Submission Number: 1

Group Number: 14

Group Members:

Full Legal Name	**Location (Country)**	**E-Mail Address**	**Non-Contributing Member (X)**
Narsihma Reddy Dharmaiahgari Paripati	India	narsimha2506@gmail.com	
Naidu Sylendra Ruthwik	India	pm18naidus@iimidr.ac.in	
Ishaan Narula	India	ishaan.narula@outlook.com	

Statement of integrity: By typing the names of all group members in the text box below, you confirm that the assignment submitted is original work produced by the group (*excluding any non-contributing members identified with an "X" above*).

Narsihma Reddy Dharmaiahgari Paripati, Sylendra Ruthwik Naidu, Ishaan Narula

Use the box below to explain any attempts to reach out to a non-contributing member. Type (N/A) if all members contributed.

NA

** Note, you may be required to provide proof of your outreach to non-contributing members upon request.*

MScFE 630 Computational Finance

Group #: __14___ Submission #: __1___

Question 1:

An up-and-out option is a kind of knock-out barrier option that expires when the underlying security's price moves over a barrier price. If the underlying price does not climb above the barrier level, the option behaves like any other option: the holder has the right but not the duty to exercise their call or put option at the striking price on or before the contract's expiration date.

Advantages of up and out barrier call option:

- Highly Customisable
- Cheaper than similar vanilla options

Disadvantages of up and out barrier option:

- Investor needs larger attention in prediction of increase/decrease in movement of asset price
- Expiry of UAO option may lead to eradication of hedge

Question 2:

UAO are traded in OTC makets

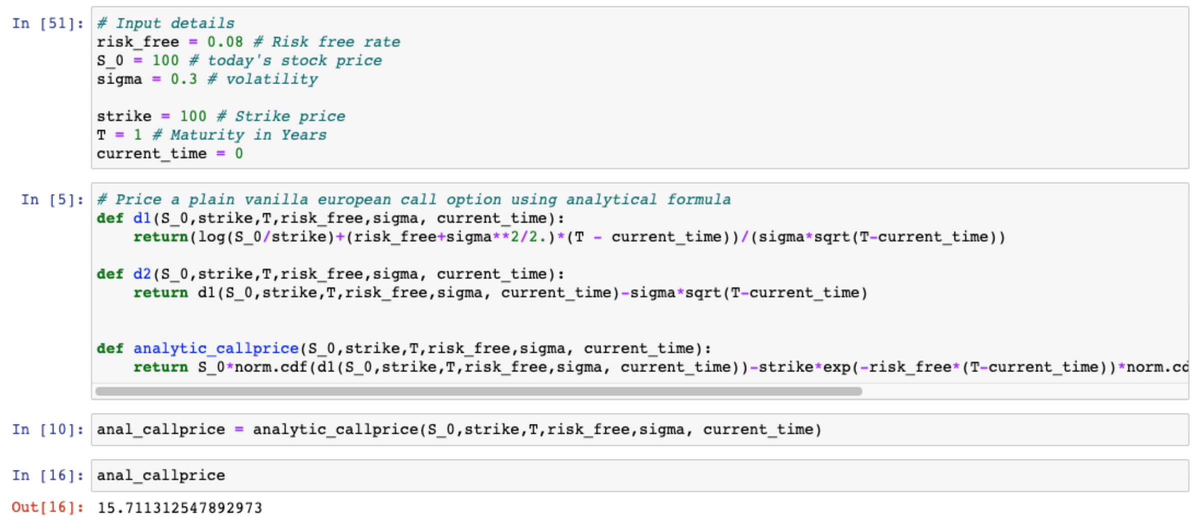
Question 3:

Yes, There is an analytical solution for the UAO call

The code for questions 4, 5, 6, and 7 are implemented in the attached notebook file named “MScFE 630 - Group 14 - Submission 1 - Qs 4-7.ipynb”.

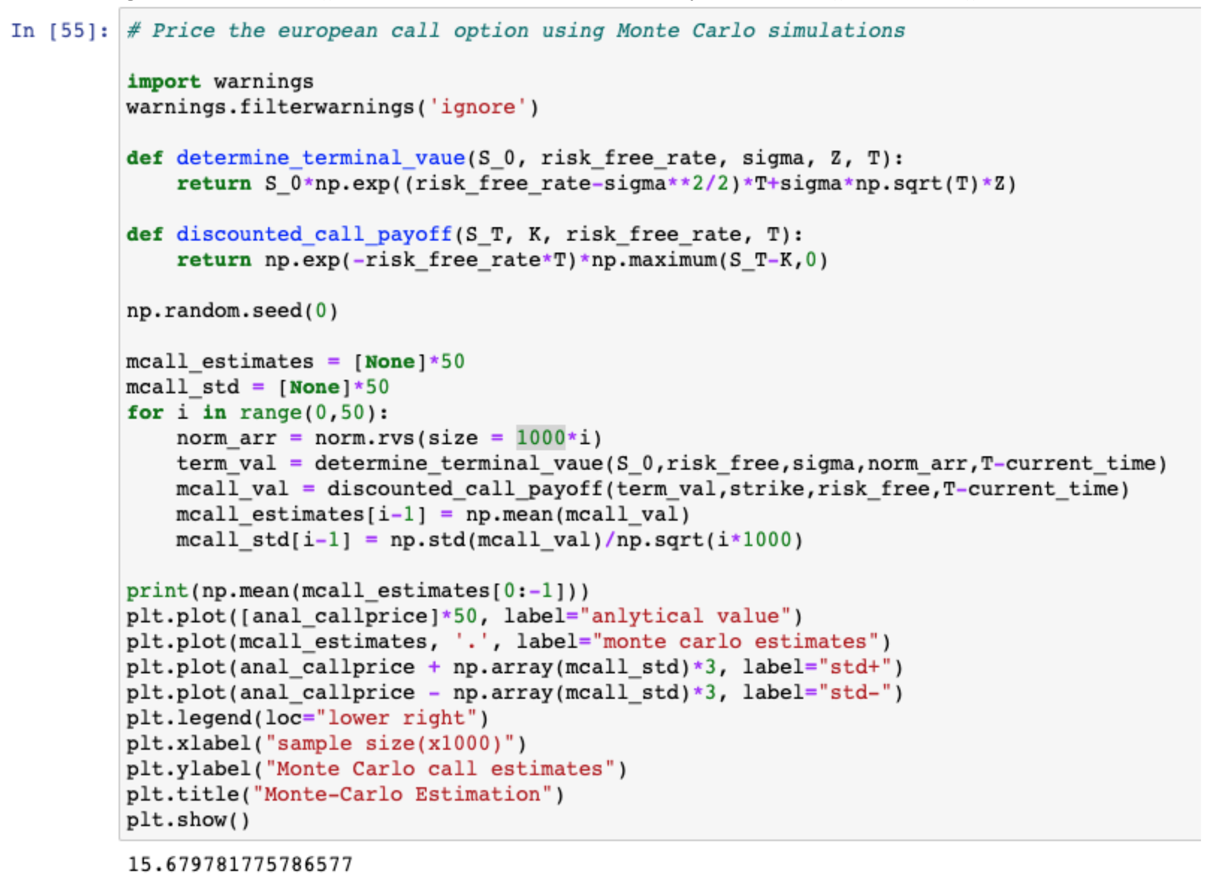
Question 4: European Vanilla option call price

