Math49111project1

Yijie Dong

November 2023

1 Introduction

In the dynamic landscape of financial markets, the accurate assessment of risk and the development of robust investment strategies are pivotal for success. Implied volatility and historical volatility stand out as critical metrics in this pursuit. Implied volatility reflects market expectations for future price fluctuations, inferred from option prices. On the other hand, historical volatility analyzes past price movements, offering insights into the asset's inherent risk.

As we navigate through the intricacies of C++ programming within the financial domain, the study accentuates the pivotal roles played by implied and historical volatility. These metrics not only serve as crucial inputs for options pricing but also play a fundamental role in risk management and strategic decision-making. Through practical examples and code implementations, we aspire to empower practitioners to leverage C++ for a comprehensive analysis of volatility, contributing to more informed and strategic financial decisions.

2 Compute Implied Volatility

First of all, we aim to employ the C++ programming language to assess the valuation of a European option by applying the Black-Scholes model.

To initiate our analysis, we commence with the widely recognized partial differential equation formulated by Black and Scholes (1) for an option:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0, \tag{1}$$

where the expression $\frac{\partial V}{\partial t}$ denotes the partial derivative of the option value (V) with respect to time (t). In this context, σ symbolizes the volatility associated with the underlying asset, S represents the prevailing price of the underlying asset, and r corresponds to the risk-free interest rate.

To solve equation (1), we first make a transformation:

$$x = log(S_0/X), \tag{2}$$
$$y = log(S_T/X), \tag{3}$$

where the variable X represents the exercise price of the option, S_0 denotes the asset value at $t = 0$, and S_T signifies the asset value at $t = T$. The parameter T represents the time to maturity.

Following that, we can obtain the value of the option at time $t = 0$ is equal to:

$$V(x,0) = A(x)\int_{-\infty}^{\infty} B(x,y)V(y,T)dy, \tag{4}$$

where

$$A(x) = \frac{1}{\sqrt{2\sigma^2}}e^{-\frac{1}{2}kx-\frac{1}{8}\sigma^2k^2T-rT} \tag{5}$$

and

$$B(x,y) = e^{-\frac{(x-y)^2}{2\sigma^2T}+\frac{1}{2}ky}, \quad V(y,T) = max(Xe^y - X, 0) \tag{6}$$

and

$$k = \frac{2r}{\sigma^2} - 1. \tag{7}$$

Hence, in the endeavour to employ C++ to evaluate the option, it becomes imperative to initiate the process by developing a C++ program designed to estimate the value of the integration.

2.1 Integration

In this section, we employ the Simpson's rule algorithm for the computation of integral values. The algorithm is represented mathematically as follows:

$$\int_b^a f(y)dy \approx \frac{h}{3}[f(a) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(b)] \tag{8}$$

where $x_j = a + jh$ and $h = (b-a)/n.$, where n determines the granularity of the approximation. Subsequently, we implement this algorithm in C++ (see Listing A.1) and apply it to the specific integral:

$$I = \int_{y=0}^{1} Xe^{-r\sigma T}e^{Ty}dy \tag{9}$$

For the verification, we consider the example with $X = 10, r = 0.05, \sigma = 0.1$ and $T = 0.5$. what is more, The exact solution is $I = 12.9420298618....$ and we aim to determine the value of n that ensures a high level of accuracy up to 6 significant figures. Adjusting the value of n enables us to achieve this precision in the computed solution. Hence, we obtain the table:

The table displays results obtained from numerical integration using Simpson's rule. Each row in the table corresponds to a different value of n. The values

n	Approximate	Exact	Error
1000	12.9332	12.942	0.880702
10000	12.9411	12.942	0.000880703
100000	12.9419	12.942	8.80703e-05
1000000	12.942	12.942	8.80705e-06
10000000	12.942	12.942	8.80722e-07

Table 1: Sample Table with Frame

of n increase logarithmically, starting from 1000 and progressing to 100,000,000. As n increases, the approximation tends to converge towards the exact solution. What is more, we can observe that when n reaches 10000000, the solution we obtain from the program is accurate to 6 s.f.

The table above illustrates outcomes derived from the application of Simpson's rule in numerical integration. Each row in the table corresponds to a distinct value of n, with n incrementing logarithmically from 1000 to 100,000,000. Notably, as n increases, the numerical approximation gradually converges toward the exact solution. Moreover, it is noteworthy that when n reaches 10,000,000, the solution computed by the program achieves the accuracy of up to 6 significant figures.

In the code, the function $f(y)$ defines the integrand, and the `integrate` function employs Simpson's rule to approximate the definite integral within a specified range $[a, b]$ using n steps. The main function executes a loop over varying values of n, computing the approximate solution and its associated error. The loop persists until the error descends below a predefined threshold ($1e-6$in this instance).

2.2 Procedural Approach

In this section, we endeavour to utilize the formula (4) along with the previously developed C++ program to assess the price of a European call option. In this context, we assume that the parameters associated with the Call Option are $S_0 = 97, X = 100, r = 0.06, \sigma = 0.2, T = 0.75$. Additionally, we perform the integration using the C++ program outlined in Section 2.1. We modify the integrated function f to evaluate B(x,y) multiplied by V(y, T), which we both define in (6). Consequently, the resulting code is represented in Listing A.2.

