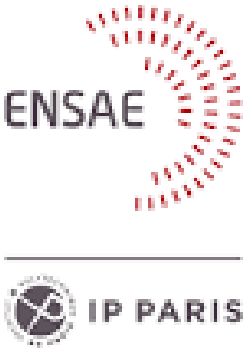
Black-Scholes-Merton and Monte Carlo Option Pricer

Authors:
Kacim Younsi
Denisa Draghia
Paul Wattellier
Mira Maamari

Teacher:
Roxana Dumitrescu

C++ Project Report

École Nationale de la Statistique et de l'Administration Économique



ENSAE Paris
Palaiseau, France

TABLE OF CONTENTS

HOW TO USE

The information in this part can be found in the README.txt file of the folder but is reported here for convenience.

Our code was mainly developed on Code::Blocks v20.03 which allowed us to free ourselves from the generation of a 'MAKEFILE', the Code Block Project folder is the "project" type file generated by Code Block.

However the file 'MAKEFILE.txt' has been written so that the code can be used by another user.

In case the MAKEFILE does not work, here is a link to access the code directly on internet without any installation needed : Online Code

1

CODE NOTATION

Here is a list of the parameters used in the code and their meaning:

- K : Strike price of the underlying asset
- S : Spot price of the underlying asset
- r : Interest rate
- *sigma* : Volatility of the underlying asset
- T : Time before maturity
- n : Number of step in the discretization before maturity
- dt : Time step when we discrete the time before maturity.
- $v0$: Initial value of the volatility
- *theta* : Average value of the volatility
- *sigma* : In StochasticEuropeanOption Class, this parameter is the volatility of the volatility
- $K1$: The trigger price of the gap option
- $T1$: The predefined date at which the holder can decide whether to exercise the call or the put
- *barrier* : the value of the barrier in barrier option
- *type*: type = 1 if the option is knock-IN and type= 0 if the option is knock-OUT
- *type*1 : type1 = 1 if the option is UP and type1= 0 if the option is DOWN

2

GLOBAL STRUCTURE OF THE CODE

The provided code is an implementation of various option pricing models in C++. It contains 3 files: a header OptionPricing.h where the class and function are declared, a source file where all the functions declared in the header are defined and a main where the code is tested through various example.

The code is class-oriented and here is a presentation of the architecture of the class : The parent class is Option, it contains all classic parameters that describe an option.

These parameters are stored as protected in the class. The class also contains getter and setter methods, that will be used to access or changed the parameters in the case of option with stochastic parameters.

```
class Option
{
protected:
    double S;       // Current price of the underlying asset
    double K;       // Strike price
    double r;       // Risk-free interest rate
    double sigma;   // Volatility of the underlying asset
    double T;       // Time to expiration (in years)

public:
    // Constructor
    Option(double S, double K, double r, double sigma, double T);
        virtual ~Option();
```

Figure 2.1: Declaration of Option Parent Class

Every class of the code that implement a type of Option inherit from this parent class because every vanilla and exotic options need almost all of these parameters.

For each type of Option we designed two sub-sub-classes for the put and the call .It should be noted that separating the two sub-sub-class for Put and Call is not mandatory and is done to make the code more readable and more convenient. It could have been avoided by adding a boolean parameter to identify whether the option is a put or a call. Moreover, this separation prevents the user from using a put as a call and vice versa.

Some types of Options need parameters to account for their specificity. These parameters are declared in their respective class so that each Option Class only contains the parameters it needs.

```
class EuropeanCall : public EuropeanOption
{
public:
    // Constructor
    EuropeanCall(double S, double K, double r, double sigma, double T);
        ~EuropeanCall();

    // Price a European call option
    double price();

    // Replication strategy
    void replicate();
    };

//***********************************
//  European Put
//***********************************
class EuropeanPut : public EuropeanOption
{
public:
    // Constructor
    EuropeanPut(double S, double K, double r, double sigma, double T);
        ~EuropeanPut();

    // Price a European put option
    double price();

    // Replication strategy
    void replicate();
};
```

Figure 2.2: Declaration of two subsubclass for pall and cut.

