Sid Bhatia ~ FE545 - Homework #2 (PDF)

March 7, 2024

0.0.1 FE545 - Homework #2

Author: Sid Bhatia

Date: March 5th, 2023

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

Professor: Steve Yang

An **Asian option** is a type of *exotic option*. Unlike a vanilla European option where the price of the option is dependent upon the price of the underlying asset at expiry, an Asian option pay-off is a function of multiple points up to and including the price at expiry. Thus it is "path-dependent" as the price relies on knowing how the underlying behaved at certain points before expiry. Asian options in particular base their price off the mean average price of these sampled points. To simplify the problem, we will consider equally distributed sample points beginning at time and ending at maturity,

In this problem, we will consider Geometric mean A of the spot prices we use the following formula:

$$A(0,T) = \exp\left(\frac{1}{N}\sum_{i=1}^{N}\log(S(t_i))\right)$$

Unlike in the vanilla European option Monte Carlo case we have learned in class, where we only needed to generate multiple spot values at expiry, we now need to generate multiple spot paths, each sampled at the correct points. Thus, instead of providing a double value representing spot to our option, we now need to provide a `std::vector<double>` (i.e. a vector of double values), each element of which represents a sample of the spot price on a particular path. We will still be modeling our asset price path via a Geometric Brownian Motion (GBM), and we will create each path by adding the correct drift and variance at each step in order to maintain the properties of GBM.

Implement **PayOff classes** according to the following template:

```
#ifndef __PAY_OFF__
#define __PAY_OFF__

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

class PayOff {
    public:
```

```
        PayOff(); // Default (no parameter) constructor
        virtual ~PayOff() {}; // Virtual destructor
        // Overloaded () operator, turns the PayOff into an abstract
        // function object
        virtual double operator(const double& S) const = 0;
};

class PayOffCall : public PayOff {
    public:
        PayOffCall(const double& K_);
        virtual ~PayOffCall() {};
        // Virtual function is now over-ridden (not pure-virtual anymore)
        virtual double operator(const double& S) const;

    private:
        double K; // Strike price
};

class PayOffPut : public PayOff {
    public:
        PayOffPut(const double& K_);
        virtual ~PayOffPut() {};
        virtual double operator(const double& S) const;
    private:
        double K; // Strike
};

#endif
```

For this assignment, you need to define a base pure abstract class called `AsianOption` and a derived class called `AsianOptionGeometric` which implements the **PayOff operator** according to formula (1).

```
#ifndef __Asian_Option__
#define __Asian_Option__

#include <vector>
#include "PayOff.h"

class AsianOption {
    public:
        AsianOption(PayOff* _pay_off);
        virtual ~AsianOption() {};
        // Pure virtual pay-off operator (this will determine arithmetic or geometric
        double OptionPayOff(const std::vector<double>& spot_prices);

    protected:
        PayOff* pay_off; // Pay-off class (in this instance call or put)
};
```

```
class AsianOptionGeometric : public AsianOption {
    public:
        AsianOptionGeometric(PayOff* _pay_off);
        virtual ~AsianOptionGeometric() {};
        // Overide the pure virtual function to produce geometric Asian Option
        virtual double OptionPayOff(const std::vector<double>& spot_prices)
};

#endif
```

Please add a function in the `Random.h` and `Random.cpp` files which generates a Geometric Brownian Motion path according to the following formula:

$$S(t_i) = S(t_{t-1}) \exp[(r - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\epsilon]$$

The price path can be generated recursively from S_0 from above equation (2). The function prototype should follow the following structure with an input argument `std::vector<double>&`

```
#ifndef __Option_Class__Random__
#define __Option_Class__Random__

double GetOneGaussianByBoxMuller();

void GetGBMSpotPricePath(std::vector<double>& spotPrices, // Vector of spot prices to be fille
                         const double& r, // Risk free interest rate (constant)
                         const double& v, // Volatility of underlying (constant)
                         const double& T // Expiry
                         );

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

You will also need to implement a `SimpleMonteCarlo` class with a function to implement the simulation procedure:

```
#ifndef __Option_Class__SimpleMonteCarlo__
#define __Option_Class__SimpleMonteCarlo__

#include <iostream>
#include "Vanilla1.h"

double SimpleMonteCarlo3(const AsianOption& TheOption,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths);

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

Overall, your Asian option pricer should include the following files: - Random.h - Random.cpp - PayOff.h - PayOff.cpp - AsianOption.h - AsianOption.cpp - SimpleMC.h - SimpleMC.cpp - AsianOptionMain.cpp

Please use the following parameter and generate Geometric Asian Put and Call option prices: - $T = 1$ - $S(0) = 50$ - $K = 50$ - $\sigma = 0.30$ - $r = 0.05$

For each path you will use 250 intervals, and please run the simulation for 1000 times to calculate expected option prices.

Random.h

```
// Random.h

#ifndef __Option_Class__Random__
#define __Option_Class__Random__

#include <vector>

// Generates a random Gaussian number.
double GetOneGaussianByBoxMuller();

// Generates a Geometric Brownian Motion spot price path.
void GetGBMSpotPricePath(std::vector<double>& spotPrices,
                         const double& r,
                         const double& v,
                         const double& T,
                         unsigned long NumberOfSteps);

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

Random.cpp

```
// Random.cpp

#include "Random.h"
#include <cstdlib> // static casting
#include <cmath> // exp, log, sqrt

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

    double sizeSquared, result;

    do
    {
        // Generate two uniform random numbers 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;

    // Continue until a point within the unit circle is found.
    } while (sizeSquared >= 1.0 || sizeSquared == 0.0);

    // Apply Box-Muller formula to produce a normally distributed value.
    result = x * sqrt(-2 * log(sizeSquared) / sizeSquared);

    return result;
};

// Generates a path for the spot price of an asset using Geometric Brownian Motion (GBM).
void GetGBMSpotPricePath(std::vector<double>& spotPrices,
                         const double& r,
                         const double& v,
                         const double& T,
                         unsigned long NumberOfSteps)
{
    double deltaT = T / NumberOfSteps; // Time increment
    double drift = exp((r - 0.5 * v * v) * deltaT); // Drift component for each step
    double vol = sqrt(v * v * deltaT); // Volatility component for each step

    for (unsigned long i = 1; i < NumberOfSteps; ++i)
    {
        // Generate a random Gaussian value
        double thisGaussian = GetOneGaussianByBoxMuller();

        // Calculate the spot price at the next step using the GBM formula
        spotPrices[i] = spotPrices[i - 1] * drift * exp(vol * thisGaussian);
    }
};
```

PayOff.h

```
// PayOff.h

#ifndef __PAY_OFF__
#define __PAY_OFF__

#include <algorithm> // For std::max, used in payoff calculations.
#include <vector>

/* PayOff: Abstract base class for option payoff calculations.

   Implements the Function Object design pattern to calculate
```

```
   the payoff for given spot price(s) of the underlying asset. */
class PayOff
{
    public:
        PayOff() {} // Default constructor
        virtual ~PayOff() {} // Virtual destructor for proper cleanup of derived classes

        /* Overloaded operator() to make this class a callable object (function object).

        Must be overridden in derived classes to compute the option payoff.

        Param S: The spot price(s) of the underlying asset.
        Returns: The calculated payoff. */
        virtual double operator()(const std::vector<double>& S) const = 0;
};

/* PayOffCall: Derived class to calculate the payoff of a call option. */
class PayOffCall : public PayOff
{
    public:
        /* Constructor to initialize the strike price of the call option.

        Param K_: The strike price. */
        PayOffCall(const double& K_);

        /* Destructor. */
        virtual ~PayOffCall() {}

        /* Calculates the payoff of the call option based on spot price(s).

        Overrides the pure virtual function of the base class.

        Param S: The spot price(s) of the underlying asset.
        Returns: The calculated call option payoff. */
        virtual double operator()(const std::vector<double>& S) const override;

    private:
        double K; // Strike price of the option
};

/* PayOffPut: Derived class to calculate the payoff of a put option. */
class PayOffPut : public PayOff
{
    public:
        /* Constructor to initialize the strike price of the put option.

