Submission Number: 3

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	
Sylendra Ruthwik Naidu	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.

N/A

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

MScFE 630 Computational Finance

Group #: 14 Submission #:3

Note: The code for questions 1, 2 and 4 is implemented in the attached notebook file named *"MScFE 630 - Group 14 - Submission 3 - Qs 1-4.ipynb"*.

Answer 1

Step 1

Initialize all the parameters as per the given information.

```
# stock parameter values
S = 100.0
v = 0.3
r = 0.08
T = 1.0
K = 100.0
B = 150.0
N = 12

# counterparty firm parameter values
Sf = 200
vf = 0.3
Debt = 175.0
c = 0.2
rr = 0.25
delta_t = T/N
```

Step 2

Calculate the analytical closed form solution and find optimal parameter values of alpha, beta, and sigma.

```
## CLosed form solution (Analytical price) using Vasicek model

def A(t1,t2,alpha):
    return (1-np.exp(-alpha*(t2-t1))) / alpha

def D(t1,t2,alpha,b,sigma):
    val1 = (t2-t1-A(t1,t2,alpha))*(sigma**2/(2*alpha**2)-b)
    val2 = sigma**2*A(t1,t2,alpha)**2/(4*alpha)
    return val1 - val2

def bond_price(r,t,T,alpha,b,sigma):
    return np.exp(-A(t,T,alpha)*r +D(t,T,alpha,b,sigma))

# actual zero-coupon bond prices & maturities
zcb_prices = np.array([100,99.38,
              98.76,98.15,97.54,
              96.94,96.34,95.74,
              95.16,94.57,93.99,
              93.42,92.85])/100.

maturity = np.array([delta_t * n for n in range(N+1)])
# difference between the Vasicek Bond Price and the actual Bond Prices
def F(x):
    r0 = x[0]
    alpha = x[1]
    b = x[2]
    sigma = x[3]
    return np.sum(np.abs(bond_price(r0,0,maturity,alpha,b,sigma) - zcb_prices))
```

```
# boundary for parameters
bounds = ((0,0.2),(0,5),(0,0.5),(0,2))

# use the minize function in the Scipy package to calibrate parameters
opt_val = scipy.optimize.fmin_slsqp(F,(0.05,0.3,0.05,0.03),bounds=bounds)
calc_r = opt_val[0]
calc_alpha = opt_val[1]
calc_beta = opt_val[2]
calc_sigma = opt_val[3]


print('\nCalculated values:')
print('Interest rate = {}'.format(calc_r))
print('Alpha = {}'.format(calc_alpha))
print('Beta = {}'.format(calc_beta))
print('Volatility = {}'.format(calc_sigma))

# estimation model bond prices
model_prices = bond_price(calc_r,0,maturity, calc_alpha, calc_beta, calc_sigma)
```

```
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.00024383601983268832
            Iterations: 11
            Function evaluations: 75
            Gradient evaluations: 11

Calculated values:
Interest rate = 0.07490754544726819
Alpha = 0.27904301286650884
Beta = 0.07065096751860526
Volatility = 0.03642093624527968
```

```
model_prices*100.0
```

```
array([100.        ,  99.37813391,  98.76100372,  98.14861882,
        97.54098348,  96.93809722,  96.3399552 ,  95.74654857,
        95.15786476,  94.5738878 ,  93.99459863,  93.41997533,
        92.84999337])
```

Step 3

Calculate the forward rates using the above estimated values and also the given input values.

```
# Input - Number of simulations
n = 100000

# initialize predictor-corrector Monte Carlo simulation forward rate
predcorr_forward = np.ones([n, N])*(model_prices[:-1]-model_prices[1:])/(delta_t*model_prices[1:])
predcorr_capfac = np.ones([n, N+1])
delta = np.ones([n, N])*delta_t
```

```
# calculate the forward rate for each steps from the bond price
for i in range(1, N):
    # Pick random number from gaussian distribution
    Z = calc_sigma*sqrt(delta[:,i:])*norm.rvs(size = [n,1])

    # predictor-corrector Monte Carlo simulation
    mu_initial = (np.cumsum(delta[:,i:]*predcorr_forward[:,i:]*calc_sigma**2/
                            (1+delta[:,i:]*predcorr_forward[:,i:]), axis = 1))
    temp = predcorr_forward[:,i:]*exp((mu_initial-calc_sigma**2/2)*delta[:,i:]+Z)
    mu_term = np.cumsum(delta[:,i:]*temp*calc_sigma**2/(1+delta[:,i:]*temp), axis = 1)
    predcorr_forward[:,i:] = predcorr_forward[:,i:]*exp((mu_initial + mu_term - calc_sigma**2)*delta[:,i:]/2+Z)
```

```
# implying capitalization factors from the forward rates
predcorr_capfac[:,1:] = np.cumprod(1+delta*predcorr_forward,axis = 1)

# inverting the capitalization factors to imply bond prices (discount factors)
predcorr_price = predcorr_capfac**(-1)

# taking averages: Forward Rate, Bond Price, Capitalization Factors
# mean Forward Rate
calc_forward_rate = np.mean(predcorr_forward,axis = 0)

# mean Price
predcorr_price = np.mean(predcorr_price,axis = 0)

# mean Capitalization Factors
calc_capfac = np.mean(predcorr_capfac,axis = 0)
```

```
calc_forward_rate
```

```
array([0.0750909 , 0.07498595, 0.07487774, 0.07475829, 0.07463964,
       0.07451499, 0.07438592, 0.07425072, 0.07411338, 0.07397028,
       0.07382609, 0.07367742])
```

```
# plot results
plt.subplots(figsize=(16, 8))
plt.xlabel('Maturity')
plt.ylabel('Bond Price')
plt.plot(maturity, zcb_prices, label = 'Actual Bond Prices')
plt.plot(maturity, model_prices, 'o', label = 'Calibration Prices')
plt.plot(maturity, predcorr_price,'x',label = "Predictor-Corrector Bond Prices")
plt.legend()
plt.show()
```