Since the goal of the library is to price different type of options, the principal function present in each sub-sub-class is a pricing function. In our code there are two types of pricing functions:

- Closed formula pricer : Only used with classic European Option, based on Black-Scholes Merton model
- Monte Carlo method pricer: When closed formula does not exist to compute the expected value of the payoff we simulate multiple scenario and average them to estimate the expected value of the payoff. This is simply done by a loop.

Here is an example of SubClass with specific parameters and a pricing function using Monte-Carlo method. In this case the parameter n is a discretisation parameter that will be detailed later on.

```cpp
class AsianOption : public Option
{
private:
    int n;  // Number of time steps
    double dt; // Time step size
public:
    // Constructor
    AsianOption(double S, double K, double r, double sigma, double T, int n);
        ~AsianOption();
    double getn();
    double getdt();
};

//***********************************
//  Asian Call Option
//***********************************

class AsianCall : public AsianOption
{
public:
    // Constructor
    AsianCall(double S, double K, double r, double sigma, double T, int n);
        ~AsianCall();
    // Function to price an Asian call option using Monte Carlo methods
    double price(int num_simulations);
};
```

Figure 2.3: Declaration of SubClass with specific parameters and pricing function via Monte Carlo method.

3

VANILLA OPTIONS

We start by pricing the plain vanilla European options. To do so, we use 2 methods: Black-Scholes-Merton formula and Monte Carlo simulation. The function *price* returns the price using the first formula, whereas price_MonteCarlo (num_simulation,n) returns the value obtained using (num_simulation) Monte Carlo simulations. The price of the underlying is computed on a finite number of date determined by the parameter n.

```
class EuropeanCall : public Option
{
public:
    // Constructor
    EuropeanCall(double S, double K, double r, double sigma, double T);
        ~EuropeanCall();

    // Price a European call option using BS formula
    double price();

    // Price a European put option using Monte Carlo simulation
    double price_MonteCarlo(int,int);

    //Price difference between the BS price and the Monte Carlo one for n simulations
    double difference(int,int);


    // Replication strategy
    void replicate();

    };
```

Figure 3.1: European Call Subclass

The next code shows the Monte Carlo simulations we used to price an European Call Option.

```
//Price a European call option by Monte Carlo Simulation

double EuropeanPut::price_MonteCarlo(int num_simulations,int n) {
        double sum = 0;
        double S_t;

        double payoff;
        for (int i = 0; i < num_simulations; i++) {
            S_t = getS();

        for (int j = 0; j < 12; j++) {
            S_t = S_t * exp((getR()-0.5 * getSigma() * getSigma()) *getT()/n + getSigma()* sqrt(getT()/n) *gaussian_box_muller());

            }
            payoff = std::max(-S_t + getK(), 0.0);
            sum += payoff;
        }
        return (sum / num_simulations) * exp(-getR() * getT());
    }
```

Figure 3.2: Monte Carlo Simulation for an European Call

The function difference computes the difference between the prices calculated using the two methods.

```
double EuropeanCall::difference(int num_simulations,int n){

    return abs(price()-price_MonteCarlo(num_simulations,n));};
```

Figure 3.3: Difference function

We notice that when the number of simulation increases, the difference decreases which is coherent with the theory.

```
Difference of prices for 6000 simulations is 0.0835121
```

Figure 3.4: Difference function.1

REPLICATION

Sometimes a portfolio manager needs a certain type of option with specific characteristics that are not available in the market, or are too expensive. Other times, the market is not liquid enough. What she or he can do to replicate the option is to take a certain position in the underlying and invest a certain amount in a risk-free asset (we considered bonds).

```
// Replication strategy
void EuropeanCall::replicate() {
    double num_shares = norm_cdf(d1()); // Number of shares of the underlying asset
    double num_bonds = exp(-getR() * getT()) * norm_cdf(d2()); // Number of risk-free bonds
    std::cout <<"To replicate a European call option, hold "<< num_shares << " shares of the underlying asset and "
    << num_bonds << " risk-free bonds." << std::endl;
    }
```

Figure 3.5: Replication of European Call

```
To replicate a European call option, hold 0.449648 shares of the underlying
asset and 0.353861 risk-free bonds.
```

