C++ For Finance, Project 2 - Implied Volatility and the Greeks

University of Birmingham
Msc Mathematical Finance

Introduction

Before running the code please ensure that the CLion working directory is set to the project directory in order to allow the input.txt file to be read and to ensure that the output.txt file will be sent to the project folder. If this is not done, the input.txt file will not be read and the output.txt file will end up being sent to the cmake-build-debug folder.

In this project we were required to implement the Newton-Raphson method and Secant root finding method in order to calculate the implied volatility of European Call and Put options. The formal problem is this: Given the price of an option F, the spot price of the underlying S, strike K, risk free rate r, expiry time T, and let $BS(S, K, r, T, \sigma)$ give the Black-Scholes price of an option, find σ such that

$$BS(\sigma) = F. \tag{1}$$

The Newton-Raphson iterative solution is formulated as follows

$$\sigma_{n+1} = \frac{F - B(\sigma_n)}{B'(\sigma_n)} + \sigma_n, \tag{2}$$

where $B'(\sigma)$ is the vega of the option. In the case of the secant method, we use the secant method to approximate $B'(\sigma)$ in the Newton-Raphson method and continue as normal. Note that for the secant method two initial guesses are required.

The approach to the implementation is centred around the use of a functor and a function template. The motivation for this approach is to keep the method as a general and extendable as possible. The goal was to create code that a client could easily extend to deal with other types of options and root finding methods as well as efficiently serving its purpose with regards to the Newton-Raphson and Secant methods.

1 Code Description

1.1 Running the Code

In order to run the code, the client will need to provide/edit the ”input.txt” file. The .txt file should contain one line with each element on the line separated by one space. Input should be in the following order: ”option_type spot_price strike_price risk_free_rate expiry(in years) option_price root_finding_method” where the types are ”string double double double double double double” respectively. The string must contain no capitals and must be either ”call” or ”put”, and the root finding method must be specified as 1 or 2 (1 for Newton-Raphson, and 2 for secant) . An example input would therefore be ”put 1157.86 1125 0.02 0.0602739726 19.7 1” for a put with the Newton-Raphson method, or perhaps ”call 1157.86 1125 0.02 0.0602739726 43 2” for a call with the secant method.

1.2 Classes and Templates

1.2.1 option_price.h

option_price.h

```
class option_price {
public:
    // Pure virtual pricing functions
    virtual double OptionPrice(double S, double K, double r, double sigma,
    double T) const = 0;
    virtual double d_j(int j, double S, double K, double r, double sigma,
    double T) const = 0;
    virtual double vega(double S, double K , double r, double sigma,
    double T) const = 0;
```

The above code shows part of the abstract base class option_price. This class serves as the base class for which different types of options can derive from. It has pure virtual member functions so that each derived class can define them separately. The reason this approach was chosen was because it means the code is easily extendable to other types of options. In this project we are just concerned with European puts and calls so the pure virtual member functions of the base class allow a derived class to implement a function to calculate the price of the option (OptionPrice), the d_1 and d_2 values for puts and calls, and the vega of an option. However, this class can easily include other member functions (for example different greeks) and different types of options would then inherit from it. The class also implements a copy constructor, a destructor, and a copy assignment operator following the rule of three that states if one of the former is required then all three should be implemented. There are two classes Call.h and Put.h that inherit publicly from this abstract base class option_price and implement the required functionality. These classes also follow the rule of three and implement a destructor, copy constructor, and a copy assignment operator.

1.2.2 black_scholes_option.h

black_scholes_option.h

```
class BlackScholesOption {
private:
    double S;  // Asset price
    double K;  // Strike price
    double r;  // Risk-free rate
    double T;  // Maturity Time
public:
    BlackScholesOption(double _S, double _K, double _r, double _T);
    double call_option_price(double sigma) const;
    double put_option_price(double sigma) const;
    double option_vega(double sigma) const;
    double strike_price() const;
```

The above code shows the header file for the black_scholes_option class. This class forms the functor that is passed to the template function that is described shortly. The member functions call_option_price and put_option_price simply instantiate call and put objects from the Call and Put classes and then call the member function OptionPrice from the respective classes to return the corresponding price of the option for a given sigma value (volatility). This class will allow us to pass a Black-Scholes option object (functor) to a function template with its spot price, strike price, risk-free rate, and expiry time attached which is crucial for our implementation since the sigma value (implied volatility) is what we are trying to work out.

