



Submission: 02

Group: 27

Members:

Name	Country	Mail	Member not contributing(X)
Vivek Verma	India	vivektheintel@gmail.com	
Christian Chihababo	Congo	aganzechihababo@gmail.com	
Harshil Sumra	India	harshilsumra1997@gmail.com	

Integrity Statement:

Vivek Verma, Christian Chihababo, Harshil Sumra

Member not contributing:

1. Pricing Vanilla European Call Option

The parameters used for the pricing:-

Symbol	Description	Value
T	Option Maturity	1.0
S_0	Current Stock price	100.0
K	Option Strike price	100.0
sigma (σ)	Volatility	0.30
r	Risk-free rate	0.08
V_0	Initial Variance	0.06
κ	Kappa	9
θ	Long run Variance	0.06
ρ	Correlation	-0.40

The assumption of the dynamics which the asset will follow under the Heston Model:

$$dS_t = \mu S dt + \sqrt{v_t} S dW_t^1$$

The vol term v_t follows: $dv_t = -\beta \sqrt{v_t} dt + \sigma dW_t^2$

The W_t^2 is Brownian Motion (standard) having a correlation, ρ with W_t^1 .

The Characteristic function can be written as:

$$\phi_{s_T} = e^{C(\tau;u) + D(\tau;u)v_t + iu \log(S_t)}$$

where

$$C(\tau;u) = ri\tau u + \theta\kappa \left[\tau x_- - \frac{1}{a} \log \left(\frac{1 - ge^{d\tau}}{1 - g} \right) \right]$$

$$D(\tau;u) = \left(\frac{1 - e^{-d\tau}}{1 - ge^{-d\tau}} \right) x_-$$

$$\tau = T - t$$

$$g = \frac{x_-}{x_+}$$

$$x_{\pm} = \frac{b \pm d}{2a}$$

$$d = \sqrt{b^2 - 4ac}$$

$$c = -\frac{u^2 + ui}{2}$$

$$b = \kappa - \rho\sigma iu$$

$$a = \frac{\sigma^2}{2}$$

Then the pricing can be done with characteristic functions

$$c = S_0 \left(\frac{1}{2} + \frac{1}{\pi} \int_0^\infty \frac{Im[e^{-it \ln K} \varphi_{M_2}(t)]}{t} dt \right) - e^{-rT} K \left(\frac{1}{2} + \frac{1}{\pi} \int_0^\infty \frac{Im[e^{-it \ln K} \varphi_{M_1}(t)]}{t} dt \right)$$

where

$$\varphi_{M_1}(t) = \phi_{s_T}(t)$$

$$\varphi_{M_2}(t) = \frac{\phi_{s_T}(u - i)}{\phi_{s_T}(-i)}$$

Python version: 3.8.10

For the given parameters,

- The Price of the Vanilla Call Option turned out to be **\$13.7349**

The code snippets are as follows:-

1. Price a European Vanilla call option with the information provided using simple Fourier pricing technique, assuming that the underlying share follows the Heston Model dynamics

```
1 T = 1.0
2 S_0 = 100.0
3 K = 100.0
4 sigma = 0.30
5 r = 0.08
6
7 v0 = 0.06
8 kappa = 9
9 theta = 0.06
10 rho = -0.40
11
12 t_max = 30
13 N = 100
14
15 k_log = np.log(K)
16 a = sigma**2 / 2
```

Figure 1: Initializing the variables

```
1 def calc_b(u, kappa, rho, sigma):
2     """Calculate the value of b using u, kappa, rho and sigma"""
3     return kappa - rho * sigma * 1j * u
4
5
6 def calc_c(u):
7     """Calculate the value of u"""
8     return -(u ** 2 + 1j * u) / 2
9
10
11 def calc_d(u, kappa, rho, sigma):
12     """Calculate the value of d using u, kappa, rho, sigma"""
13     return np.sqrt(calc_b(u, kappa, rho, sigma) ** 2 - 4 * a * calc_c(u))
14
15
16 def calc_xminus(u, kappa, rho, sigma):
17     """Calculate the value of x- using u, kappa, rho, sigma"""
18     return (calc_b(u, kappa, rho, sigma) - calc_d(u, kappa, rho, sigma)) / (2 * a)
19
20
21 def calc_xplus(u, kappa, rho, sigma):
22     """Calculate the value of x+ using u, kappa, rho, sigma"""
23     return (calc_b(u, kappa, rho, sigma) + calc_d(u, kappa, rho, sigma)) / (2 * a)
24
25
26 def calc_g(u, kappa, rho, sigma):
27     """Calculate the value of g using u, kappa, rho, sigma"""
28     return calc_xminus(u, kappa, rho, sigma) / calc_xplus(u, kappa, rho, sigma)
29
30
31 def calc_C(u, kappa, rho, sigma):
32     """Calculate the value of C using u, kappa, rho, sigma"""
33     val1 = T * calc_xminus(u, kappa, rho, sigma) - np.log((1 - calc_g(u, kappa, rho, sigma)) * np.exp(-T * calc_d(u, kappa, rho, sigma))) / (1 - calc_g(u, kappa, rho, sigma)) / a
34     return r * T * 1j * u + theta * kappa * val1
35
36
37 def calc_D(u, kappa, rho, sigma):
38     """Calculate the value of D using u, kappa, rho, sigma"""
39     val1 = 1 - np.exp(-T * calc_d(u, kappa, rho, sigma))
40     val2 = 1 - calc_g(u, kappa, rho, sigma) * np.exp(-T * calc_d(u, kappa, rho, sigma))
41     return (val1 / val2) * calc_xminus(u, kappa, rho, sigma)
42
```