Figure 3.6: Replication of European Call.1

THE GREEKS

In order to estimate the sensibility of the option price to changes in factors such as the price of the underlying, the volatility, the interest rate, we computed the Greeks.

```
// Function to calculate the delta of the option
double Option::delta() { return norm_cdf(d1()); }

// Function to calculate the gamma of the option
double Option::gamma() { return norm_pdf(d1()) / (S * sigma * sqrt(T)); }

// Function to calculate the theta of the option
double Option::theta() {
    return (-S * norm_pdf(d1()) * sigma) / (2 * sqrt(T)) - (r * K * exp(-r * T) * norm_cdf(d2()));
}

// Function to calculate the vega of the option
double Option::vega() { return S * sqrt(T) * norm_pdf(d1()); }

// Function to calculate the rho of the option
double Option::rho() { return K * T * exp(-r * T) * norm_cdf(d2()); }
```

Figure 3.7: The Greeks

4

EXOTIC OPTIONS

In this part, we will detail the pricing of exotic options. Although, we have mostly used the Monte Carlo simulation, in some cases we were able to price the options using a slightly changed Black-Scholes-Merton formula.

We priced Asian, Loockback, Binary, Gap, Chooser and Barrier Options.

ASIAN OPTION

An Asian Option is an option where the payoff is based on the average price of an underlying asset over a set period of time, rather than the asset's price at a single point in time. This type of option is commonly used in options trading to manage the risk associated with volatile underlying assets. In our code the average price is computed on a finite number of date determined by the parameter n.

```
double AsianCall::price(int num_simulations) {
        double sum = 0;
        double S_t;
        double average;
        double payoff;
        for (int i = 0; i < num_simulations; i++) {
            S_t = getS();
            average = 0;
            for (int j = 0; j < getn(); j++) {
                S_t = S_t * exp((getR() - 0.5 * getSigma() * getSigma()) * getdt() + getSigma() * sqrt(getdt()) * gaussian_box_muller());
                average += S_t;
            }
            average /= getn();
            payoff = std::max(average - getK(), 0.0);
            sum += payoff;
        }
        return (sum / num_simulations) * exp(-getR() * getT());
    }
```

Figure 4.1: Asian Option pricing

LOOKBACK OPTION

A lookback option is a type of Option that allows the holder to choose the best price of an underlying asset over a specified period of time as the strike price for the option. For a call lookback option, the chosen price would be the lowest price of the underlying asset over the specified period of time. For a put lookback option, the chosen price would be the highest price of the underlying asset over the specified period of time. To determine this chosen price we evaluate the price on a finite number of date, similarly as the Asian Option pricing. The parameter that accounts for this number of date is again n.

```
double LookbackPut::price(int num_simulations) {
        double sum = 0;
        double S_t;
        double payoff;
        for (int i = 0; i < num_simulations; i++) {
            S_t = getS();
            double Min_S = getS();
            double Max_S = getS();
            for (int j = 0; j < getn(); j++) {
                S_t = S_t * exp((getR() - 0.5 * getSigma() * getSigma()) * getdt() + getSigma() * gaussian_box_muller());
                Min_S = std::min(Min_S, S_t);
                Max_S = std::max(Max_S, S_t);
            }
            payoff = std::max(Max_S - S_t, 0.0);
            sum += payoff;
        }
        return (sum / num_simulations) * exp(-getR() * getT());
    }
```

Figure 4.2: Lookback Option pricing

BINARY OPTION

A binary option can be cash-or-nothing or asset-or-nothing. We analyse a cash-or nothing option that pays a fixed amount (we choose it to be equal to 1 for simplicity) if the option finishes in-the-money. To check this condition we use the fonction *heaviside*.

```
//Function Heaviside that return 1 if its argument is positive and 0 otherwise

double heaviside(const double& x) {
  if (x >= 0) {
      return 1.0;
  } else {
    return 0.0;
  }
}
```

