] & & udS \\
 & dS \arrow[ur, "\hat{p}"'] \arrow[dr, "1-\hat{p}"'] & \\
 & & d^2S
\end{tikzcd}$$

Figure 1.2: Binomial Lattice Tree for Stock Price with 2 Steps

As Figure 1.2 shows, we will obtain a total of $n+1$ nodes at maturity T. For each node at the final step, we can obtain the corresponding payoff of an option related to that stock price. For example, recall from the above section, the $n+1$ possible payoffs $P(S(T))$ of an vanilla European call option would be:

$$P(S(T)) = (S(T) - K)^+ \tag{1.6}$$

The probability $\hat{p}$ is set to be "risk neutral" in a way such that:

$$\begin{aligned}
S(t_i) &= e^{-r\Delta t}\mathbb{E}[S(t_{i+1})] \\
&= e^{-r\Delta t}[\hat{p}uS(t_i) + \hat{q}dS(t_i)], \quad \text{for } i = 0, ..., n-1
\end{aligned}$$

And it can be calculated using the formula:

$$\hat{p} = \frac{e^{r\Delta t} - d}{u - d} \tag{1.7}$$

Meaning that the current node's stock price is equal to the expected value of the two future nodes it can go in the next time step, discounting the interest. In such way, a portfolio which consist only of this stock would have a fixed return independent from price movement.

Under such setting, the option payoff at each node is computed in similar way. Suppose payoff P_{ij} denotes the payoff of the jth node at time step i, it can be computed via recursive form:

$$P_{ij} = e^{-rT}[\hat{p}P_{(i+1),j+1} + \hat{q}P_{(i+1),j}] \tag{1.8}$$

As mentioned above, the payoff of an option at maturity T is obvious to obtain. By applying this recursive method forward from $t = T$ back to $t = 0$, we will obtain an deterministic estimate of the current payoff (fair value) of the option of interest.

1.5.5 Monte Carlo Option Pricing

As the financial market develops and more complicated options emerge, in many realistic cases, one cannot find a deterministic solution for pricing option. However, thanks to the advancement of computer power, one can simulate price trajectories for enormous times, and estimate the payoff of the option of interest by simply taking the average of the option payoff under each simulated trajectory. Such method is known as the Monte Carlo method. The very first attempt of applying computational method in option pricing is by Phelim Boyle in 1977. More serious (and effective) approach was introduced by Paul Glasserman in the 1990s based on the Black-Scholes model. Until now, the field of Monte Carlo option pricing is still under active development and is widely used by "quants". The contents below will elaborate implementation method as the one introduced in Glasserman's *Monte Carlo Methods in Financial Engineering* [2].

We know that a GBM is a continuous process, and its dynamics is described by an SDE with respect to the t through infinitesimal dt. In order to simulate the process, we need to discretise dt to the computable Δt.

Recall the explicit form of the GBM (Equation 1.3) introduced in the Black-Scholes model, we can expand on it to describe the change in asset price $S_{t+\Delta t}$ as:

$$S_{t+\Delta t} = S_t \exp\left\{(\mu - \frac{\sigma^2}{2})\Delta t + \sigma W_{\Delta t}\right\}$$

Knowing that $W_t \sim N(0,t)$, suppose we have $Z :\sim N(0,1)$, we have $W_t = Z\sqrt{t}$. So we can modify the above equation to:

$$S_{t+\Delta t} = S_t \exp\left\{(\mu - \frac{\sigma^2}{2})\Delta t + \sigma Z\sqrt{\Delta t}\right\}$$

Suppose we discretise the continuous time interval $[0,T]$ into m time steps with fixed length $\Delta t = \frac{T}{m}$. We can derive the iterative formula:

$$S_i = S_{i-1} \exp\left\{(\mu - \frac{\sigma^2}{2})\Delta t + \sigma Z_i\sqrt{\Delta t}\right\}, \qquad i = 1, ..., m \tag{1.9}$$

We can also modify the equation to address the effect of interest rate and dividend yield rate according to what's mentioned below Equation 1.3:

$$S_i = S_{i-1} \exp\left\{(r - q - \frac{\sigma^2}{2})\Delta t + \sigma Z_i\sqrt{\Delta t}\right\}, \qquad i = 1, ..., m \tag{1.10}$$

By simulating m standard normal random variables $Z_1, ..., Z_m$, we will be able to generate a full trajectory of the price movements.

```
for $i \leftarrow 1, ..., m$ do
    $Z_i \leftarrow N(0,1)$
    $S_i \leftarrow S_{i-1} \exp\left\{(\mu - \frac{\sigma^2}{2})\Delta t + \sigma Z_i \sqrt{\Delta t}\right\}$
end for
```

Noticing that as $W_T \sim N(0, T)$, and that individual increments for Brownian motions are independent, the approximation of W_T using Z_i's is lossless (i.e. the number of time steps / length of Δt, has no effect on how well the approximation will be). For path-independent options[6], it may be wise to take $m = 1$, i.e. $\Delta t = T$ to minimise the computational cost.

If we are to simulate n times, for the ith trajectory, $i = 1, ..., n$, one may employ formulas introduced in List 1.5.1 and discounting the effect of interest rate and dividend rate to obtain the corresponding discounted payoff (i.e. fair price) of the option, C_i. For example, the discounted payoff of an Vanilla call option, at maturity, can be calculated as:

$$C_i = e^{-rT}(S_T - K)^+ \tag{1.11}$$

The Monte Carlo estimator for the option price is then simply the arithmetic mean $\hat{C}_n := \frac{1}{n}\sum_{i=1}^{n} C_i$. This estimator is *unbiased* and *strongly consistent* [2].

For large enough n, we can construct a confidence interval using the standard error, given by [2]:

$$SE_C = \frac{s_C}{\sqrt{n}} \tag{1.12}$$

$$s_C = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(C_i - \hat{C}_n)^2} \tag{1.13}$$

Glasserman in his work [2] also indicated the criterion for the simulation estimators' efficiency: computing time, bias, and variance. The estimators considered in our package are unbiased, and variance reduction techniques are beyond our scope. So we will only discuss some R programming methods in reducing the computing time to increase estimation efficiency, as we will see in Chapter 3.

[6]Options whose payoff only depends on the stock price at maturity, S_T.

