FE 545 - Homework 1: Option Pricing C++ Code and Output

Amod Lamichhane

February 11, 2025

Option Parameters

- **Time to Maturity:** 1 year
- **Current Stock Price (Spot):** 50.0
- **Lower Strike Price (k1):** 43.0
- **Upper Strike Price (k2):** 57.0
- **Volatility:** 0.30
- **Risk-Free Interest Rate (r):** 0.05
- **Number of Monte Carlo Paths:** 200

Final Output

```
The price of the Call option is: 7.7523
The price of the Put option is: 4.33214
Double Digital Option Price: 0.342443
```

C++ Code

Main Program (DDMain.cpp)

```cpp
//DDMain.cpp

#include <iostream>
#include <vector>

#include "PayOff.h"
#include "Random.h"
#include "SimpleMC.h"
#include "DoubleDigital.h"

int main() {
    // Define option parameters
    double Expiry = 1.0;      // Time to expiration (in years)
    double Spot = 50.0;       // Current stock price
    double k1 = 43.0;         // Lower strike price for Double
        Digital option
    double k2 = 57.0;         // Upper strike price for Double
        Digital option
    double Vol = 0.30;        // Volatility (sigma)
    double r = 0.05;          // Risk-free interest rate
    unsigned long NumberOfPath = 200; // Number of Monte Carlo
        simulation paths

    // Create parameter object for the double digital option
    DoubleDigitalPayOffParameters PayOffParams(k1, k2);

    // Initialize PayOffDoubleDigital object with defined
        parameters
    PayOffDoubleDigital PayOff(PayOffParams);

    // Run Monte Carlo simulation to estimate the option price for
        Double Digital
    double price = SimpleMonteCarlo(PayOff, Spot, Vol, r, Expiry,
        NumberOfPath);

    // Call Option Parameters
    PayOffParameters callParams(Spot);  // Strike price of 50 for
        the Call option
    PayOffCall callPayOff(callParams);
    double callResult = SimpleMonteCarlo(callPayOff, Spot, Vol, r,
        Expiry, NumberOfPath);
    std::cout << "The price of the Call option is: " << callResult
        << std::endl;

    // Put Option Parameters
    PayOffParameters putParams(Spot);  // Strike price of 50 for
        the Put option
    PayOffPut putPayOff(putParams);
    double putResult = SimpleMonteCarlo(putPayOff, Spot, Vol, r,
        Expiry, NumberOfPath);
    std::cout << "The price of the Put option is: " << putResult <<
         std::endl;

```

```
    // Output the calculated Double Digital option price
    std::cout << "Double Digital Option Price: " << price << std::
        endl;

    return 0;
}
```

DoubleDigital.cpp

```cpp
// DoubleDigital.cpp

#include "DoubleDigital.h"

// Constructor: Initializes lower and upper strike prices
DoubleDigitalPayOffParameters::DoubleDigitalPayOffParameters(const double& K_1, const double& K_2)
    : K1(K_1), K2(K_2) {}

// Getter for lower strike price
double DoubleDigitalPayOffParameters::GetLowerStrike() const {
    return K1;
}

// Getter for upper strike price
double DoubleDigitalPayOffParameters::GetUpperStrike() const {
    return K2;
}

// Constructor: Initializes payoff object with strike boundaries
PayOffDoubleDigital::PayOffDoubleDigital(const DoubleDigitalPayOffParameters& Param_)
    : K1(Param_.GetLowerStrike()), K2(Param_.GetUpperStrike()) {}

// Payoff function: Returns 1 if the spot price is within the strike boundaries, otherwise 0
double PayOffDoubleDigital::operator()(const double& S) const {
    return (S >= K1 && S <= K2) ? 1.0 : 0.0;
}
```

DoubleDigital.h

```cpp
// DoubleDigital.h

#ifndef __PayOffDoubleDigital__
#define __PayOffDoubleDigital__

#include "PayOff.h"

// Class representing parameters for a double digital option
class DoubleDigitalPayOffParameters : public BasePayOffParameters {
public:
    // Constructor: Initializes lower and upper strike prices
    DoubleDigitalPayOffParameters(const double& K_1, const double& K_2);
    virtual ~DoubleDigitalPayOffParameters() {}; // Destructor

    // Getter functions for strike prices
    double GetLowerStrike() const;  // Returns lower strike price
    double GetUpperStrike() const;  // Returns upper strike price

private:
    double K1; // Lower strike price
    double K2; // Upper strike price
};

// Class representing the payoff structure for a double digital option
class PayOffDoubleDigital : public PayOff {
public:
    // Constructor: Initializes payoff structure using given parameters
    PayOffDoubleDigital(const DoubleDigitalPayOffParameters& Param_);
    virtual ~PayOffDoubleDigital() {}; // Destructor

    // Overloaded operator(): Determines payoff based on spot price
    virtual double operator()(const double& S) const override; // Payoff function

private:
    double K1; // Lower strike price
    double K2; // Upper strike price
};

#endif
```