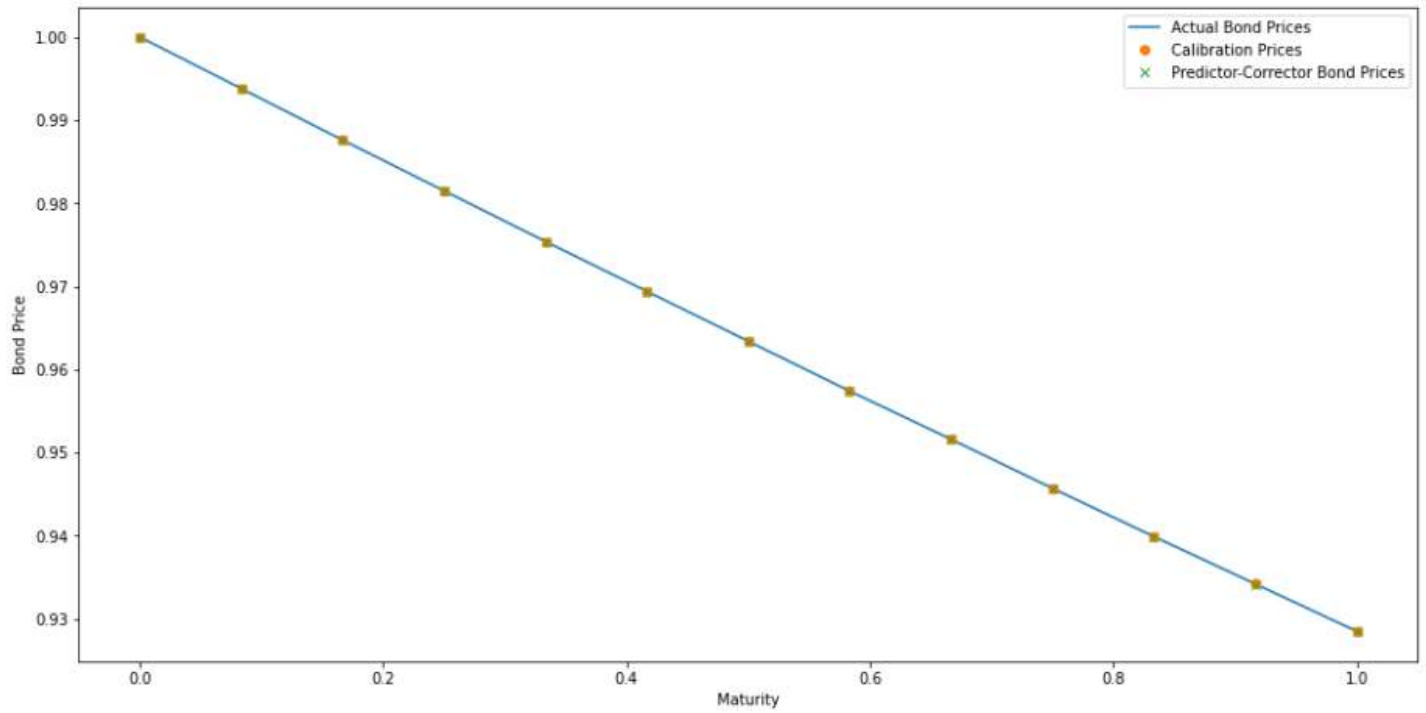


Figure: Plot of normalized values of actual bond prices and the calibrated prices against the maturity time

Step 4

Run the simulation using above estimated values and the given input values using predictor-corrector technique.

```
stock_paths = [] # stock paths using simulation
firm_paths = []  # firm paths using simulation

def simulate(n = 100000, gamma = 0.75):
    # Constans calculate outside the loop for optimization
    corr_matrix = np.linalg.cholesky(np.array([[1,c],[c,1]]))

    for i in range(n):
        stock_path = []
        firm_path = []
        S_j = S
        Sf_j = Sf
        for j in range(N):
            stock_path.append(S_j)
            firm_path.append(Sf_j)
            xi = np.matmul(corr_matrix, stats.norm.rvs(size = 2))

            # local volatilities
            v_dt = v*(S_j)**(gamma-1)
            vf_dt = vf*(Sf_j)**(gamma-1)

            # continuously compounded interest rate
            r = np.log(1 + calc_forward_rate[j]*delta_t)/delta_t

            # stock price
            S_j *= exp((r-1/2*(v_dt**2))*delta_t +
                       v_dt*sqrt(delta_t)*xi[0])

            # firm price
            Sf_j *= exp((r-1/2*(vf_dt**2))*delta_t +
                       vf_dt*sqrt(delta_t)*xi[1])

        # Saving the stock and firm path
        stock_paths.append(stock_path)
        firm_paths.append(firm_path)

simulate(n)
```