Given that the precise value can be obtained using the calculator (2) to calculate the exact value of this call option is equal to 7.36937. We then generate a table contrasting our computed solutions with the exact values for various settings of n.

The output table illustrates the convergence behaviour of the numerical approximation for different values of n. The columns present the number of steps (n), the approximate option price, the exact solution, and the absolute error. With an increase in n, the numerical approximation progressively approaches the exact solution, thereby underscoring the numerical stability inherent in the

n	Approximate	Exact	Error
100	7.7708	7.36937	0.401432
1000	7.36939	7.36937	1.94993e-05
10000	7.36937	7.36937	3.85165e-08

Table 2: Sample Table with Frame

implemented methodology.

2.3 Object Orientated Approach

To enhance modularity, and extensibility, and maintain a consistent structure in the design of classes related to financial options, we employ a pure virtual function to establish a protocol for the option's payoff function. The class, denoted as `Payoff`, operates as an abstract base class, furnishing an interface for diverse payoff functions associated with different types of financial options. Consequently, the `Payoff` class is utilized as a template or blueprint, requiring that any derived subclass representing a specific type of option must conform to the predefined interface by implementing its distinct version of the payoff function. This approach ensures a standardized and flexible framework for accommodating various option types within a cohesive design paradigm.

Hence, we compose the C++ code in Section 2.2 and incorporate `Payoff &payoff` into the argument list of each function. Furthermore, we can establish both the `PutOption` and `CallOption` classes, each inheriting from the `Payoff` base class. Subsequently, we can instantiate new functions, `optionValue`, and formulate a class that encapsulates all the requisite methods and parameters. Thus, we obtain the code A.3 and use this to generate the option value.

2.4 Implied Volatility

Before delving into the discussion of implied volatility, we introduce the Newton-Secant method, as outlined in (3).

The Newton-Secant method is a root-finding algorithm employed to determine a root of $f(x) = 0$. The iterative process for this method is defined as follows:

$$x_n = x_{n-1} - f(x_{n-1}) \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}. \tag{10}$$

To implement the Newton-Secant method in C++, a function is introduced to return a Boolean value indicating the success of the iteration process. Additionally, a tolerance value (`tol`) is required to terminate the iteration when $|f(x)| < tol$. In this project, we assume 10^{-6} in this project. Furthermore, a maximum number of iterations (`maxiter`) is set to halt the iteration if a root is not found.

Therefore, the implementation of the Newton-Secant method in C++ is presented in Code A.4. Additionally, the accuracy of the program can be assessed

by solving the equation $f(x) = x^2 - 2 = 0$, for which the known solutions are $x = \pm\sqrt{2}$. Across various initial guesses, the program consistently yielded results matching the expected solutions. Hence, the algorithm's accuracy has been successfully validated.

Next, we introduce the concept of implied volatility as outlined by (4). Implied volatility serves as an estimate of the future volatility of an underlying asset's price movements, inferred from the market prices of its associated options. This forward-looking metric is derived from option prices, encapsulating the collective market consensus on the anticipated magnitude of future price fluctuations.

The calculation of implied volatility involves equating the option value $V(S, X, r, t, T, \sigma$ to the market value of the option $\hat{V}(S, X, r, T)$. Here, $V(S, X, r, t, T, \sigma$ represents the option price computed using the Black-Scholes model, for which we have implemented a C++ program in Section 2.3.

Simultaneously, we define $\hat{V}(S, X, r, T)$ as the observed price for a European call (or put) option in the market. These prices are calculated based on selected call and put options for the same underlying assets used in the historical volatility calculation. Consequently, the market value is determined as follows:

$$\hat{V}(S, X, r, T) = (bidprice + askprice)/2 \tag{11}$$

As an illustration, on the website (5), the option selected is as follows:

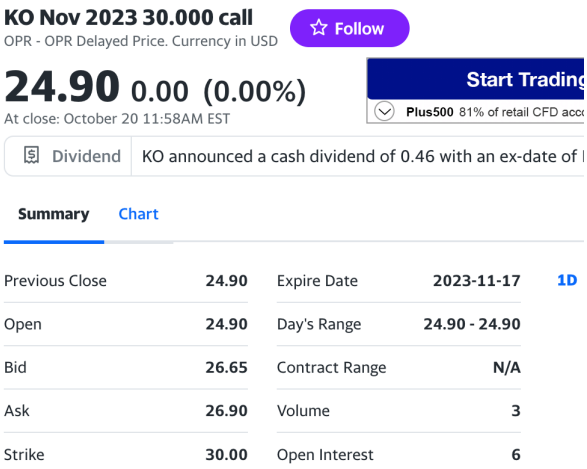

Figure 1: Option information

Upon examination of the aforementioned table, the market value for this call option is determined to be

$$\hat{V}(S, X, r, T) = (bidprice + askprice)/2 = \frac{26.65 + 26.9}{2} = 26.775$$

Therefore, the implied volatility (σ_{im}) can be determined by solving the function

$$g(\sigma_{im}) = V(S, t; X, T, r, \sigma) - \hat{V}(S, t; x, T) = 0 \tag{12}$$

Utilizing the previously developed C++ program A.5 for the Newton-Secant method, we can find the value of σ_{im} that satisfies $g(\sigma_{im}) = 0$. Consequently, the implied volatility is obtained and denoted as σ_{im}, and we can write the code of C++ to calculate the implied volatility.