PayOff.cpp

```cpp
#include "PayOff.h"

// Constructor for BasePayOffParameters (empty)
BasePayOffParameters::BasePayOffParameters() {}

// Constructor for PayOffParameters, initializes K (strike price)
PayOffParameters::PayOffParameters(const double& K) : K(K) {}

// Getter for strike price
double PayOffParameters::GetStrike() const {
    return K;
}

// Constructor for PayOff (abstract class)
PayOff::PayOff() {}

// Constructor for PayOffCall, initializes strike price from PayOffParameters
PayOffCall::PayOffCall(const PayOffParameters& Param_) : K(Param_.GetStrike()) {}

// Constructor for PayOffPut, initializes strike price from PayOffParameters
PayOffPut::PayOffPut(const PayOffParameters& Param_) : K(Param_.GetStrike()) {}

// Payoff for Call option: max(Spot - Strike, 0)
double PayOffCall::operator()(const double& S) const {
    return std::max(S - K, 0.0); // Call option payoff is max(Spot - Strike, 0)
}

// Payoff for Put option: max(Strike - Spot, 0)
double PayOffPut::operator()(const double& S) const {
    return std::max(K - S, 0.0); // Put option payoff is max(Strike - Spot, 0)
}
```

PayOff.h

```cpp
// Header guard to prevent multiple inclusions
#ifndef __PAY_OFF__
#define __PAY_OFF__

#include <algorithm> // This is needed for the std::max comparison function, used in the pay-off calculations

// Abstract base class for PayOff options
class PayOff {
public:
    PayOff(); // Default constructor
    virtual ~PayOff() {}; // Virtual destructor
    virtual double operator()(const double& S) const = 0; // Pure virtual function for PayOff calculation
};

// Base class for PayOff parameters
class BasePayOffParameters {
public:
    BasePayOffParameters(); // Default constructor
    virtual ~BasePayOffParameters() {}; // Virtual destructor
};

// Derived class for PayOff parameters, stores the strike price
class PayOffParameters : public BasePayOffParameters {
public:
    PayOffParameters(const double& K); // Constructor with strike price
    virtual ~PayOffParameters() {}; // Virtual destructor
    double GetStrike() const; // Function to get the strike price
private:
    double K; // Strike price
};

// Class representing the PayOff for a Call option
class PayOffCall : public PayOff {
private:
    double K; // Strike price
public:
    PayOffCall(const PayOffParameters& Param_); // Constructor with PayOffParameters
    virtual ~PayOffCall() {}; // Virtual destructor
    virtual double operator()(const double& S) const; // Call option pay-off function
};

// Class representing the PayOff for a Put option
class PayOffPut : public PayOff {
private:
    double K; // Strike price
public:
    PayOffPut(const PayOffParameters& Param_); // Constructor with PayOffParameters
    virtual ~PayOffPut() {}; // Virtual destructor
    virtual double operator()(const double& S) const; // Put option
```

```
            pay-off function
50  };
51
52  #endif /* defined(__PAY_OFF__) */
```

Random.cpp

```cpp
// Random.cpp

#include "Random.h"
#include <cstdlib>
#include <cmath>

#if !defined(_MSC_VER)
using namespace std;
#endif

// Generate a Gaussian random variable using the summation method
double GetOneGaussianBySummation()
{
    double result = 0;

    for(unsigned long j = 0; j < 12; j++)
        result += rand() / static_cast<double>(RAND_MAX);

    result -= 6.0;

    return result;
}

// Generate a Gaussian random variable using the Box-Muller transform
double GetOneGaussianByBoxMuller()
{
    double result;
    double x, y;
    double sizeSquared;

    do
    {
        // Generate two independent random variables in the range [-1, 1]
        x = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
        y = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
        sizeSquared = x * x + y * y;
    }
    while (sizeSquared >= 1.0);

    // Box-Muller transform: generate a Gaussian random variable
    result = x * sqrt(-2 * log(sizeSquared) / sizeSquared);

    return result;
}
```

Random.h

```
1  //Random.h
2
3  #ifndef __Random_Class__
4  #define __Random_Class__
5
6  double GetOneGaussianBySummation();
7  double GetOneGaussianByBoxMuller();
8
9  #endif /* defined(__Random_Class__) */
```

SimpleMC.cpp

```cpp
// SimpleMC.cpp

#include <cmath>
#include "SimpleMC.h"
#include "Random.h"

double SimpleMonteCarlo(const PayOff& ThePayOff, double Spot,
    double Vol, double r, double Expiry, unsigned long
    NumberOfPaths)
{
    // Calculate the variance of the asset price over the expiry
        period
    double variance = Vol * Vol * Expiry;

    // Calculate the square root of the variance for later use
    double rootVariance = sqrt(variance);

    // Adjust the spot price for the drift (r - 0.5 * variance)
        over the expiry period
    double movedSpot = Spot * exp((r - 0.5 * variance) * Expiry);

    // Variable to accumulate the sum of all payoffs
    double sum = 0.0;

    // Loop over the number of paths (simulations)
    for (unsigned long i = 0; i < NumberOfPaths; i++)
    {
        // Generate a random number from a Gaussian distribution
            using the Box-Muller method
        double gaussian = GetOneGaussianByBoxMuller();

        // Simulate the final spot price by applying the random
            Gaussian to the adjusted spot price
        double thisSpot = movedSpot * exp(rootVariance * gaussian);

        // Calculate the payoff for this simulated spot price
        double thisPayOff = ThePayOff(thisSpot);

        // Add the payoff to the total sum
        sum += thisPayOff;
    }

    // Discount the average payoff by the risk-free rate and return
         the result
    return exp(-r * Expiry) * (sum / NumberOfPaths);
}
```

SimpleMC.h

```
//SimpleMC.h

#ifndef __Option_Class__SimpleMonteCarlo__
#define __Option_Class__SimpleMonteCarlo__

#include <iostream>
#include "DoubleDigital.h"

double SimpleMonteCarlo(const PayOff& ThePayOff_,
                        double Spot,
                        double Vol,
                        double r,
                        double Expiry,
                        unsigned long NumberOfPaths);

#endif /* defined(__Option_Class__SimpleMonteCarlo__) */
```