Step 5

Visualize all the estimated stock price paths. The price from these simulated stock paths is ranged between 70$ and 150$.

```
"""Plot stock simulations"""
dates = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"]
plt.subplots(figsize=(16, 8))
paths = pd.DataFrame(stock_paths, columns = dates)
for index, row in paths.iterrows():
    plt.plot(row, marker='o')

plt.plot([B]*len(dates), linestyle='--', color ='r')
plt.figtext(0.16,0.85,'Barrier = '+ str(B))
plt.title('Stock Price Paths')
plt.xlabel('Time (in months)')
plt.ylabel('Price ($)')
plt.grid(True)
plt.show()
```

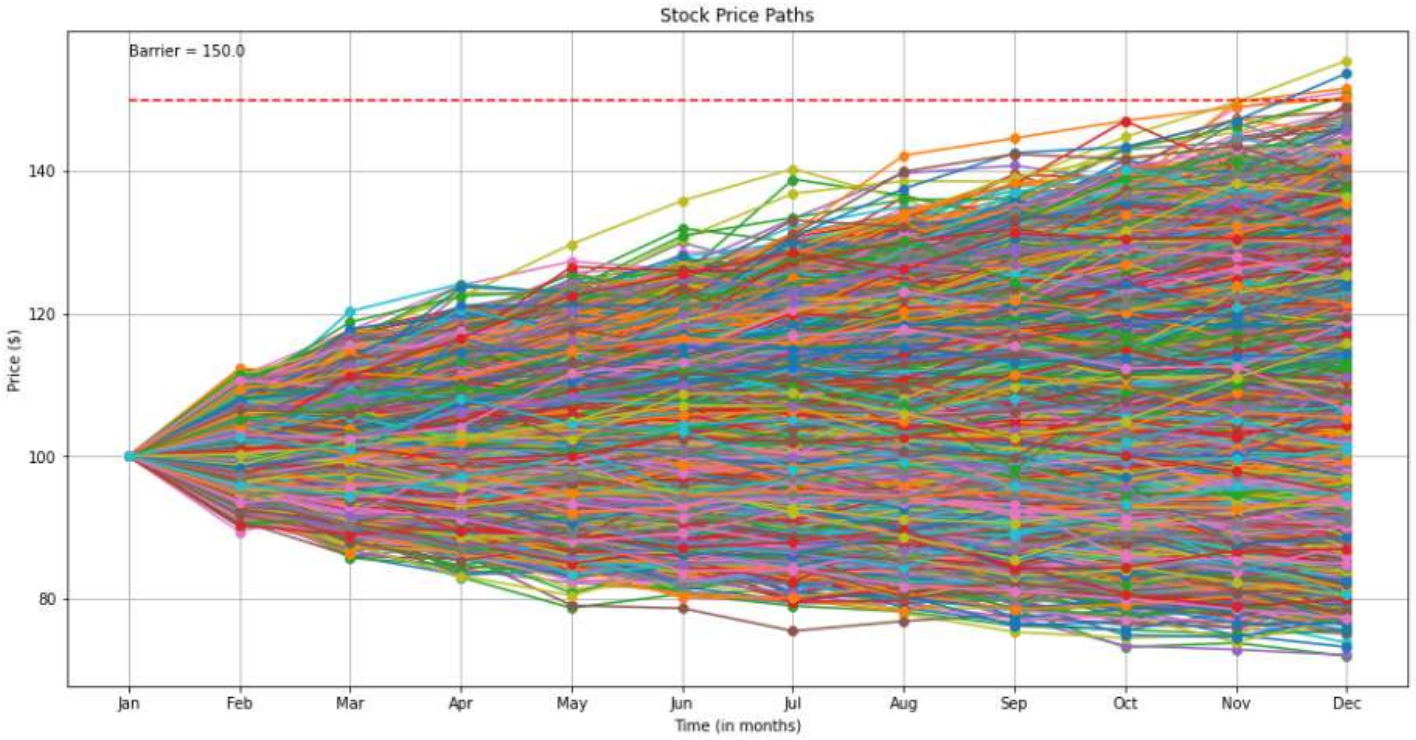


Figure: Line plot visualization of all the stock price paths

Step 6

Visualize all the estimated counterparty firm price paths.The simulated firm price paths is ranged between 160$ and 280$.

```
"""Plot firm simulations"""
plt.subplots(figsize=(16, 8))
paths = pd.DataFrame(firm_paths, columns = dates)
for index, row in paths.iterrows():
    plt.plot(row, marker='o')

plt.plot([Debt]*len(dates), linestyle='--', color ='r')
plt.figtext(0.16,0.22,'Debt = '+ str(Debt))
plt.title('Firm Price Trajectory')
plt.xlabel('Time (in months)')
plt.ylabel('Price ($)')
plt.grid(True)
plt.show()
```