3 Stock Analysis

To analyze a company, the initial step involves selecting a specific stock from (5). In this study, we opt for the Coca-Cola Company's stock. Subsequently, we can download the historical data and utilize R code A.6 to generate a time-series graph representing the stock performance of the Coca-Cola Company. The resulting graph is presented in Figure 2.

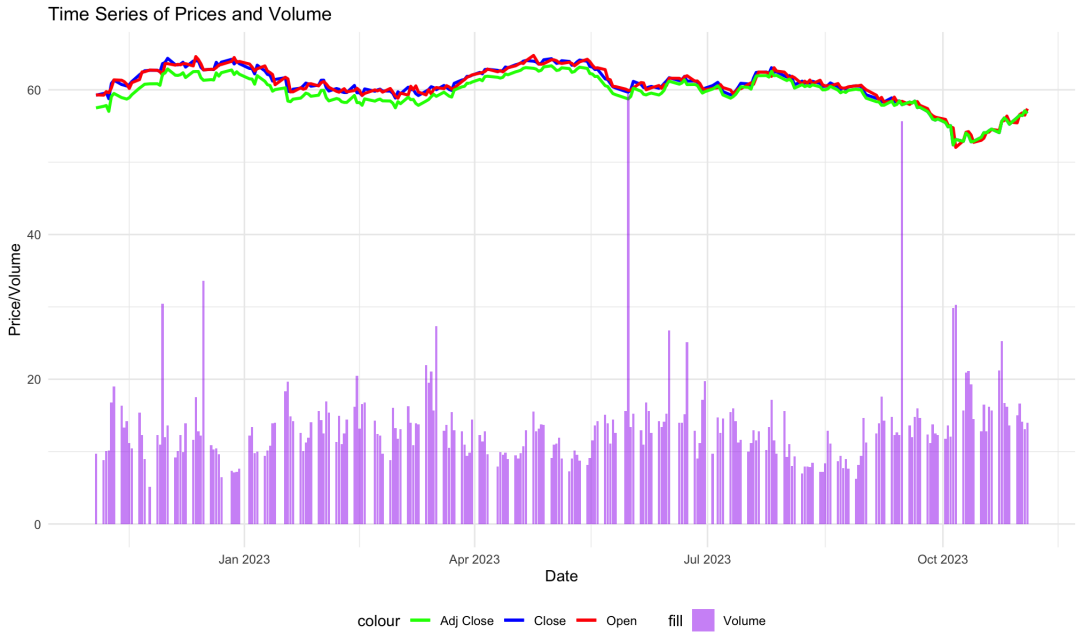

Figure 2: Graph of time series

The presented graph illustrates a time series of stock prices and trading volume for Coca-Cola spanning from January to October 2023. The lines delineate the opening price (in red), closing price (in blue), and adjusted closing price (in

green), the latter being adjusted for dividends and splits. These lines provide insights into the daily price fluctuations and the overarching trend of the stock.

The purple bars on the graph depict trading volume, indicating the number of shares traded each day. Elevated volume levels often signify increased interest in the stock and can potentially precede price changes. Noteworthy spikes in volume are observed, potentially corresponding to significant corporate events or market reactions.

Upon thorough examination of the graph, several notable observations emerge. Firstly, the stock price demonstrated a state of relative stability throughout the observed period, with a slight downward trend as the period concluded. Secondly, a marked spike in trading volume was observed around July, implying a significant event or the dissemination of noteworthy news during that period. Lastly, the adjusted close price closely mimics the pattern of the closing price, suggesting the absence of substantial adjustments related to corporate actions within this time frame. This comprehensive analysis provides insights into the stock's performance and potential influencing factors during the specified duration.

In summary, the stock experienced a phase of stability with some volatility, particularly around the mid-year period.

4 Implied Volatility Analysis

In this section, we aim to analyze the variation in implied volatility concerning different strike prices. To achieve this, we will create a table enumerating the strike prices and their corresponding implied volatilities using the provided C++ code (refer to Code A.7). Specifically, we will focus on a call option scenario where $S = 30.23, r = 0.002, \hat{V} = 1.425$ and $T = 0.1205$. The code will be executed to generate the table, covering strike prices ranging from 29 to 35, which will then be input into a file.

Subsequently, we will utilize this data file, along with an R script (refer to Code A.8), to construct a graphical representation illustrating the relationship between implied volatility and various strike prices. This comprehensive analysis aims to provide insights into how implied volatility evolves in response to changes in strike prices.

