GW_Submission_2

May 3, 2021

1 Group Work Submission 2 : Price a Vanilla European Call Option

1.1 Team Members Name.

- Xin He (xinhe9701@gmail.com)
- Ali Kone (ali.kone@rotmanalum.utoronto.ca)
- David Akanji (david.o.akanji@gmail.com)

1.1.1 Importing the Library

[1]:
```
import pandas as pd
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
from scipy import stats

plt.style.use('fivethirtyeight')
```

1.2 Question 1

We have the following parameter values: - $V_0 = 0.06$ - $\kappa = 9$ - $\theta = 0.06$ - $\rho = -0.4$

1.2.1 Initializing Parameters

[2]:
```
# Risk-free Continuously Compounded Interest rate
r = 0.08

#current share price
S0 = 100
v_0 = 0.06
kappa = 9
theta = 0.06
sigma = 0.3
rho = -0.4

#Call Specific Information
K = 100
```

```python
T = 1
k_log = np.log(K)

#Approximation Information

t_max = 30
N = 100
```

[3]:
```python
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)*v_0 + 1j*u*np.log(S0))

def adj_char(u):
    return log_char(u-1j)/log_char(-1j)
```

[4]:
```python
delta_t = t_max/N
from_1_to_N = np.linspace(1,N,N)
t_n = (from_1_to_N-1/2)*delta_t
```

2 Estimating the Integral

[5]:
```
integral_1 = sum((((np.exp(-1j*t_n*k_log)*adj_char(t_n)).imag)/t_n)*delta_t)
integral_2 = sum((((np.exp(-1j*t_n*k_log)*log_char(t_n)).imag)/t_n)*delta_t)
print(integral_1)
print(integral_2)
```

```
0.5719530211708514
0.28298394762296175
```

Calculating The Fourier Estimate of Call Price

[6]:
```
fourier_call_price = S0*(1/2 + integral_1/np.pi)-np.exp(-r*T)*K*(1/2 +␣
↪integral_2/np.pi)
print(f"The call value determined with Heston Model is {fourier_call_price:.
↪2f}")
```

```
The call value determined with Heston Model is 13.73
```

2.1 Question 2

Initializing the Parameters and the Assumptions

[7]:
```
# Major Parameters
sigma_const = 0.30
gamma = 0.75


# Assumptions

S0 = 100
T = 1
r = 0.08
timesteps = 12
sample_sizes = range(1000, 50001, 1000)
```

3 3 procedures will be adopted to solve this problem identified above

4 1st

We create a function Next_SharePrice to calculate the evolution of share prices at time `t+1` from the share price at time `t`.

[8]:
```
def Next_SharePrice(prev_price, r, dT, sigma_const, gamma, sample_size,␣
↪varying_vola = True):
    Z = stats.norm.rvs(size=sample_size)
    if varying_vola:
        sigma = sigma_const*(prev_price)**(gamma-1)
```

```
    else:
        sigma = sigma_const*(S0)**(gamma-1)

    return prev_price*np.exp((r-(sigma**2)/2)*(dT)+(sigma)*(np.sqrt(dT))*Z)
```

[9]:
```
def Next_SharePrice(prev_price, r, dT, sigma_const, gamma, sample_size,␣
 ↪const_vola = True):
    Z = stats.norm.rvs(size=sample_size)
    sigma = sigma_const*(S0)**(gamma-1) if const_vola else␣
 ↪sigma_const*(prev_price)**(gamma-1)

    return prev_price*np.exp((r-(sigma**2)/2)*(dT)+(sigma)*(np.sqrt(dT))*Z)
```

5 2nd

We created a function `share_price_path` using an empty numpy array with the shape of `X*Y` i.e `Sample size * timestamps-1`.

[10]:
```
def share_price_path(S0, r, T, sigma_const, gamma, sample_size, timesteps,␣
 ↪const_vola = True):
    df = pd.DataFrame([S0]*sample_size)
    for t in range(1, timesteps+1):
        df[t] = Next_SharePrice(df[t-1], r, 1/timesteps, sigma_const, gamma,␣
 ↪sample_size, const_vola)
    return df.T
```

