Submission: 01
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:

Question 1. (ANSWER)

	Advantages	Disadvantages
Up-and-out barrier call option	- Low Price of the Option: Cheaper than vanilla European call options as there is a risk of the option being knocked out in case of exceeding the barrier level. - Less expensive for hedging: As long as the security price is below the barrier level, UAO barrier call options are less expensive in hedging against losses on a short position - Customizable: As it is traded over the counter, the UAO barrier call option could be customized per buyer's requirements.	- Risk of Knock-out: The option worthlessly expires if the price of the underlying goes beyond the knock-out price (barrier level). It is knocked out and no longer comes into existence when it goes below the barrier level. - Limited liquidity: Compared to vanilla European call options with the same strike and expiration, UAO barrier call options have a lower premium given that they are traded over the counter and don't give a baseline estimate. - Imperfect hedging method: in case of the price of the underlying rising beyond the barrier level, the option ceases to exist and becomes worthless.

Question 2. (ANSWER)

UAO barrier call options like other options trade OTC (Over-The-Counter). Therefore, they are to be found in the OTC market as they are not standardized.

Question 3. (ANSWER)

In discrete-time, closed-form solutions do not exist for UAO barrier call options. However, in continuous-time, there exists a closed-form analytical solution.

4. 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
vol	Volatility	0.30
r	Risk-free rate	0.08
B_{UO}	Up-and-Out Barrier	150.0

Name	Formula	Equivalent Python code f()
d1	$\frac{\ln(S_0/K)+(r+\frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}$	calc_d1
d2	$d1 - \sigma\sqrt{T}$	calc_d2
cdf	$\Phi()$	calc_cdf
call option price	$S_0\Phi(d1) - Ke^{-rT}\Phi(d2)$	calc_price_analytical_eur_call_option
S_T	$S_0 e^{(r-\frac{\sigma^2}{2})T+\sigma\sqrt{T}Z}$	calc_terminal_shareprice
discounted call payoff	$(S_T - K)^+ e^{-rT}$	calc_discounted_call_payoff
numerical call option price		calc_price_numerical_eur_call_option

Python version: 3.8.10
For the given parameters,

- The Analytical price of the Call Option turned out to be **$15.711**
- The Numerical (Monte-Carlo) price of the Call Option turned out to be **$15.679**

The code snippets are as follows:-

4. Price a European call option with the information provided

```python
import numpy as np
from scipy.stats import uniform, norm
import matplotlib.pyplot as plt
```
✓ 7.1s

```python
def calc_d1(S0, K, r, vol, T):
    """Function to calculate d1"""
    return (np.log(S0 / K) + (r + .5 * vol ** 2) * T) / (vol * np.sqrt(T))


def calc_d2(d1, vol, T):
    """Function to calculate d2"""
    return d1 - vol * np.sqrt(T)


def calc_cdf(val):
    """Function to calculate cumulative distributive function using scipy norm pkg"""
    return norm.cdf(val)


def calc_price_analytical_eur_call_option(S0, r, T, vol, K, verbose=False):
    """Function to calculate price of Call option"""
    d1 = calc_d1(S0, K, r, vol, T)
    d2 = calc_d2(d1, vol, T)
    if verbose: print(f"d1: {d1}; d2: {d2}")
    price = S0 * calc_cdf(d1) - K * np.exp(-r * T) * calc_cdf(d2)  # compute price using all input and variables
    if verbose: print(f"Call option price: $ {price}, with a standard deviation of {vol} for time {T} and S0={S0} & K={K}")
    return price


def calc_terminal_shareprice(S_0, r, vol, Z, T):
    """Function to calculate the terminal share price using Z"""
    return S_0*np.exp((r - vol**2/2)*T + vol*np.sqrt(T)*Z)


def calc_discounted_call_payoff(S_T, K, r, T):
    """Function to calculate the Discounted Call Payoff"""
    return np.exp(-r*T)*np.maximum(S_T - K, 0)


def calc_price_numerical_eur_call_option(S_0, K, r, vol, T, cycles=50, iterations=1000):
    """Function to calculate the price of European Call Option using Monte Carlo Simulation"""
    np.random.seed(0)
```