The Analytical calculation answer is 15.711
The Monte Carlo simulated value is 15.679



```
In [51]: # Input details
         risk_free = 0.08 # Risk free rate
         S_0 = 100 # today's stock price
         sigma = 0.3 # volatility

         strike = 100 # Strike price
         T = 1 # Maturity in Years
         current_time = 0

In [5]:  # Price a plain vanilla european call option using analytical formula
         def d1(S_0,strike,T,risk_free,sigma, current_time):
             return(log(S_0/strike)+(risk_free+sigma**2/2.)*(T - current_time))/(sigma*sqrt(T-current_time))

         def d2(S_0,strike,T,risk_free,sigma, current_time):
             return d1(S_0,strike,T,risk_free,sigma, current_time)-sigma*sqrt(T-current_time)


         def analytic_callprice(S_0,strike,T,risk_free,sigma, current_time):
             return S_0*norm.cdf(d1(S_0,strike,T,risk_free,sigma, current_time))-strike*exp(-risk_free*(T-current_time))*norm.cd

In [10]: anal_callprice = analytic_callprice(S_0,strike,T,risk_free,sigma, current_time)

In [16]: anal_callprice

Out[16]: 15.711312547892973
```

Figure 4.1 Code piece of calculation of analytical european call price



```
In [55]: # Price the european call option using Monte Carlo simulations

         import warnings
         warnings.filterwarnings('ignore')

         def determine_terminal_vaue(S_0, risk_free_rate, sigma, Z, T):
             return S_0*np.exp((risk_free_rate-sigma**2/2)*T+sigma*np.sqrt(T)*Z)

         def discounted_call_payoff(S_T, K, risk_free_rate, T):
             return np.exp(-risk_free_rate*T)*np.maximum(S_T-K,0)

         np.random.seed(0)

         mcall_estimates = [None]*50
         mcall_std = [None]*50
         for i in range(0,50):
             norm_arr = norm.rvs(size = 1000*i)
             term_val = determine_terminal_vaue(S_0,risk_free,sigma,norm_arr,T-current_time)
             mcall_val = discounted_call_payoff(term_val,strike,risk_free,T-current_time)
             mcall_estimates[i-1] = np.mean(mcall_val)
             mcall_std[i-1] = np.std(mcall_val)/np.sqrt(i*1000)

         print(np.mean(mcall_estimates[0:-1]))
         plt.plot([anal_callprice]*50, label="anlytical value")
         plt.plot(mcall_estimates, '.', label="monte carlo estimates")
         plt.plot(anal_callprice + np.array(mcall_std)*3, label="std+")
         plt.plot(anal_callprice - np.array(mcall_std)*3, label="std-")
         plt.legend(loc="lower right")
         plt.xlabel("sample size(x1000)")
         plt.ylabel("Monte Carlo call estimates")
         plt.title("Monte-Carlo Estimation")
         plt.show()

         15.679781775786577
```

Figure 4.2 Code piece of calculation by using Monte Carlo simulation.