1.2.3 newton_raphson.h

newton_raphson.h

```
template<typename T,
        double (T::*f)(double) const,
        double (T::*f_prime)(double) const>

double newton_raphson(double target,double init1,
                      double init2, double tol,
                      const T& func, double method)
```

The above code is from the newton_raphson.h header file and it shows the function template and the declaration of the Newton-Raphson root finding function. A pointer to a member function is used in order allow us to specify the function that is to be called at compile time in conjunction with the template. The template accepts an object of type T (this is the functor) and two pointers to member functions of the functor, f and f_prime (the derivative of f). To invoke these member functions which each take 1 double as parameters we simply use the syntax (func.*f)(x) and (func.*f_prime)(x) for some double x. Pointers to member functions were used since calls to these function pointers can be inlined as they form the template parameters and can be evaluated at compile time. This is one of the reasons why a function template was used instead of inheritance for this part. Inheritance would require the use of virtual functions which would be significantly slower since virtual functions cannot be inlined. We will use the black_scholes_option object as the functor, and the two pointers to member functions will be for call_option_price or put_option_price (depending on whether we are working with a call or a put) and option_vega, these will then be used by the newton_raphson method to calculate the implied volatility.

The Newton-Raphson root finding function itself takes in 6 parameters, target is the target value of the option, init1 is the first initial guess of implied volatility, init2 is the second guess, tol is the tolerance used for the termination condition, func is the functor (black scholes option in our case), and method is the type of method used, Newton-Raphson or secant (1 for Newton-Raphson, 2 for secant). The actual implementation of the Newton-Raphson and secant method can be found in the newton_raphson.h file, the client has the option of choosing which root finding method to use. If they select 1 the Newton-Raphson method will be used with init1 as the initial guess, if they select 2 then the secant method will be used with init1 and init2 as the two initial guesses. This is where some error handling is necessary since the programme takes user input in the form of a .txt file. If a value other than 1 or 2 is provided then a runtime error is thrown prompting the client to correct the input and subsequently caught in the main file.

newton_raphson.h

```
double y = (func.*f)(init1);  // Initial option price
double x = init1; // Initial volatility - first guess
double x2 = init2; // Second guess

// If method == 1 do the Newton-Raphson method
if(method == 1)
{
    while (std::fabs(y - target) > tol) {
        double dx = (func.*f_prime)(x); // vega
        x += (target - y) / dx; // update initial guess
        y = (func.*f)(x); // current option value
    }
    return x;
}
// If method == 2 do the Secant method
else if(method == 2)
{
    y = (func.*f)(init2);
    while (std::fabs(y - target) > tol) {
        double dx = ( (func.*f)(x) - (func.*f)(x2) ) / (x - x2); // vega
        double x_temp = x2 + (target - y) / dx; // calculate current vol
        x = x2; // Update first guess
        x2 = x_temp; // Update second guess
        y = (func.*f)(x2); // current option value
    }
    return x2;
}
```

The above code is from the newton_raphson.h file and it shows the implementation of the Newton-Raphson method (method 1) and the secant method (method 2). For the Newton-Raphson method we require one initial guess for the volatility, this is given by init1 which the method takes and assigns it to a variable x. The initial option price under this initial guess for the volatility is then calculated by invoking the member function f (which is a function to calculate the price of the option) as (func.*f)(x) and assigning the price to a variable y. The method also takes in a target value, this is the market price of the option. The while loop works in a three step procedure, while the absolute value of the difference between the target option price and the current price y is greater than the tolerance: calculate the vega by invoking the member function f_prime as (func.*f_prime)(x), update the initial guess according to the Newton-Raphson rule and calculate the current option price under the updated guess.