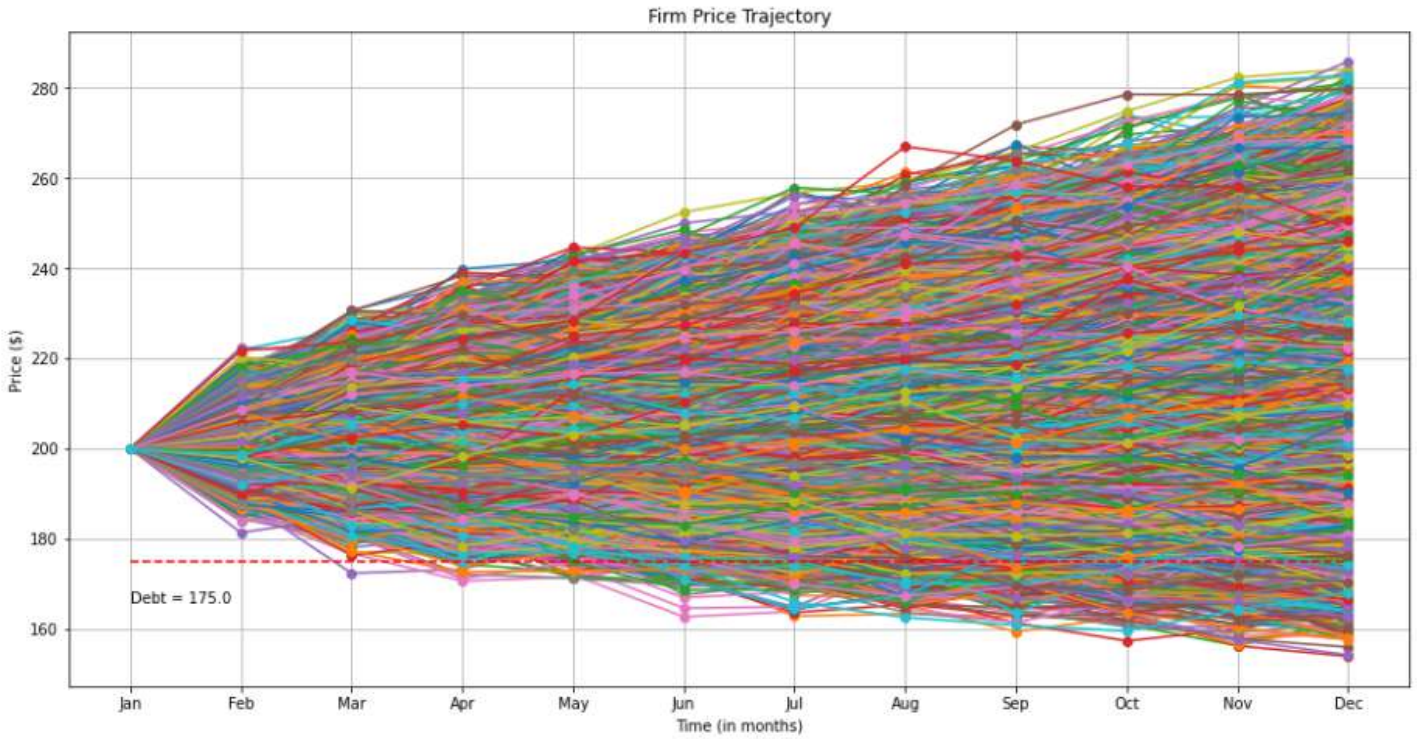


Figure: Line plot visualization of all the counterparty firm value paths

Answer 2

Part 1: Calculating the 1-year Discount Factors

Step 1.1

To calculate this metric for each simulation, we first derive the continuously compounded interest rates for each month across all simulations from the Libor Forward Market Model (LFMM) implemented above. The following equation is used:

$$e^{r_{ti}(t_{i+1}-t_i)} = 1 + \mathrm{L}(t_i, t_{i+1})(t_{i+1} - t_i)$$

This can be solved for the continuously compounded rate r_{ti} by taking the natural log on both sides:

$$\ln\left(e^{r_{ti}(t_{i+1}-t_i)}\right) = \ln\left(1 + \mathrm{L}(t_i, t_{i+1})(t_{i+1} - t_i)\right)$$

$$r_{ti}(t_{i+1} - t_i) = \ln\left(1 + \mathrm{L}(t_i, t_{i+1})(t_{i+1} - t_i)\right)$$

$$r_{ti} = \frac{\ln\left(1 + \mathrm{L}(t_i, t_{i+1})(t_{i+1} - t_i)\right)}{(t_{i+1} - t_i)}$$

The code for executing the above is as follows:

```
r_sim = np.log(1 + predcorr_forward*delta_t)/delta_t
r_sim
```

A plot of 100,000 simulations of this continuously compounded rate is shown below:

```
"""Plot interest rate simulations"""
plt.subplots(figsize=(16, 8))
paths = pd.DataFrame(r_sim, columns = dates)
for index, row in paths.iterrows():
    plt.plot(row, marker='o')

plt.title('Continuously Compounded Interest Rates Based on LFMM')
plt.xlabel('Time (in years)')
plt.ylabel('%')
plt.grid(True)
plt.show()
```

