C++ For Finance, Project 1 - Heston Model with Monte-Carlo Method

Student ID: 1423101

University of Birmingham
Msc Mathematical Finance

1. Introduction

Important: Before running the code, please ensure that the working directory in CLion is set to the project directory to allow the additional .txt files to be read. This can be done by clicking the drop down arrow between the build icon and the run icon in the top right of the CLion window and selecting "edit configurations", a window will pop up with a section called "Working Directory", click the three dots to the right and another window will open that allows you to select the working directory, from here at the top of the window the third icon from the left can be clicked to automatically select the project directory (alternatively using Ctrl+2 also does this), click "ok", and then "apply".

This project required an implementation of the Heston stochastic volatility model with a Monte-Carlo method. The Heston Stochastic volatility model assumes that the stock follows the stochastic process

$$dS_t = rS_t dt + \sqrt{\nu_t} S_t dW_t^S$$

while the variance ν_t is governed by the following equation

$$d\nu_t = \kappa(\theta - \nu_t)dt + \sigma\sqrt{\nu_t} dW_t^\nu.$$

r is the risk free interest rate, κ volatility mean reversion rate, θ long run average volatility, and σ volatility of volatility. W_t^S, W_t^ν are Brownian motion with correlation coefficient ρ.

2. Description Of The Code

2.1. Running The Code

Attention must first be drawn to the Input Data.txt file. This file allows the user to specify the stochastic parameters, the Monte-Carlo parameters along with the option parameters, and finally, it allows the user to choose between a standard GBM or Heston model simulation. The Input Data.txt can be edited in a text editor or can be edited directly in CLion (or any other IDE). It's structure is as follows, the first line denotes the Heston model parameters in the following order: rho, volatility of volatility, kappa, and theta, each separated by one space. The second line denotes the Monte-Carlo and option data in the following order, initial stock price at time 0, risk free interest rate, initial volatility, strike price, expiry time (in years), number of simulations, and the number of time-steps, each separated by one space. The third line denotes the type of option being simulated, there are 6 choices, Call, Put, LookbackCallFloatingStrike, LookbackCallFixedStrike, LookbackPutFloatingStrike, and LookbackPutFixedStrike, only one is to be specified at a time and must be entered in the exact way they have just been typed. The fourth and final line denotes the type of simulation being performed, the user has the option of either using a heston model, in which case the word "heston" (lower case) is typed in, or a standard geometric Brownian motion simulation, in which case "gbm" (lower case) is typed in. It imperative that the user fills out the Input Data.txt file accurately and in the correct order in order for the code to run. In addition, all lines of the file must be completed and none left blank, for example the stochastic parameters for the Heston model line should be left there even if the user is using a geometric Brownian motion simulation. Two examples are given to illustrate it's use.

```
-0.5 0.5 5.0 0.04
100 0.05 0.04 100 1 100000 100
LookbackPutFixedStrike
gbm
```

Figure 1: Input Data.txt for gbm simulation of a European Put lookback option with fixed strike.

```
-0.5 0.5 5.0 0.04
100 0.05 0.04 100 1 100000 100
Call
heston
```

Figure 2: Input Data.txt for heston simulation of a European Call option.

2.2. Structures

There are several variables related to either Geometric Brownian Motion(GBM), options, or the Heston model. For this reason two structures were used to group related information together and make this information easily accessible. The structure *MC_data* holds the Monte-Carlo and option parameters, namely, the initial value of the stock, risk free interest rate, initial volatility, strike price, expiry date, number of sample paths, and the number of time steps. The second structure, *Heston_data*, contains all of the parameters used in the Heston model, namely, the correlation between the Brownian motions, volatility of mean reversion rate, long run average volatility, and volatility of volatility.

2.3. Header files/namespaces

This project made use of the library code provided in addition to extra namespaces being created for various components.