6 3rd

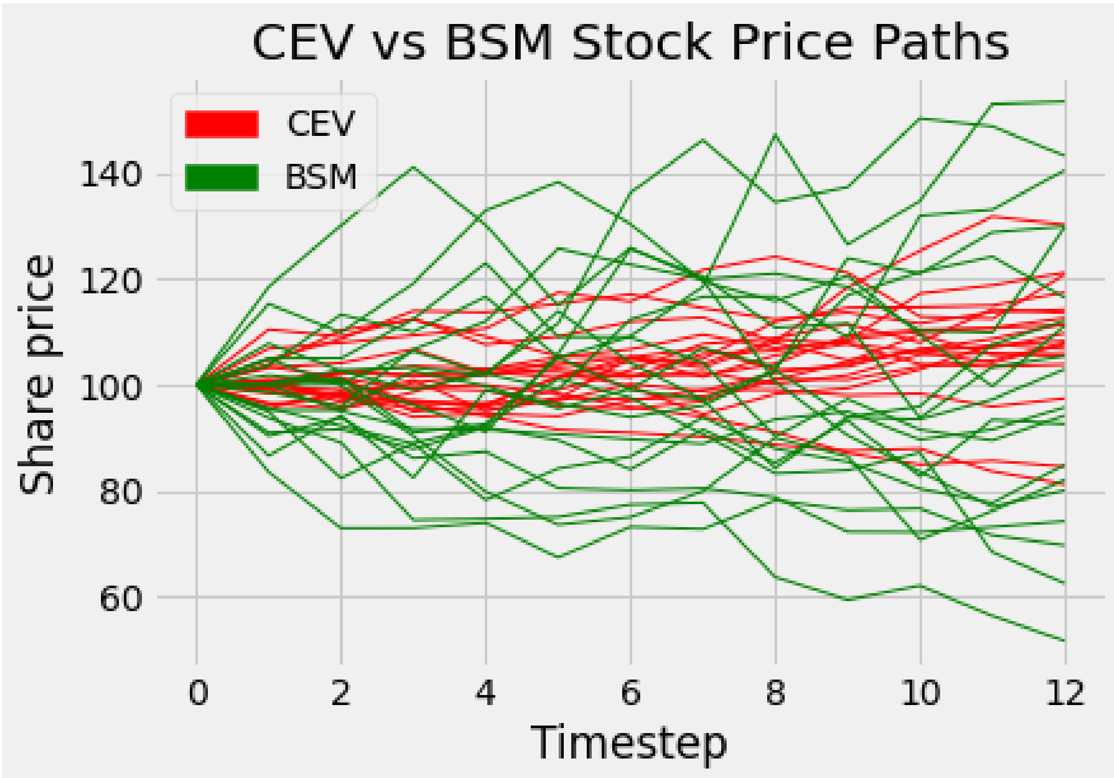
Plotting the `CEV` over the Black Scholes model (BSM), by generating the values for `Share price path CEV` and `Share price path Black Scholes`

[11]:
```
import matplotlib.patches as mpatches
T = 10
sample_size = 20

share_price_path_cev = share_price_path(S0, r, T, sigma_const, gamma,␣
 ↪sample_size, timesteps)
share_price_path_black_scholes = share_price_path(S0, r, T, sigma_const, 1.0,␣
 ↪sample_size, timesteps, const_vola=True)

plt.plot(share_price_path_cev, linewidth = 1.0, color='red')
plt.plot(share_price_path_black_scholes, linewidth = 1.0,color='green')
plt.xlabel("Timestep")
plt.ylabel("Share price")
red_patch = mpatches.Patch(color='red', label='CEV')
blue_patch = mpatches.Patch(color='green', label='BSM')
```

```
plt.legend(handles=[red_patch, blue_patch], loc='upper left')
plt.title("CEV vs BSM Stock Price Paths")
plt.show();
```



Creating a dictionary having the sample size as the key of this dictionary

[12]:
```
import time

T = 1
sample_sizes = range(1000, 50001, 1000)

Share_Price_Paths = {}

print("Generating share price path")
start = time.time()
for sample_size in sample_sizes:
    share_val = share_price_path(S0, r, T, sigma_const, gamma, sample_size,␣
 ↪timesteps, const_vola=True)

    Share_Price_Paths[sample_size] = share_val
end = time.time()
print(f"Generating all samples paths takes {(end - start):.2f}s")
```

```
Generating share price path
Generating all samples paths takes 4.52s
```