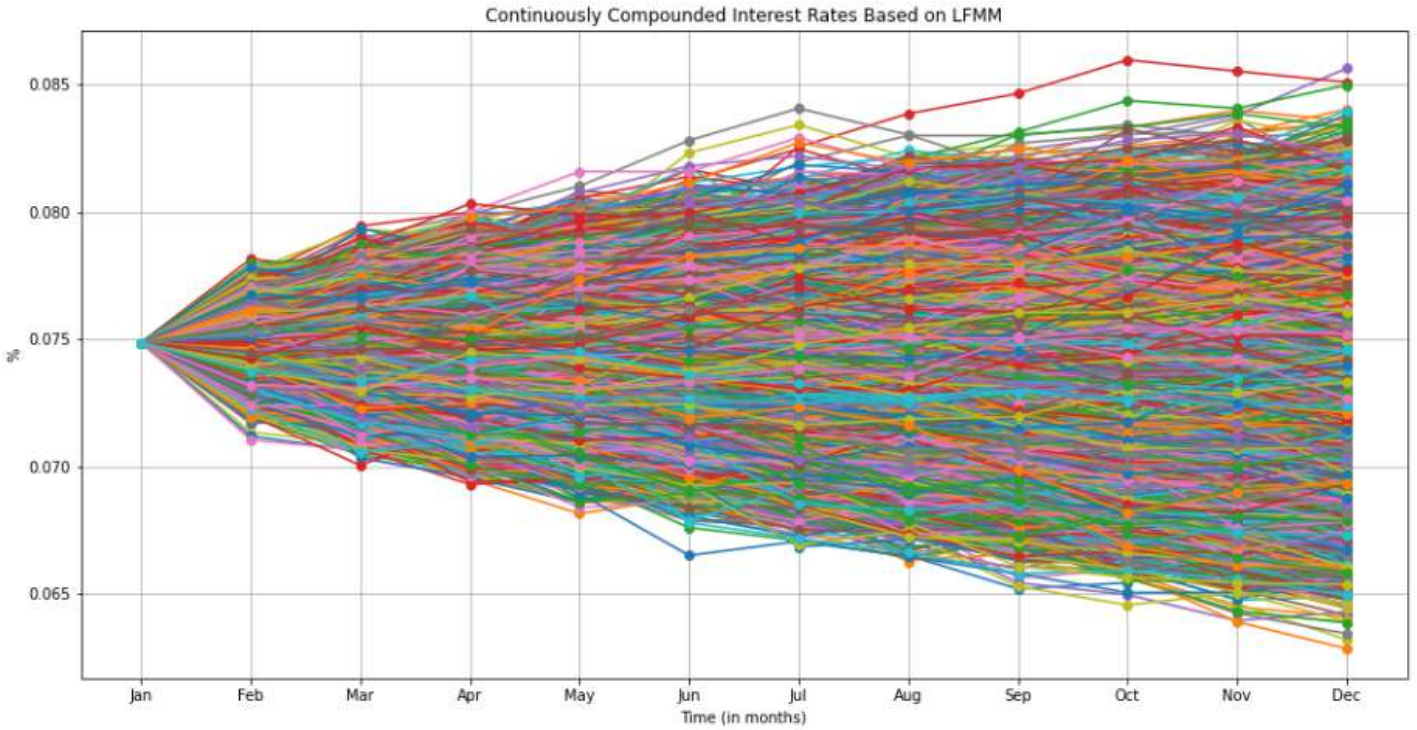


Figure: Line plot visualization of all the continuously compounded interest rate paths

Step 1.2

We now derive the 1-month, 2-month, …, 12-month capitalization factors which are obtained by compounding the interest rates of the respective months. The discount factor for a given month is then simply the reciprocal of that month's capitalisation factor.

Finally, the 1-year (or the 12-month) discount factor is the last column of the `discfac_r_sim` matrix, which gives the discount factors for each month across all 100,000 simulations.

```
capfac_r_sim = np.cumprod(1 + delta_t*r_sim, axis = 1)
discfac_r_sim = capfac_r_sim**(-1)

discfac_1y = discfac_r_sim[:, -1]
discfac_1y
```

```
array([0.92701989, 0.92824836, 0.92847842, ..., 0.92666556, 0.9280432 ,
       0.92922043])
```

Part 2: Default-free Value of a European Up-and-Out (UAO) Call Option

As the next step, the above 1-year discount factors across the various simulations are used to find the default-free value of the UAO call for the jointly simulated stock and firm paths.

Step 2.1

We first write the function `uao_call_disc_payoffs` to calculate the discounted payoff of the UAO call for each simulation:

```
def uao_call_disc_payoffs(price_paths, discfacs, strike, barrier):
    disc_payoffs = np.zeros([n, 2])
    price_paths = np.array(price_paths)
    path_no = -1

    for path in price_paths:
        path_no += 1
        if sum((path >= barrier)) == 0:
            disc_payoffs[path_no, 0] = np.maximum(path[-1] - strike, 0) * discfacs[path_no]
            disc_payoffs[path_no, 1] = 1

    return disc_payoffs
```

The `for` loop in the above function implements the following logic of the UAO call's feature (assuming no rebate): if at any point throughout the 12 months, the underlying's price breaches the USD 150 barrier, the option ceases to exist. Otherwise, it offers the payoff of a standard European call, i.e. $\max(S_T - K, 0)$.

Note that we test the above for each simulated stock price path, and discount the payoff by the relevant 1-year discount factor for those simulations where the barrier is not touched. The discounted payoffs of such simulations are marked 1.