Figure 2: Functions written for the entire compute - 1

```

36
37 def calc_D(u, kappa, rho, sigma):
38     """Calculate the value of D using u, kappa, rho, sigma"""
39     val1 = 1 - np.exp(-T * calc_d(u, kappa, rho, sigma))
40     val2 = 1 - calc_g(u, kappa, rho, sigma) * np.exp(-T * calc_d(u, kappa, rho, sigma))
41     return (val1 / val2) * calc_xminus(u, kappa, rho, sigma)
42
43
44 def calc_log_char(u, kappa, rho, sigma, S0):
45     """Calculate the value of log_char using u, kappa, rho, sigma and S_0"""
46     return np.exp(calc_C(u, kappa, rho, sigma) + calc_D(u, kappa, rho, sigma) * v0 + 1j * u * np.log(S0))
47
48
49 def calc_adj_char(u, kappa, rho, sigma, S0):
50     """Calculate the value of adj_char using u, kappa, rho, sigma, S_0"""
51     return calc_log_char(u - 1j, kappa, rho, sigma, S0) / calc_log_char(-1j, kappa, rho, sigma, S0)
52
53
54 def calc_first_integral(t_n, delta_t, k_log, kappa, rho, sigma, S0):
55     """Calculate the value of first integral using t_n, delta_t, k_log, kappa, rho, sigma, S_0"""
56     return ((np.exp(-1j * t_n * k_log) * calc_adj_char(t_n, kappa, rho, sigma, S0)).imag / t_n) * delta_t
57
58
59 def calc_second_integral(t_n, delta_t, k_log, kappa, rho, sigma, S0):
60     """Calculate the value of second integral using t_n, delta_t, k_log, kappa, rho, sigma, S_0"""
61     return ((np.exp(-1j * t_n * k_log) * calc_log_char(t_n, kappa, rho, sigma, S0)).imag / t_n) * delta_t
62
63
64 def calc_fourier_call_price(S0, r, T, K, t_n, delta_t, k_log, kappa, rho, sigma):
65     """Calculate the value of the Call price using Fourier method which follows the Heston Dynamics"""
66     first_integral = calc_first_integral(t_n, delta_t, k_log, kappa, rho, sigma, S0)
67     first_integral_sum = sum(first_integral)
68     second_integral = calc_second_integral(t_n, delta_t, k_log, kappa, rho, sigma, S0)
69     second_integral_sum = sum(second_integral)
70     return S0 * (1 / 2 + first_integral_sum / np.pi) - np.exp(-r * T) * K * (1 / 2 + second_integral_sum / np.pi), first_integral, second_integral
71
72
73 def plot_fourier_first_second_integral(first_intg, second_intg):
74     """Plotting function of the 2 integrals"""
75     plt.plot(first_intg, label="1st integral")
76     plt.plot(second_intg, label="2nd integral")
77     plt.xlabel("t")
78     plt.ylabel("Integrand value")
79     plt.legend()
80     plt.show()

