**WQU Master’s in Financial Engineering**

**Course: Computational Finance**

**Group work Assignment Submission 2 M 5**

**Group members**:

[**Tewodros Ayalew (ttaayyll32@gmail.com)**](https://masters.wqu.org/user/view.php?id=2308&course=99)[**Nishan Agarwal (agarwalnishan5@gmail.com)**](https://masters.wqu.org/user/view.php?id=14514&course=99)[**Akshay Kulkarni (akshaypk2012@gmail.com)**](https://masters.wqu.org/user/view.php?id=14760&course=99)[**Dhruv Agrawal (dhruva1@stanfordalumni.org)**](https://masters.wqu.org/user/view.php?id=21612&course=99)

**Nicholas Munu (nicholasmunu@yahoo.com)**

**Contents**

**1. Introduction**

**2. Task 1**

**3. Task 2**

**4. Task 3**

**5. Task 4**

**6. Summary and Conclusion**

**7. References**

# **Introduction**

## Background of the project

In this project, we are expected to price a vanilla Europeancall option using a simple Fourier pricing technique assuming that the underlying share follows the Heston model dynamics. Then we price a vanilla Europeancall option assuming CEV dynamics using Monte Carlo simulations.

**Task 1**: Using a simple Fourier pricing technique (using 𝑁= 100 intervals, and using an effective upper bound of integration of 30), price a vanilla call option assuming that the underlying share follows the Heston model dynamics

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt |

We import the necessary packages.

|  |
| --- |
| #Initializing the variables from previous section r = 0.08    # risk-free rate  T = 1               # Time to maturity of option K = 100        # Strike price of option S0 = 100             #Initial stock value  sigma = 0.3             # Stock volatility  k\_log = np.log(K) |

The relevant variables are initialized:

* Risk-free rate, *𝑟* of 8%
* Initial stock value, 𝑆\_0 of $100
* Strike price of option, *K* of $100
* Time to maturity of option, T, of 1 year
* Stock volatility, sigma, 30%

k\_log is initialized as as log of strike price

|  |
| --- |
| #Initializing the variable values given in this project v0 = 0.06 kappa = 9 theta = 0.06 rho = -0.4 |

New parameters are introduced in this section for time varying volatility under Heston model.

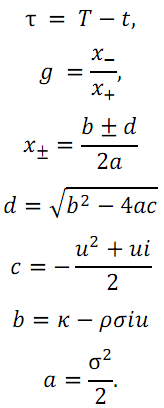
|  |
| --- |
| #Approximation information t\_max = 30 N = 100 |

For Fourier technique, t\_max (=30) is the upper bound of our discretized interval for t, and N (=100) is the upper bound for the sum.

|  |
| --- |
| #Characteristic function code  a = sigma\*\*2/2  def b(u):     return kappa - rho\*sigma\*1j\*u  def c(u):     return -(u\*\*2+1j\*u)/2  def d(u):     return np.sqrt(b(u)\*\*2-4\*a\*c(u))  def xminus(u):     return (b(u)-d(u))/(2\*a)  def xplus(u):     return (b(u)+d(u))/(2\*a)  def g(u):     return xminus(u)/xplus(u)  def C(u):     val1 = T\*xminus(u) - np.log((1-g(u)\*np.exp(-T\*d(u)))/(1-g(u)))/a     return r\*T\*1j\*u +theta\*kappa\*val1  def D(u):     val1 = 1 - np.exp(-T\*d(u))     val2 = 1 - g(u)\*np.exp(-T\*d(u))     return (val1/val2)\*xminus(u)  def log\_char(u):     return np.exp(C(u)+ D(u)\*v0 + 1j\*u\*np.log(S0))  def adj\_char(u):     return log\_char(u-1j)/log\_char(-1j) |

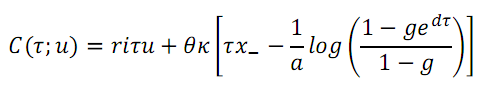
We need to define the above functions so that we can create a function (based on these functions) for the characteristic function.

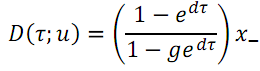
From a = sigma\*\*2/2 to def g(u):, we are simply defining the following functions



For def C(u) to def D(u), we are defining the functions below with the idea that t = 0 thus

τ = T

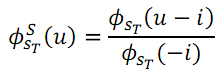




For def log\_char(u) we are defining the characteristic equation below



def adj\_char(u) is the function corresponding to



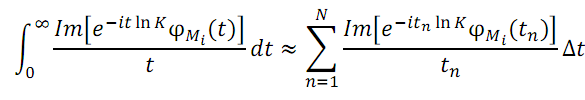
|  |
| --- |
| delta\_t = t\_max/N from\_1\_to\_N = np.linspace(1,N,N) t\_n = (from\_1\_to\_N -1/2)\*delta\_t |