Step 2.2

The value of the UAO call now is simply the average of the payoffs across all those simulations for which the underlying call is not deactivated (i.e. the barrier is not touched). The standard error of this estimated call value is given by:

$$\sigma_{UAO\ call\ value} = \frac{\sigma_{discounted\ payoffs\ |\ call\ active}}{no.of\ simulations\ for\ which\ the\ call\ is\ active}$$

This is implemented below:

```
disc_payoffs = uao_call_disc_payoffs(stock_paths, discfac_1y, K, B)
```

```
uao_call_meanval = np.mean(np.extract(disc_payoffs[:, 1] == 1, disc_payoffs[:, 0]))
uao_call_stderr = np.std(np.extract(disc_payoffs[:, 1] == 1, disc_payoffs[:, 0]))/np.sqrt(np.sum(disc_payoffs[:, 1]))

print('Default-free European UAO Call Value:', uao_call_meanval.round(3))
print('Default-free European UAO Call Std. Error:', uao_call_stderr.round(3))
```

```
Default-free European UAO Call Value: 7.677
Default-free European UAO Call Std. Error: 0.023
```

Part 3: Default-adjusted Value of a European Up-and-Out (UAO) Call Option

We now make a Credit Valuation Adjustment (CVA) which captures for each simulation, the loss which we expect to incur in case our counterparty defaults (assumed to take place when their firm value < debt level).

Step 3.1

Typically, in the presence of a correlation between the underlying price and firm value, the CVA for a standard European call is given by:

$$CVA = e^{-rT}(1-\delta)X_T\mathbb{I}_{\{V_T<D\}}$$

where
r: constant continuously compounded interest rate,
T: time to maturity,
δ: recovery rate
X_T: call option payoff at maturity
V_T: counterparty firm value
D: counterparty debt level

In our implementation, we replace the exponential function with constant continuously compounded interest rate with our 1-year discount factor.

Step 3.2
The above formula is implemented as follows:

```
#CVA estimate
Sf_T = np.column_stack((np.array(firm_paths)[:, -1], disc_payoffs[:, 1]))

#We only calculate the loss for those price paths where the call is activated, i.e. barrier is not breached
loss = (np.extract(disc_payoffs[:, 1] == 1, disc_payoffs[:, 0]) * (np.extract(Sf_T[:, 1] == 1, Sf_T[:, 0]) < Debt) *
        (1 - rr))
```

We first extract the terminal firm values across all simulations, and mark those for which the underlying price has not touched the barrier (i.e. the call option is active) with a 1.

Then, we implement the formula in step 3.1 only for those simulations for which the call option is active. This is because for those scenarios under which the underlying price has touched the barrier, the option holder's right to the call's payoff expires. Since he/ she loses his right to the payoff, looking at the default risk associated with it is of no use.

Step 3.3
Finally, the expected loss across all simulations under which the call remains active is averaged to get a CVA estimate and a standard error of this estimate is calculated on the same lines as the one calculated for the UAO call's default-free value.

```
cva_meanval = np.mean(loss)
cva_stderror = np.std(loss)/np.sqrt(np.sum(disc_payoffs[:, 1]))

print('CVA Mean Value:', cva_meanval.round(3))
print('CVA Std. Error:', cva_stderror.round(3))
print(' ')
print('Default-adj UAO Call Value:', (uao_call_meanval - cva_meanval).round(3))
```

```
CVA Mean Value: 0.015
CVA Std. Error: 0.001

Default-adj UAO Call Value: 7.662
```

Conclusion
The various values have been summarised below.

Default-free Eur. UAO Call Value	CVA	Default-adjusted Eur. UAO Call Value
7.677 (0.023)	0.015 (0.001)	7.662

Note: The brackets contain standard errors of the respective estimates

Answer 3

The value of the option estimated using various models and techniques typically assumes that neither side defaults. The possibility of counterparty default is captured by the Credit Valuation Adjustment (CVA), as discussed above and throughout the course.

As a result, if f_{nd} is the no-default value of the option, the resulting value incorporating the selling counterparty's default risk is given by $f_{nd} - CVA$.

However, default risk is bilateral in nature, i.e. both counterparties could default. The buying counterparty's credit risk is captured by what is called the Debit or the Debt Valuation Adjustment (DVA). Since the buying counterparty would default only when this is profitable DVA captures the present value of the expected gains from the buyer's own default.

In conclusion, the buyer's own default risk increases the option's value by the DVA. The net value of the option then becomes $f_{nd} - CVA + DVA$.

Answer 4

A 25bps increase in 'interest rates' has been interpreted as an increase in the continuously compounded interest rates. The simulations of these were stored as a `numpy` matrix in the variable `r_sim` previously.