Chapter 2

Existing Option Pricing Packages within R Ecosystem

2.1 Introduction

In this chapter, we will introduce three common R packages which are used for pricing financial options.

2.2 Packages Review

2.2.1 derivmkts

2.2.2 fOptions

2.2.3 RQuantLib

The QuantLib is a C++ based open-source library for quantitative finance. The package `RQuantLib` is simply an R interface to QuantLib

An R interface to the QuantLib library, which embeded C++ programming.

2.3 General Comments

Chapter 3

Package Development

3.1 Introduction

In this chapter, we will elaborate on the development of `rmcop`. We will first present the general structure of the package. Then, we will discuss the implementation of pricing models mentioned in the literature review in Section 1.5 in details.

3.2 Package Structure

The overlying workflow of using `rmcop` package to price financial option is displayed in Figure 3.1 below. One shall see that with the OOP framework, we encapsulate most arguments into two classes of objects: `option` and `env`.

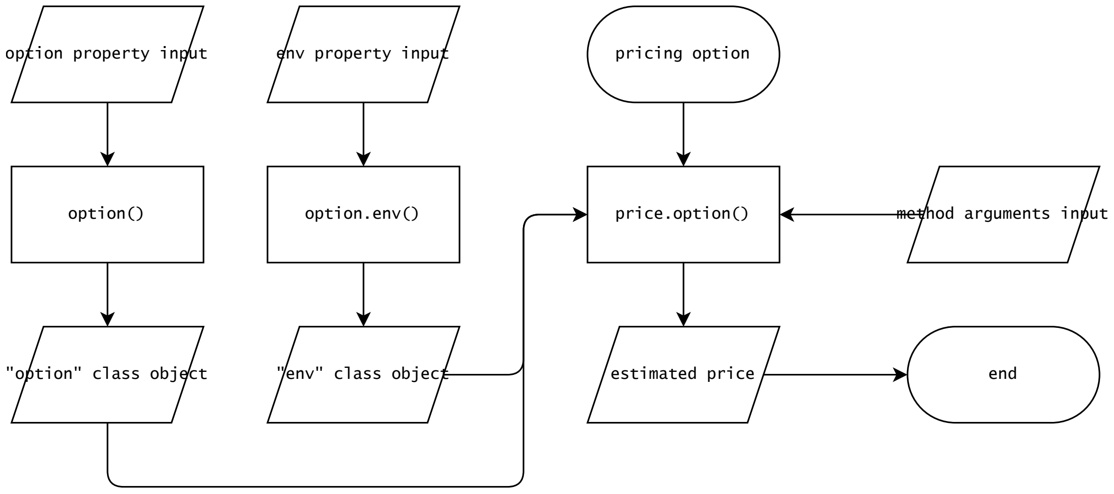


Figure 3.1: Flowchart of option pricing using rmcop package

As Figure 3.1 indicates, there are only three exported[1] functions. Their purpose is shown in

[1]When installing an R package, only functions that are exported can be directly called and executed by the

Table 3.1 below:

Function	Description
`option()`	Create new `"option"` class object, which encapsulates the characteristics of an option
`option.env()`	Create new `"env"` class object, which encapsulates the market environment setup
`price.option()`	Pricing the option based on specified option, market environment, and method input

Table 3.1: Functions to Call

To keep the functions (exported and internal) organised and to make each script restricted to a maintainable length, they are divided to 7 R scripts listed below:

File	Contents
`Option.R`	Methods creating and updating option and option.env objects
`Price.R`	Pricing functions takes objects input and calls specific pricing engines
`MonteCarlo.R`	Monte Carlo option pricing method engine
`BlackScholes.R`	Black-Scholes option pricing method engine
`Binomial.R`	Binomial option pricing method engine
`Tools.R`	Supplementary functions used in the package

Table 3.2: Package R Scripts

`Option.R` and `Price.R` contains all constructor functions for S3 class objects. `Price.R` contains the overlying pricing functions which will trigger the pricing engine functions that are contained in `MonteCarlo.R`, `BlackScholes.R`, and `Binomial.R`. `Tools.R` includes supplementary functions, including checking of inputs integrity and plotting of Monte Carlo price trajectories.

3.3 Objective Oriented Programming

3.3.1 R Framework for OOP

Existing packages in R ecosystem provides comprehensive pricing algorithms for financial derivatives. However, their function are used in a Procedural Oriented (POP) way. POP functions can be called directly by passing in required arguments. For simple options pricing cases, such as pricing individual options, using POP functions is intuitive. However, many real scenarios require pricing options in a complex formulation, such as compounded options (i.e. options with underlying assets being another option) and combination of options (i.e. spread, straddle, stringle, and other option strategies). In these situations, managing numerous arguments for POP functions can be difficult and inefficient.

The `rmcop` package proposed an Objective Oriented Programming (OOP) approach for pricing financial options in R. It encapsulates multiple arguments, such as option style, type, strike price, and maturity time, into an `option` class object. It also allows encapsulation of other market environemnt arguments, such as interest rate, dividend yield rate, and volatility measure

user, other "unexported" functions might serve as internal components of the package.

into an `option.env` class object. The OOP structure enables easier variables managements and facilitates the comparison of prices among different sets of options and market environments.

The codes below demonstrates the calculation of an theoretical European vanilla call option with strike price $K = 20$, maturity $t = 0.75$, under the market condition such that the current price is $S = 20$, fixed interest rate is $r = 1\%$, and market volatility measured by $\sigma = 0.1$. We use Monte Carlo (i.e. `mc`) method with $n = 100$ replications and number of time steps per replication $steps = 1$.

OOP approach requires extra steps declaring objects before using the function, but once the declaration is completed, calling the pricing funcion is much simpler than the POP approach. As above, it only required two (object) arguments, `obj` and `env`.

R provides two ways to perform OOP, the S3 and S4 methods. The S3 class objects are based on R `list` objects. An R list contains a `class` attribute, which can be customised into string (or vector of strings) that can be interpreted as a list's corresponding S3 class (i.e. so that the list itself became an object of that class). Other items within the list can be treated as object's properties under the OOP scope. Here is an example on how to implement OOP using S3 method in R.

```r
student <- function(age, gender) {
    obj <- list(
        "age" = age,
        "gender" = gender
    )
    class(obj) <- "student"
    obj
}

John <- student(
    "age" = 20,
    "gender" = "male"
)

John
> $age
> [1] 20
>
> $gender
> [1] "male"
>
> attr(,"class")
> [1] "student"
```

We first define a new `list` object named “John”, which contains two items/properties, age and gender. Then, we redefined the class of this list using the `class()` function to “student”. Now, we have created a new object named “John” of the class “student” under the S3 scheme.