The secant method can be thought of as the Newton-Raphson method with a finite difference approximation for the derivative of a function f (this is the vega in our case). As such, the code for the secant method (method 2) is identical to the code for the Newton-Raphson method except for the fact that instead if invoking the member function f_prime (analytical formula for the vega of the option) we instead approximate the vega using a finite difference at each iteration, from there, the algorithm continues in the same way that the Newton-Raphson method does except we update a second guess too.

1.3 Operator Overloading

The data.h and data.cpp files define a struct to hold all of the input data from the input.txt that is used in the code. Within the struct we overload the extraction operator, », for input streams so that it precisely assigns each element of the struct to the corresponding input provided by the client in the input.txt file. This makes the code much more readable and efficient in the main file. In addition to this, we also implememnt some exception handling as part of overloading the extraction operator. The code requires the user to enter ”call” or ”put” in the first element of the input.txt file, if the client fails to do this an invalid argument exception is thrown which prompts the client to specify the option type correctly. Furthermore, if the client fails to provide an input file completely then a cerr is thrown. The exceptions are caught in the main file.

As part of the project we were required to automatically output an ”output.txt” file in a specific format. The output_file.h and output_file.cpp files define a struct to hold all of the data that will be outputted. Within this struct we overload the operator, «, in order to send to the output file the strike price and the implied volatility under both root finding methods for a specific call and a put as required by the specification. This again makes the code much more readable and efficient in the main file.

1.4 Structure of the main.cpp file

The main.cpp file collates all of the code together. First, the Newton-Raphson and secant method parameters are initialised. Two arrays each of length 2 used to hold the implied volatilities for the output file are declared, these are not related to the user’s input they are simply used for the required output.txt file. Finally, The double ”sigma” will hold the value of the implied volatility of the option under the user specified input. The remaining bulk of the main.cpp file is encapsulated in a try-catch block to catch any exceptions that may be thrown. Firstly a struct is instantiated and populated with the input data from input.txt using the overloaded extraction operator, ». Secondly, a Black-Scholes option functor is constructed by instantiating an object of the black_scholes_option class with the parameters from the struct holding the client input data. From here, there is an if-else statement that executes the corresponding code depending on whether the user specified a call or a put. In either case, the template function takes in the Black-Scholes option functor along with the 2 member functions giving the price of the option and the vega of the option. The newton_raphson method then calculates the implied volatility which is output to the console. The last part of the main.cpp creates another Black-Scholes option functor with the parameters required by the specification for the output file. The implied volatilities are calculated and all of the data is stored in an output_file struct, finally, the contents of the output_file struct are output to file called ”output.txt” using the overloaded operator, «. All of the exceptions are then caught after the try block and the code terminates.

2 Results and Analysis

2.1 Initial Testing

In order to test the program it is a good idea to test its individual components first before testing it as one entity. We started this by first testing that the d_1 and d_2 values where being correctly calculated for some known values with predefined input parameters, this is easy to check since there exists analytical formulas for these. Next, the option pricing functions for the calls and puts where tested for predefined input values with a known call/put price and checking that the function returns the correct price. In addition to this, we employ the same method with the function that calculates the vega of the options, this is also easy to check since there exists an analytical formula for the vega of calls and puts. The different root finding methods where also tested against known solutions and performed accurately as would be expected.

2.2 Speed/Efficiency

The code executes in a very short period of time when using either root finding method. This was tested using the chrono library which provides the system_clock() method. The run time varies every time the program is ran but it always runs very fast, usually taking between 3000000 ns and 5000000 ns (nanoseconds) for either root finding method with varying parameters. This is to be expected as a result of the use of the various methods used such as inline calls to function pointers, templates, and the fact that the actual task at hand is not very computationally expensive in the first place. With regards to the two root finding methods, the speed and efficiency of both of them are extremely good and neither significantly outperforms the other in this metric.