Plotting samples of the generated share paths

[13]:
```
Share_Price_Paths[1000].iloc[:,:10]
```

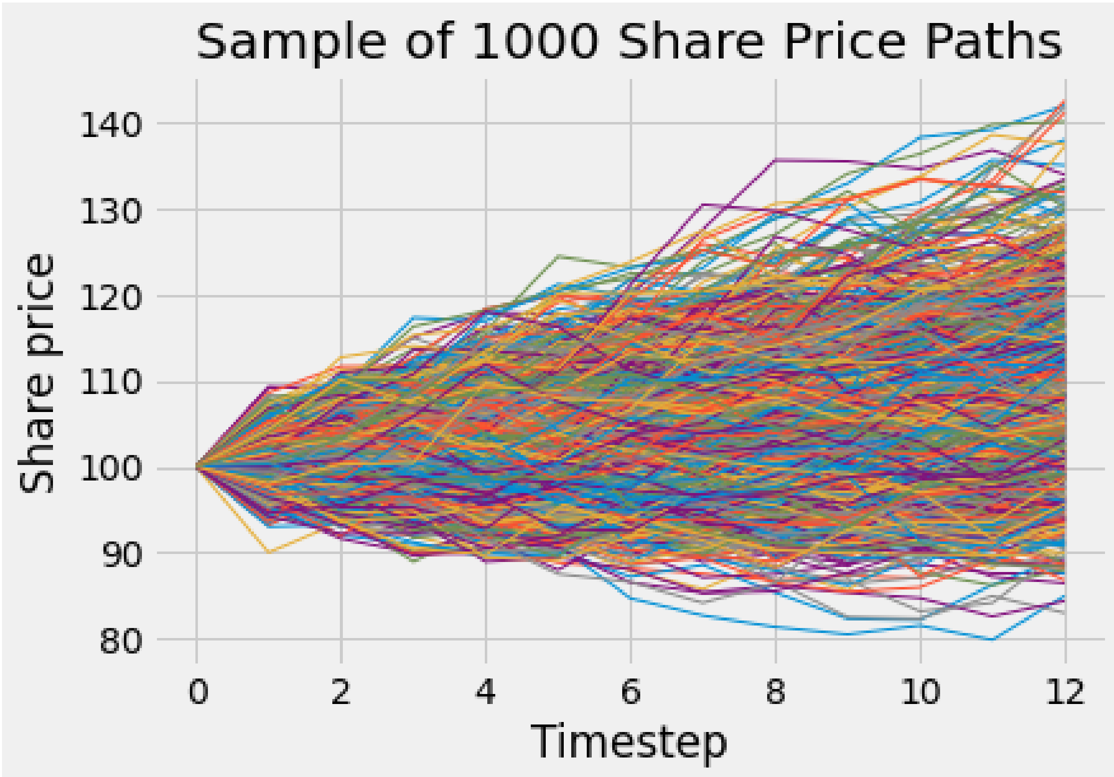
[13]:

	0	1	2	3	4	5 \
0	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000
1	97.064112	103.322274	100.066583	101.148340	96.290072	96.822147
2	94.587113	110.644481	103.951831	101.859269	98.253920	99.506238
3	95.837578	108.050270	107.304880	103.047917	96.458883	102.106216
4	96.183658	112.320481	103.517508	109.194048	97.872657	101.837479
5	100.389627	105.434896	104.161457	105.781640	102.747775	101.463430
6	102.429058	103.744416	102.275675	105.387191	105.781999	103.984721
7	103.982519	103.537309	99.990198	102.854386	104.885006	101.100964
8	108.477220	103.804703	103.799643	107.128828	102.520758	97.563342
9	112.865772	101.385120	102.651916	107.699804	103.667627	102.130954
10	108.682859	104.359796	105.194058	107.884152	104.716872	101.405335
11	108.521308	104.570242	111.519803	105.664692	111.397975	101.568720
12	108.050577	100.627111	108.779020	105.570714	111.685269	104.721817

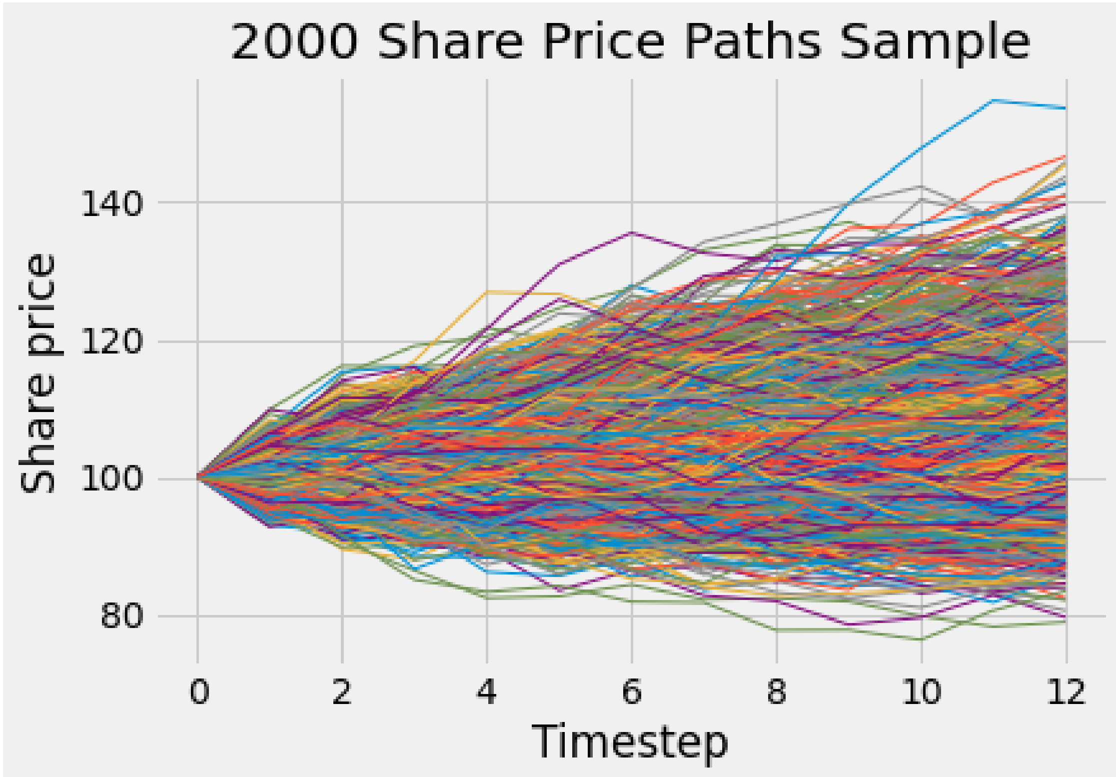
	6	7	8	9
0	100.000000	100.000000	100.000000	100.000000
1	99.456933	99.119506	99.671583	104.379974
2	99.341823	102.822658	102.437761	109.185517
3	96.258782	110.936342	100.892159	113.684443
4	99.024743	114.787503	102.190049	113.394112
5	102.295844	113.525831	98.649091	111.472110
6	99.795935	115.505711	102.958581	111.834419
7	105.825442	119.096436	101.925610	115.725620
8	108.792807	122.797181	104.560550	114.288026
9	109.910461	121.244902	109.161464	114.650830
10	112.226426	124.075784	108.575024	114.292640
11	107.990693	127.838313	107.709837	113.068575
12	111.348723	129.676034	109.031584	114.717712