R's S3 method also provides a naive way of expressing inheritance, which is by expressing the `class` attribute of an object with a vector instead of a single string. Continuing our previous example, let's now create a new object ”Sara” using the following code:

```r
international.student <- function(age, gender, nationality, language)
{

    obj <- student(age, gender)
    obj <- c(
        obj,
        "nationality" = nationality,
        "language" = language
    )
    class(obj) <- c("international", "student")
    obj
}

Sara <- international.student(
    age = 21,
    gender = "female",
    nationality = "Singapore",
    language = "Chinese"
)

Sara
> $age
> [1] 21
>
> $gender
> [1] "female"
>
> $nationality
> [1] "Singapore"
>
> $language
> [1] "Chinese"
>
> attr(,"class")
> [1] "international" "student"
```

Here, we see that the object representing Sara not only contains the properties as a ”student” class object has, but also contains two extra properties ”nationality” and ”language”. In this case, the objects from class `international student` are inherited from the father class `student`, which is reflected by the `class` attribute being a vector and has the last item named ”student”. To check whether an object is from a class, we can use the `inherits` function. For example:

```r
inherits(Sara, "student") # True
inherits(John, "student") # True
inherits(Sara, c("international", "student"), which = TRUE) # 1 1
inherits(John, c("international", "student"), which = TRUE) # 0 1
```

The S4 methods requires more rigorous class and object definition, as one would typically see in an OOP language such as Python and Java. The development of our pricing functions does not require rigorous OOP structure or defining generic functions, so S3 method is sufficient for its development.

3.3.2 Option and Environment Class Objects

As we mentioned in Section 3.2, `rmcop` requires users to encapsulate most of their input arguments as two classes of objects, `option` and `env`. Thus, we first need to build functions (constructors) that allows user to create/initialise new objects.

The option class object should contains properties that defines an option, or elements that are common to be pre-specified when the option is issued. This includes the option's style, type, strike price K, and maturity T.

First, it is reasonable and convenient to assume that all options should have specified style, type, strike price, and maturity. We should therefore make these properties contained by the overlying class.

```r
option <- function(style, type, K, t) {
    obj <- list(
        "style" = style,
        "type" = type,
        "K" = K,
        "t" = t
    )
    class(obj) <- "option" # Specifying class attribute to the S3 object
    obj
}
```

This would be sufficient to deal with vanilla options in the market. However, when defining exotic options, extra properties may be required. For instance, a binary option would require an additional property to specify the fixed payoff it would generate; a barrier option, in contrast, would require a different property to specify a price barrier and another property to classify whether it is a “knock-in” or “knock-out” option.

To address these differences, creating “sub-classes” and using inheritance techniques via OOP framework is an intuitive and proper approach. We shall update our previous function by including a new argument named `option`, which allows user to specify the family/name of the (exotic) option the user wish to define[2].

```r
option <- function(style, option = "vanilla", type, K, t, ...) {

    obj <- list(
        "style" = style,
        "type" = type,
        "K" = K,
        "t" = t
    )

    # Direct to corresponding sub-class constructors
```

[2]The default value is set to `"vanilla"`, since this is the most common family of options.

```
    if (option == "vanilla") {
        obj <- option.vanilla(obj)
    } else {
        obj <- option.exotic(obj, option, ...)
    }
    obj
}
```

As shown by the codes above, depending on the value of input for the argument `option`, the constructor calls one of the two sub-class constructors. The function `option.vanilla` will take the list created with four basic properties and assign it with a `"vanilla" "option"`[3] class attribute. On the other hand, the function `option.exotic` will take the input for argument `option` together with additional user inputs, and return an object of class `"XXX" "option"`, where `XXX` is the family of option that the user specified. An example is demonstrated below:

```
option(style = "european",
       option = "barrier",
       type = "call",
       K = 20,
       t = 1,
       barrier = 21,
       is.knockout = TRUE)
> $style
> [1] "european"
>
> $type
> [1] "call"
>
> $K
> [1] 20
>
> $t
> [1] 1
>
> $barrier
> [1] 21
>
> $is.knockout
> [1] TRUE
>
> attr(,"class")
> [1] "barrier" "option"
```

In this example, the user specified the option to be a "barrier" option, which required additional parameters `barrier` and `is.knockout` (detailed discuss see sections below on individual pricing functions). As we can see, the `class` attribute became `"barrier" "option"`, which follows what we have discussed earlier.

After completing the constructor for option class, we need to build the constructor for the

[3]This is the printted form of the vector `c("vanilla", "option")`

environment class. For an object of the class `env`, we wish to encapsulate some market conditions that will be considered during our pricing procedure. The parameters we considered here are: pricing method, current underlying asset price S, interest rate r, dividend yield rate q, volatility measure $sigma$, and two supporting parameters `n` and `steps`. Following the standard procedure of creating an S3 object, we have the following algorithm:

```
option.env <- function(method = "mc", S, r, q = 0, sigma, n = NULL,
steps = NULL) {
    env <- list(
        "method" = method,
        "S" = S,
        "r" = r,
        "q" = q,
        "sigma" = sigma,
        "n" = n,
        "steps" = steps
    )
    class(env) <- "env"
    env
}
```

Since Monte Carlo method is the most commonly used pricing method (and the core of this package), we set the default value of the argument `method` to the abbreviation `"mc"`. The parameters of current price, interest rate, and volatility measure are mandatory, as they are essensial inputs for the pricing functions. The dividend yield rate is set to a default zero to cover the majority of non-dividend bearing options. Finally, the two supplementary parameters `n` and `steps` are allowed to be left as `NULL` depending on whether the specified pricing method requires these arguments to be specified.

3.4 price.option

As we mentioned in Section 3.2, `price.option` is the overlying function in the script `Price.R`. It act as a medium that connects user input and pricing algorithms, making the package structure more operable.