2.3.1. heston.h, heston.cpp

The heston namespace was created to keep the calculations regarding the stochastic data together. In this namespace there is 1 structure *Heston_data* used to store the parameters as described above, and 3 functions, *volatility_path*, *stock_path*, and ReadInHestonData which calculate the volatility path, the stock price path and read in the parameters into the structure respectively. The function *volatility_path* takes in vectors vol_draws and vol_path by reference, and 2 structures each containing Monte-Carlo data and heston data respectively. Figure 3 shows an image of the function declaration in the heston.cpp file. The entire volatility path was generated at once in a vector and was simulated using an Euler method as follows:

$$\nu_{i+1} = v_i + \kappa(\theta - \nu_i^p)\delta t + \sqrt{\nu_i^p(\delta t)} W_i^v$$

The stock path was generated in a similar way, again in one step as a vector so as to make use of vector methods to find the maximum and minimum stock prices over the entire path which will be used for some of the lookback options. Figure 4 shows the function declaration in the heston.cpp file of the *stock_path* function. It uses the volatility path to simulate the stock path using the exact solution in the same manner as before:

$$S_{i+1} = S_i exp((r - 0.5\nu_i^p)\delta t + \sqrt{\nu_i^p \delta t} W_i^S$$

2.3.2. monte_carlo.h, monte _carlo.cpp

The Monte-Carlo namespace (mc) has a structure *MC_data* which holds the Monte-Carlo and option parameters as described above, there are also 5 prototype functions in *monte_carlo.h*, namely *ReadInMonteCarloData* which reads in the Monte-Carlo and option parameter data from the second line of Input Data.txt, and *Accumulate, ComputeValues, DoOutput* which are all associated with combining outputs of various functions in order to return the atual option value to the console, these 3 prototype functions are used in both the heston simulation and the standard GBM simulation. *Accumulate* sums the payoff values calculated and the squared payoff values, *ComputeValues* computes the option value and the standard error, and *DoOutput* prints to the screen the option value, the standard error, and the time taken for the code to run. Finally, *Next_S* (used only for GBM simulation) works out the next *S* for geometric brownian motion which assumes a constant volatility, and uses normal random variables generated from the *rv_library*.

2.3.3. payoff.h, payoff.cpp

The payoff namesapce (pf) was created in order to make it easy for the user to add or alter any of the payoff functions should this be required. It also makes it much easier to change between payoffs in the main function for testing and benchmarking purposes. There are 5 function declarations in the *payoff.h* file, *Call, Put, LookbackCallFloatingStrike, LookbackCallFixedStrike, LookbackPutFloatingStrike*, and *LookbackPutFixedStrike*. The payoff functions are extremely simple to calculate and the function definitions are present in the *payoff.cpp* file, each function returns a double.

2.4. main.cpp

Before discussing the structure of the *main.cpp* two important features are highlighted, the generation of correlated normal random variables, and reading the Input Data.txt file. In order to work out the stock path and volatility path described above, we require W_i^S and W_i^ν to be random N(0,1) variables which are correlated with coefficient ρ. In order to do this we simply generate two random normal variables Y_1, Y_2 using the random variable library provided, and then set $X_1 = Y_1$ and $X_2 = \rho Y_1 + \sqrt{1 - \rho \cdot \rho} Y_2$ so that X_1 and X_2 are now correlated with coefficient ρ. This was done in a for loop and each time the correlated normal random variables were added to vectors to store the normal draws as can be seen in figure 5 which shows the implementation in CLion.

The next feature in the *main.cpp* is reading the Input Data.txt file and processing the data. This was done using *ifstream*, which simply reads each line of the text file until a new line has been reached where-after it moves to the next line. Once the file is open, the *ReadInHestonData* function from the heston namespace is used to fill up the heston structure, *ReadInMonteCarloData* from the Monte-Carlo namespace is used to fill up the mc structure, and finally the option type and the model type specified by the user are stored in strings for later use. If the file could not be opened an error is thrown. This can all be seen in figure 6.