Upon scrutiny of the graph 3, it is evident that it portrays a curve commencing from the bottom-left and ascending consistently towards the top-right. The depicted relationship between the strike price and implied volatility is positive, signifying that with an increase in the strike price, the implied volatility also demonstrates an upward trend.

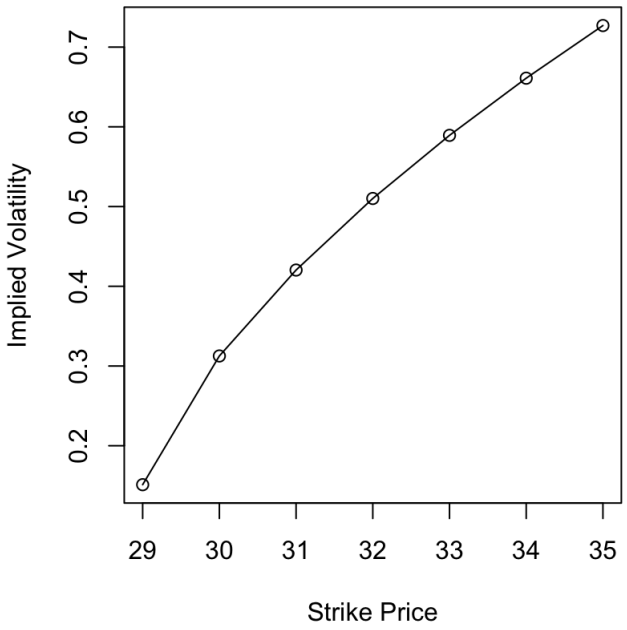

Figure 3: Implied Volatility against different Strike Price

5 Compare the Implied Volatility with Historical Volatility

5.1 Historical Volatility

To commence, we will elucidate the historical volatility algorithm, employing historical stock data for Coca-Cola as an illustrative example. We consider a vector of values $\mathbf{S} = (S_0, S_1, ..., S_{n-1})$ governed by the following discrete stochastic differential equation (SDE):

$$\frac{dS_i}{S_i} = \mu dt + \sigma\sqrt{dt}\varphi_i.$$

Here, μ represents the drift term, σ is the volatility, dt denotes the time step, and φ_i is a random variable.

In the case study of the Coca-Cola company, we employ the "adj value" notation to represent S_i, aiming to calculate historical volatility using the provided C++ code.

Initially, the adj close prices are read from the file and incorporated into C++, storing the data in the `MVector`. Subsequently, the vector $\mathbf{dln(S)} = ln(S_1) - ln(S_0), ln(S_2) - ln(S_1), ..., ln(S_{n-1}) - ln(S_{n-2})$ is generated by applying logarithmic transformations to the data within the `MVector`. The variance `dln(S)` through the equation:

$$Var(\mathbf{X}) = \frac{1}{n-1}(\sum_{i=0}^{n-1}(x_i)^2) - \frac{1}{n^2-n}(\sum_{i=0}^{n-1} x_i)^2. \tag{13}$$

After this calculation, the daily volatility is determined by:

$$\sigma_{daily} = \sqrt{var(\mathbf{dln(S)})} \tag{14}$$

Utilizing this approach, we leverage the historical price data of the Coca-Cola company and execute the C++ code A.9 to ascertain the historical volatility.

5.2 Contrast the Implied Volatility Against Historical Volatility

To compare implied volatility with historical volatility, we need to analyze the behaviour of these two metrics concerning the stock's price movements over time. As we have discussed before, the implied volatility is derived from option prices and reflects market expectations for future price fluctuations. On the other hand, the historical volatility is calculated from past stock price data and represents the actual volatility observed over a specific historical period.

Following this, the data can be sourced from the website (5), and the previously developed C++ program can be employed to compute both implied and historical volatilities. Subsequently, it is noteworthy that, in the majority of instances, the implied volatility closely approximates the historical volatility. The comparison revealed that, for the most part, the values of implied volatility closely align with historical volatility.

6 Conclusion

This study encompassed a diverse array of subjects, ranging from algorithm implementation to intricate financial concepts such as options pricing, implied volatility, and historical volatility analysis within the context of C++ programming. In the course of these discussions, we scrutinized the development of C++ programs for option pricing and explored methodologies such as the Newton-Secant root-finding algorithm. Furthermore, we conducted an in-depth analysis of historical volatility calculation using financial data, compared implied and historical volatilities, and employed graph plotting techniques to visually represent financial trends.

References

[1] Fischer Black and Myron Scholes.(1973) The pricing of options and corporate liabilities. *Journal of political economy* **81**, 637–654.

[2] Option calculator: `https://personalpages.manchester.ac.uk/staff/paul.johnson-2/pages/blackscholesCalculator.html`.

[3] Burden, Richard L.(2011)Numerical analysis. *Brooks/Cole Cengage Learning*

[4] Natenberg, Sheldon. (1994)"Option volatility and pricing: advanced trading strategies and techniques.".