```

Figure 3: Functions written for the entire compute - 2

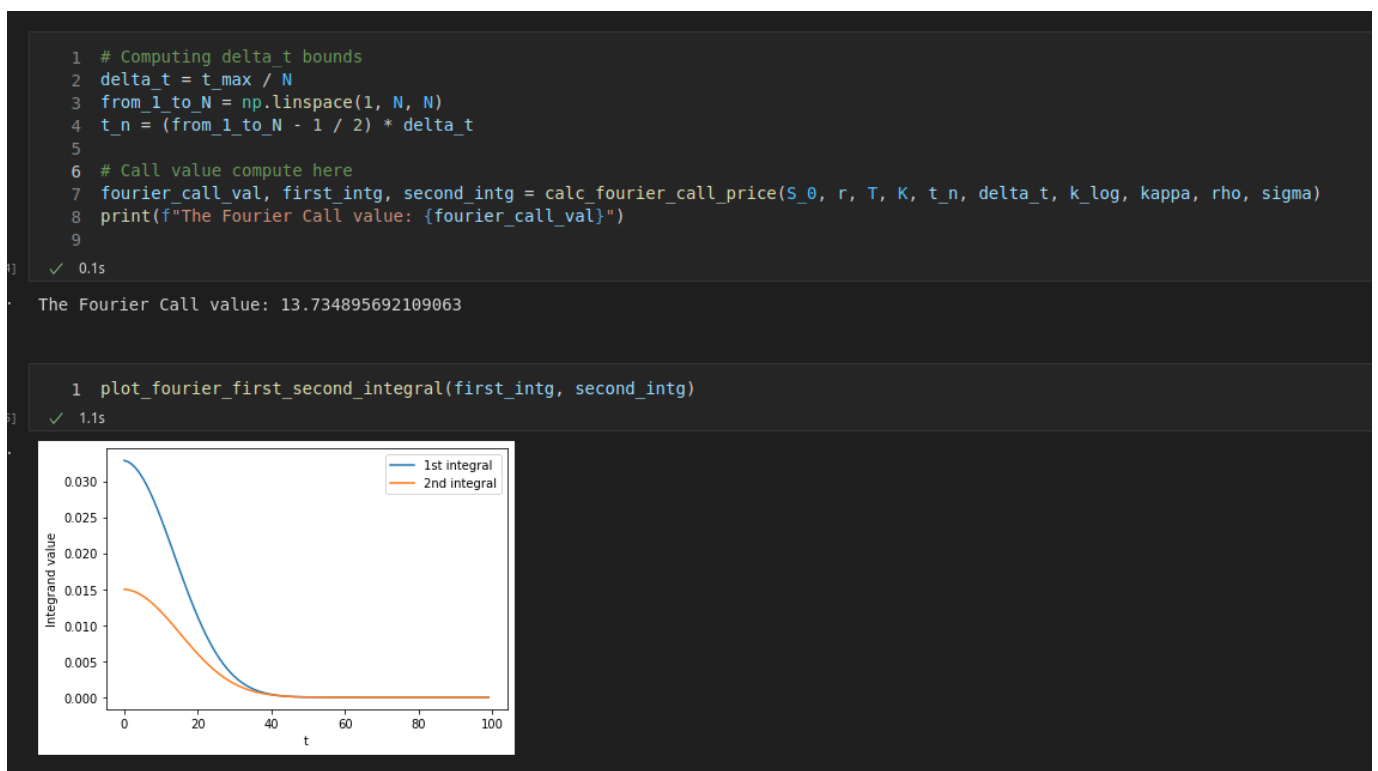


Figure 4: Displaying the Vanilla Call Option pricing result

∴ Concludes the Vanilla Call Option Pricing!

2. Simulate share path using the CEV Model

Using $\sigma = 0.3, \gamma = 0.75$

$$\sigma(t_i, t_{i+1}) = \sigma(S_{t_i})^{\gamma-1}$$

Share path simulation formula:

$$S_{t_{i+1}} = S_{t_i} e^{(r - \frac{\sigma^2(t_i, t_{i+1})}{2})(t_{i+1} - t_i) + \sigma(t_i, t_{i+1})\sqrt{t_{i+1} - t_i}Z}$$

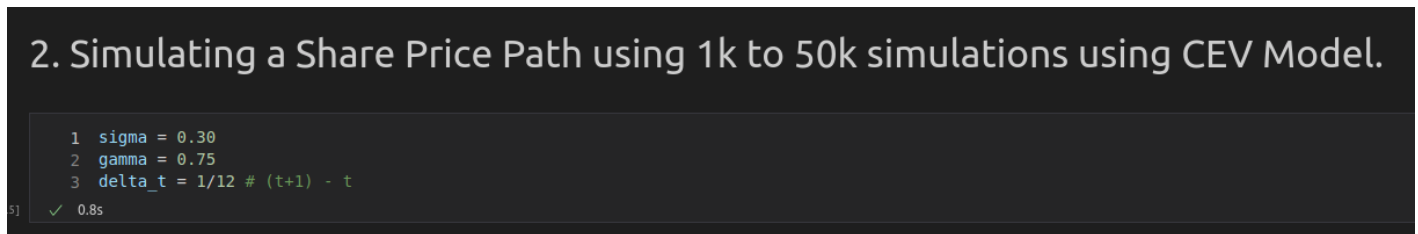
where, $Z \sim N(0, 1)$

For the given parameters,

- The Average Share price turned out to be **\$108.335**

Python version: 3.8.10

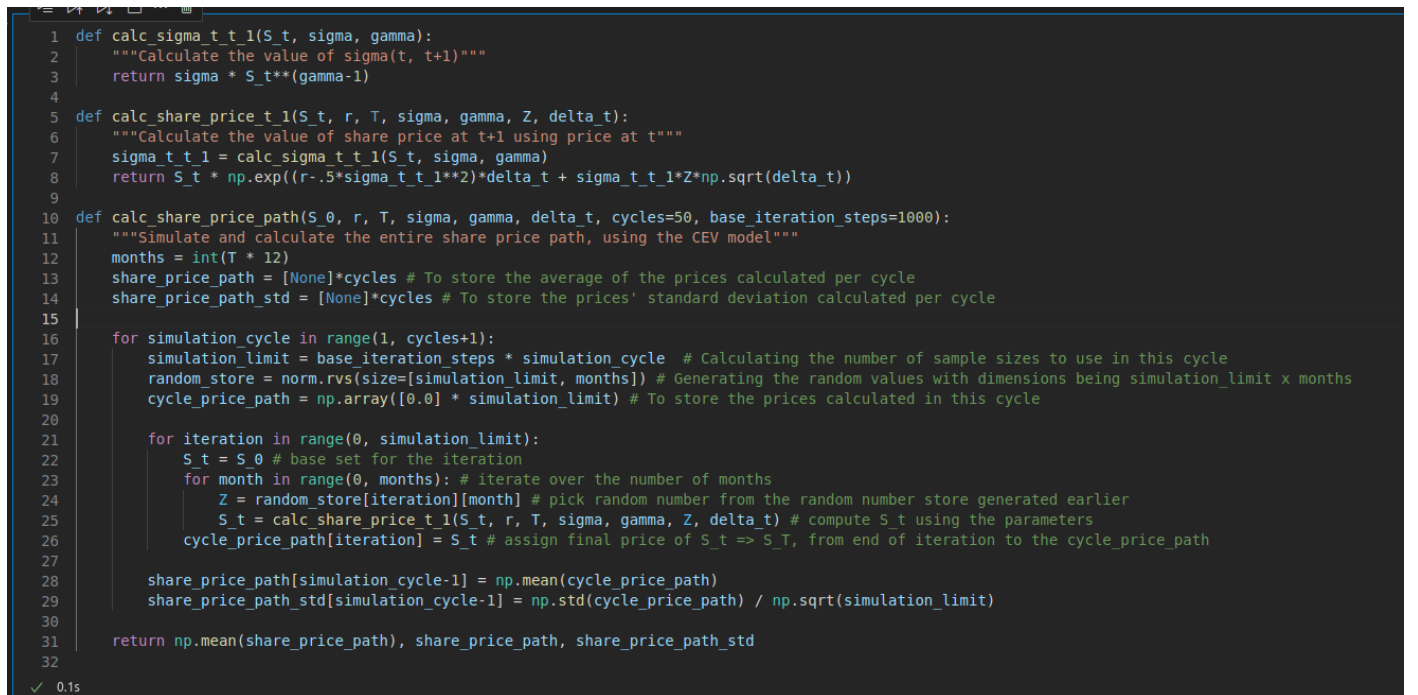
The code snippets are as follows:-



```
2. Simulating a Share Price Path using 1k to 50k simulations using CEV Model.

1 sigma = 0.30
2 gamma = 0.75
3 delta_t = 1/12 # (t+1) - t
5] ✓ 0.8s
```