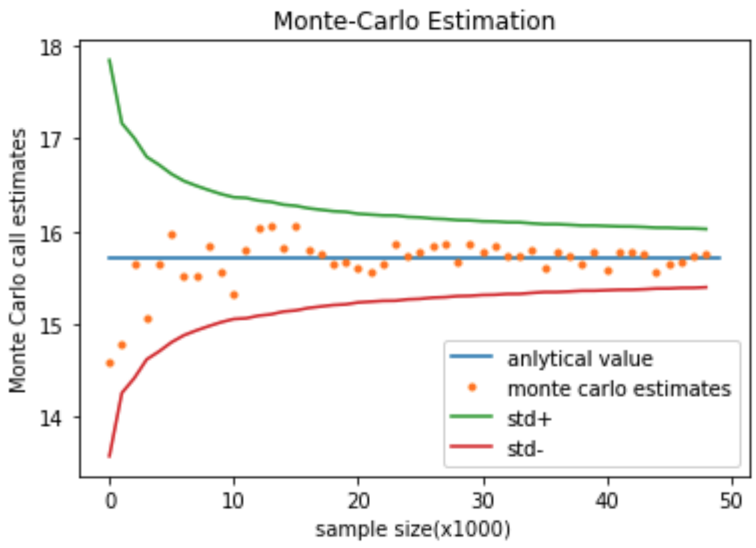
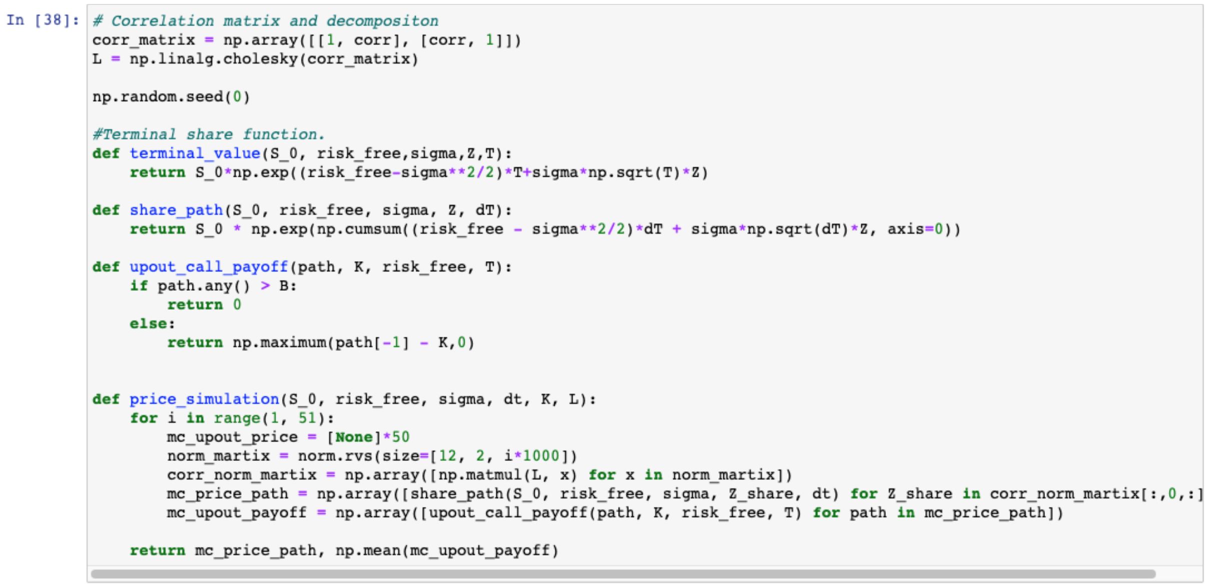


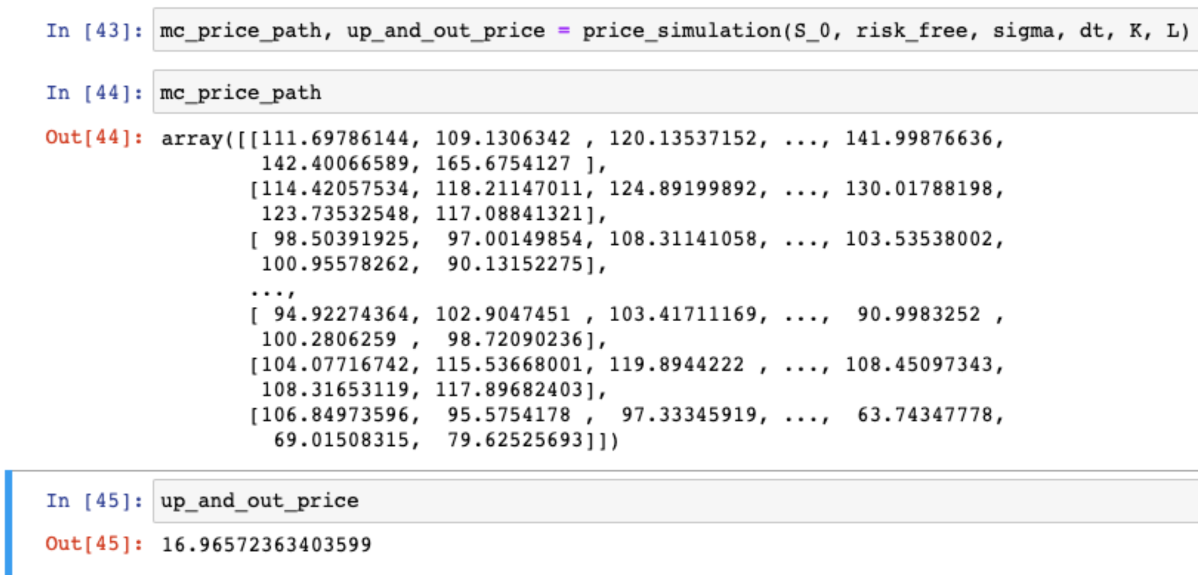
Figure 4.3 Plot showing the Analytical value and the Simulation values.

Question 5: Up and Out Barrier Option



```
In [38]: # Correlation matrix and decompositon
         corr_matrix = np.array([[1, corr], [corr, 1]])
         L = np.linalg.cholesky(corr_matrix)

         np.random.seed(0)

         #Terminal share function.
         def terminal_value(S_0, risk_free,sigma,Z,T):
             return S_0*np.exp((risk_free-sigma**2/2)*T+sigma*np.sqrt(T)*Z)

         def share_path(S_0, risk_free, sigma, Z, dT):
             return S_0 * np.exp(np.cumsum((risk_free - sigma**2/2)*dT + sigma*np.sqrt(dT)*Z, axis=0))

         def upout_call_payoff(path, K, risk_free, T):
             if path.any() > B:
                 return 0
             else:
                 return np.maximum(path[-1] - K,0)


         def price_simulation(S_0, risk_free, sigma, dt, K, L):
             for i in range(1, 51):
                 mc_upout_price = [None]*50
                 norm_martix = norm.rvs(size=[12, 2, i*1000])
                 corr_norm_martix = np.array([np.matmul(L, x) for x in norm_martix])
                 mc_price_path = np.array([share_path(S_0, risk_free, sigma, Z_share, dt) for Z_share in corr_norm_martix[:,0,:]
                 mc_upout_payoff = np.array([upout_call_payoff(path, K, risk_free, T) for path in mc_price_path])

             return mc_price_path, np.mean(mc_upout_payoff)
```