The rest of the main function uses the namespace's and structures described above to simulate the option price of a user specified option using either Heston model or a standard geometric Brownian motion model. First the Input Data.txt is opened and read and the corresponding operations executed. From there on the main function is split into an if-else statement, the if statement corresponding to the user specifying "heston" as the model type in Input Data.txt, and the else statement corresponding to the user specifying "gbm" as the model type. It is important to note that for the lookback options the maximum value of the stocks over the entire path are required, in order to achieve this we use the vector methods std::min_element and std::max_element.

3. Results and Analysis

To begin the analysis and justification of the simulations we start by noting the fact there are exact solutions for prices of a European call and put under GBM given by the following:

$$C = S_0 N(d_1) - Kexp(-rT)N(d_2)$$

$$P = Kexp(-rT)N(-d_2) - S_0 N(-d_1).$$

This is used as a benchmark to compare the implementation of the standard GBM with Monte-Carlo method against for calls and puts. With the parameters set as $S_0 = 100, r = 0.05, sigma = 0.04, X = 100, T = 1, M = 1000000, N = 1000$, the implementation returns, on average, a call price of 5.08272 and a put price of 0.197335. The exact solutions to the above equations indicate that the exact values of the call and put under GBM are 5.07438 and 0.197324 respectively. This amounts to only a 0.16422% difference between the exact value of the call and the Monte-Carlo value, and a 0.00557% difference between the exact value of the put and the Monte-Carlo value. This is strong evidence that the Monte-Carlo method is accurate under standard GBM.

There are no simple closed form solutions to the price of the options under the Heston model as can be seen by the fact that the volatility path and stock paths have to be generated using correlated normal random variables and simulated using the Euler method detailed earlier. For this reason it is necessary to carry out checks and analyse the effects of increasing the number of simulations and the time-step on the option prices. We will consider the results for the vanilla call first, the same methodology and results are extended to the other types of options valued in this implementation. Consider the European

Vanilla call under the Heston model, starting with just 5 simulations and increasing up to 10000000 keeping the time-step constant at 100, initially it can be seen that the model returns option prices for the call that are varying in size and there does not seem to be any obvious answer, however, once the number of simulations reaches around 1000 and above, it is clear that the option prices are converging. This data is shown in the table 1. It is clear that the call price is converging to a value of about 10.xxx.

# Simulations	Call price
0	0
5	8.027
10	6.226
20	14.562
40	11.271
80	9.601
100	10.032
500	10.186
1000	10.191
10000	10.213
10000000	10.319

Table 1: Option prices for Call for increasing number of simulations

Now, we consider changing the time-step and keeping the number of simulations constant at 1000000, and we can see that there a similar pattern emerges. Table 2 shows that as the number of time-steps is increased, the price of the call under the Heston model again begins to converge after a point to around 10.xxx, this is indicative that the prices are accurate. This methodology of increasing the number of simulations and the number of time steps can be used on all of the other options to yield the exact same trend, after a certain point they all begin to converge to a price and this tells us that the software is producing the correct price for the model.

Time-step	Call price
5	9.056
10	9.751
20	9.981
30	10.114
40	10.236
50	10.280
80	10.270
100	10.288
500	10.311
1000	10.299

Table 2: Option prices for Call for increasing # of time-steps

The above data has been shown graphically in figures 7 and 8 which show that in the long run increasing the number of simulations and increasing the number of time-steps deos indeed cause the Call option under the heston model to converge to a price, again it is reiterated that this same methodology and trend is seen amongst the other options, puts and lookbacks for both standard GBM and the Heston model simulations. From the tables and graphs it can be seen that the optimal number of time-steps is around 100 time-steps, after this point the price of the option remains fairly constant and one could apply Occam's razor to argue that any further increase in the number of time-steps after 100 is not worth the added computational cost and run time since there are minimal performance improvements after this point. The same can be said of the number of simulations, it can be seen that the optimal number of simulations is around 10,000 since beyond this point the option price does not oscillate much.