        Param K_: The strike price. */
        PayOffPut(const double& K_);
```

```
        /* Destructor. */
        virtual ~PayOffPut() {}

        /* Calculates the payoff of the put option based on spot price(s).

        Overrides the pure virtual function of the base class.

        Param S: The spot price(s) of the underlying asset.
        Returns: The calculated call option payoff. */
        virtual double operator()(const std::vector<double>& S) const override;

    private:
        double K; // Strike price of the option
};

#endif
```

PayOff.cpp

```
// PayOff.cpp

#include "PayOff.h"
#include <cmath> // For std::exp
#include <numeric> // For std::accumulate

/* Constructor for PayOffCall, initializing the strike price (K). */
PayOffCall::PayOffCall(const double& K_) : K(K_) {}

/* Calculates the payoff for a call option.

   The payoff is the maximum of zero or the difference between the
   average spot price and the strike price.

   Param S: Vector of spot prices.
   Returns: Calculated payoff for the call option. */
double PayOffCall::operator()(const std::vector<double>& S) const
{
    double average = std::exp(std::accumulate(S.begin(), S.end(), 0.0,
                              [](double a, double b) { return a + std::log(b); }) / S.size());
    return std::max(average - K, 0.0); // Max between 0 and (average spot price - K)
}

/* Constructor for PayOffPut, initializing the strike price (K). */
PayOffPut::PayOffPut(const double& K_) : K(K_) {}

/* Calculates the payoff for a put option.
```

```
   The payoff is the maximum of zero or the difference between the
   strike price and the average spot price.

   Param S: Vector of spot prices.
   Returns: Calculated payoff for the put option. */
double PayOffPut::operator()(const std::vector<double>& S) const {
    double average = std::exp(std::accumulate(S.begin(), S.end(), 0.0,
                              [](double a, double b) { return a + std::log(b); }) / S.size());
    return std::max(K - average, 0.0); // Max between 0 and (K - average spot price)
}
```

AsianOption.h

```
// AsianOption.h

#ifndef __Asian_Option__
#define __Asian_Option__

#include <vector>
#include "PayOff.h"

/* AsianOption: Abstract base class for Asian options.

   It provides a common interface for different types of Asian options,
   encapsulating the behavior through the PayOff class for calculating payoffs. */
class AsianOption
{
    public:
        /* Constructor that initializes the PayOff object for the option.

        Param _pay_off: Pointer to a PayOff object, which encapsulates the payoff calculation
        AsianOption(PayOff* _pay_off);

        /* Virtual destructor ensures derived class destructors are called correctly. */
        virtual ~AsianOption() {}

        /* Pure virtual function for calculating the option payoff.

        This function must be implemented by derived classes to specify how the payoff is calc

        Param spot_prices: A vector of double values representing spot prices of the underlyin
        Returns: The calculated option payoff as a double. */
        virtual double OptionPayOff(const std::vector<double>& spot_prices) const = 0;

    protected:
        PayOff* pay_off; // Pay-off class (in this instance call or put), used to calculate th
};
```

```
/* AsianOptionGeometric: Derived class from AsianOption to implement payoff calculation
   for geometric Asian options. */
class AsianOptionGeometric : public AsianOption
{
    public:
        /* Constructor that initializes the PayOff object for the geometric Asian option.

        Param _pay_off: Pointer to a PayOff object, encapsulating the payoff calculation logic
        AsianOptionGeometric(PayOff* _pay_off);

        /* Destructor. */
        virtual ~AsianOptionGeometric() {}

        /* Overridden function to calculate the payoff for a geometric Asian option.

        Utilizes the geometric mean of spot prices for calculation.

        Param spot_prices: A vector of double values representing spot prices of the underlying
        Returns: The calculated option payoff as a double. */
        virtual double OptionPayOff(const std::vector<double>& spot_prices) const override;
};

#endif
```

AsianOption.cpp

```
// AsianOption.cpp

#include "AsianOption.h"
#include <numeric> // For std::accumulate
#include <cmath>   // For std::exp and std::log

/* Constructor for the AsianOption class. Initializes the pay_off object.