We align ourselves to apply Fourier Technique to the integrals in the vanilla call price by creating new variables that we are going to use in the summation approximations.

|  |
| --- |
| first\_integral = sum((((np.exp(-1j\*t\_n\*k\_log)\*adj\_char(t\_n)).imag)/t\_n)\*delta\_t) second\_integral = sum((((np.exp(-1j\*t\_n\*k\_log)\*log\_char(t\_n)).imag)/t\_n)\*delta\_t) |

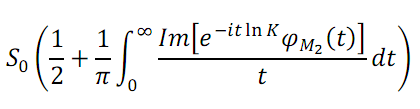
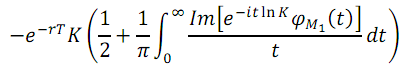
Then we calculate an estimate for the integrals in the above code.

For the first integral in particular the estimate is calculated as



|  |
| --- |
| fourier\_call\_val = S0\*(1/2+first\_integral/np.pi) - np.exp(-r\*T)\*K\*(1/2+second\_integral/np.pi) |

Finally, we can calculate the Fourier estimate of our call price. The formula used is below:



C =

**Output**

**13.734895692109077**

**So the value of the vanilla call option is 13.73**

**Task 2 -** Assume that 𝜎 (𝑡𝑖,𝑡𝑖+1)=𝜎(𝑆𝑡𝑖)𝛾−1, where σ= 0.3 and 𝛾 = 0.75. Using the formula below



*where* 𝑆𝑡𝑖 *is the share price at time* 𝑡𝑖,σ(𝑡𝑖,𝑡𝑖+1) *is the volatility for the period* [𝑡𝑖,𝑡𝑖+1],𝑟

*is the risk-free interest rate, and* 𝑍~𝑁(0,1).

simulate the paths for the underlying share using sample sizes of 1000, 2000, …, 50000.

|  |
| --- |
| import matplotlib.pyplot as plt from scipy.stats import ncx2 from scipy.stats import norm |

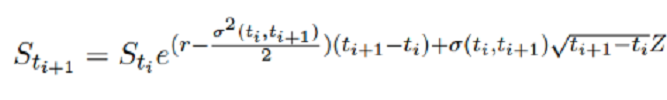
Here we import the necessary packages. Pyplot is imported to help in visualization. The noncentral Chi-squared distribution is imported to help in CEV calculations. The normal distribution is imported because we are going to generate normal random numbers for Z.

|  |
| --- |
| def terminal\_value(St, r, sigma, Z,T, gamma):          sigmat = sigma\*((St)\*\*(gamma - 1))     return St\*np.exp((r  - sigmat\*\*2/2)\*T + sigmat\*np.sqrt(T)\*Z) #Calculating next month's stock value using previous month's stock value,  #risk free rate, sigma,simulated matrix of N(0,1) random variables,  #time frame of 1 month (from the previos month), and gamma      def option\_payoff(S\_T,K):     return np.maximum(S\_T - K,0) #The option\_payoff function finds the payoff for a call option at terminal  #time for given terminal stock price and strike price. |

The function terminal\_value() is used to calculate the share price value for the underlying share using the value at the previous time step after updating the value of volatility for this time step.

The equation for volatility is given by



We return the stock value for the next month using the value in the previous month with the equation below :

The function option\_payoff() defines the payoff for a call option at a terminal share price for a given strike price.

|  |
| --- |
| np.random.seed(0) #fixing the seed for result replication  moption\_estimates = [None]\*50 #monte carlo option estimates not incorporating default risk moption\_std = [None]\*50    #standard deviation of monte carlo option estimates  gamma = 0.75 |

In the above code, we have fixed the seed for result replication. We initialize the arrays of size 50 for Monte Carlo option estimates, along with each of their respective standard deviations. We have also fixed the value of gamma for taking into account variable volatility.

|  |
| --- |
| for i in range (1,51):       size = i\*1000 #sample sizes set ranging from 1000 to 50,000 with increments of 1000     #in each loop this is the number of paths simulated          stock\_val = np.zeros(size)     # The value of stock is initialized to zero for each simulation.          prevstock\_val = S0          for month in range(1,12\*T+1):  #simulation from month 1 to month 12\*T where T is number of years              norm\_matrix = norm.rvs(size = np.array([1,size]))                          stock\_val = terminal\_value(prevstock\_val,r, sigma,                                    norm\_matrix,1/12.0, gamma)                  #We calculate the stock's value for this month using              #previos month's stock value, risk free rate, sigma,              #simulated matrix of N(0,1) random variables, time frame of 1 month             # (from the previos month), and gamma                          prevstock\_val = stock\_val |
|  |

We will simulate paths for the underlying share using sample sizes of 1000, 2000, …, 50000.