[5] Finance website: `https://finance.yahoo.com/`

A Computer Code

A.1 C++ Code

The code uses Simpson's rule algorithm to estimate the integration:

```
#include <iostream>
#include <cmath>

int X=10;
double r = 0.05, sigma = 0.1, T = 0.5;

double f(double y)
{
    return X * (std::exp(-r * sigma * T)) * (std::exp(T * y));
}

// function to implement Simpsons rule from y=a to y=b with n steps
double integrate(double a, double b, int n)
{
    double sum = 0;
    double h = (b - a) / n;

    for (int i = 1; i < n; i += 2){
        sum += 4*f(a+i*h);
    }

    for (int i = 2; i < n-1; i += 2){
        sum += 2*f(a+i*h);
    }

    return sum * h/3;
}
```

```
28
29  int main()
30  {
31
32      // Exact solution (calculated separately)
33      double exact_solution = 12.9420298618;
34
35      std::cout << "n\tApproximate\tExact\t\tError" << std::endl;
36      for (int n = 1000; n <= 100000000; n *= 10)
37      {
38          double approximate = integrate(0, 1, n);
39          double error = std::abs(approximate - exact_solution);
40
41          std::cout << n << "\t" << approximate << "\t" << exact_solution
                << "\t" << error << std::endl;
42
43          // If the error is within 1e-6, consider it accurate enough
44          if (error < 1e-6)
45          {
46              std::cout << "n = " << n << " gives a solution accurate to 6 s
                    .f." << std::endl;
47              break;
48          }
49      }
50
51      return 0;
52  }
```

A.2 C++ Code

The code to calculate the option value:

```
1   #include <iostream>
2   #include <cmath>
3
4   double payoff(double y, double X)
5   {
6       return std::max(X * std::exp(y) - X, 0.0);
7   };
8
9   double A(double x, double r, double sigma,double T, double k)
10  {
11      return (1/sqrt(2*sigma*sigma* M_PI*T))*(std::exp(-0.5*k*x-0.125*sigma
            *sigma*k*k*T-r*T));
12  };
13
14  double B(double x, double y, double sigma, double T, double k)
15  {
16      return exp((-pow(x - y, 2) / (2 * pow(sigma, 2) * T)) + 0.5 * k * y);
```

```
}


double f(double x,double y,double X,double r,
double sigma,double T,double k)
{
    return B(x,y,sigma,T,k)*payoff(y,X);
}

// function to implement Simpsons rule from y=a to y=b with n steps
double integrate(double a,double b,int n,double x,
double X,double r,double sigma,double T,double k)
{
    double sum = 0;
    double h = (b - a) / n;

    for (int i = 1; i < n; i += 2){
        sum += 4*f(x, a+i*h, X, r, sigma, T,k);
    }

    for (int i = 2; i < n-1; i += 2){
        sum += 2*f(x, a+i*h, X, r, sigma, T,k);
    }

    return sum * h/3;
}


int main()
{
    double exact_solution = 7.369373;

    double X = 100;
    double r = 0.06;
    double sigma = 0.2;
    double T = 0.75;
    double S_0 = 97;


    double x = std::log(S_0/X);
    double k = 2*r/(sigma*sigma)-1;

    std::cout << "n\tApproximate\tExact\t\tError" << std::endl;
    for (int n = 100; n <= 1000000; n *= 10)
    {
        double value = integrate(-10,10,n,x,X,r,sigma,T,k);
        double approximate = A(x,r,sigma,T,k)*value;
        double error = std::abs(approximate - exact_solution);

        std::cout << n << "\t" << approximate << "\t" << exact_solution
```

```
            << "\t" << error << std::endl;

        // If the error is within 1e-6, consider it accurate enough
        if (error < 1e-6)
        {
            std::cout << "n = " << n << " gives a solution accurate to 6 s
                .f." << std::endl;
            break;
        }
    }

    return 0;
}
```