Plotting for 1000, 2000, 10000 and 50000

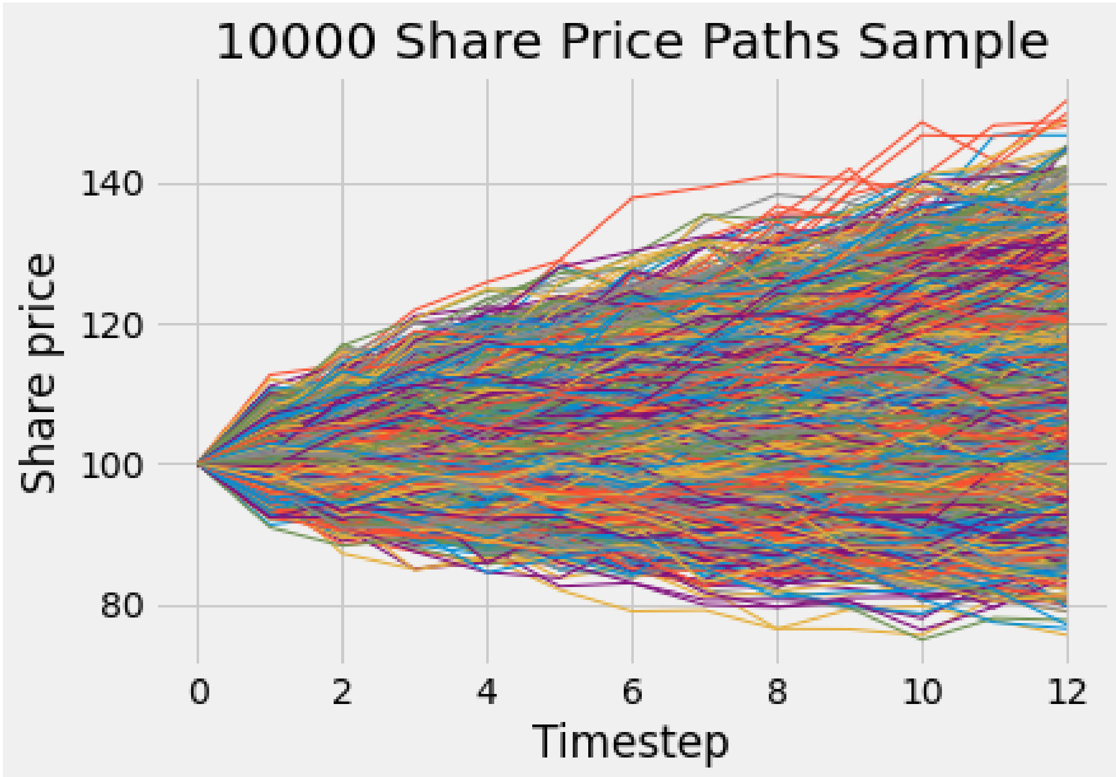
[14]:
```
plt.plot(Share_Price_Paths[1000], linewidth = 1.0)
plt.xlabel('Timestep')
plt.ylabel('Share price')
plt.title('Sample of 1000 Share Price Paths')
plt.show();
```