We now add 25bps to each value in this matrix to reflect this interest rate increase. Based on this new matrix r_sim_plus25bps, the one-year discount factors, discounted call payoffs, default-free call value and default-adjusted call value are calculated in the same manner as done in answer 2. The code snippets and results are shown below.

```
r_sim_plus25bps = r_sim + 0.0025
r_sim_plus25bps
```

```
array([[0.07735693, 0.07864214, 0.07813986, ..., 0.07867244, 0.07825892,
        0.07851377],
       [0.07735693, 0.07756146, 0.07846396, ..., 0.07562996, 0.07611304,
        0.07682557],
       [0.07735693, 0.07801802, 0.07779863, ..., 0.07556912, 0.0752583 ,
        0.07562622],
       ...,
       [0.07735693, 0.07781839, 0.07823008, ..., 0.07963325, 0.07883884,
        0.078065  ],
       [0.07735693, 0.07633794, 0.07543879, ..., 0.07855531, 0.07910264,
        0.07856142],
       [0.07735693, 0.07624716, 0.07586699, ..., 0.07507093, 0.07597509,
        0.07709168]])
```

```
capfac_r_sim_plus25bps = np.cumprod(1 + delta_t*r_sim_plus25bps, axis = 1)
discfac_r_sim_plus25bps = capfac_r_sim_plus25bps**(-1)

discfac_1y_plus25bps = discfac_r_sim_plus25bps[:, -1]
discfac_1y_plus25bps
```

```
array([0.92472003, 0.9259452 , 0.92617464, ..., 0.92436664, 0.92574058,
       0.92691465])
```

```
disc_payoffs_plus25bps = uao_call_disc_payoffs(stock_paths, discfac_1y_plus25bps, K, B)
```

```
uao_call_meanval_plus25bps = np.mean(np.extract(disc_payoffs_plus25bps[:, 1] == 1, disc_payoffs_plus25bps[:, 0]))
uao_call_stderr_plus25bps = (np.std(np.extract(disc_payoffs_plus25bps[:, 1] == 1, disc_payoffs_plus25bps[:, 0]))/
                             np.sqrt(np.sum(disc_payoffs_plus25bps[:, 1])))

print('Default-free European UAO Call Value after Interest Rate Increase:', uao_call_meanval_plus25bps.round(3))
print('Default-free European UAO Call Std. Error after Interest Rate Increase:', uao_call_stderr_plus25bps.round(3))
```

```
Default-free European UAO Call Value after Interest Rate Increase: 7.658
Default-free European UAO Call Std. Error after Interest Rate Increase: 0.023
```

```
#CVA estimate
Sf_T_plus25bps = np.column_stack((np.array(firm_paths)[:, -1], disc_payoffs_plus25bps[:, 1]))

#We only calculate the loss for those price paths where the call is activated, i.e. barrier is not breached
loss_plus25bps = (np.extract(disc_payoffs_plus25bps[:, 1] == 1, disc_payoffs_plus25bps[:, 0]) *
                  (np.extract(Sf_T_plus25bps[:, 1] == 1, Sf_T_plus25bps[:, 0]) < Debt) * (1 - rr))

cva_meanval_plus25bps = np.mean(loss_plus25bps)
cva_stderror_plus25bps = np.std(loss_plus25bps)/np.sqrt(np.sum(disc_payoffs_plus25bps[:, 1]))

print('CVA Mean Value after Interest Rate Increase:', cva_meanval_plus25bps.round(3))
print('CVA Std. Error after Interest Rate Increase:', cva_stderror_plus25bps.round(3))
print(' ')
print('Default-adj UAO Call Value after Interest Rate Increase:', (uao_call_meanval_plus25bps -
                                                                   cva_meanval_plus25bps).round(3))
```

```
CVA Mean Value after Interest Rate Increase: 0.015
CVA Std. Error after Interest Rate Increase: 0.001

Default-adj UAO Call Value after Interest Rate Increase: 7.643
```

The following table summarises the various values before and after the interest rate increase:

Scenario	Default-free Eur. UAO Call Value	CVA	Default-adjusted Eur. UAO Call Value
Before Interest Rate Increase (+25 bps)	7.677 (0.023)	0.015 (0.001)	7.662
After Interest Rate Increase (+25 bps)	7.658 (0.023)	0.015 (0.001)	7.643

Note: The brackets contain standard errors of the respective estimates

Answer 5

Part a

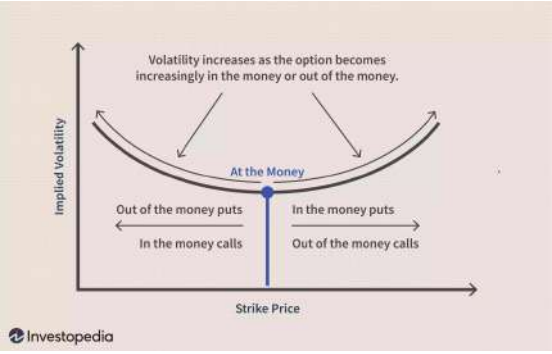
Basis of Difference	Vanilla Options	Barrier Options
Overview	A financial derivative which gives the holder the right but not the obligation to buy (call) or sell (put) the underlying asset on which it is written at (European) or until (American) a specific date	Derivative contracts the right to exercise of which depends on whether the underlying's price touches a pre-defined fixed barrier throughout the life of the contract