2.3 Accuracy

Here we present our results as well as an assessment of their accuracy. In subsequent discussion S is the asset price, K is the strike price, r is the risk-free interest rate, T is the expiry time in years, and F is the market price of the option. The columns of the results tables is the tolerance level used for each run of the program and the rows of the table indicate the root finding method used, the entries of the table are the implied volatilities computed for the corresponding tolerance value and root finding method used.

Table 1: Call 1, implied volatilities with decreasing tolerance from 1 to 0.0001. Call option parameters : S = 1157.86, K = 1250, r = 0.02, T = 0.0602739726 (years), F = 1.5.

	1	0.8	0.5	0.2	0.1	0.05	0.01	0.001	0.0001
Newton-Raphson	0.2203	0.1997	0.1997	0.1997	0.1997	0.1973	0.1973	0.1973	0.1973
Secant	0.1982	0.1982	0.1982	0.1982	0.1982	0.1982	0.1973	0.1973	0.1973

Table 2: Call 2, implied volatilities with decreasing tolerance from 1 to 0.0001. Call option parameters : S = 1157.86, K = 1125, r = 0.02, T = 0.0602739726 (years), F = 43.

	1	0.8	0.5	0.2	0.1	0.05	0.01	0.001	0.0001
Newton-Raphson	0.1991	0.1991	0.1991	0.1947	0.1947	0.1947	0.1947	0.1946	0.1946
Secant	0.1950	0.1950	0.1950	0.1950	0.1950	0.1950	0.1947	0.1946	0.1946

Table 3: Put 1, implied volatilities with decreasing tolerance from 1 to 0.0001. Put option parameters : S = 1157.86, K = 1125, r = 0.02, T = 0.0602739726 (years), F = 19.7.

	1	0.8	0.5	0.2	0.1	0.05	0.01	0.001	0.0001
Newton-Raphson	0.3000	0.3000	0.3054	0.3054	0.3054	0.3054	0.3054	0.3054	0.3054
Secant	0.3000	0.3000	0.3056	0.3056	0.3056	0.3056	0.3054	0.3054	0.3054

Table 4: Put 2, implied volatilities with decreasing tolerance from 1 to 0.0001. Put option parameters : S = 1157.86, K = 1100, r = 0.02, T = 0.0602739726 (years), F = 6.60.

	1	0.8	0.5	0.2	0.1	0.05	0.01	0.001	0.0001
Newton-Raphson	0.2414	0.2414	0.2414	0.2375	0.2375	0.2375	0.2375	0.2374	0.2374
Secant	0.2338	0.2338	0.2338	0.2371	0.2371	0.2371	0.2374	0.2374	0.2374

From the results tables it can be seen that for all of the options tested the value of the implied volatility computed by the program converges to a value as the tolerance is gradually decreased under both root finding methods: Newton-Raphson and the secant method. Furthermore, the implied volatilities computed under both root finding methods converge to the same value, this is indicative of the accuracy of the implementation. Additionally, it is important to point out that there exits an analytical formula for the vega of European calls and puts and the implementation of the Newton-Raphson method makes use of this. Nonetheless, the secant method is also implemented which approximates the vega of the option and since under the secant method the implied volatilities converge to the same values of those under the Newton-Method which uses the analytical form of the vega, we can be sure that the approximation has been accurately implemented. In addition the program returns an implied volatility value that is non-negative and between 0 and 1 which is a requirement.

This analysis is extended by the graphs shown in figure 1. Each graph shows the implied volatilities computed under both root finding methods (blue line is Newton-Rapson, orange line is secant method) as the degree of tolerance is decreased from 1 to 0.0001. Looking at the graphs from right to left we can clearly see that as the tolerance is decreased the path of the implied volatility under both methods intersect as the tolerance gets closer and closer to zero. In some cases, as can be seen for the case of put option 1, the path taken by both root finding methods is very similar indeed. In general it seems as though the Newton-Raphson method starts higher than the secant method but there is not much absolute difference between the implied volatilities at all.