To do this, we require this function to: 1. take objects/arguments input from user, 2. trigger corresponding pricing engines based on specified method, and 3. return the estimation of option price. We can do so starting from the codes below:

```
price.option <- function(obj, env, method = env$method, n = env$n, ...
, all = FALSE) {
    check.method(method)
    res <- get(paste0("price.option.", method))(obj = obj, env = env,
    n = n, ..., all = all)
    res
}
```

As we see, the input argument of the function are:

- `obj` the pre-defined option object.

- `env` the pre-defined environment object.
- `method` a string to determine which pricing method to use.
- `...` additional arguments that are required/provided by specific pricing methods.
- `all` a boolean[4] value to determine whether only a number of estimate should be returned or should all relevant data used in the computation should be returned as well.

Going through the code with more details, in line 2, `check.method` is a function in `Tools.R` which checks if a pricing method is specified and if it is supported by the package.

The following `get(<function name>)(<arguments>)` structure in line 3 executes the function with the name provided and takes specified argument inputs. This line triggers corresponding sub-functions that evoke specific pricing engines. For example, the sub-function for Binomial tree option pricing is shown below:

```r
price.option.binomial <- function(obj, env, n, u, d, all) {
    res <- get(paste0(class(obj)[1], ".binomial"))(obj = obj, env =
    env, n = n, u = u, d = d, all = all)
    res
}
```

These sub-functions are all named in the format `"price.option.XXX"`, where `"XXX"` is the string defined by argument `method`. These sub-functions also used the similar structure to trigger corresponding pricing engines for different family of options. The pricing engines are all named in the format `"family.method"`. For instance, the Monte Carlo pricing engine for lookback option is named `lookback.mc`. Using the trick of `get()()` allowed us to decide which subordinate function to call without using lengthy if-else statements.

The input objects and method-specific arguments will then be passed to the pricing engine, where the information encapsulated within the objects will be extracted and used to compute the estimation of the option's price. The detailed construction of these engines will be discussed in the sections below.

3.5 Deterministic Methods

Deterministic methods provide closed-form formulas which builds upon their relevant assumptions to price options. When pricing an option using deterministic method, given the same initial conditions, the result estimates should always be the same.

The deterministic methods that are included in `rmcop` are Black-Scholes formula and Binomial Lattice Tree. The package's implementation of them are limited to pricing American and European Vanilla options.

3.5.1 Black-Scholes Model

`vanilla.bs` is the pricing engine function for Black-Scholes formula. The solution of call and put options are specified by Equation 1.4 and 1.5, respectively.

[4] i.e. logical value, which evaluated to either true or false

The implementation is simple, as we shall only “translate” the formula to R code. This can be done with the algorithm below:

```r
d1 <- bs.d1(S, K, t, r, q, sigma)
d2 <- d1 - sigma * sqrt(t)

if (type == "call") {
    price <- exp(-q * t) * S * pnorm(d1) - exp(-r * t) * K * pnorm(d2)
} else if (type == "put") {
    price <- exp(-r * t) * K * pnorm(-d2) - exp(-q * t) * S *
    pnorm(-d1)
}
```

Where d_1 is computed by the function:

```r
bs.d1 <- function(S, K, t, r, q, sigma) {
    (log(S / K) + (r - q + sigma^2 / 2) * t) / (sigma * sqrt(t))
}
```

Which is the excat same formula that is specified under Equation 1.5. The engine will then simply return the variable `price` as the output.

Notice that by definition seen in Section 1.5, only non-dividend bearing American call option can be priced using Black-Scholes formula as well. This is because under such case, the option isn't worthwile to be exercised earlier than maturity, so the resulted price will be the same as a European call with other arguments being identical. In other cases, American option's price cannot be expressed in a closed-form, and should thus be calculated using numerical methods.

To identify this issue and prevent users from pricing American style options wrongly, we add the following code to generate a warning message:

```r
if (style == "american") {
    if (q != 0 | type != "call") {
        warning("Black-Scholes formula not suitable, numerical
        evaluation should be used instead.")
    }
}
```

3.5.2 Binomial Lattice Tree

`vanilla.binomial` prices vanilla options via CRR's Binomial Lattice Tree method, as introduced in Section 1.5.

The R algorithm can be roughly divided into four segments: 1. compute the risk-neutral probability 2. generating binomial tree; 3. computing the payoff at the end step of the binomial tree; and 4. using backward recursive methods to compute the expected payoff of the option at current time under the risk-neutral assumption.

First, after extracting the information from input objects, we can compute the risk-neutral probability using Euqation 1.7. This can be translated to the following code:

```r
p <- (exp(r * dt) - d) / (u - d) # Compute riskless probability for an upward price movement
q <- 1 - p # Compute riskless probability for a downward price movement
```

Where variables `p` and `q` represents $\hat{p}$ and $\hat{q}$, respectively.

Second, to generate a binomial tree, we require four known parameters: the current underlying asset price S (or S_0), ratio of an upward jump u, ratio of an downward jump d, and the number of steps to compute n. By observing the Binomial tree through Graph 1.2, we see we can store all nodes of the tree into an $(n+1) \times (n+1)$ triangular matrix, as displayed in the following form:

$$\begin{array}{ccccc}
S & & & & \\
dS & uS & & & \\
d^2S & duS & u^2S & & \\
 & & \cdots & & \\
d^nS & d^{n-1}uS & d^{n-2}u^2S & \cdots & u^nS
\end{array}$$

Figure 3.2: Matrix Representing the Binomial Tree

As shown by Graph 3.2, each row of the lower triangular matrix represent a step of the binomial tree. To create such matrix shown in Figure 3.2, we can use the algorithm below:

```r
1  Binomial.tree <- function(S0, u, d, n) {
2      dim <- n + 1
3      S <- matrix(nrow = dim, ncol = dim) # Predefine the size
4      temp <- c(S0)
5      for (i in 1:dim) {
6          S[i, ] <- c(temp, rep(NA, dim - i))
7          temp <- c(temp[1] * d, temp * u)
8      }
9      S
10 }
```

We first build an empty matrix of the required size $(n+1) \times (n+1)$[5] and assign the current price S_0 to be the only element of row 1.