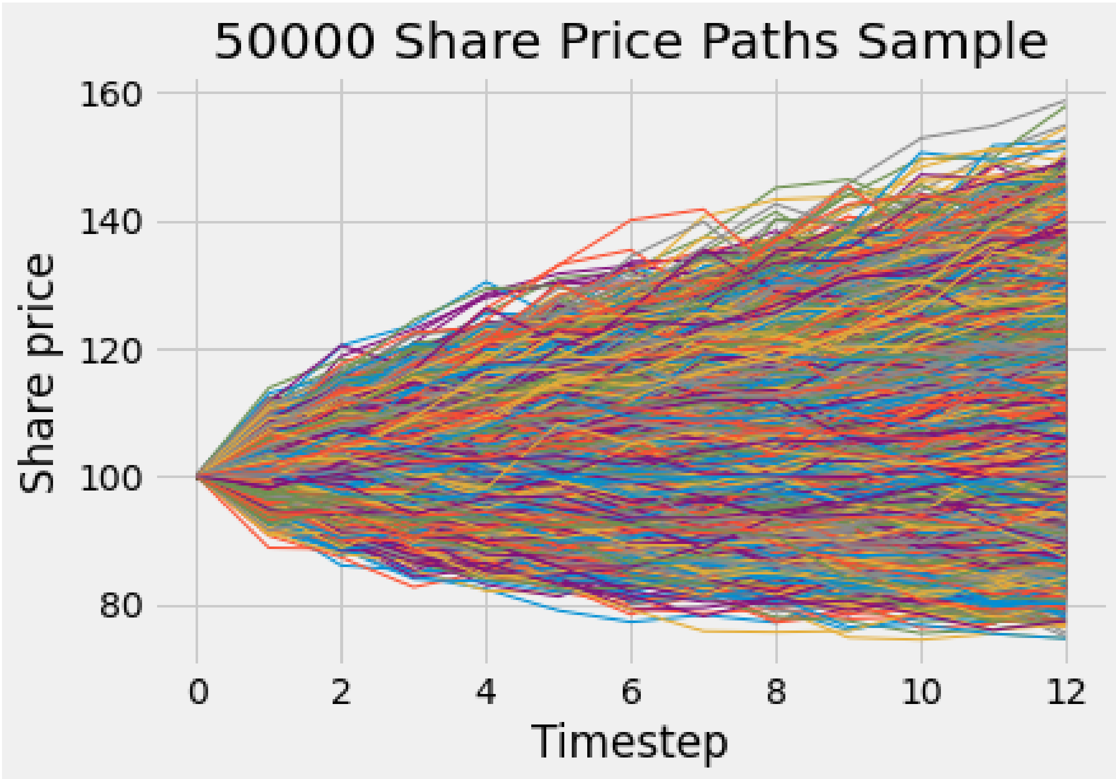
[15]:
```
plt.plot(Share_Price_Paths[2000], linewidth = 1.0)
plt.xlabel('Timestep')
plt.ylabel('Share price')
plt.title('2000 Share Price Paths Sample')
plt.show();
```



[16]:
```
plt.plot(Share_Price_Paths[10000], linewidth = 1.0)
plt.xlabel('Timestep')
plt.ylabel('Share price')
plt.title('10000 Share Price Paths Sample')
plt.show();
```




```
[17]: plt.plot(Share_Price_Paths[50000], linewidth = 1.0)
      plt.xlabel('Timestep')
      plt.ylabel('Share price')
      plt.title('50000 Share Price Paths Sample')
      plt.show();
```



7 Question 3

[18]:
```
price_estimate = []
price_std = []


for size in sample_sizes:
    S_Ts = Share_Price_Paths[size].iloc[12, :]
    payoff = np.maximum(S_Ts - K, 0)
    discounted_price = np.exp(-r*T)*payoff
    price_estimate.append(discounted_price.mean())
    price_std.append(discounted_price.std()/np.sqrt(size))
print("The price estimated by Monte Carlo when using sample size of 50,000 is :␣
 ↪{:.3f}".format(price_estimate[-1]))
```

```
The price estimated by Monte Carlo when using sample size of 50,000 is : 8.716
```

[19]:
```
from statistics import mean
Avg_price_estimate = mean(price_estimate)
Avg_price_std = mean(price_std)
print(Avg_price_estimate)
print(Avg_price_std)
```

```
8.696229143735518
0.06522544299527223
```