Figure 5: Initializing the variables



```
1 def calc_sigma_t_t1(S_t, sigma, gamma):
2     """Calculate the value of sigma(t, t+1)"""
3     return sigma * S_t**(gamma-1)
4
5 def calc_share_price_t1(S_t, r, T, sigma, gamma, Z, delta_t):
6     """Calculate the value of share price at t+1 using price at t"""
7     sigma_t_t1 = calc_sigma_t_t1(S_t, sigma, gamma)
8     return S_t * np.exp((r - .5*sigma_t_t1**2)*delta_t + sigma_t_t1*Z*np.sqrt(delta_t))
9
10 def calc_share_price_path(S_0, r, T, sigma, gamma, delta_t, cycles=50, base_iteration_steps=1000):
11     """Simulate and calculate the entire share price path, using the CEV model"""
12     months = int(T * 12)
13     share_price_path = [None]*cycles # To store the average of the prices calculated per cycle
14     share_price_path_std = [None]*cycles # To store the prices' standard deviation calculated per cycle
15
16     for simulation_cycle in range(1, cycles+1):
17         simulation_limit = base_iteration_steps * simulation_cycle # Calculating the number of sample sizes to use in this cycle
18         random_store = norm.rvs(size=[simulation_limit, months]) # Generating the random values with dimensions being simulation_limit x months
19         cycle_price_path = np.array([0.0] * simulation_limit) # To store the prices calculated in this cycle
20
21         for iteration in range(0, simulation_limit):
22             S_t = S_0 # base set for the iteration
23             for month in range(0, months): # iterate over the number of months
24                 Z = random_store[iteration][month] # pick random number from the random number store generated earlier
25                 S_t = calc_share_price_t1(S_t, r, T, sigma, gamma, Z, delta_t) # compute S_t using the parameters
26                 cycle_price_path[iteration] = S_t # assign final price of S_t => S_T, from end of iteration to the cycle_price_path
27
28             share_price_path[simulation_cycle-1] = np.mean(cycle_price_path)
29             share_price_path_std[simulation_cycle-1] = np.std(cycle_price_path) / np.sqrt(simulation_limit)
30
31     return np.mean(share_price_path), share_price_path, share_price_path_std
32
✓ 0.1s
```

Figure 6: Functions written for the entire compute

```

1 cycles = 50
2
3 from datetime import datetime
4 dt1 = datetime.now()
5 # triggering the actual compute for 50 cycles
6 mean_share_price_path, share_price_path, share_price_path_std = calc_share_price_path(
7     S_0=S_0,
8     r=r,
9     T=T,
10    sigma=sigma,
11    gamma=gamma,
12    delta_t=delta_t,
13    cycles=cycles
14 )
15
16 print(f"Mean Share Price from the paths: $ {mean_share_price_path:.3f}")
17 delta = datetime.now() - dt1
18 print(f"Time taken for the compute: {delta}")
19
[17] ✓ 1m 58.1s
... Mean Share Price from the paths: $ 108.335
Time taken for the compute: 0:01:57.969789