As we see from Graph 3.2, each row can be created via the upper row by: 1. Multiply the upper row's first element by the downward jump ratio d; 2. Multiply all elements (including the first) by an upward jump, and; 3. storing these new values into a vector which is 1 unit longer than the previous row.

We encoded the above procedure using the for-loop shown in code lines 5-8 above. Here, `temp` is a record of each "upper row's" non-NA elements in the above context, and `c(temp, rep(NA, dim - i))`

[5] In R, this is represented by a matrix containing only `NA`s. By defining the size of the matrix ahead allows us to speed up the computation.

allows us to fill each new row's length to $n+1$, so that we can directly update the rows in the predefined empty matrix.

A typical output of the `Binomial.tree` function is shown below:

```r
Binomial.tree(S0 = 8, u = 2, d = 0.5, n = 4)
>       [,1] [,2] [,3] [,4] [,5]
> [1,]  8.0   NA   NA   NA   NA
> [2,]  4.0   16   NA   NA   NA
> [3,]  2.0    8   32   NA   NA
> [4,]  1.0    4   16   64   NA
> [5,]  0.5    2    8   32  128
```

The third step is calculating the payoff at the terminal state of the binomial tree. In our setup, this would be the payoff calculuated using the prices shown in the last row of the matrix. Recall the payoff of different types of options form Section 1.5, we will use an if-else statement to decide which payoff function to use:

```r
if (type == "call") {
    fn <- sapply(Sn, function(x) {max(x - K, 0)}) # Payoff for call option
} else if (type == "put") {
    fn <- sapply(Sn, function(x) {max(K - x, 0)}) # Payoff for put option
}
```

Here, variable `Sn` is a vector storing the last row of the constructed price matrix, and `fn` would represent the corresponding final step's payoffs. This will be used as an initial vector of our next recursive price valuation.

The fourth and final step is to use risk-neutral valuation to compute the current payoff from the final steps payoffs. For each step backwards in time, the recursive formula is shown by Equation 1.8. A simple pseudocode of the function enconding this formula would be:

rnv ¡- function(step i payoffs) if (length of step i payoffs == 1) return result payoff else compute step i-1 payoffs from step i payoffs rnv(step i-1 payoffs)

For each layer of recurrsion, we will take the current step's payoff vector as imput. Using the Equation 1.8, the output of each recurrsion will be a vector that is 1 unit shorter than the input vector (meaning we are moving forward by one step in the binomial tree). When the length of the vector finally reduced to 1, we will be reaching step 0, or the current time under the binomial tree setup. The result singular payoff will be the desired current payoff, or the fair price of the target option.

The detailed implementation of the recursive method can be found in the package's source code under Appendix A.1.

Noticing that when the computer runs a recursive algorithm, the data will not be returned until the recurrsion reaches to an end. This means the memory useage will accumulate continuously as the recurrsing layers stacks deeper. Therefore, a binomial tree with steps $n = 1000$ will consume over 6GB of system memory and noticably long time to complete, and a binomial tree with steps $n = 10000$ cannot be executed in R as the number of recurrsing layers will exceed the default limit (and RAM capacity of many personal devices).

To demonstrate this effect better, an example code is presented under Section 4.1.2.

3.6 Monte Carlo Methods

3.6.1 Simulating Price Trajectories

The core of Monte Carlo option pricing is simulating the price movements of the underlying asset. Recall the computable form of the Black-Scholes model (Equation 1.9) and the pseudocode given, we can directly implement this in R as follows.

```r
# Setup arguments for calculation
t <- 2 # Expiration
n <- 100 # Number of trajectories to simulate
m <- 100 # Number of time steps per trajectory
S0 <- 20 # The current asset price
mu <- 0.05 # The drift coefficient
sigma <- 0.03 # The diffusion coefficient
dt <- t / m # The length of each time step
```

We will first directly implement the pseudocode in R, following Glasserman's framework [2].

```r
mc.for <- function() {
    S.mat <- matrix(nrow = m + 1, ncol = n)
    for (j in 1:n) {
        S <<- vector(length = m)
        S[1] <<- S0
        for (i in 2:(m + 1)) {
            Z <<- rnorm(1)
            S[i] <<- S[i - 1] * exp((mu + sigma^2 / 2) * dt + sigma *
            Z * sqrt(dt))
        }
        S.mat[, j] <<- S
    }
}
```

The resulting `S.mat` is a $(m + 1) \times n$ matrix, where each column records a simulated price trajectory from $t = 0$ to $t = T$ of $m + 1$ steps (including the the current stock price at time $t = 0$). We can plot the result trajectories using the `matplot()` function from the `graphics` package.

```r
matplot(S.mat, type = "l", col = rgb(0, 0, 0, 0.2),
        xlab = "dt", ylab = "price")
```

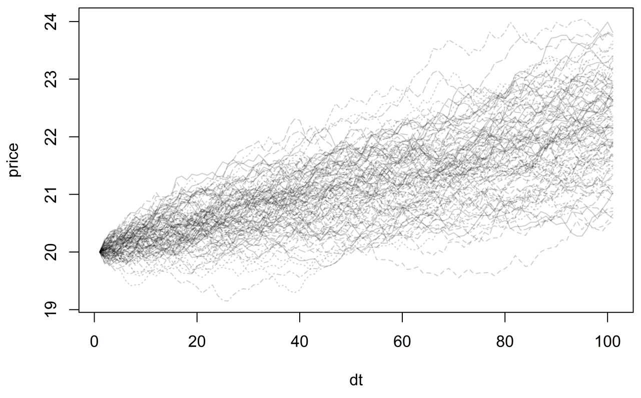


Figure 3.3: Monte Carlo Price Trajectories

From the result shown in Figure 3.3, we see an collectively upward movement for prices. This corresponds to our setup of a positive drift coefficient of 0.05.

However, the above code is not computationally efficient. R is relatively slow in running for-loops, and we can speed up the process by using matrix method.