A.3 C++ Code

The code is used to compute the option value with the virtual function:

```
#include <iostream>
#include <cmath>
#include <algorithm> // For std::max function

// Payoff class remains the same
class Payoff {
public:
    virtual double operator()(double y) = 0;
    virtual ~Payoff() {}
};

// CallOption and PutOption classes remain the same

class OptionPricer {
private:
    double r; // Risk-free interest rate
    double sigma; // Volatility
    double T; // Time to expiration
    int n; // Number of steps for numerical integration

    // Helper function for integration
    double f_object(double x, double y, double X, double k, Payoff &
        payoff) {
        double V = std::max(X * std::exp(y) - X, 0.0);
        double B = exp((-pow(x - y, 2) / (2 * pow(sigma, 2) * T)) + 0.5 *
            k * y);
        return B * V;
    }

    // Simpson's rule integration
```

```
29      double integrate(double a, double b, double x, double X, double k,
            Payoff &payoff) {
30          double sum = 0;
31          double h = (b - a) / n;
32
33          sum += 0.5 * (f_object(x, a, X, k, payoff) + f_object(x, b, X, k,
                payoff));
34
35          for (int i = 1; i < n; i++) {
36              sum += f_object(x, a + i * h, X, k, payoff) * (i % 2 == 0 ? 2
                    : 4);
37          }
38
39          return sum * h / 3;
40      }
41
42  public:
43      // Constructor
44      OptionPricer(double r_, double sigma_, double T_, int n_) : r(r_),
            sigma(sigma_), T(T_), n(n_) {}
45
46      // Method to calculate the option value
47      double optionValue(double S, double X, Payoff &payoff) {
48          double x = std::log(S/X);
49          double k = 2 * r / (sigma * sigma) - 1;
50          double A = (1 / sqrt(2 * sigma * sigma * M_PI * T)) * (std::exp
                (-0.5 * k * x - 0.125 * sigma * sigma * k * k * T - r * T));
51          double value = integrate(-10, 10, x, X, k, payoff);
52          return A * value;
53      }
54  };
55  class CallOption : public Payoff {
56  private:
57      double X;
58
59  public:
60      CallOption(double X_) : X(X_) {}
61      double operator()(double y) {
62          return std::max(X * std::exp(y) - X, 0.0);
63      }
64  };
65
66  class PutOption : public Payoff {
67  private:
68      double X;
69
70  public:
71      PutOption(double X_) : X(X_) {}
72      double operator()(double y) {
73          return std::max(X - X * std::exp(y) , 0.0);
```

```
    }
};

int main() {
    // Parameters for option pricing
    double S = 100.0; // Current stock price
    double X = 100.0; // Strike price
    double r = 0.05; // Risk-free interest rate
    double sigma = 0.1; // Volatility
    double T = 0.5; // Time to expiration
    int n = 1000; // Number of steps for numerical integration
    // Create an instance of the pricer
    OptionPricer pricer(r, sigma, T, n);

    // Create a payoff instance, for example, a CallOption
    CallOption callPayoff(X); // Calculate the call option value
    double callValue = pricer.optionValue(S, X, callPayoff);
    std::cout << "The value of the call option is: " << callValue << std
        ::endl;
    // Similarly for a put option
    PutOption putPayoff(X);
    double putValue = pricer.optionValue(S, X, putPayoff);
    std::cout << "The value of the put option is: " << putValue << std::
        endl;


    return 0;
    }
```

A.4 C++ Code

The code of Newton-Secant method:

```
#include <iostream>
#include <cmath>

double newton_secant_f(double x)
{
    return x*x - 2;
}

bool NewtonSecant(double &result, double x0, double x1, double tol, int
maxiter)
{
int i;
double x;
for (i=0; i<maxiter; i++)
{
x = x1 - newton_secant_f(x1)*(x1-x0)/
```

```
(newton_secant_f(x1)-newton_secant_f(x0));
if (std::abs(newton_secant_f(x))<tol) break;
x0=x1;
x1=x;
}
result = x;

if (i==maxiter)
return false;
else return true;
}

int main() {
    double result;
    double x0 = 1.0;
    double x1 = 2.0;
    double tol = 1e-6;
    int maxiter = 100;

    bool success = NewtonSecant(result, x0, x1, tol, maxiter);

    if (success) {
        std::cout << "Root found: " << result << std::endl;
    } else {
        std::cout << "Root not found within maximum iterations." << std::
            endl;
    }

    return 0;
}
```