```

Figure 7: Calculating and displaying the Mean Share price path compute result

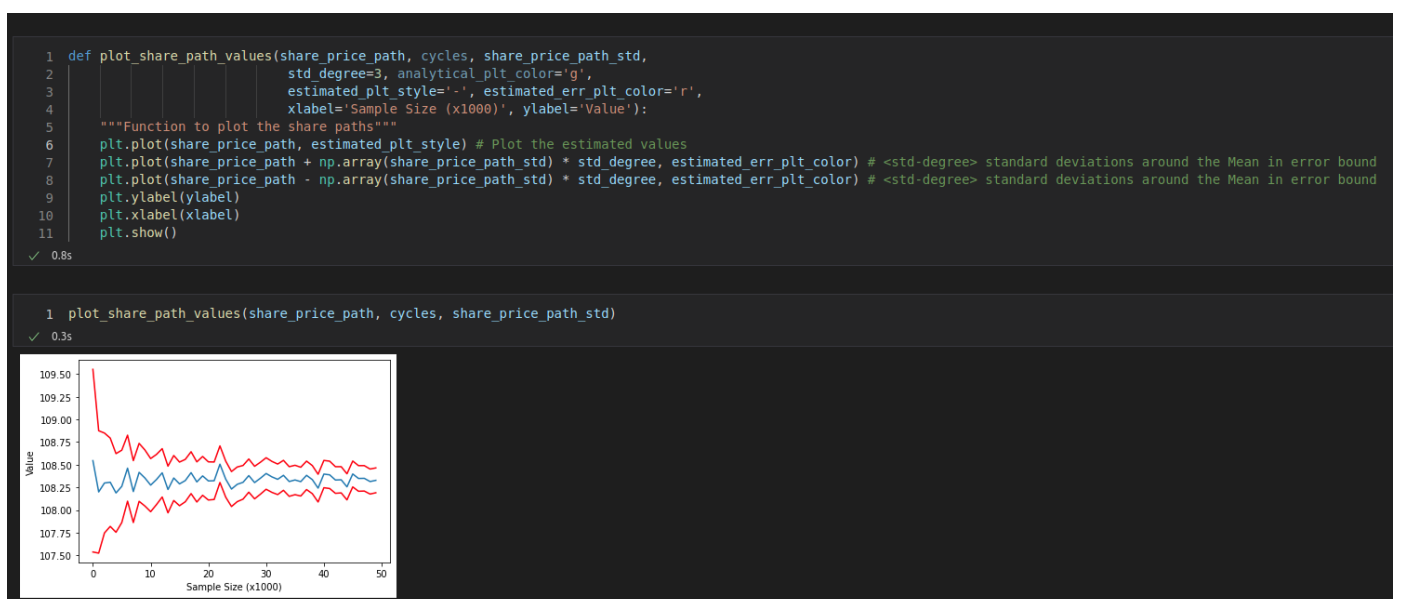


Figure 8: Plotting the entire Share price computation over the 50 cycles

Ans.3.

Here, we define the a function by the name of 'call_price_and_stddev' which helps us generate Monte Carlo estimates and their standard deviation for a combination of (t,N) -> (time left in years, no of simulations).

```
In [11]: #pricing call option based on above model
dt=1/12
gamma=0.75
def call_price_and_stddev(t, N):
    n = int(t/dt)
    Z = norm.rvs(size =[N, n])
    price_path = np.array([[np.float64(S_0)]*(n+1)]*N)
    for i in range(n):
        vol = sigma*price_path[:,i]**(gamma-1)
        power = (r-vol**2/2)*dt+vol*np.sqrt(dt)*Z[:,i]
        price_path[:,i+1]=price_path[:,i]*np.exp(power)
    pay_off = np.maximum(price_path[:, -1]-K, 0)*np.exp(-r*t)
    return np.mean(pay_off), np.std(pay_off)/np.sqrt(N)
```

Then we store the corresponding mean and standard deviation value in output lists. These lists are Monte Carlo estimates and standard deviations for different simulation sample sizes(N).

```
In [12]: #o/p variables
call_price = [None]*50
call_stddev = [None]*50

# price estimates
for i in range(1, 51):
    call_price[i-1], call_stddev[i-1] = call_price_and_stddev(T, i*1000)
```

Then we calculate closed form call using CEV model equations which are as follows:

$$C(S_0, K, T, \sigma, \gamma) = -S_0 \chi(y; z, x) + K e^{-rT} (1 - \chi(x; z - 2, y))$$

Where

$$\kappa = \frac{2r}{\sigma^2(1-\gamma)e^{2r(1-\gamma)T}-1}$$

$$x = \kappa S_0^{2(1-\gamma)} e^{2r(1-\gamma)T}$$

$$y = \kappa K^{2(1-\gamma)}$$

$$z = 2 + \frac{1}{1-\gamma}$$

$\chi(.,d,\lambda)$ is cumulative distribution function for non-central chi-square distribution with 'd' degrees of freedom and 'λ' pertaining to measure of non-centrality. Then call price under CEV is calculated as follows:

```
In [13]: # CEV closed form call price
z = 2+1/(1-gamma)
def closed_form_call_price(t):
    kappa = 2*r/(sigma**2*(1-gamma)*(np.exp(2*r*(1-gamma)*t)-1))
    x = kappa*S_0**(2*(1-gamma))*np.exp(2*r*(1-gamma)*t)
    y = kappa*K**(2*(1-gamma))
    return S_0*(1-ncx2.cdf(y,z,x))-K*np.exp(-r*t)*ncx2.cdf(x,z-2,y)
```

Output of both methods are as follows:

```
In [18]: print(f'CEV closed form call option price is {closed_form_call_price(T)}')
print(f'Monte Carlo estimate of call option price is {call_price[-1]}')
print(f'Difference between the two approaches = {round(call_price[-1]-closed_form_call_price(T),3)}')
```