Recall Equation 1.2, where we have the explicit form of $\log(S_t)$. Using similar discretisation method, we can use standard normal random variable Z to substitute W_t to form a recursive equation, such that:

$$\log(S_i) = \log(S_{i-1}) + (\mu - \frac{\sigma^2}{2})\Delta t + \sigma Z_i \sqrt{\Delta t}$$

Notice that assume Δt is fixed, the part $\log(\Delta S)_i := (\mu - \frac{\sigma^2}{2})\Delta t + \sigma Z_i \sqrt{\Delta t}$ is time independent. Therefore:

$$\log(S_{i+k}) = \log(S_{i-1}) + \sum_{j=i}^{k} \left[(\mu - \frac{\sigma^2}{2})\Delta t + \sigma Z_i \sqrt{\Delta t}\right] \tag{3.1}$$

$$= \log(S_{i-1}) + \sum_{j=i}^{k} \log(\Delta S_i) \tag{3.2}$$

For $i, k \in \mathbb{N}$ and $i + k \leq m$.

This allows us to reduce the two layers of for-loops into matrix operations.

```r
mc.mat <- function() {
    Z <<- matrix(rnorm(n * m), ncol = n)
    logdS <<- (mu - sigma^2 / 2) * dt + sigma * Z * sqrt(dt)
    logS <<- log(S0) + apply(logdS, 2, cumsum)
    S <<- exp(logS)
    S <<- rbind(S0, S) # Add back the missing row of starting point S0
}
```

Here, we first define `Z` to be a matrix that constains all standard normal random variables required for a total of $n \times m$ number of steps. Then we define `logdS` as the matrix of all increments. Based on Equation 3.2, by using $\log(S_0)$ as the starting point and use cumulative sum `cumsum` function to add increament $\log(\Delta S)$ along steps of each trajectory, we will obtain the movement of prices in time. By some further post process shown above, we will get the same result as using the for-loops.

To test the difference in computing speed between the two methods, we will test the average running time using the R package `microbenchmark`. The device configuration which the test runs on can be find in Section 1.4.

The `microbenchmark` function runs individual functions for 100 times and records the summary statistics of the running times, as shown below:

```r
microbenchmark::microbenchmark(
    mc.for(),
    mc.mat()
)

>     Unit: microseconds
>     expr     min      lq      mean   median      uq     max neval
> mc.for() 18327.2 19872.8 26257.504 25396.45 27853.6 60920.1   100
> mc.mat()   831.8   964.9  1200.425  1039.30  1094.5  8045.3   100
```

From the result we see that `mc.mat` is substantially faster than `mc.for`. We shoud thus prefer the former to obtain better estimation efficiency.

The matrix method can also be implemented via the form shown in Equation 1.9 using cumulative product `cumprod` function. However, the running time for single multiplication is slightly longer than addition. One may decide to use either methods considering redability and efficiency.

The full look of the `mc.engine` is shown below:

```r
mc.engine <- function(type, K, t, S0, r, q, sigma, n, steps) {

    dt <- t / steps

    # Generate n random samples from N(0,1)
    Z <- matrix(rnorm(n * steps), ncol = n)

    # Get logarithm of price change per step
    increment <- (r - q - sigma^2 / 2) * dt + sigma * sqrt(dt) * Z

    # Use vectorised MC method to compute logarithm of S(t) at each
    time step
    logS <- log(S0) + apply(increment, 2, cumsum)
    S <- exp(logS) # Obtain the simulated stock price by taking
    exponential
    S <- S * exp(-q * t) # Adjust the asset price for dividends
    S <- rbind(S0, S) # Add column of current price
    S
}
```

Notice that the argument specifying the number of time steps to simulate per trajectory, m, is now specified under the name `steps`.

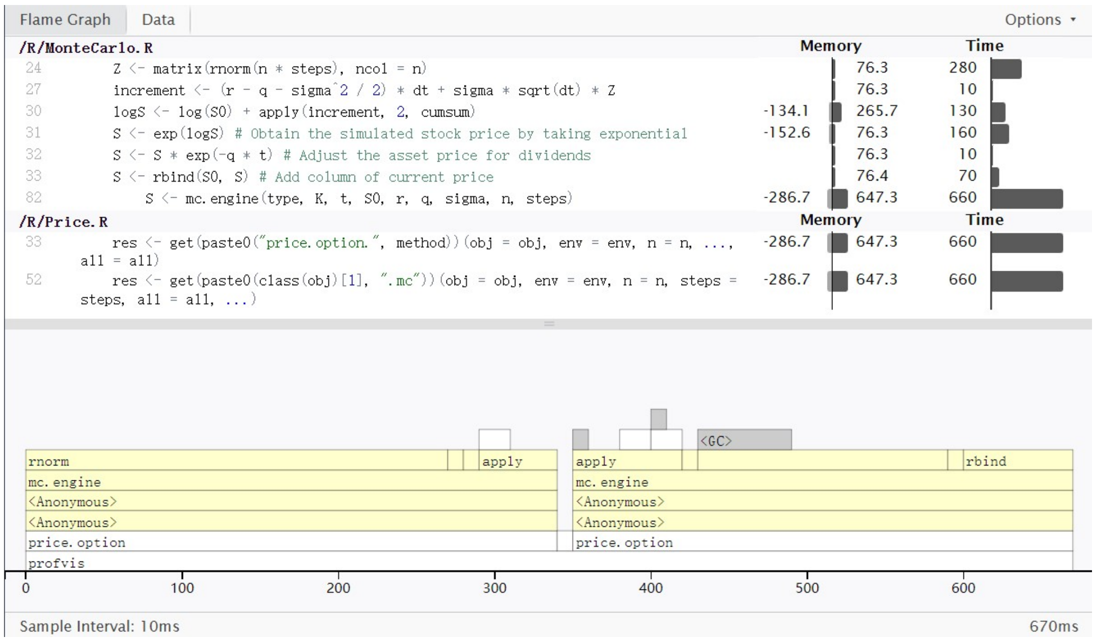
Here, the drift coefficient is decomposed to (fixed) annual interest rate r and (fixed) annual dividend yield rate q according to Equation 1.10. The number of time steps m is directly named as `steps`.

Adapting this formulation, `rmcop` has a function named `mc.engine`, which takes the mentioned input arguments and returns a $(m + 1) \times n$ matrix recording all generated trajectories. The rest of the functions described in the following subsections takes `mc.engine` as an internal component and use the returned matrix to perform further calculations of option payoff.