Figure 5.1 Code piece of calculation of prices of share path and up and out option



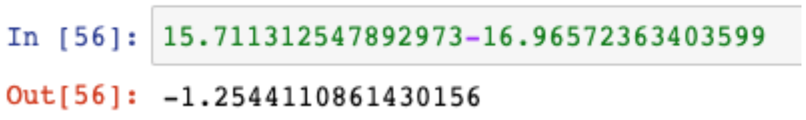
```
In [43]: mc_price_path, up_and_out_price = price_simulation(S_0, risk_free, sigma, dt, K, L)

In [44]: mc_price_path

Out[44]: array([[111.69786144, 109.1306342 , 120.13537152, ..., 141.99876636,
                 142.40066589, 165.6754127 ],
                [114.42057534, 118.21147011, 124.89199892, ..., 130.01788198,
                 123.73532548, 117.08841321],
                [ 98.50391925,  97.00149854, 108.31141058, ..., 103.53538002,
                 100.95578262,  90.13152275],
                ...,
                [ 94.92274364, 102.9047451 , 103.41711169, ...,  90.9983252 ,
                 100.2806259 ,  98.72090236],
                [104.07716742, 115.53668001, 119.8944222 , ..., 108.45097343,
                 108.31653119, 117.89682403],
                [106.84973596,  95.5754178 ,  97.33345919, ...,  63.74347778,
                  69.01508315,  79.62525693]])

In [45]: up_and_out_price

Out[45]: 16.96572363403599
```

Figure 5.2 Result of all the path prices and the Up and Out price.

Question 6: Up and In Barrier Option

$$Call_{up-and-out} + Call_{up-and-in} = Call$$

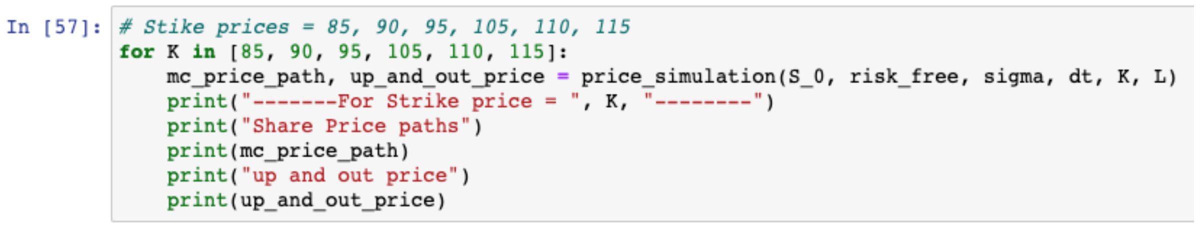
$$Call_{up-and-in} = Call - Call_{up-and-out}$$



```
In [56]: 15.711312547892973-16.96572363403599

Out[56]: -1.2544110861430156
```

Figure 6.1 Calculation of Up and In call option price

Question 7: Repeat #5 with the different Strike prices



```
In [57]: # Stike prices = 85, 90, 95, 105, 110, 115
         for K in [85, 90, 95, 105, 110, 115]:
             mc_price_path, up_and_out_price = price_simulation(S_0, risk_free, sigma, dt, K, L)
             print("-------For Strike price = ", K, "--------")
             print("Share Price paths")
             print(mc_price_path)
             print("up and out price")
             print(up_and_out_price)
```

Figure 7.1 Code piece of calculation of question 5 using the different strike prices.

Strike Price	Option Price
85	26.619
90	23.153
95	19.732
100	16.965
105	14.392
110	12.278
115	10.235

Questions 8 & 9:

Brief Overview on Barrier Options

As mentioned earlier, barrier options, in brief, are derivatives the right to exercise of which depends on whether the underlying's price touches a pre-defined fixed barrier throughout the life of the contract. Barrier options may be of the knock-in or knock-out variety, whereby the right to exercise either appears or disappears respectively, upon breach of the barrier.

We price a European Up-and-Out (UAO) Call whose barrier is set above (up) the underlying's spot price. If the barrier is touched/ breached throughout the life of the contract, the holder loses the right (out) to receive the European call's payoff upon maturity. He may get nothing or a rebate, although we assume the former for our analysis. If the barrier is not touched, the holder gets the standard call payoff.

Steps in Monte Carlo Analysis

Step 1:

We use the following inputs in our simulations, whose variable names are self-explanatory.

We then create empty arrays to store the UAO call's default-free price, CVA estimate and the option's default-adjusted price along with their respective standard errors. 50

simulations are run with each subsequent simulation taking on 1000 additional data points in its sample.

```
#Market-specific inputs
r_f = 0.08

#Stock-specific inputs
S_0 = 100
sigma_stock = 0.3

#Option-specific inputs
T = 1
months = 12*T
K = 100 #Option struck at-the-money
B = 150

#Counterparty-specific inputs
sigma_cp = 0.25
debt = 175 #Due in one year, same as the option's maturity
corr_stock_cp = 0.2
recovery_rate = 0.25
V_0 = 200
```

```
#Ensure same set of pseudo-random numbers upon each code execution
import numpy as np
np.random.seed(0)

#Create empty arrays to store European UAO call and CVA mean value and standard error of mean for 50 sample sizes
european_uao_call_meanval = [None]*50
european_uao_call_stderror = [None]*50

cva_meanval = [None]*50
cva_stderror = [None]*50

default_adj_call_val = [None]*50
default_adj_call_val_stderror = [None]*50
```