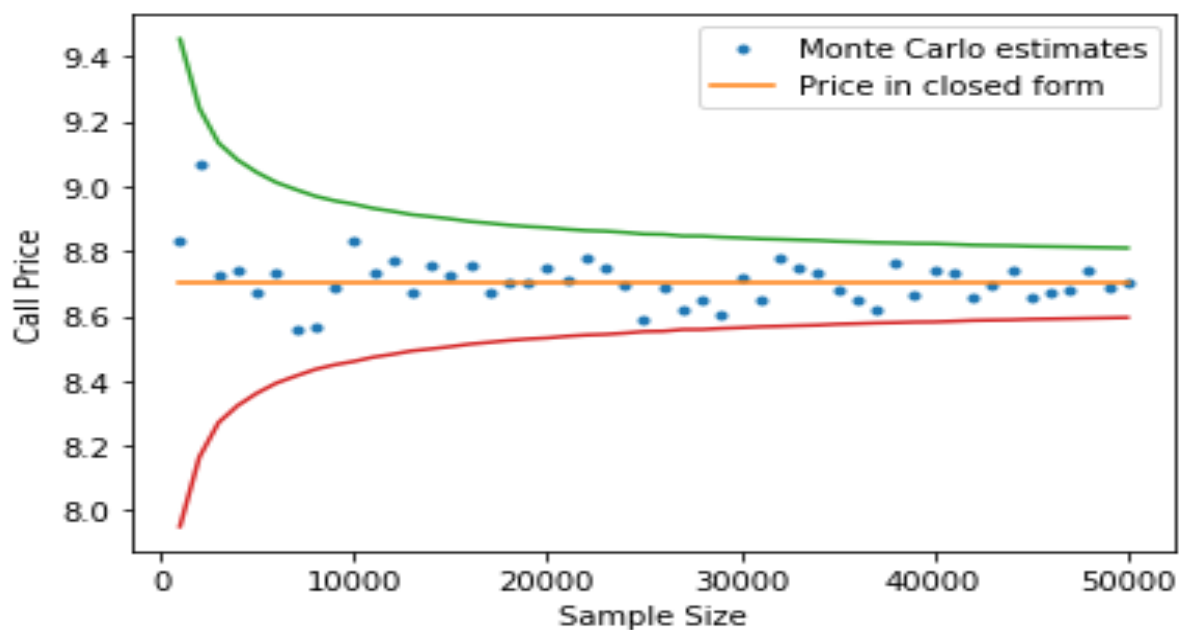
CEV closed form call option price is 8.702333534327622
Monte Carlo estimate of call option price is 8.711744547527417
Difference between the two approaches = 0.009

Ans.4.

Now we plot the both the Monte Carlo estimates and closed form price along with both above and below '3σ' boundaries about the closed form price.

```
In [14]: #Ans.4. plot of above results
plt.plot(np.array(range(1,51))*1000, call_price, '.', label="Monte Carlo estimates")
plt.plot(np.array(range(1,51))*1000, [closed_form_call_price(T)*50, label="Price in closed form")
plt.plot(np.array(range(1,51))*1000, [closed_form_call_price(T)+3*s for s in call_stddev])
plt.plot(np.array(range(1,51))*1000, [closed_form_call_price(T)-3*s for s in call_stddev])
plt.xlabel("Sample Size")
plt.ylabel("Call Price")
plt.legend()
plt.show();
```

Plot output



Question 5: Graphing the Volatility Smile

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.stats import ncx2
import random
```

```
# Package for fetching historical market data from Yahoo Finance API
!pip install yfinance
```

```
Requirement already satisfied: yfinance in ./opt/anaconda3/lib/python3.8/site-packages (0.1.70)
Requirement already satisfied: requests>=2.26 in ./opt/anaconda3/lib/python3.8/site-packages (from yfinance) (2.27.1)
Requirement already satisfied: lxml>=4.5.1 in ./opt/anaconda3/lib/python3.8/site-packages (from yfinance) (4.6.3)
Requirement already satisfied: multitasking>=0.0.7 in ./opt/anaconda3/lib/python3.8/site-packages (from yfinance) (0.0.10)
Requirement already satisfied: numpy>=1.15 in ./opt/anaconda3/lib/python3.8/site-packages (from yfinance) (1.20.1)
Requirement already satisfied: pandas>=0.24.0 in ./opt/anaconda3/lib/python3.8/site-packages (from yfinance) (1.2.4)
Requirement already satisfied: python-dateutil>=2.7.3 in ./opt/anaconda3/lib/python3.8/site-packages (from pandas>=0.24.0->yfinance) (2.8.1)
Requirement already satisfied: pytz>=2017.3 in ./opt/anaconda3/lib/python3.8/site-packages (from pandas>=0.24.0->yfinance) (2021.1)
Requirement already satisfied: six>=1.5 in ./opt/anaconda3/lib/python3.8/site-packages (from python-dateutil>=2.7.3->pandas>=0.24.0->yfinance) (1.15.0)
Requirement already satisfied: certifi>=2017.4.17 in ./opt/anaconda3/lib/python3.8/site-packages (from requests>=2.26->yfinance) (2020.12.5)
Requirement already satisfied: charset-normalizer<=2.0.0 in ./opt/anaconda3/lib/python3.8/site-packages (from requests>=2.26->yfinance) (2.0.12)
Requirement already satisfied: idna<4,>=2.5 in ./opt/anaconda3/lib/python3.8/site-packages (from requests>=2.26->yfinance) (2.10)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in ./opt/anaconda3/lib/python3.8/site-packages (from requests>=2.26->yfinance) (1.26.4)
```

```
# Fetching Historical 1 year stock price data
import yfinance as fin
data = fin.download('FB', '2018-01-01', '2019-01-01')['Adj Close']
# Sorting by Date
data.sort_index(ascending=False, inplace=True)
prices = data.reset_index()
prices.columns = ['date', 'adjClose']