   Param _pay_off: Pointer to a PayOff object which will be used to calculate payoffs. */
AsianOption::AsianOption(PayOff* _pay_off) : pay_off(_pay_off) {}

/* Constructor for the AsianOptionGeometric class. Calls the base class constructor to initial

   Param _pay_off: Pointer to a PayOff object for payoff calculation logic. */
AsianOptionGeometric::AsianOptionGeometric(PayOff* _pay_off) : AsianOption(_pay_off) {}

/* Calculates the option payoff for a geometric Asian option.

   This method overrides the pure virtual function in the base class.

   It computes the payoff using the geometric mean of spot prices.
```

```
   Param spot_prices: A vector of double values representing the sampled spot prices of the un
   Returns: The calculated option payoff as a double value. */
double AsianOptionGeometric::OptionPayOff(const std::vector<double>& spot_prices) const
{
    return (*pay_off)(spot_prices);
}
```

SimpleMC.h

```
// SimpleMC.h

#ifndef __Option_Class__SimpleMonteCarlo__
#define __Option_Class__SimpleMonteCarlo__

#include "AsianOption.h"

/* SimpleMonteCarlo3:
   This function performs the Monte Carlo simulation to price an Asian option.

   Parameters:
   - TheOption: A reference to an AsianOption object. This object encapsulates the
     option type (call or put) and the payoff calculation method. The Monte Carlo
     simulation will use this information to compute the option price.
   - Spot: The initial spot price of the underlying asset.
   - Vol: The volatility of the underlying asset. This is a measure of the asset's
     price variation and is a crucial input for the simulation.
   - r: The risk-free interest rate. This rate is used to discount the option payoff
     back to the present value.
   - NumberOfPaths: The number of simulation paths. This parameter determines how many
     times the underlying asset price path will be simulated. A higher number of paths
     can increase the accuracy of the simulation but also increases computational time.
   - NumberOfSteps: The number of steps in each simulation path. This determines how many
     times the asset price will be sampled in each path, which is relevant for path-dependent
     options like Asian options.

   Returns:
   - The estimated price of the Asian option as a double.
*/
double SimpleMonteCarlo3(const AsianOption& TheOption,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths,
                         unsigned long NumberOfSteps);

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

```
#include "SimpleMC.h"
```

```cpp
#include "Random.h"
#include <vector>
#include <cmath> // sqrt, exp

/* SimpleMonteCarlo3 function implementation.

   Performs a Monte Carlo simulation to estimate the price of an Asian option.

   Parameters:
   - TheOption: A reference to an AsianOption object, which specifies the type of option and h
   - Spot: The initial spot price of the underlying asset.
   - Vol: The volatility of the underlying asset, indicating the standard deviation of its ret
   - r: The risk-free interest rate, used for discounting the payoff to present value.
   - NumberOfPaths: The number of paths to simulate in the Monte Carlo method.
   - NumberOfSteps: The number of steps to use in each path for simulating the underlying asse

   Returns:
   - The estimated price of the option as a double. */
double SimpleMonteCarlo3(const AsianOption& TheOption,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths,
                         unsigned long NumberOfSteps)
{
    double expiry = 1.0; // Fixed expiry of 1 year for the option.

    double variance = Vol * Vol * expiry; // Total variance over the option's life.
    double rootVariance = sqrt(variance); // Standard deviation of returns.
    double itoCorrection = -0.5 * variance; // Correction term for drift due to Ito's lemma.

    double movedSpot = Spot * exp(r * expiry + itoCorrection); // Adjust initial spot price fo
    double runningSum = 0; // Accumulator for payoffs across all paths.

    for (unsigned long i = 0; i < NumberOfPaths; i++)
    {
        std::vector<double> spotPrices(NumberOfSteps); // Vector to store simulated spot price
        spotPrices[0] = movedSpot; // Initialize the first spot price.

        // Generate the spot price path.
        GetGBMSpotPricePath(spotPrices, r, Vol, expiry, NumberOfSteps);

        // Calculate the payoff for this path and add it to the running sum.
        double thisPayoff = TheOption.OptionPayOff(spotPrices);
        runningSum += thisPayoff;
    }

    double mean = runningSum / NumberOfPaths; // Calculate the mean payoff.
```

```
    mean *= exp(-r * expiry); // Discount the mean payoff to present value.

    return mean; // Return the estimated option price.
}
```