Figure 1: Functions declared for Calculating compute for Options

```python
def calc_price_numerical_eur_call_option(S_0, K, r, vol, T, cycles=50, iterations=1000):
    """Function to calculate the price of European Call Option using Monte Carlo Simulation"""
    np.random.seed(0)

    mcall_estimates = [None]*cycles
    mcall_std = [None]*cycles

    for i in range(1, cycles + 1):
        norm_array = norm.rvs(size=i * iterations)
        term_val = calc_terminal_shareprice(S_0, r, vol, norm_array, T)
        mcall_val = calc_discounted_call_payoff(term_val, K, r, T)
        mcall_estimates[i-1] = np.mean(mcall_val)
        mcall_std[i-1] = np.std(mcall_val)/np.sqrt(i * iterations)

    numerical_call_price = np.mean(mcall_estimates)
    print(f"The Numerical EUR CALL option computed via Monte-Carlo simulation is $ {numerical_call_price}")
    return numerical_call_price, mcall_estimates, mcall_std


def plot_estimated_and_analytical_values(analytical_value, cycles, estimated_values, estimated_values_std,
                                         std_degree=3, analytical_plt_color='g',
                                         estimated_plt_style='.', estimated_err_plt_color='r',
                                         xlabel='Sample Size (x1000)', ylabel='Value'):
    """Function to plot the estimated values v/s the Analytical value"""
    plt.plot([analytical_value] * cycles, analytical_plt_color) # Plotting the actual value from Analytical approach.
    plt.plot(estimated_values, estimated_plt_style) # Plot the values estimated via Monte-Carlo Estimation
    plt.plot(analytical_value + np.array(estimated_values_std) * std_degree, estimated_err_plt_color) # <std-degree> standard deviations around the Mean in error bound
    plt.plot(analytical_value - np.array(estimated_values_std) * std_degree, estimated_err_plt_color) # <std-degree> standard deviations around the Mean in error bound
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.show()
```
✓ 0.1s

```python
analytic_call_price = calc_price_analytical_eur_call_option(S_0, r, T, vol, K, verbose=True)
```
✓ 0.1s

```
d1: 0.4166666666666667; d2: 0.1166666666666667
Call option price: $ 15.711312547892973, with a standard deviation of 0.3 for time 1.0 and S0=100.0 & K=100.0
```

```python
cycles = 50
numerical_call_price, mcall_estimates, mcall_std = calc_price_numerical_eur_call_option(S_0, K, r, vol, T, cycles=cycles)
```
✓ 0.3s

```
The Numerical EUR CALL option computed via Monte-Carlo simulation is $ 15.678885422394455
```

Figure 2: Functions declared, and the actual compute

```
1  plot_estimated_and_analytical_values(analytic_call_price, cycles, mcall_estimates, mcall_std)
```
[6] ✓ 0.6s

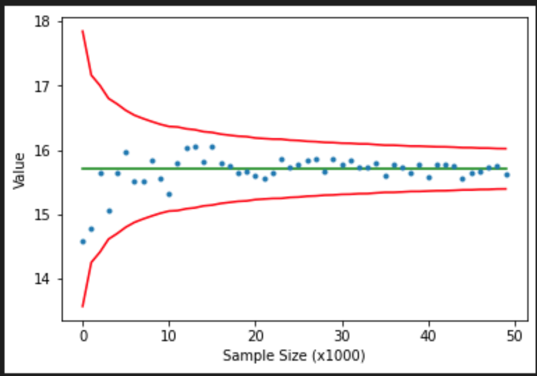


Figure 3: Displaying the Vanilla Call Option pricing result from Analytical and Numerical Methods

$\therefore$ Concludes the Vanilla Call Option Pricing!

5. Pricing European Up-and-Out Barrier Option

S_0, the starting spot price, starts below the barrier. The option is active initially. Before option expiry, if the underlying stock price breaches the barrier even once, then the option gets knocked-out.

$$\text{payoff}_{up-and-out-barrier-call-option} = \begin{cases} \phi & iff \max(S(t)) > B \ \& \ 0 \leq t \leq T \\ [S(T) - K]^+ & otherwise \end{cases}$$

$$\text{payoff}_{up-and-out-barrier-put-option} = \begin{cases} \phi & iff \max(S(t)) > B \ \& \ 0 \leq t \leq T \\ [K - S(T)]^+ & otherwise \end{cases}$$

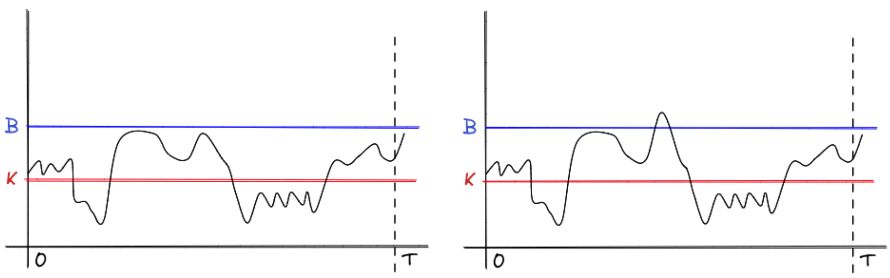


Figure 4: Left option will be exercised; Right option will be knocked-out.

In the pricing below, the simulation for the path of the UND stk price is in increasing sample sizes of 1K to 50K.
The parameters used for this section are the ones used in Q4.