Another method to asses the performance of the Heston model is to use the software developed to price options based on real past market data. The following paper [1] gives the information needed to calculate the price of a European Vanilla call option for Google Inc. (GOOG). However, it is important to point out that the stochastic parameters of the Heston model we have implemented have not been calibrated to any market data, despite this we will see that the program still manages to fairly accurately price the option. The market price of European call option for Google Inc.(GOOG) recorded on Apr 6, 2013 was 272.90 for the following parameters, $S\_0 = 783.05, r = 0.000151644, \sigma = 0.04, X = 510, T = 0.112328767$. Using our implementation (with the default Heston parameters in Input Data.txt) we find that the Heston model with 100000 simulations and 100 time steps returns an option price of 273.126, and the standard GBM model returns an option price of 273.159, for either of models it can be seen that increasing the number of simulations and the number of time-steps again causes the option price to converge. There are other option prices present in the paper [1] that can also be tested and similar results are obtained. Table 3 shows the results of the option price for each of the European options this program can price, using 1000000 simulations and 100 time-steps, and using the stock data and Heston model parameters given to us in the project specification.

Option	Heston Price	GBM price
Call	10.3471	5.0705
Put	5.4892	0.1992
LookbackCallFixedStrike	16.3261	6.1028
LookbackCallFloatingStrike	16.7319	6.0602
LookbackPutFixedStrike	11.7722	1.1815
LookbackPutFloatingStrike	11.5134	1.2506

Table 3: European option prices under Heston and GBM model

4. Discussion and Conclusions

The program has the required functionality in that it is able to take input data from a user and calculate the price of European vanilla calls puts and lookback options under either a standard geometric Brownian motion simulation or a Heston simulation. The degree to which the code is generalisbale is also fairly high, there are namespace's and structs used which can easily be modified and built upon in order to incorporate other payoff func-

tions or other models. However, there is room for improvement when it comes to the run time of the program, one possible area of improvement could be to use a lookup table, or using a hash function to map the strings to an enum in order to use switch statements for checking which payoff function is to be used. Also, instead of using vector methods to find the minimum and maximum values of the stocks over the entire path an improvement in performance may be achieved by keeping track of the minimum and maximum values throughout the path generation processes.

References

[1] Y. Yang. Valuing a european option with the heston model. 2013.

```
void heston::volatility_path(const std::vector<double> &vol_draws,
        std::vector<double> &vol_path,
        mc::MC_data mc,
        heston::Heston_data heston)
{
    size_t size =  vol_draws.size();
    double dt = mc.T/mc.N;

    for (size_t i=0; i<size-1; i++) {
        double max = std::max(vol_path[i], 0.0);
        vol_path[i+1] = vol_path[i] + heston.kappa * dt * (heston.theta - max) +
                heston.vol_of_vol * sqrt(max * dt) * vol_draws[i];
    }
}
```

Figure 3: Function to calculate volatility path.

```
void heston::stock_path(const std::vector<double> &spot_draws, const std::vector<double> &vol_path,
       std::vector<double> &spot_path, mc::MC_data mc)
{
    size_t size = spot_draws.size();
    double dt = mc.T/mc.N;

    for (size_t i=0; i<size-1; i++) {
        double max = std::max(vol_path[i], 0.0);
        spot_path[i+1] = spot_path[i] * exp( (mc.r - 0.5*max)*dt +
                                        sqrt(max*dt)*spot_draws[i]);
    }
}
```

Figure 4: Function to calculate stock path.

```
for (int i = 0; i < mc.N-1; i++)
{
    double y_1 = rv::GetNormalVariate();
    double y_2 = rv::GetNormalVariate();
    double x_1 = y_1;
    double x_2 = heston.rho * y_1 + sqrt(1 - heston.rho * heston.rho) * y_2;
    spot_draws[i] = x_1;
    vol_draws[i] = x_2;
}
```