Figure 4.3: Function Heaviside

The declaration of the class BinaryOption and of Subclasses BinaryCall and Binary-Put are similar to those of Asian Options presented in figure 2.3. We priced the Binary Option using Monte Carlo methods, in the same way used for Asian options. We simulated multiple paths, found the payoff for each one and discount the averaged payoff to the present value.

```
// Function to price an Binary call option using Monte Carlo methods
double BinaryCall::price(int num_simulations) {
    double sum = 0;
    double S_t;
    double payoff;
    for (int i = 0; i < num_simulations; i++) {
        S_t = getS();
        for (int j = 0; j < 12; j++) {
            S_t = S_t * exp((getR()-0.5 *getSigma() * getSigma())* getT()/12+getSigma()*sqrt(getT()/12) * gaussian_box_muller());
            }

            payoff = heaviside(S_t - getK());
            sum += payoff;
        }
        return (sum / num_simulations) * exp(-getR() * getT());
    }
```

Figure 4.4: Pricing of a Binary Call

GAP OPTION

A Gap Option inherits the Option class. Its specificity is the second strike price K_1 that needs to be reached for the option to have a positive payoff. A Gap Call option pays

its usual payoff (S-K) when its final price is bigger than the second strike price K_1. Its theoretical price is the sum of the price of an European call option with a strike price of K_1 (calculated using BSM formula) and (K_1-K) e^{-rT} N(d_2).

```
// Constructor
GapCall::GapCall(double S, double K, double r, double sigma, double T, double K1_2) :
    Option(S, K, r, sigma, T), K1(K1_2) {}
GapCall::~GapCall() {}

//Getter methods
double GapCall::getK1(){return K1;};

//Function to price a Gap Call option
double GapCall::price()
{return EuropeanCall(getS(),getK1(),getR(),getSigma(),getT()).price()+(getK1()-getK())*exp(-getR()*getT())*norm_cdf(d2());};
```

Figure 4.5: Pricing of a Gap Call Option

As for a put, it pays its usual payoff (K_1-S_T) when its final price is smaller than the second strike price K_2. Its theoretical price is K_1e^{-rT} N(-d_2) - $S_0N(-\mathrm{d}_1)$.

```
//**********************************
//  Gap Put Option
//**********************************

// Constructor
GapPut::GapPut(double S, double K, double r, double sigma, double T, double K1_2) :
    Option(S, K, r, sigma, T), K1(K1_2) {}
GapPut::~GapPut() {}

//Getter methods
double GapPut::getK1(){return K1;};
// Function to price a Gap Put option
double GapPut::price() { return getK1()*exp(-getR()*getT())*norm_cdf(-d2())-getS()*norm_cdf(-d1());};
```

Figure 4.6: Pricing of a Gap Put Option

CHOOSER OPTION

A Chooser Option inherits the Option class. Its specificity is the second time T_1, when one can decide whether to exercise a call or a put option. Its theoretical price is the sum of a call with a maturity of T-T_1 and a put with a strike of K$e^{-r(T-T_1)}$ and a maturity of T-T_1.

```
// Constructor
ChooserOption::ChooserOption(double S, double K, double r, double sigma, double T, double T12) :
    Option(S, K, r, sigma, T), T1(T12) {}
ChooserOption::~ChooserOption() {}

// Getter methods
double ChooserOption::getT1(){return T1;};

// Function to price an Chooser option
// The price of a chooser option is the price of an European Call Option with maturity of T-T1+
//the price of an European Put option with a strike of K*exp(-r(T-T1)) and a maturity of T-T1

double ChooserOption::price()
{return EuropeanCall(getS(),getK(),getR(),getSigma(),getT()-getT1()).price()
+EuropeanPut(getS(), getK()*exp(-getR()*(getT()-getT1())),getR(),getSigma(),getT()-getT1()).price();};
```

Figure 4.7: Pricing of a Chooser Option

BARRIER OPTION

Barrier options are options where the payoff depends on whether the underlying asset's price reaches a certain level during a certain period of time. A Barrier Option inherits the Option class. It can be a knock-in if *type=1* or knock-out if *type=0*

```
class BarrierOption : public Option
{
protected:
    int type;// type=1 if the option is knock-in and type=0 if the option is knock-out
    double n;
    double dt;
    double barrier; // the value of the barrier
```