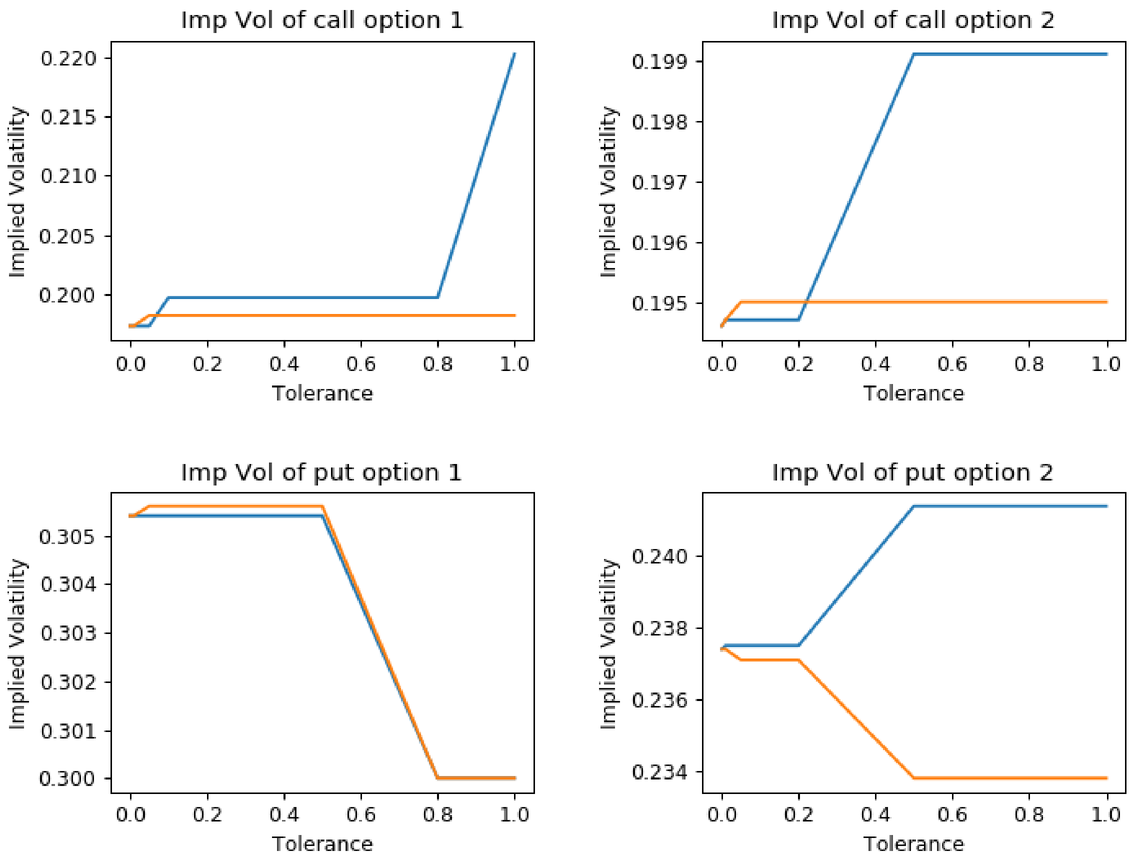

Figure 1: The blue line represents the Newton-Raphson method and the orange line represents the secant method

3 Discussion and Conclusions

This program has the required functionality in an efficiently implemented manner. The use of a template function with a functor and class inheritance make the code easily extendable to deal with other types of options or root finding methods. For instance, the option pricing base class could easily have a derived class that implements a monte-carlo simulation in order to calculate the prices of options where analytical solutions do not exist. An improvement to the code could be made by perhaps creating a separate exception handler class that derives from std::exception, this could make exception handling more flexible, easier to use, and enhance readability. A further extension could be to put the programme into wrappers so as to keep parts of the programme conceptually separate.