Step 2:
Based on the assumption that both stock and counterparty firm values follow a Geometric Brownian motion, the function stock_price_path is created using the following equation:

$$S_T = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)T + \sigma W_T^Q\right)$$

where S_T is the stock price at time T, S_0 is the initial stock price, r is the risk-free rate (constant drift) and σ is the stock's volatility. W_T^Q is a Brownian motion such that $W_T^Q \sim N(0, T)$. Also, $\sqrt{T}Z \sim N(0, T)$, where $Z \sim N(0,1)$. So, we can simulate W_T^Q by first simulating a standard normal random variable, Z, and then multiplying this Z by $\sqrt{T}$. The equation then becomes:

$$S_T = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}Z\right)$$

```
#Stock price path generator based on geometric Brownian motion
def stock_price_path(periods_per_path, current_stock_price, risk_free_rate, stock_vol, time_increment):

    series = np.zeros(periods_per_path)
    series[0] = current_stock_price

    for i in range(1, periods_per_path):
        dWt = np.random.normal(0, 1) * np.sqrt(time_increment) #Brownian motion
        series[i] = series[i-1] * np.exp((risk_free_rate - stock_vol**2/2)*time_increment + stock_vol*dWt)

    return series
```

Note that we use the above equation to predict monthly stock prices on a given stock price path, which is why time increment is taken to be 1/12 years.

Using the above equation, we also write a function to generate terminal stock prices with given inputs.

```
#Terminal stock price
def terminal_value(initial_stock_price, risk_free_rate, volatility, Z, time_to_maturity):
    return initial_stock_price * np.exp((risk_free_rate - volatility**2/2)*time_to_maturity
                                         + volatility*np.sqrt(time_to_maturity)*Z)
```

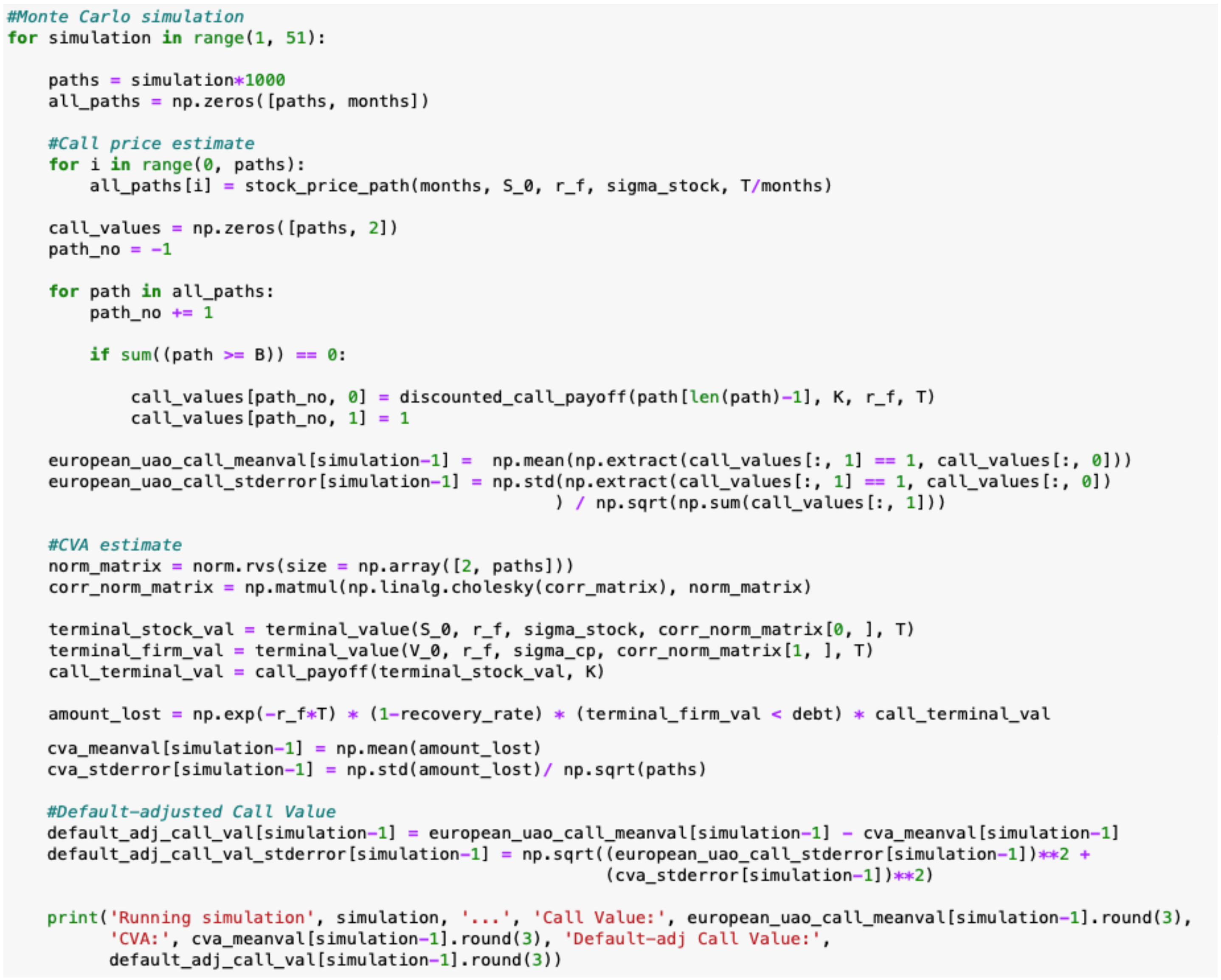
Step 3:
We now write functions to generate a vanilla call's payoff upon maturity and its discounted value at a given time point. A correlation matrix for CVA estimation is also constructed. The use of these and the above code snippets will become apparent later.

```
#Vanilla call payoff
def call_payoff(terminal_stock_price, strike):
    return np.maximum(terminal_stock_price - strike, 0)
```

```
#Discounted vanilla call payoff
def discounted_call_payoff(terminal_stock_price, strike, risk_free_rate, time_to_maturity):
    return np.exp(-risk_free_rate*time_to_maturity)*np.maximum(terminal_stock_price - strike, 0)
```

```
corr_matrix = np.array([[1, corr_stock_cp],[corr_stock_cp, 1]])
```

Step 4:
We now run 50 Monte Carlo simulations. The first simulation is run based on 1000 stock price paths, with every subsequent simulation taking on an additional 1000 paths. The code for these is shown below.