7.0.1 CEV modelling with Non Central Chi-Square Distribution Comparison

Initializing Parameters

[20]:
```
S0 = 100
sigma = 0.3
gamma = 0.75
r = 0.08
T = 1
```

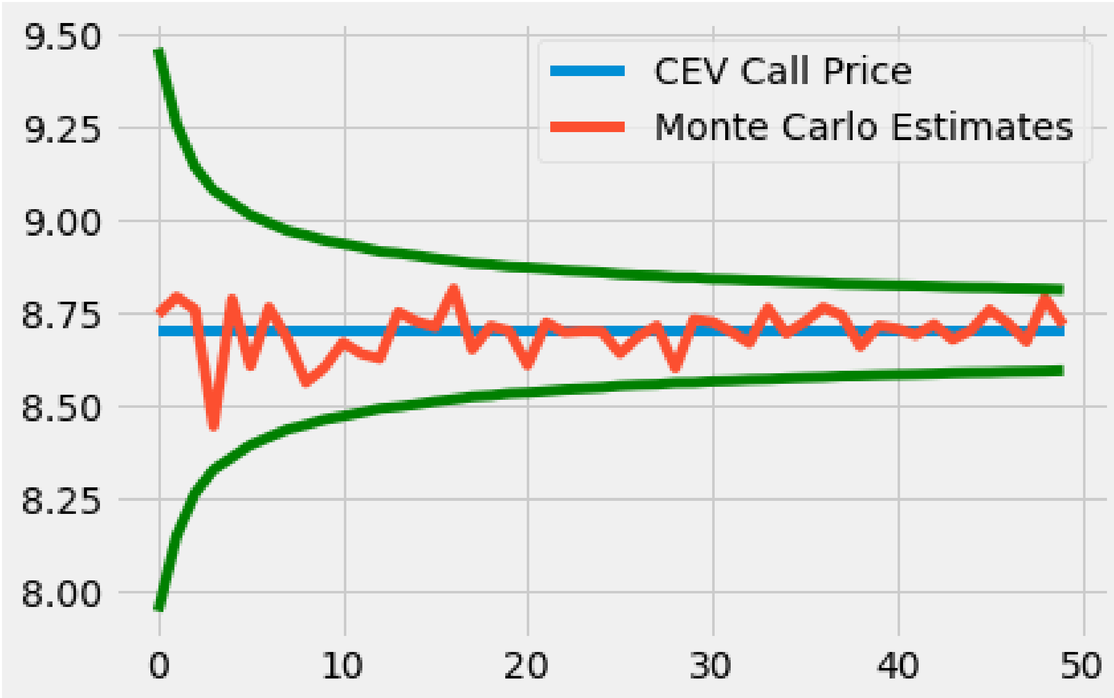
[21]:
```
from scipy.stats import ncx2
z = 2 + 1/(1-gamma)
def C(t,K):
    kappa = 2*r/(sigma**2*(1-gamma)*(np.exp(2*r*(1-gamma)*t)-1))
    x = kappa*S0**(2*(1-gamma))*np.exp(2*r*(1-gamma)*t)
    y = kappa*K**(2*(1-gamma))
    return S0*(1-ncx2.cdf(y,z,x))-K*np.exp(-r*t)*ncx2.cdf(x,z-2,y)
cev_call_price = C(T, 100)
print("The CEV call price according to Noncentral chi-squared distribution is:␣
 ↪{:.3f}".format(cev_call_price))
```

```
The CEV call price according to Noncentral chi-squared distribution is: 8.702
```

8 Question 4

The Monte Carlo Estimates was plotted against the CEV Non Central Chi-Squared Distribution.

[22]:
```
plt.plot([cev_call_price]*50, label='CEV Call Price')
plt.plot(price_estimate, '-', label='Monte Carlo Estimates')
plt.plot(cev_call_price + 3*np.array(price_std), 'g')
plt.plot(cev_call_price - 3*np.array(price_std), 'g')
plt.legend()
plt.show();
```



[23]:
```python
import yfinance as yf
from scipy import stats
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

8.1 Utility function

[24]:
```python
def plot_implied_volatilities(ticker:str, last_stock_price_date:datetime,␣
↪expiry_date:datetime)-> tuple([float, np.array]):

    #Downloading stock price info form Yahoo
    data = yf.download(ticker, start=last_stock_price_date)
    ticker_price = data.loc[:,'Close'].values[0]

    #Getting options data
    ticker= yf.Ticker(ticker)
    ticker_options_expiries = ticker.options #options expiries date
    idx = ticker_options_expiries.index(expiry_date.strftime("%Y-%m-%d"))
    ticker_options = ticker.option_chain(ticker_options_expiries[idx])
    call_chain = ticker_options.calls  # calls options
    strikes = call_chain.loc[:,'strike'].values
```