SimpleMC.cpp

```
#include "SimpleMC.h"
#include "Random.h"
#include <vector>
#include <cmath> // sqrt, exp

/* SimpleMonteCarlo3 function implementation.

   Performs a Monte Carlo simulation to estimate the price of an Asian option.

   Parameters:
   - TheOption: A reference to an AsianOption object, which specifies the type of option and h
   - Spot: The initial spot price of the underlying asset.
   - Vol: The volatility of the underlying asset, indicating the standard deviation of its ret
   - r: The risk-free interest rate, used for discounting the payoff to present value.
   - NumberOfPaths: The number of paths to simulate in the Monte Carlo method.
   - NumberOfSteps: The number of steps to use in each path for simulating the underlying asse

   Returns:
   - The estimated price of the option as a double. */
double SimpleMonteCarlo3(const AsianOption& TheOption,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths,
                         unsigned long NumberOfSteps)
{
    double expiry = 1.0; // Fixed expiry of 1 year for the option.

    double variance = Vol * Vol * expiry; // Total variance over the option's life.
    double rootVariance = sqrt(variance); // Standard deviation of returns.
    double itoCorrection = -0.5 * variance; // Correction term for drift due to Ito's lemma.

    double movedSpot = Spot * exp(r * expiry + itoCorrection); // Adjust initial spot price fo
    double runningSum = 0; // Accumulator for payoffs across all paths.

    for (unsigned long i = 0; i < NumberOfPaths; i++)
    {
        std::vector<double> spotPrices(NumberOfSteps); // Vector to store simulated spot price
        spotPrices[0] = movedSpot; // Initialize the first spot price.

        // Generate the spot price path.
```

```
        GetGBMSpotPricePath(spotPrices, r, Vol, expiry, NumberOfSteps);

        // Calculate the payoff for this path and add it to the running sum.
        double thisPayoff = TheOption.OptionPayOff(spotPrices);
        runningSum += thisPayoff;
    }

    double mean = runningSum / NumberOfPaths; // Calculate the mean payoff.
    mean *= exp(-r * expiry); // Discount the mean payoff to present value.

    return mean; // Return the estimated option price.
}
```

AsianOptionMain.cpp

```
#include "SimpleMC.h"
#include "PayOff.h"
#include <iostream>

int main() {
    // Define parameters for the option pricing.

    double Expiry = 1.0; // Option expiry time in years
    double Strike = 50;  // Strike price of the option
    double Spot = 50;    // Initial spot price of the underlying asset
    double Vol = 0.30;   // Volatility of the underlying asset
    double r = 0.05;     // Risk-free interest rate

    unsigned long NumberOfPaths = 1000; // Number of paths for Monte Carlo simulation
    unsigned long NumberOfSteps = 250;  // Number of steps in each path

    // Create PayOff objects for call and put options.
    PayOffCall callPayOff(Strike);
    PayOffPut putPayOff(Strike);

    // Create AsianOptionGeometric objects for the call and put options.
    AsianOptionGeometric callOption(&callPayOff);
    AsianOptionGeometric putOption(&putPayOff);

    // Perform Monte Carlo simulations to estimate the prices of the call and put options.
    double callPrice = SimpleMonteCarlo3(callOption, Spot, Vol, r, NumberOfPaths, NumberOfSteps
    double putPrice = SimpleMonteCarlo3(putOption, Spot, Vol, r, NumberOfPaths, NumberOfSteps)

    // Output the results.
    std::cout << "The price of the Asian geometric call option is " << callPrice << std::endl;
    std::cout << "The price of the Asian geometric put option is " << putPrice << std::endl;

    return 0;
```

```
}
```