Figure 5: Calculating correlated normal variables

```
double rho, vol_of_vol, kappa, theta;
double S_0, r, sig, X, T;
long M, N;
std::string option, option_type, model, model_type;
std::ifstream file_("Input Data.txt");
if(file_.is_open())
{
    while(file_ >> rho >> vol_of_vol >> kappa >> theta >> S_0 >> r >> sig >> X >> T >> M >> N >> option >> model)
    {
        heston::ReadInHestonData(heston, rho, vol_of_vol, kappa, theta);
        mc::ReadInMonteCarloData(mc, S_0, r, sig, X, T, M, N);
        option_type = option;
        model_type = model;
    }
    file_.close();
}
else
    throw std::runtime_error("File cannot be opened");
```

Figure 6: Reading input data from Input Data.txt

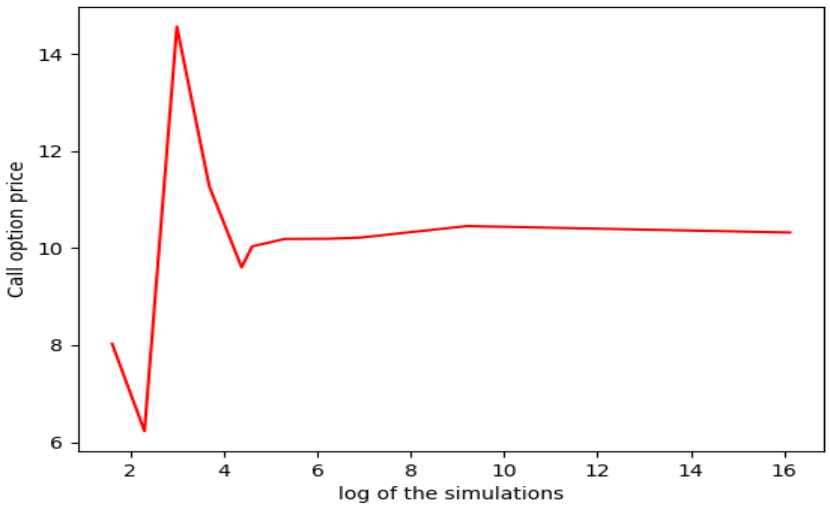

Figure 7: Convergence of call option price with increasing simulations under Heston model

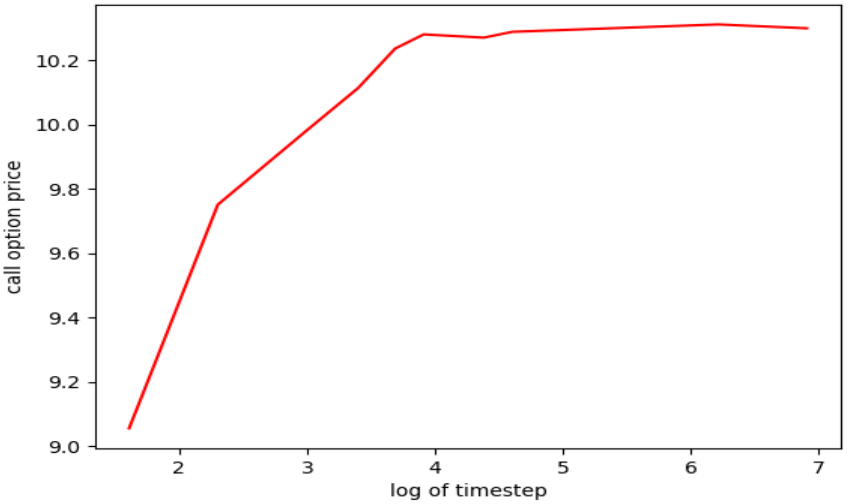

Figure 8: Convergence of call option price with increasing # of timesteps under Heston model