A.5 C++ Code

The code calculates the implied volatility:

```
#include <iostream>
#include <cmath>
#include <algorithm> // For std::max function

// Payoff class remains the same
class Payoff {
public:
    virtual double operator()(double y) = 0;
    virtual ~Payoff() {}
};

// CallOption and PutOption classes remain the same

class OptionPricer {
```

```
15 private:
16     double r; // Risk-free interest rate
17     double sigma; // Volatility
18     double T; // Time to expiration
19     int n; // Number of steps for numerical integration
20 
21     // Helper function for integration
22     double f_object(double x, double y, double X, double k, Payoff &
           payoff) {
23         double V = std::max(X * std::exp(y) - X, 0.0);
24         double B = exp((-pow(x - y, 2) / (2 * pow(sigma, 2) * T)) + 0.5 *
               k * y);
25         return B * V;
26     }
27 
28     // Simpson's rule integration
29     double integrate(double a, double b, double x, double X, double k,
           Payoff &payoff) {
30         double sum = 0;
31         double h = (b - a) / n;
32 
33         sum += 0.5 * (f_object(x, a, X, k, payoff) + f_object(x, b, X, k,
               payoff));
34 
35         for (int i = 1; i < n; i++) {
36             sum += f_object(x, a + i * h, X, k, payoff) * (i % 2 == 0 ? 2
                   : 4);
37         }
38 
39         return sum * h / 3;
40     }
41 
42 public:
43     // Constructor
44     OptionPricer(double r_, double sigma_, double T_, int n_) : r(r_),
           sigma(sigma_), T(T_), n(n_) {}
45 
46     // Method to calculate the option value
47     double optionValue(double S, double X, Payoff &payoff) {
48         double x = std::log(S/X);
49         double k = 2 * r / (sigma * sigma) - 1;
50         double A = (1 / sqrt(2 * sigma * sigma * M_PI * T)) * (std::exp
               (-0.5 * k * x - 0.125 * sigma * sigma * k * k * T - r * T));
51         double value = integrate(-10, 10, x, X, k, payoff);
52         return A * value;
53     }
54 };
55 
56 double g(double S,double X,double r,double sigma, double T,Payoff &payoff
       )
```

```
{
    double bidprice = 21.95;
    double askprice = 22.1;
    double marketValue = 0.05;
    int n = 1000;
    OptionPricer pricer(r, sigma, T, n);
    double optionValue = pricer.optionValue(S,X,payoff);
    return optionValue - marketValue;

}

bool NewtonSecant(double &result, double sigma0, double sigma1, double
     tol, int
maxiter, double S,double X,double r, double T,Payoff &payoff)
{
int i;
double sigma;
for (i=0; i<maxiter; i++)
{
sigma = sigma1 - g(S, X,r,sigma1, T, payoff)*(sigma1-sigma0)/
(g(S, X,r,sigma1, T, payoff)-g(S, X,r,sigma0, T, payoff));
if (std::abs(g(S, X,r,sigma, T, payoff))<tol) break;
sigma0=sigma1;
sigma1=sigma;
}
result = sigma;

if (i==maxiter)
return false;
else return true;
}

class CallOption : public Payoff {
private:
    double X;

public:
    CallOption(double X_) : X(X_) {}
    double operator()(double y) {
        return std::max(X * std::exp(y) - X, 0.0);
    }
};


int main()
{
    double result;
    double S = 1;
    double X = 1;
    double T = 0.5;
```

```
    double r = 0.05;
    double sigma0 = 0.1; // Adjusted initial guess
    double sigma1 = 0.2; // Adjusted initial guess
    double tol = 1e-6;
    int maxiter = 100000; // Increased maximum iterations

    CallOption callOption(X);

    bool success = NewtonSecant(result, sigma0,sigma1, tol, maxiter, S,X,
         r, T,callOption);

    if (success) {
        std::cout << "Root found: " << result << std::endl;
    } else {
        std::cout << "Root not found within maximum iterations." << std::
            endl;
    }

    return 0;
}
```

A.6 R Code

The R code to plot the graph of time series:

```
# Install and load required packages if not already installed
# install.packages("ggplot2")
# install.packages("dplyr")
library(ggplot2)
library(dplyr)

# Convert the 'Date' column to Date format
df$Date <- as.Date(df$Date)

# Create a time series plot using ggplot2
ggplot(df, aes(x = Date)) +
  geom_line(aes(y = Close, color = 'Close'), size = 1, linetype = 'solid'
      ) +
  geom_line(aes(y = Open, color = 'Open'), size = 1, linetype = 'solid')
      +
  geom_line(aes(y = `Adj.Close`, color = 'Adj Close'), size = 1, linetype
       = 'solid') +

  labs(title = 'Time Series of Prices and Volume',
       x = 'Date',
       y = 'Price/Volume') +
  scale_y_continuous(labels = scales::comma) + # Format y-axis labels
  theme_minimal() +
```

```
  scale_color_manual(values = c('Close' = 'blue', 'Open' = 'red', 'Adj
      Close' = 'green')) +

  theme(legend.position = 'bottom') # Move the legend to the bottom
```