Using R package `profvis`, we can profile the run-time of the function `price.option` under a reasonably time-consuming Monte Carlo setup. Here we will simulate 10000 trajectories, each with 1000 time steps. The result trajectory matrix will contain $10000 \times 1000 = 10000000$ simulated prices. The code for benchmarking is shown below, and the output is presented in Figure 3.4.

```r
# Creating option and environment objects
op <- option("european", "vanilla", "call", 20, 0.75)
op.env <- option.env(S = 20, r = 0.03, q = 0.01, sigma = 0.05)

# Profiling
profvis::profvis(
    price.option(op, op.env, n = 10000, steps = 1000, all = FALSE,
    plot = FALSE)
)
```


```
Flame Graph   Data                                                      Options ▾
/R/MonteCarlo.R                                              Memory         Time
24        Z <- matrix(rnorm(n * steps), ncol = n)                         76.3    280
27        increment <- (r - q - sigma^2 / 2) * dt + sigma * sqrt(dt) * Z  76.3     10
30        logS <- log(S0) + apply(increment, 2, cumsum)           -134.1  265.7    130
31        S <- exp(logS) # Obtain the simulated stock price by taking exponential  -152.6  76.3  160
32        S <- S * exp(-q * t) # Adjust the asset price for dividends     76.3     10
33        S <- rbind(S0, S) # Add column of current price                 76.4     70
82            S <- mc.engine(type, K, t, S0, r, q, sigma, n, steps)  -286.7  647.3    660
/R/Price.R                                                   Memory         Time
33        res <- get(paste0("price.option.", method))(obj = obj, env = env, n = n, ...,   -286.7  647.3  660
    all = all)
52        res <- get(paste0(class(obj)[1], ".mc"))(obj = obj, env = env, n = n, steps =   -286.7  647.3  660
    steps, all = all, ...)

Sample Interval: 10ms                                                              670ms
```

Figure 3.4: Profiling output of Monte Carlo pricing function

The named strips in Figure 3.4, represents different components that are triggered when running `price.option`. Each function represented by the corresponding strip is the internal component of the one below it. Here, we see `mc.engine` is the most (and only) time consuming component of the pricing function. This means our prior adjustment of replacing for-loops with more efficient matrix operation is essensial for reducing the computing cost of the whole pricing procedure.

An additional component is added to calculate the Standard Error of estimation. The formula used is shown by Equation 1.12:

```r
mc.SE <- function(C_i, C.mean, n) {
    s_c <- sqrt(sum(C_i - C.mean)^2 / (n - 1))
    SE <- s_c / sqrt(n)
    SE
}
```

The resulting standard error can be used to construct confidence intervals of the estimate of the option price.

One should be aware that Monte Carlo option pricing for American options is beyond the scope of this report. In the pricing functions below, we will only consider the pricing of European options.

3.6.2 vanilla.mc

The pricing of Vanilla options is examplified in Equation 1.11. To obtain the simulated S_T's, one shall simply take the last row of the matrix output from the `mc.engine` and directly translate the formula to R code:

```r
if (type == "call") {
    C <- exp(-r * t) * pmax(S[steps + 1, ] - K, 0)
} else if (type == "put") {
    C <- exp(-r * t) * pmax(K - S[steps + 1, ], 0)
}
```

Where the resulting discounted payoffs for each trajectory is stored in the vector `C`.

Notice that for a path-independent option such as an Vanilla option, we are only considering the option price at expiration. Thus, it is reasonable to take `step = 1` to reduce computational cost.

3.6.3 asian.mc

Exotic options such as Asian options and those included below are path-dependent. This implies capturing the price movements throughout time $t \in [0, T]$ is necessary.

Recall from Section 1.5, an Asian option's payoff is revelent to the underlying asset's average price through $t = [0, T]$. Thus, we will first compute and store the average prices of each trajectory.

```r
S.mean <- apply(S, 2, mean)
```

The `apply` function with the second argument set to `2` allows us to apply the operation `mean` on each column of `S`, thus giving us the average price of each trajectory.

After obtaining the average, we will directly implement the payoff functions showed in List 1.5.1 in R as shown below:

```r
if (is.avg_price) { # Compute payoff for average price Asian option
    if (type == "call") {
        C <- exp(-r * t) * pmax(S.mean - K, 0)
    } else if (type == "put") {
        C <- exp(-r * t) * pmax(K - S.mean, 0)
    }
} else { # Compute payoff for average strike Asian option
    if (type == "call") {
        C <- exp(-r * t) * pmax(S[steps + 1, ] - S.mean, 0)
    } else if (type == "put") {
        C <- exp(-r * t) * pmax(S.mean - S[steps + 1, ], 0)
    }
}
```

Notice that for path dependent cases, using large enough number of steps for each simulation to better resemble the real-world scenario is necessary to obtain an accurate approximation.

3.6.4 barrier.mc

As we see from List 1.5.1, a barrier option is “switched on and off” depending on whether the price movements crossed some pre-defined price barrier. To record this pre-specified quantity, we require user to include a property named `barrier` when they define their option object. We also require users to specify a boolean argument named `is.knockout` to classify whether the barrier option is “knock-in” or “knock-out”. This is demonstrated by the below example:

```r
option(style = "european", option = "barrier", type = "call", K = 20,
t = 1, barrier = 21, is.knockout = TRUE)
```

Here, besides from the regular parameters to see in any option, the user defined the barrier price to be 21 and the option to be a “knock-out” barrier option, meaning that the option payoff will be zero if the price movement crossed the value 21.

To determine whether a price trajectory crossed certain level, we can record the minimum and maximum through the trajectory and see if the barrier value lies between two extremas. If so, because the price trajectory is continuous, we know that the trajectory must have crossed the barrier at least once through the option’s life time.

We will use similar `apply` method as mentioned in the previous section. Only now, we are computing the minimums and maximums of the trajectories. We then use a boolean vector `is.cross` to determine whether the barrier is bounded by the extremas.

```r
S.min <- apply(S, 2, min)
S.max <- apply(S, 2, max)
is.cross <- (S.min < barrier & S.max > barrier)
```

We know that a barrier option would behave exactly the same as a vanilla option if ignoring the “knock-in/out” effect, so we will first calculate its discounted payoff with the exact same methods in the above vanilla option pricing section.

```r
if (type == "call") {
    C <- exp(-r * t) * pmax(S[steps + 1, ] - K, 0)
} else if (type == "put") {
    C <- exp(-r * t) * pmax(K - S[steps + 1, ], 0)
}
```

Then, we take into account of the special effect and decide whether to zero the option payoff.

```r
if (is.knockout) { # Case for knock-out barrier option
    C <- ifelse(is.cross, 0, C)
} else { # Case for knock-in barrier option
    C <- ifelse(is.cross, C, 0)
}
```

The function `ifelse(condition, A, B)` is simply a quick command for the regular if-else statement. It execute expression `A` if `condition` is evaluated to `TRUE`, and execute `B` otherwise.