Varieties	European and American calls and puts (features covered in the 'Overview' part)	Down-and-in calls and puts: Barrier is set below *(down)* the spot price of the underlying. In case this barrier is touched at any point throughout the life of the contract, the underlying derivative (call or put) gets activated *(in)*. If it is not, the holder gets nothing or, sometimes, a rebate upon maturity of the contract Up-and-in calls and puts: Compared to down-and-in options, barrier is set above *(up)* the spot price of the underlying upon the derivative's inception. Everything else remains the same Down-and-out and up-and-out options: Compared to down-and-in and up-and-in options respectively, the underlying derivative gets deactivated in case the barrier is touched. All other features are the same
Payoffs	$Payoff = \max(\omega S_T - \omega K, 0)$ where $\omega = 1$ for a call option and -1 for a put option In case of European/ American options, these payoffs are realized upon option exercise at/ before the contract's maturity	Down-and-in: $max\{[\omega S(t^*) - \omega K, 0] \mid S(t) > H \text{ and } S(T) \leq H\}, for \text{ some } t < T \leq t^*$ or $Rm(\tau) \mid S(t) > H \text{ and } S(T) > H, for \text{ all } t < T \leq t^*$ Up-and-in: $max\{[\omega S(t^*) - \omega K, 0] \mid S(t) < H \text{ and } S(T) \geq H\}, for \text{ some } t < T \leq t^*$ or $Rm(\tau) \mid S(t) < H \text{ and } S(T) < H, for \text{ all } t < T \leq t^*$ Down-and-out: $max\{[\omega S(t^*) - \omega K, 0] \mid S(t) > H \text{ and } S(T) > H\}, for \text{ all } t < T \leq t^*$ or $Rd(\tau) \mid S(t) > H \text{ and } S(T) \leq H, for \text{ some } t < T \leq t^*$ Up-and-out: $max\{[\omega S(t^*) - \omega K, 0] \mid S(t) < H \text{ and } S(T) < H\}, for \text{ all } t < T \leq t^*$ or $Rd(\tau) \mid S(t) < H \text{ and } S(T) \geq H, for \text{ some } t < T \leq t^*$ where

		$\omega = 1$ for a call option and -1 for a put option $H =$ barrier $t =$ current time $t^* =$ expiration time $Rm(\tau) =$ rebate paid on maturity in case the barrier is not touched

Part b:

Basis of Difference	Pricing Out-of-the-money call options	Pricing At-the-money call Options
Characteristic	Strike price is higher than the trading price of the underlying asset	Strike price is approximately equals to the trading price of the underlying asset
Volatility	Out the money call options have larger implied volatility. Farthest the out-of-the-money options higher the volatility	At the money calls have lowest implied volatility
	Higher the implied volatility, Higher the option premium/price. Volatility smile suggests that Money-ness, options tends to have higher premium when compared to the options ATM or near the money. Thus when you move from left to right premiums decrease i.e ITM are expensive than ATM and OTM options # discussion is pertained to options with different strike prices and ceteris paribus	
Price	OTM call options doesn't have any intrinsic value it's premium consists only of Time value	ATM call options premium doesn't have any intrinsic value and has only Time Vale



Source: Investopedia

Part c:

Many financial instruments have only unilateral credit risk and this unidirectional risk is considered for their respective pricing. Many other derivative instruments have bilateral credit risk i.e. bidirectional, hence adjustment needs to be made accounting for the credit risk of the both parties.

Default-free option pricing	CVA option pricing
Default-Free options are suitable for Exchange related trading as exchange traded derivatives are guaranteed that trade will go through	CVA adjusts for the counterparty default risk. This adjustment accounts for additional premium when the trade is executed

	CVA pricing is equal to the sum of expected value at each step weighted by the default probability of the counterparty at the step

Part d:

Both the methods are used to model the price under certain assumptions and conditions.

Analytical pricing methods	Monte Carlo simulation methods
Analytical pricing models are mathematical models that can be extended to various other working conditions. They provide a generalized solution which act as a proxy to many real scenarios. These models need validation by considering the assumptions used while deriving the mathematical formulation.	Though Monte Carlo simulation methods, considers the assumptions in the mathematical model, it also makes further assumptions of the behavior of the process. It is used in the scenario where mathematical formulation cannot be derived. Simulation models provide results for a specific use case and should be run many times to counterbalance the effect of numerical calculations. For a different functioning use case, the simulation should be run over again. A simulation model can be accepted when results are validated in a number of working conditions under various input assumptions Sometimes Simulation results may vary from and cannot be accepted due to variation in simulation environments

Part e:

Black-Scholes Volatility	Heston volatility	Local volatility
BS model is built on the constant volatility, the only randomness is considered to be in Stock price $dS_t = \mu S_t dt + \sigma(S_t) d\mathbb{W}_t$	In Stochastic Volatility models, asset prices and their corresponding volatilities are both assumed to be generated via random processes with time variance $\frac{dS_t}{S_t} = \mu(t)dt + \sigma_t d\mathbb{W}_t^S$ $d\sigma_t = \alpha(t,\sigma_t)dt + \beta(t,\sigma_t)d\mathbb{W}_t^\sigma$	In Local Volatility models, the volatility of the underlying security is modelled as a time-varying, deterministic function of t and the t-indexed-spot-price of the underlying asset (S) $dS_t = \mu S_t dt + \sigma(S_t, t) d\mathbb{W}_{t_1}$

References:

[1] Hull J.C. Options, Futures and Other Derivatives, 9th Edition. Pearson

[2] Lee, D., (2018), "Pricing Financial Derivatives Subject to Counterparty Risk and Credit Value Adjustment,"

[3] "Barrier Option: Definition," Investopedia, 2021.

[4] "At-The-Money (ATM)," Corporate Finance Institute From: https://corporatefinanceinstitute.com/resources/knowledge/tradinginvesting/at-the-money-atm/

[5] "Moneyness of an Option Contract," Zerodha University From: https://zerodha.com/varsity/chapter/moneyness-of-an-optioncontract/

[6] Duffie, D., (1996), "Dynamic asset pricing theory," Princeton NJ: Princeton university press.