```
    implied_vols = call_chain.loc[:,'impliedVolatility'].values
    closest_strikes = np.abs(strikes - ticker_price)
    idx = np.argmin(closest_strikes)
    closest_strike = strikes[idx]
    iv_closest_strike = implied_vols[idx]
    closest_strikes_below = strikes[idx-3: idx]
    closest_strikes_above = strikes[idx+1: idx+4]
    iv_closest_strikes_below = implied_vols[idx-3: idx]
    iv_closest_strikes_above = implied_vols[idx+1: idx+4]

    strikes_array = np.concatenate((closest_strikes_below,[closest_strike],␣
↪closest_strikes_above)) # arrays of strikes closest to the current stock␣
↪price
    implied_vols_array = np.
↪concatenate((iv_closest_strikes_below,[iv_closest_strike],␣
↪iv_closest_strikes_above)) # arrys of related IV for strikes above
    option_price = call_chain[call_chain.strike==closest_strike].lastPrice.
↪values[0]

    return option_price, ticker_price, closest_strike, last_stock_price_date,␣
↪strikes_array, implied_vols_array
```

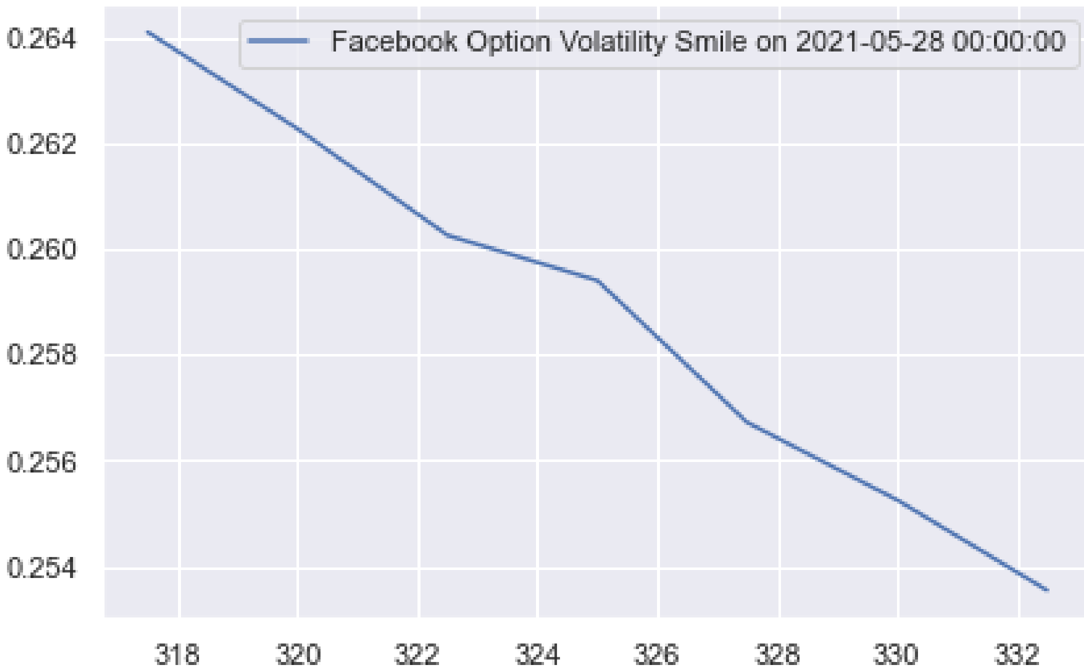
8.2 Question 5 Graphing Implied volatilities

[25]:
```
data_for_iv_calc = {}
expiry_dates = [datetime(2021,5,28), datetime(2021,12,17)]
last_stock_price_date= datetime(2021,4,30)
for expiry_date in expiry_dates:
    res = plot_implied_volatilities('FB', last_stock_price_date, expiry_date)
    data_for_iv_calc[expiry_date] = res[:-2]
    #plt.figure();
    plt.plot(res[-2], res[-1], label=f"Facebook Option Volatility Smile on␣
↪{expiry_date}");
    plt.legend(loc='upper right');
    plt.show();
```

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

[25]:
```
[<matplotlib.lines.Line2D at 0x1a1486ccf48>]
```

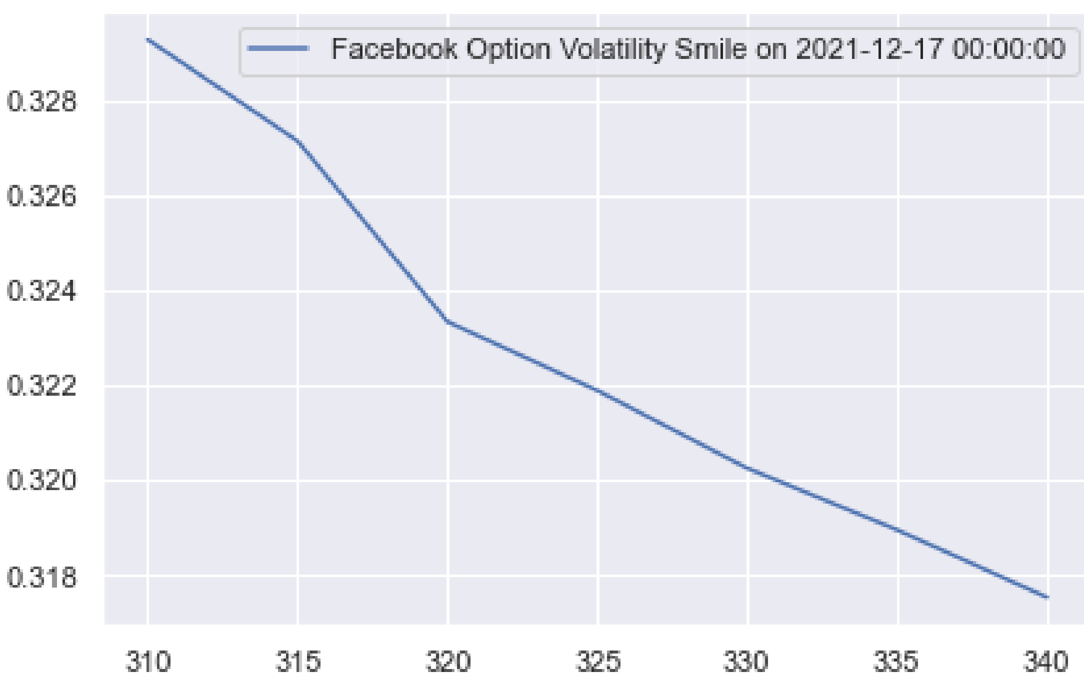
[25]:
```
<matplotlib.legend.Legend at 0x1a1486e0588>
```




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

[25]: [<matplotlib.lines.Line2D at 0x1a148742948>]

[25]: <matplotlib.legend.Legend at 0x1a148742bc8>



8.2.1 Implied Volatility function

[26]:
```
def bsm_price(option_type, sigma, s, k, r, T, q):
    # calculate the bsm price of European call and put options
    sigma = float(sigma)
    d1 = (np.log(s / k) + (r - q + sigma ** 2 * 0.5) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    if option_type == 'c':
        price = np.exp(-r*T) * (s * np.exp((r - q)*T) * stats.norm.cdf(d1) - k␣
↪*  stats.norm.cdf(d2))
        return price
    elif option_type == 'p':
        price = np.exp(-r*T) * (k * stats.norm.cdf(-d2) - s * np.exp((r - q)*T)␣
↪*  stats.norm.cdf(-d1))
        return price
    else:
        print('No such option type %s') %option_type
def implied_vol(option_type, option_price, s, k, r, T, q):
    # apply bisection method to get the implied volatility by solving the BSM␣
↪function
    precision = 0.00001
    upper_vol = 500.0
    max_vol = 500.0
    min_vol = 0.0001
    lower_vol = 0.0001
    iteration = 0