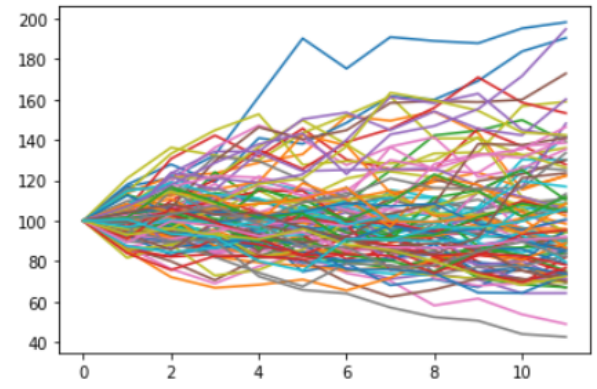


```
#Monte Carlo simulation
for simulation in range(1, 51):

    paths = simulation*1000
    all_paths = np.zeros([paths, months])

    #Call price estimate
    for i in range(0, paths):
        all_paths[i] = stock_price_path(months, S_0, r_f, sigma_stock, T/months)

    call_values = np.zeros([paths, 2])
    path_no = -1

    for path in all_paths:
        path_no += 1

        if sum((path >= B)) == 0:

            call_values[path_no, 0] = discounted_call_payoff(path[len(path)-1], K, r_f, T)
            call_values[path_no, 1] = 1

    european_uao_call_meanval[simulation-1] =  np.mean(np.extract(call_values[:, 1] == 1, call_values[:, 0]))
    european_uao_call_stderror[simulation-1] = np.std(np.extract(call_values[:, 1] == 1, call_values[:, 0])
                                               ) / np.sqrt(np.sum(call_values[:, 1]))

    #CVA estimate
    norm_matrix = norm.rvs(size = np.array([2, paths]))
    corr_norm_matrix = np.matmul(np.linalg.cholesky(corr_matrix), norm_matrix)

    terminal_stock_val = terminal_value(S_0, r_f, sigma_stock, corr_norm_matrix[0, ], T)
    terminal_firm_val = terminal_value(V_0, r_f, sigma_cp, corr_norm_matrix[1, ], T)
    call_terminal_val = call_payoff(terminal_stock_val, K)

    amount_lost = np.exp(-r_f*T) * (1-recovery_rate) * (terminal_firm_val < debt) * call_terminal_val

    cva_meanval[simulation-1] = np.mean(amount_lost)
    cva_stderror[simulation-1] = np.std(amount_lost)/ np.sqrt(paths)

    #Default-adjusted Call Value
    default_adj_call_val[simulation-1] = european_uao_call_meanval[simulation-1] - cva_meanval[simulation-1]
    default_adj_call_val_stderror[simulation-1] = np.sqrt((european_uao_call_stderror[simulation-1])**2 +
                                                  (cva_stderror[simulation-1])**2)

    print('Running simulation', simulation, '...', 'Call Value:', european_uao_call_meanval[simulation-1].round(3),
          'CVA:', cva_meanval[simulation-1].round(3), 'Default-adj Call Value:',
          default_adj_call_val[simulation-1].round(3))
```

We start with a loop on the no. of simulations. For each simulation, we carry out a call price estimate and a CVA estimate and then subtract the latter from the former to get the default-adjusted call price.

To get the call price estimate, we first generate a certain number of stock price paths dictated by the simulation number (1000 for the 1st, 2000 for the 2nd, …). For example, 100 stock price paths would look as follows:

```
import matplotlib.pyplot as plt
paths_100 = []

for sample_path in range(0,100):
    plt.plot(stock_price_path(months, S_0, r_f, sigma_stock, T/months))

plt.show()
```



100 stock price sample paths based on GBM

For each path, we then check if the stock price as breached the barrier at any point of time. If the barrier is touched or breached, the call option expires and its value is zero. But the price remains below the barrier, the call's value is simply the present value of its payoff, which is calculated as follows:

$$(S_T - K)^+ = max(S_T - K, 0)$$

We then average the values of the call option only for those price paths for which the stock price remains below the barrier to get an estimate of the default-free UAO call's value. The standard error of this estimate is then calculated as the standard deviation of the call's value for each non-breached price path divided by the number of such price paths. This exercise is repeated for each simulation.

Step 5:
Within the loop, the next step is to estimate the CVA. We do so by first constructing a matrix of 2 uncorrelated standard normal random variables. The number of RVs depends on the number of paths under a certain simulation.

We then use Cholesky decomposition on the correlation matrix constructed earlier and multiply the resulting matrix with the matrix of uncorrelated standard normal RVs, to get a matrix of correlated standard normal RVs.