For each path we will simulate the stock price for a year using monthly simulations. Storing the previous value of the stock price on this path (and using the random normal variable that has been generated for that particular path) we calculate the next period value of the stock price using the terminal\_value() function.

**Task 3 -** Augment your code in part 2 to calculate Monte Carlo estimates, as well as the standard deviations for these estimates, for the price of a vanilla call option (with the same strike term as in Submission 1).

|  |
| --- |
| option\_val = np.zeros(size)             for j in range(0,size):         option\_val = np.exp(-r\*T)\*option\_payoff(stock\_val,K)              #finds the discounted call payoff           moption\_estimates[i-1] = np.mean(option\_val)     moption\_std[i-1] = np.std(option\_val)/np.sqrt(size) |

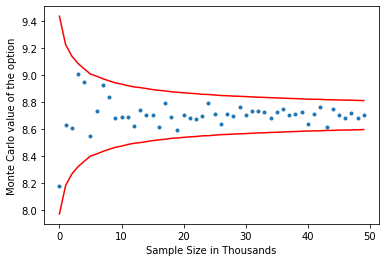
In the above block of code, we find the discounted call option payoff by using the function option\_payoff() for each simulation. We then calculate the Monte Carlo estimates (as the mean of the option values corresponding to the different share price paths in each simulation) and the standard deviations of those estimates for each sample size.

**TASK 4:** Plot the Monte Carlo estimates generated in part 3 with respect to sample size, as well as three standard deviation error bounds around these estimates.

|  |
| --- |
| #Plotting the graph for Monte Carlo estimates plt.plot(moption\_estimates,'.') plt.plot(np.mean(moption\_estimates)+np.array(moption\_std)\*3, 'r') plt.plot(np.mean(moption\_estimates)-np.array(moption\_std)\*3, 'r') plt.xlabel("Sample Size in Thousands") plt.ylabel("Monte Carlo value of the option") |

The above is the code for plotting the Monte Carlo value of the optionalong with 3 standard deviation error bounds for each simulation.

**Output**



The above is the plot for the **Monte Carlo value of the option** along with 3 standard deviation error bounds for the various simulations – we can see that the mean of the data points comes to be around **8.7** which is also the closed form CEV value of the option that is calculated in the below code

|  |
| --- |
| #closed-form solution to the value of a call option z = 2 + 1/(1-gamma)  def CF(t,K):  kappa = 2\*r/(sigma\*\*2\*(1-gamma)\*(np.exp(2\*r\*(1-gamma)\*t)-1))  x = kappa\*S0\*\*(2\*(1-gamma))\*np.exp(2\*r\*(1-gamma)\*t)  y = kappa\*K\*\*(2\*(1-gamma))  return S0\*(1-ncx2.cdf(y,z,x)) - K\*np.exp(-r\*t)\*ncx2.cdf(x,z-2,y)  CF(T,K) |

**Output:** **8.702**

Conclusion and Summary

In this exercise we are pricing European vanilla call option considering time varying volatility.

In Task 1, we used a Fourier pricing technique to price a vanilla European call option

assuming that the underlying share follows the Heston model dynamics and got the price to be **13.73**.

In Tasks 2,3, and 4, we calculated the Monte Carlo estimates for the price of the same option with CEV dynamics and get the mean of the estimates to be around **8.70** which is the same as the closed form solution of the option calculated under CEV dynamics. We plot the Monte Carlo estimates along with 3 standard deviation error bounds for the various simulations (in the plot as we increase the sample size, the standard deviation error bounds decrease in value).

References

1) Chen James (2019) Vanilla Option Definition, Investopedia. Available at: https://www.investopedia.com/terms/v/vanillaoption.asp (Accessed: 10 October 2019).

2) QuantStart (2019) European vanilla option pricing with C++ via Monte Carlo methods | QuantStart, QUANT ACADAMY. Available at: https://www.quantstart.com/articles/European-vanilla-option-pricing-with-C-via-Monte-Carlo-methods (Accessed: 10 October 2019).

3) Rouah, F. D. (2013) The Heston Model and Its Extensions in Matlab and C#, The Heston Model and Its Extensions in Matlab and C#. doi: 10.1002/9781118656471.

4) Schmelzle, M. (2010) ‘Option pricing formulae using Fourier transform: Theory and application’, Preprint, http://pfadintegral. com. Available at: http://pfadintegral.com/docs/Schmelzle2010 Fourier Pricing.pdf.

5) Heston, S. L. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. The review of financial studies, 6(2), 327-343.

6) Albrecher, H., Mayer, P., Schoutens, W., & Tistaert, J. (2007). The little Heston trap. Wilmott, (1), 83-92