[*****100%*****] 1 of 1 completed
```

```
# Daily log returns
prices['returns'] = (np.log(prices.adjClose /
    prices.adjClose.shift(-1)))

# Daily STD of returns
d_std = np.std(prices.returns)

# Annualized daily STD with SIGMA = Standard Deviation
std = d_std * 252 ** 0.5

print("Sigma (Volatility) = ", std)
```

Sigma (Volatility) = 0.39050002816034096

```
import datetime
current_date = datetime.datetime(2022, 5, 2)
stock_price = 200.47
option_expiration_date = datetime.datetime(2023, 3, 17)
# Maturity
T_days = option_expiration_date - current_date
T = T_days.days/365 # Time to maturity in years
# K = 200 , implied_volatility = 0.4910, market_price = 35.77
K = 200
sigma = 0
bK_1 = 195
bk_1_implied_volatility = 0.4936
bk_1_market_price = 38.00

bK_2 = 190
bk_2_implied_volatility = 0.5012
bk_2_market_price = 43.30

bK_3 = 185
bk_3_implied_volatility = 0.5075
bk_3_market_price = 46.17

aK_1 = 210
ak_1_implied_volatility = 0.4756
aK_1_market_price = 34.50
```

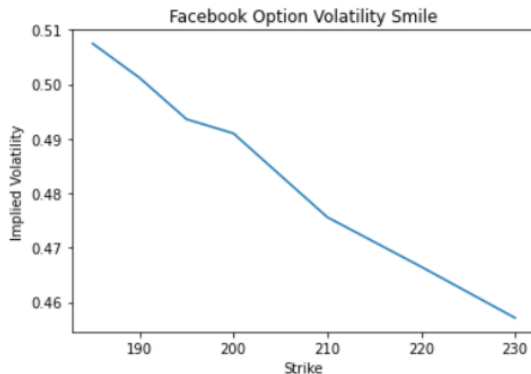
```

aK_2 = 220
ak_2_implied_volatility = 0.4665
ak_2_market_price = 26.88

aK_3 = 230
ak_3_implied_volatility = 0.4571
ak_3_market_price = 25.65

fb_option_strikes = [185, 190, 195, 200, 210, 220, 230]
fb_option_imp_vols = [0.5075, 0.5012, 0.4936, 0.4910, 0.4756, 0.4665, 0.4571]
pt.title("Facebook Option Volatility Smile")
pt.plot(fb_option_strikes, fb_option_imp_vols)
pt.ylabel("Implied Volatility")
pt.xlabel("Strike")
pt.show()

```



Question 6: Function for Computing Implied Volatility

```

# Installing library for calculating option prices, implied volatility and greeks
!pip install py_vollib

```

```

Requirement already satisfied: py_vollib in ./opt/anaconda3/lib/python3.8/site-packages (1.0.1)
Requirement already satisfied: numpy in ./opt/anaconda3/lib/python3.8/site-packages (from py_vollib) (1.20.1)
Requirement already satisfied: scipy in ./opt/anaconda3/lib/python3.8/site-packages (from py_vollib) (1.6.2)
Requirement already satisfied: py-lets-be-rational in ./opt/anaconda3/lib/python3.8/site-packages (from py_vollib) (1.0.1)
Requirement already satisfied: simplejson in ./opt/anaconda3/lib/python3.8/site-packages (from py_vollib) (3.17.6)
Requirement already satisfied: pandas in ./opt/anaconda3/lib/python3.8/site-packages (from py_vollib) (1.2.4)
Requirement already satisfied: python-dateutil>=2.7.3 in ./opt/anaconda3/lib/python3.8/site-packages (from pandas->py_vollib) (2.8.1)
Requirement already satisfied: pytz>=2017.3 in ./opt/anaconda3/lib/python3.8/site-packages (from pandas->py_vollib) (2021.1)
Requirement already satisfied: six>=1.5 in ./opt/anaconda3/lib/python3.8/site-packages (from python-dateutil>=2.7.3->pandas->py_vollib) (1.15.0)

```

```

from py_vollib.black_scholes import black_scholes as bls
from py_vollib.black_scholes.greeks.analytical import vega

tolerance = 0.00001
option_type = 'c' # Vanilla European Call Option
risk_free_rate = 0.0037 # May 02's U.S. 1 Month Treasury Rate

def implied_vol(S0,K,T,r,market_price, flag = option_type, tol = tolerance):
    """Calculating Implied Volatility for a Vanilla European Option, using Newton-Raphson Method of root finding
    S0: Stock price
    K: Strike price
    T: Time to maturity
    r: risk free rate
    market_price: Option price in the market
    """
    max_iter = 1000 # Maximum Number of iterations
    vol_old = 0.3 # Initial guess
    for j in range(max_iter):
        bls_price = bls(flag,S0,K,T,r,vol_old)
        Cprime = vega(flag,S0,K,T,r,vol_old)*100
        c = bls_price - market_price

        vol_new = vol_old - c/Cprime
        new_bls_price = bls(flag,S0,K,T,r,vol_new)
        if(abs(vol_old - vol_new) < tol or abs(new_bls_price - market_price) < tol):
            break