We then use this matrix with the terminal_value function to generate terminal values for the stock price and the firm for every path. Lastly, in the event of default, the amount lost for each path is estimated using the following formula:

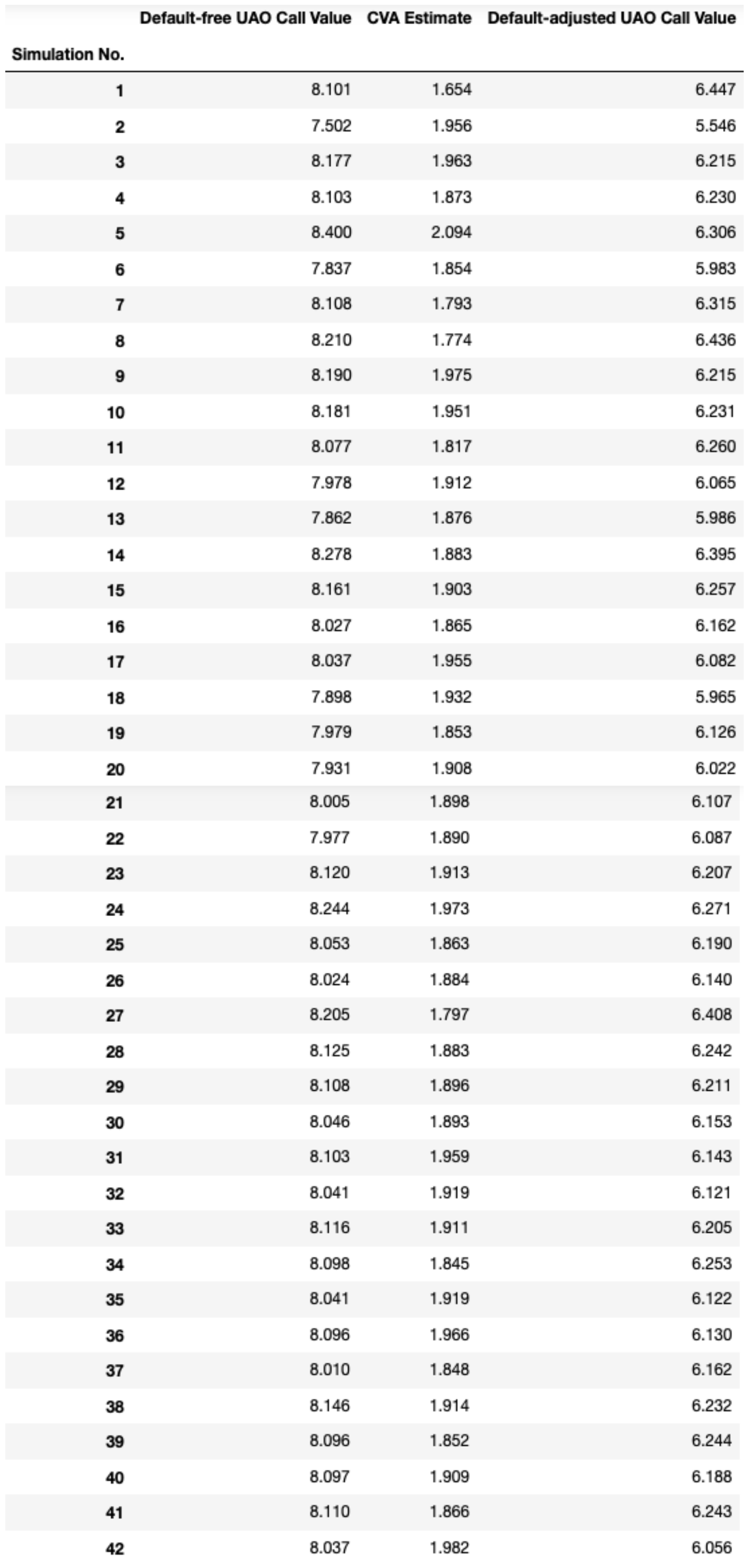
$$\mathit{Amount\ Lost} = \begin{cases} \exp\left(r_f . T\right) \times (1 - \mathit{recovery\ rate}) \times \mathit{call\ payoff\ if\ terminal\ firm\ value} < \mathit{debt} \\ 0\ \mathit{otherwise} \end{cases}$$

The CVA estimate is then the average of the amounts lost across the various price paths. Standard error is calculated analogously to the one for default-free option price.

Step 6:
Finally, the default-adjusted price (Answer 9) is calculated by subtracting the CVA estimate from the default-free option price.

Conclusion

The following table summarises the estimates obtained for the default-free UAO call price, the CVA and the default-adjusted UAO call price:



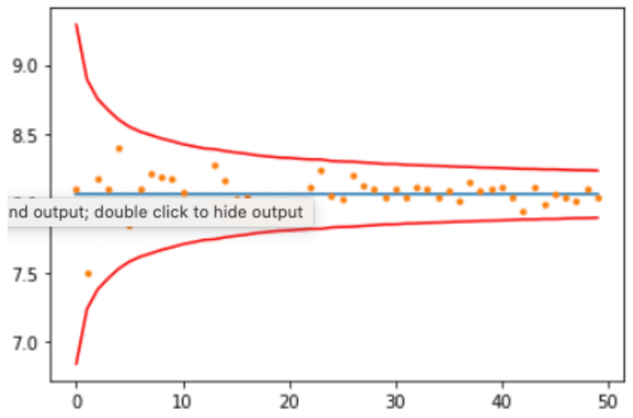
Simulation No.	Default-free UAO Call Value	CVA Estimate	Default-adjusted UAO Call Value
1	8.101	1.654	6.447
2	7.502	1.956	5.546
3	8.177	1.963	6.215
4	8.103	1.873	6.230
5	8.400	2.094	6.306
6	7.837	1.854	5.983
7	8.108	1.793	6.315
8	8.210	1.774	6.436
9	8.190	1.975	6.215
10	8.181	1.951	6.231
11	8.077	1.817	6.260
12	7.978	1.912	6.065
13	7.862	1.876	5.986
14	8.278	1.883	6.395
15	8.161	1.903	6.257
16	8.027	1.865	6.162
17	8.037	1.955	6.082
18	7.898	1.932	5.965
19	7.979	1.853	6.126
20	7.931	1.908	6.022
21	8.005	1.898	6.107
22	7.977	1.890	6.087
23	8.120	1.913	6.207
24	8.244	1.973	6.271
25	8.053	1.863	6.190
26	8.024	1.884	6.140
27	8.205	1.797	6.408
28	8.125	1.883	6.242
29	8.108	1.896	6.211
30	8.046	1.893	6.153
31	8.103	1.959	6.143
32	8.041	1.919	6.121
33	8.116	1.911	6.205
34	8.098	1.845	6.253
35	8.041	1.919	6.122
36	8.096	1.966	6.130
37	8.010	1.848	6.162
38	8.146	1.914	6.232
39	8.096	1.852	6.244
40	8.097	1.909	6.188
41	8.110	1.866	6.243
42	8.037	1.982	6.056