3.6.5 binary.mc

As we see from List 1.5.1, a binary option, unlike other options discussed in the report, has a fixed payoff. To record this characteristic, we require user to passin a value for argument `payout` when defining a binary option object, as shown in the example below:

```r
option(style = "european", option = "binary", type = "call", K = 20, t
= 1, payout = 5)
```

For each trajectory, to decide whether the payoff should be zero or the specified value, we simply compare the simulated price at maturity S_T with the strike price K. This characteristic also makes a regular binary option path-independent.

Since the payoff is fixed by the variable `payout`, for simplicity, we can first set the discounted payoff of every trajectory to be equal, and reduced the "turned-off" cases to zero afterwards:

```r
# Genearte a column vector with the same discounted payoff
C <- matrix(exp(-r * t) * payout, nrow = n, ncol = 1)

# Based on the specific type of the option, reduced corresponding rows
to zero based on the payoff formula for binary option
if (type == "call") {
    C <- ifelse(S[steps + 1, ] > K, C, 0)
} else if (type == "put") {
    C <- ifelse(S[steps + 1, ] < K, C, 0)
}
```

3.6.6 lookback.mc

A lookback option is another case of a path-dependent option. As we see from List 1.5.1, a fixed strike lookback's payoff is determined by comparing the price extramas and the strike price throughout the life time, and a floating strike lookback's payoff is by comparing the price extramas and the price at maturity. This makes the lookback option being obviously path-dependent.

To determine whether the pre-specified lookback option is fixed or floating, we require the user to passin a value for the boolean variable `is.fixed`, which indicates the option is "fixed strike" if evaluated to `TRUE`, and being "floating strike" otherwise. This is shown in the codes below:

```r
option(style = "european", option = "lookback", type = "call", K = 20,
t = 1, is.fixed = TRUE)
```

To allow us to calculate the payoff for each trajectory, we will first store the required value of minimum and maximum prices for each simulation, this could be done using the similar `apply` method as demonstrated above:

```r
S.min <- apply(S, 2, min)
S.max <- apply(S, 2, max)
```

And the vector containing the simulated prices at maturity can be done with the same `S[steps + 1, ]` command.

After obtaining these elements, based on the classification of lookback options, there are 4 scenarios for the payoff to be calculated. By directly implement the formulas as displayed in 1.5.1, we will obtain the vector `C` of discounted payoffs.

```r
if (type == "call") {
    if (is.fixed) { # Fixed strike
        C <- exp(-r * t) * pmax(S.max - K, 0)
    } else { # Float strike
        C <- exp(-r * t) * (S[steps + 1, ] - S.min)
    }
} else if (type == "put") {
    if (is.fixed) { # Fixed strike
        C <- exp(-r * t) * pmax(K - S.min, 0)
    } else { # Float strike
        C <- exp(-r * t) * (S.max - S[steps + 1, ])
    }
}
```

Chapter 4

Results

This Chapter will discuss the outcome of the programming implementation discussed in previous chapters. The "outcome" will be presented using numerous option pricing examples based on our package, as well as some comments on the effectiveness of the result.

The second part of the chapter will discuss the limitations of the package and potential directions of furture development.

4.1 Examples

4.1.1 Black-Scholes

4.1.2 Binomial Tree

The below example uses the binomial tree method to price an option undering an

4.1.3 Monte Carlo

4.2 Limitations and Future Development

Currently, `rmcop` only supports a few types of options. The derivative market is varying, and it is infesible for a single package to contain tools that can address all possible scenarios. One of the necessary future development of `rmcop` will be extending its support to more cases. This includes the support of pricing American and Bermudian styles options via Monte Carlo method, and the support of more types of exotic options pricing.

Though the development of the package includes several R-programming tricks to speed up the computation, one cannot guarantee the implementation here is the most efficient way of encoding the pricing methods. Also, as we discussed in Section 2, `rmcop` being an R-based package has fundamentally no computational advantage comparing to algorithms developed via lower level programming languages. The main focus of this pacakge is for demonstration of model implementations, and the C++ based programmes should be more desirable when conducting more practical and computationally-intensive option pricing projects.

Until the time when this report is completed, the development of `rmcop` is still active. One may found resources in Appendix A useful in terms of tracking future updates of the package.

Appendix A

Appendix 1

A.1 Package Source Code

Bibliography

[1] D. Higham, *An Introduction to Financial Option Valuation.* Cambridge University Press, 2004. Exotic Options Pricing.

[2] P. Glasserman, *Monte Carlo methods in financial engineering.* Springer, 2003. Explaination on Monte Carlo method, MC European vanilla option pricing, MC Asian option pricing.

[3] S. M. Iacus, *Stochastic Processes and Stochastic Differential Equations.* Springer New York, 2008. Stochastic Processes¡br/¿Brownian Motion.

[4] S. E. Shreve, *Stochastic calculus for finance. II, Continuous-time models.* Springer, 2004. Geometric Brownian Motion¡br/¿Ito Calculus.

[5] L. Bachelier, "Théorie de la spéculation," *Annales scientifiques de l'École normale supérieure*, vol. 17, pp. 21–86, 1900. Earliest attempt of quantitative method in option pricing.

[6] C. M. Sprenkle, "Warrant prices as indicators of expectations and preferences," *Yale economic essays*, vol. 1, pp. 179–232, 1961. Sprenkle Formula.

[7] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *The Journal of political economy*, vol. 81, pp. 637–654, 1973. Definition of Financial Options¡br/¿Black-Scholes Model.

[8] J. C. Cox, S. A. Ross, and M. Rubinstein, "Option pricing: A simplified approach," *Journal of financial economics*, vol. 7, pp. 229–263, 1979. Binomial Lattice Model.