A.7 C++ Code

The C++ code to calculate the implied volatility with different strike price:

```
#include <iostream>
#include <cmath>
#include <algorithm> // For std::max function
#include <fstream>

// Payoff class remains the same
class Payoff {
public:
    virtual double operator()(double y) = 0;
    virtual ~Payoff() {}
};

// CallOption and PutOption classes remain the same

class OptionPricer {
private:
    double r; // Risk-free interest rate
    double sigma; // Volatility
    double T; // Time to expiration
    int n; // Number of steps for numerical integration

    // Helper function for integration
    double f_object(double x, double y, double X, double k, Payoff &
        payoff) {
        double V = std::max(X * std::exp(y) - X, 0.0);
        double B = exp((-pow(x - y, 2) / (2 * pow(sigma, 2) * T)) + 0.5 *
            k * y);
        return B * V;
    }

    // Simpson's rule integration
    double integrate(double a, double b, double x, double X, double k,
        Payoff &payoff) {
        double sum = 0;
        double h = (b - a) / n;

        sum += 0.5 * (f_object(x, a, X, k, payoff) + f_object(x, b, X, k,
            payoff));

        for (int i = 1; i < n; i++) {
```

```
37              sum += f_object(x, a + i * h, X, k, payoff) * (i % 2 == 0 ? 2
                    : 4);
38          }
39
40          return sum * h / 3;
41      }
42
43  public:
44      // Constructor
45      OptionPricer(double r_, double sigma_, double T_, int n_) : r(r_),
            sigma(sigma_), T(T_), n(n_) {}
46
47      // Method to calculate the option value
48      double optionValue(double S, double X, Payoff &payoff) {
49          double x = std::log(S/X);
50          double k = 2 * r / (sigma * sigma) - 1;
51          double A = (1 / sqrt(2 * sigma * sigma * M_PI * T)) * (std::exp
                (-0.5 * k * x - 0.125 * sigma * sigma * k * k * T - r * T));
52          double value = integrate(-10, 10, x, X, k, payoff);
53          return A * value;
54      }
55  };
56
57  double g(double S,double X,double r,double sigma, double T,Payoff &payoff
        )
58  {
59      double bidprice = 21.95;
60      double askprice = 22.1;
61      double marketValue = 1.425;
62      int n = 1000;
63      OptionPricer pricer(r, sigma, T, n);
64      double optionValue = pricer.optionValue(S,X,payoff);
65      return optionValue - marketValue;
66
67  }
68
69  bool NewtonSecant(double &result, double sigma0, double sigma1, double
        tol, int
70  maxiter, double S,double X,double r, double T,Payoff &payoff)
71  {
72  int i;
73  double sigma;
74  for (i=0; i<maxiter; i++)
75  {
76  sigma = sigma1 - g(S, X,r,sigma1, T, payoff)*(sigma1-sigma0)/
77  (g(S, X,r,sigma1, T, payoff)-g(S, X,r,sigma0, T, payoff));
78  if (std::abs(g(S, X,r,sigma, T, payoff))<tol) break;
79  sigma0=sigma1;
80  sigma1=sigma;
81  }
```

```
result = sigma;

if (i==maxiter)
return false;
else return true;
}

class CallOption : public Payoff {
private:
    double X;

public:
    CallOption(double X_) : X(X_) {}
    double operator()(double y) {
        return std::max(X * std::exp(y) - X, 0.0);
    }
};


int main() {
    double S = 30.23;
    double T = 0.1205;
    double r = 0.002;
    double sigma0 = 0.1; // Adjusted initial guess
    double sigma1 = 0.2; // Adjusted initial guess
    double tol = 1e-6;
    int maxiter = 1000; // Increased maximum iterations

    // Output file
    std::ofstream outputFile("implied_volatility_results.csv");
    outputFile << "Strike Price,Implied Volatility\n"; // Header

    // Loop over a range of strike prices
    for (double X = 29.0; X <= 35.0; X += 1.0) {
        CallOption callOption(X);

        double result; // Implied volatility result
        bool success = NewtonSecant(result, sigma0, sigma1, tol, maxiter,
             S, X, r, T, callOption);

        if (success) {
            // Print the result to the console
            std::cout << "Strike Price: " << X << ", Implied Volatility: "
                 << result << std::endl;

            // Write the result to the file
            outputFile << X << "," << result << "\n";
        } else {
            std::cout << "Root not found within maximum iterations for
                Strike Price: " << X << std::endl;
```

```
        }
    }

    // Close the output file
    outputFile.close();

    return 0;
}
```

A.8 R Code

The R code to plot the graph of the implied Volatility against different Strike Price:

```
df<-read.csv("implied_volatility_results.csv")
plot(df$Strike.Price,df$Implied.Volatility, xlab ="Strike Price", ylab =
    "Implied Volatility")
lines(df$Strike.Price,df$Implied.Volatility)
```

A.9 C++ Code

The C++ code to calculate the historical volatility:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <cmath>

#ifndef MVECTOR_H
#define MVECTOR_H

// Class that represents a mathematical vector
class MVector {
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}
    MVector(std::initializer_list<double> l) : v(l) {}

    // access element (lvalue) (see example sheet 5, q5.6)
    double &operator[](int index) {
        return v[index];
    }

    // access element (rvalue) (see example sheet 5, q5.7)
    double operator[](int index) const {
```

```
26          return v[index];
27      }
28
29      int size() const { return v.size(); } // number of elements
30
31      // Add data to the vector
32      void push_back(double x) {
33          v.push_back(x);
34      }
35
36  private:
37      std::vector<double> v;
38  };
39
40  #endif
41
42  double variance(const MVector &S) {
43      double squareSum = 0.0;
44      double DeltaS = 0.0;
45      double sumLogs = 0.0;
46      int n = S.size();
47      for (int i = 0; i < n - 1; i++) {
48          DeltaS = std::log(S[i + 1]) - std::log(S[i]);
49          squareSum += DeltaS * DeltaS;
50          sumLogs += DeltaS;
51      }
52      double results = (1.0 / (n - 1)) * squareSum - (1.0 / (n * n - n)) *
            sumLogs * sumLogs;
53
54      return sqrt(n*results);
55  }
56
57  int main() {
58      std::ifstream input("output.txt");
59      MVector stockPrices; // Use MVector to store all data
60
61      if (!input) {
62          std::cout << "Could not open file for reading" << std::endl;
63          return 1;
64      }
65
66      double value;
67      while (input >> value) {
68          // Assuming all values are in the file
69          stockPrices.push_back(value);
70      }
71
72      input.close();
73
74      // Print all values in the vector
```

```
    for (int i = 0; i < stockPrices.size(); i++) {
        std::cout << stockPrices[i] << " ";
    }

    // Calculate and print the variance
    double stockVariance = variance(stockPrices);
    std::cout << "\nVariance: " << stockVariance << std::endl;

    return 0;
}
```