42	8.037	1.982	6.056
43	7.943	1.973	5.970
44	8.110	1.834	6.276
45	7.986	1.836	6.150
46	8.064	1.869	6.195
47	8.037	1.865	6.172
48	8.011	1.855	6.156
49	8.097	1.936	6.162
50	8.040	1.938	6.101

Summary of Default-free and Default-adjusted Call Values and CVA Values

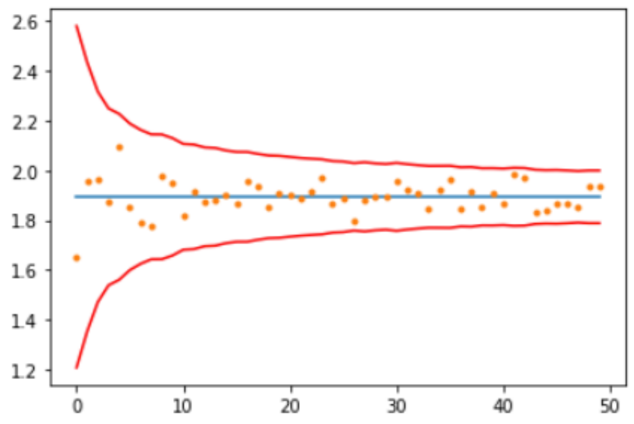
We then plot these estimates along with their standard error bounds

```
plt.plot([sum(european_uao_call_meanval)/len(european_uao_call_meanval)]*50)
plt.plot(european_uao_call_meanval, '.')
plt.plot(sum(european_uao_call_meanval)/len(european_uao_call_meanval) +
         np.array(european_uao_call_stderror) * 3, 'r')
plt.plot(sum(european_uao_call_meanval)/len(european_uao_call_meanval) -
         np.array(european_uao_call_stderror) * 3, 'r')
plt.show()
```



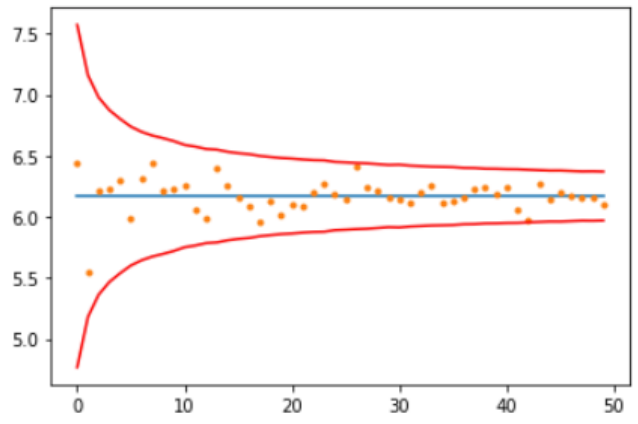
Monte Carlo Estimates of Default-free European UAO Call Price

```
plt.plot([sum(cva_meanval)/len(cva_meanval)]*50)
plt.plot(cva_meanval, '.')
plt.plot(sum(cva_meanval)/len(cva_meanval) + np.array(cva_stderror) * 3, 'r')
plt.plot(sum(cva_meanval)/len(cva_meanval) - np.array(cva_stderror) * 3, 'r')
plt.show()
```



Monte Carlo Estimates of CVA

```
plt.plot([sum(default_adj_call_val)/len(default_adj_call_val)]*50)
plt.plot(default_adj_call_val, '.')
plt.plot(sum(default_adj_call_val)/len(default_adj_call_val) + np.array(default_adj_call_val_stderror) * 3, 'r')
plt.plot(sum(default_adj_call_val)/len(default_adj_call_val) - np.array(default_adj_call_val_stderror) * 3, 'r')
plt.show()
```



Monte Carlo Estimates of Default-adjusted European UAO Call Price

Question 10:

In a derivative market, when two parties enter into a trade, besides the market risk, they are subject to default risk against each other. The risk of incurring financial loss due to the default of the counterparty is known as Counterparty Credit Risk. During the 2008 financial crisis, many financial institutions incurred huge losses due to unfair adjustments in the derivative value. When it became clear that counterparty will likely default on their obligations, value of outstanding options were corrected[1]

Default-Free Value of the option doesn't account for the counter party credit risk while valuing a option hence the name Default-Free value as it assumes that counterparty adheres to the obligations. Whereas Credit value Adjustments is an adjustment made to the value of the option to consider counterparty credit risk i.e. reduction of CVA to the default -Free Value of the option.[3]

How CVA is used in valuing an option?

Lets assume that there are two entities A and B who are likely to trade a contract of options over OTC, A to sell call option to B. when valuing the trade it is observed that A is likely to default its obligations in future, hence B demands for the discount to address this likely default.

Value of the option = Default-Free value of the option – CVA

Reference:

[1] Basel Committee on Banking Supervision – Review of CVA Risk Framework –

https://www.bis.org/bcbs/publ/d325.pdf

[2] Credit Value Adjustments In theory and Practice –

https://www.diva-portal.org/smash/get/diva2:692743/FULLTEXT01.pdf

[3] Alavian, Shahram, et al. "Credit valuation adjustment (CVA)." Available at SSRN 1310226 (2008).

[4] Wystup, U. (2002). Ensuring Efficient Hedging of Barrier Options. Frankfurt: commerzbank Trasury and Financial products.

[5] Reiner ,E. and Rubinstein, M. (1991). Breaking down the barriers. Risk, vol4.pp. 28– 35.