        vol_old = vol_new
    implied_volatility = vol_new
    return implied_volatility

K_1 = bK_1

```

```

K_2 = aK_2
market_price_1 = bK_1_market_price
market_price_2 = aK_2_market_price

implied_vol_1 = implied_vol(S0 = stock_price, K = K_1, T = T, r = risk_free_rate, market_price = market_price_1,
                             flag = option_type, tol = tolerance)

implied_vol_2 = implied_vol(S0 = stock_price, K = K_2, T = T, r = risk_free_rate, market_price = market_price_2,
                             flag = option_type, tol = tolerance)

strike_levels = [K_1, K_2]
web_imp_vol = [bk_1_implied_volatility, ak_2_implied_volatility]
cal_imp_vol = [implied_vol_1, implied_vol_2]
import pandas as pd
DF_Implied_Volatilities = pd.DataFrame([[strike_levels[0], web_imp_vol[0], cal_imp_vol[0]],
                                       [strike_levels[1], web_imp_vol[1], cal_imp_vol[1]]],
                                       columns = ["Strike Level", "Implied Volatility from the net",
                                                "Computed Implied Volatility"])

DF_Implied_Volatilities

```

	Strike Level	Implied Volatility from the net	Computed Implied Volatility
0	195	0.4936	0.477498
1	220	0.4665	0.457270

Question 7: Calculating of the Volatility Skew for Facebook

```

# Volatility skewness
delta_y = bk_3_implied_volatility - bk_1_implied_volatility
delta_x = bK_3 - bK_1
v_skew = delta_y/delta_x
print(" FB Volatility Skewness = ", v_skew )

```

FB Volatility Skewness = -0.0013899999999999967

Question 8: In Black Scholes, does the volatility depend on the strike level? Why or why not?

ANSWER

No, it does not and it is so because each asset is assumed to have constant volatility. In other words, in the Black Scholes, volatility is constant despite the maturity and/or the strike level.

Question 9: How does the Heston Model better estimate the volatility smile? Specifically, what is different in Heston's model from Black Scholes that allows more estimate option pricing?

ANSWER

Because of its extension of the relaxation of the constant volatility assumption, the Heston model performs better in estimating the volatility smile than the Black Scholes model. The model takes into account a possible correlation between the stock price and its volatility. Also, the Heston Model does not require of the stock prices to follow a lognormal probability distribution but gives a closed-form solution.

Question 10 (DISCUSSION): How the Heston model better prices the volatility smile than the Black-Scholes model.

ANSWER

Brief Story

Significant contributions to the option pricing theory were made by Economists Fischer Black and Myron Scholes in 1968 while in 1973 Robert C. Merton contributed to continuous-time finance, especially the first continuous-time option pricing model, the Black-Scholes-Merton model. They provided a simple solution for European options by considering a Geometric Brownian Motion for the underlying (Heston, 1993). Later, the Black-Scholes-Merton model was considered insufficient for real life option pricing; therefore, in 1994 Mark Edward Rubinstein rejected the theory stating that the data was a risk-free process.

Volatility Smile

Regarding volatility smile, a volatility smile curve is a plot of the different strike prices of an option contract along the abscissa axis and the implied volatility given by the Black-Scholes Model (BSM) along the ordinate axis; and as one moves from the option at-the-money to the option at-the-money or out-of-the-money, the implied volatility increases significantly. This character of volatility is observable in the market. By observing the volatility smile, the BSM assumption of constant volatility yields a straight line graph instead of a smile curve, indicating that the BSM assumption of constant volatility is incorrect.

In addition, stochastic models such as the Heston model, the SABR model, and the constant elasticity of variance (CEV) are more appropriate for option pricing since they incorporate changing volatility. In 1987, both Johnson and Shanno conducted a study of stochastic volatility. Their study of stochastic volatility attempted to address the limitations of the Black Scholes by allowing volatility to fluctuate over time.

Since then, Hull and White and Scott and Wiggins have further developed the concept of stochastic volatility and proved that volatility is a random process that fluctuates over time. In 1991, Jeremy C. Stein derived for the first time an analytical solution by assuming that volatility follows a mean-reverting process and is uncorrelated with the underlying asset returns.

In 1993, Steven "Steve" L. Heston introduced a stochastic volatility model in his paper, a model that allows for an arbitrary correlation between volatility and cash asset returns. His model involves two stochastic differential equations: (1) geometric Brownian motion that accounts for the evolution of the asset price, and (2) a Cox-Ingersoll-Ross model that accounts for the evolution of volatility. The Heston model allows the volatility of the underlying asset price to fluctuate as a random variable, and confirms the claims of Stein & Stein.

Also, in letting the price vary, the Heston model gives more accurate computations and forecasts, and also incorporates the volatility smile. While the BSM produces option prices identical to those of stochastic volatility models for at-the-money options, the Heston model allows for a greater implied volatility weighting for downward versus upward strikes.

Done By

Part	Creator
1, 2	Vivek
3, 4	Harshil
5 - 10	Christian
Compilation	Vivek

References

- [1] Ganti, Akhilesh. “How Implied Volatility (IV) Helps You to Buy Low and Sell High.” Investopedia, Investopedia, 22 Apr. 2022, <https://www.investopedia.com/terms/i/iv.asp>
- [2] Heston, S. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. The Review of Financial Studies
- [3] Hayes, Adam. “What Is the Black-Scholes Model?” Investopedia, Investopedia, 3 Apr. 2022, <https://www.investopedia.com/terms/b/blackscholes.asp>