    while True:
        iteration +=1
        mid_vol = (upper_vol + lower_vol)/2.0
        price = bsm_price(option_type, mid_vol, s, k, r, T, q)
        if option_type == 'c':

            lower_price = bsm_price(option_type, lower_vol, s, k, r, T, q)
            if (lower_price - option_price) * (price - option_price) > 0:
                lower_vol = mid_vol
            else:
                upper_vol = mid_vol
            if abs(price - option_price) < precision: break
            if mid_vol > max_vol - 5 :
                mid_vol = 0.000001
                break

        elif option_type == 'p':
```

```
            upper_price = bsm_price(option_type, upper_vol, s, k, r, T, q)

            if (upper_price - option_price) * (price - option_price) > 0:
                upper_vol = mid_vol
            else:
                lower_vol = mid_vol
            if abs(price - option_price) < precision: break
            if iteration > 50: break

    return mid_vol
```

[27]:
```
r =0.01/100 # US 1 month T-bill rate
for date in data_for_iv_calc.keys():
    option_price= data_for_iv_calc[date][0]
    option_strice_price = data_for_iv_calc[date][2]
    time_to_maturity =(date-data_for_iv_calc[date][3]).days/365
    fb_price = data_for_iv_calc[date][1]
    print(f"The implied volatility for {date} is {implied_vol('c', option_price␣
 ↪, option_strice_price, fb_price, r, time_to_maturity, 0)}")
```

```
The implied volatility for 2021-05-28 00:00:00 is 0.2155032268957235
The implied volatility for 2021-12-17 00:00:00 is 0.3181319272942375
```

8.3 Question 7

Let's chose the expiry date of '2021-12-17' to calculate the Skewness

[28]:
```
data = plot_implied_volatilities('FB', last_stock_price_date, expiry_dates[1])
strikes = data[-2]
ivs = data[-1]

skew = (ivs[1] -ivs[0])/(strikes[1] - strikes[0])

print(f"The skewness is {skew*100:.3f}%")
```

```
[*********************100%***********************]  1 of 1 completed
The skewness is -0.043%
```

8.4 Question 8

The volatility does not depend on the strilke level because it is assumed constant

8.5 Question 9

The Heston Model is noteworthy because it seeks to provide for one of the main limitations of the Black-Scholes model which holds volatility constant. The use of stochastic variables in the Heston Model provides for the notion that volatility is not constant but arbitrary.