Figure 4.8: Barrier Option Class

A barrier option can be either up (*type1=1*, if the barrier is superior to the initial spot price) or down (*type1=0*, if the barrier is inferior to it). A knock-in option pays the ordinary payoff if a certain barrier has been reached and 0 otherwise. A knock-out option pays the ordinary payoff if a certain barrier has NOT been reached and 0 otherwise.

```
class BarrierCall: public BarrierOption
{private:
int type_call_1; // type_call_1=1 if the option is UP-and- and
//type_call_1=0 if the option is DOWN-and-
```

Figure 4.9: Barrier Call SubClass

There are 8 types of option: 4 calls (in-and-down, in-and-up, out-and-in, out-and-up) and 4 puts. We price them using Monte Carlo simulation as we did for Asian Options.

```
{if ((getType()==1)&&(getType1()==1)) // The call is up-and-in
{if (getbarrier()<getS()) {std::cout<<"Wrong parameters ";
    return -1;
}
else {double sum=0;
        double S_t;
        double payoff;
        int kick=0;
        for (int i = 0; i < num_simulations; i++) {
            S_t = getS();

            for (int j = 0; j < getn(); j++) {
                S_t = S_t * exp((getR() - 0.5 * getSigma() * getSigma()) * getdt() + getSigma() * gaussian_box_muller());
                if (S_t>getbarrier()){kick+=1;}
            }
            if (kick==0){payoff =0;}
            else payoff=std::max(S_t-getK(), 0.0);
            sum += payoff;
        }
        return (sum / num_simulations) * exp(-getR() * getT());
    }}
```

Figure 4.10: Pricing of an up-and-in Barrier Call

5

VANILLA OPTIONS WITH STOCHASTIC PARAMETERS

In the standard Black-Scholes-Merton (BSM) model, the volatility and interest rate are assumed to be constant over the life of the option. However, in reality, these parameters can change over time, which can affect the option's price. To account for this, one can use one variant of the BSM model that allows for stochastic volatility and interest rates.

In a model with stochastic volatility, the volatility of the underlying asset is not constant but follows a certain probability distribution. This means that the option price will depend not only on the current volatility but also on the volatility's future evolution. To price options under this model, one can use numerical methods like Monte Carlo simulation or finite difference methods.

Similarly, in a model with stochastic interest rates, the risk-free rate is not constant but follows a certain probability distribution. This means that the option price will depend not only on the current interest rate but also on the interest rate's future evolution. To price options under this model, one can use numerical methods like Monte Carlo simulation or tree methods.

In general, pricing options with stochastic parameters can be more complex and computationally intensive than pricing options with constant parameters. However, these models can provide more accurate prices for options, especially when the underlying asset exhibits significant volatility or interest rate changes over time.

In our code this change is simply implemented by computing the volatility and interest rate at each loop iteration and then computing the underlying price.

```
double StochasticEuropeanCall_volatility ::price(int num_simulations) {
        double sum = 0;
        double S_t;
        double v_t;
        double payoff;
        for (int i = 0; i < num_simulations; i++) {
            S_t = getS();
            v_t = getV0();
            for (int j = 0; j < getn(); j++) {
                S_t = S_t * exp((getR() - 0.5 * v_t) * getdt() + sqrt(v_t) * gaussian_box_muller());
                v_t = v_t + getk() * (getTheta() - v_t) * getdt() + getSigma() * gaussian_box_muller();
            }
            payoff = std::max(S_t - getK(), 0.0);
            sum += payoff;
        }
        return (sum / num_simulations) * exp(-getR() * getT());
    }
```

Figure 5.1: Implementation of Option with stochastic parameters

The implementation of the put and call for stochastic interest rate is very similar, that is why we won't detail it here.

In this code *theta* stand for the average value of the volatility while k is the mean reversion speed of the volatility and *sigma* is the volatility of the volatility.

REFERENCES

1. Hull, J Options, Futures and Other Derivatives. 9th Edition 2014
2. Raimonda Martinkutė-Kaulienė, Exotic Options: a Chooser Option and its Pricing. 2012
3. Patrick Boyle,Jesse McDougall, Trading and Pricing Financial Derivatives. 2019
4. Daniel J. Duffy, Introduction to C++ for Financial Engineers: An Object-Oriented Approach (The Wiley Finance Series). 2006