Name	Formula	Equivalent Python code f()
discounted call payoff	$(S_T - K)^+ *$ precomputed_exp_r_T	calc_discounted_call_payoff
S_T	precomputed_part_one * precomputed_part_two^Z	calc_terminal_shareprice
Numerical barrier UO option price		calc_price_barrier_call_up_and_out
Alternate Numerical barrier UO option price		calc_price_barrier_call_up_and_out_alternate_app
d1	$\frac{\ln(S_0/K)+(r+\frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}$	calc_d1
d2	$d1 - \sigma\sqrt{T}$	calc_d2
cdf	$\Phi()$	calc_cdf
call option price	$S_0\Phi(d1) - Ke^{-rT}\Phi(d2)$	calc_price_analytical_eur_call_option
v	$r - \frac{1}{2}\sigma^2$	calc_v
d	$\frac{\ln(a/b)+vT}{\sigma\sqrt{T}}$	calc_d
Analytical Barrier UO option	FORMULA_BO_UO	calc_analytical_up_and_out_barrier

precomputed_part_one $= S_0 e^{(r-\frac{\sigma^2}{2})dt_i}$
precomputed_part_two $= e^{\sqrt{dt_i}\sigma}$
FORMULA_BO_UO $= C_{BS}(S,K) - C_{BS}(S,B) - (B-K)e^{-rT}\Phi(d_{BS}(S,B)) - \frac{B}{S}^{\frac{2v}{\sigma^2}}\{C_{BS}(\frac{B^2}{S},K) - C_{BS}(\frac{B^2}{S},B) - (B-K)e^{-rT}\Phi(d_{BS}(B,S))\}$ [Westermark, "Barrier Option Pricing"]
where,

- S_t: Price of Underlying Stock at time t
- K: Strike Price of the Option
- r: risk-free rate of return
- σ: Volatility / Standard deviation
- T: Option time
- Φ: Standard Normal Cumulative Distribution Function
- d1, d2, call option price, put option price: Calculated in line with the Black-Scholes equation
- B: Barrier
- C_{BS}: Call option price computed from Black-Scholes equation

For the given parameters,

- The Analytical price of the Barrier Up-and-Out Option turned out to be **$5.313**
- The Numerical (Monte-Carlo) price of the Barrier Up-and-Out Option turned out to be **$5.279**

Python version: 3.8.10

The code snippets are as follows:-

5. Price a European up-and-out barrier call option: Simulate paths for the underlying share and for the counterparty's firm value using sample sizes of 1000, 2000, …, 50000. Do monthly simulations for the lifetime of the option

```python
def calc_discounted_call_payoff(S_T, K, precomputed_exp_r_T):
    """
    Function for evaluating the discounted payoff of a call option
    :param S_T: terminal stk price
    :param K: Strike price
    :param precomputed_exp_r_T: equivalent of the np.exp(-rT)
    :return: Discounted payoff of a call option
    """
    return np.maximum(S_T - K, 0) * precomputed_exp_r_T


def calc_terminal_stock_price(precomputed_part_one, precomputed_part_two, Z):
    """
    Calculates the terminal stock price, given the pre-computed values and the random value of Z. Note this is using the same formula as defined in the previous calc_terminal_stock_price, but has
    been reinforced to utilize the precomputed values, to speed up compute.
    :param precomputed_part_one: equivalent of the S_0*e^{(r - .5 * vol^2) * dt_i}, i being the index variable
    :param precomputed_part_two: equivalent of the e^{sqrt{dt_i} * vol}, i being the index variable
    :param Z: Random value to use for stk terminal price compute
    :return: Terminal stk price
    """
    return precomputed_part_one * precomputed_part_two ** Z


def calc_price_barrier_call_up_and_out(S_0, K, T, r, vol, B, cycles=1, base_iteration_steps=1000):
    """
    Function to compute the price for a Barrier Up And Out Call Option
    :param S_0: stk starting price
    :param K: Option Strike Price
    :param T: Option Life
    :param r: risk-free rate of interest
    :param vol: Volatility of underlying stk
    :param B: Barrier level
    :param cycles: number of epochs type iteration to perform
    :param base_iteration_steps: base number of iterations to perform in a cycle
    :return:
        Price of a Barrier Up And Out Call Option,
        Cycle length list of mean of the call option prices computed,
        Cycle length list of std dev of the call option prices computed,
    """

    months = int(T) * 12  # total months
```

Figure 5: Functions declared for Calculating Actual Numerical compute for Barrier Up-Out pricing

Some values were pre-computed in order to save the CPU time otherwise the calculations would be redundant in each simulation iteration.
Below are the explanation for the ones presented in 6

1. The terminal stock price formula is given by: $S_0 e^{(r-\frac{\sigma^2}{2})dt+\sigma\sqrt{dt}Z}$
$\Rightarrow S_0 e^{(r-\frac{\sigma^2}{2})dt} * e^{\sigma\sqrt{dt}Z}$
$\Rightarrow S_0 e^{(r-\frac{\sigma^2}{2})dt} * (e^{\sigma\sqrt{dt}})^Z$
Thus, we can actually precompute these parts as dt & Z are the only changing factors.
This is stored in the 2 arrays of dt_S0_exp_r_vol & dt_sqrt_vol.

2. Another is the discounted call payoff value: $e^{-rT}(S_T - K)^+$
$\Rightarrow e^{-rT} * (S_T - K)^+$
Thus, the e^{-rT} is pre-computed as the S_T is the only varying factor.
This is stored in a variable exp_r_t.

The code snippet below contains the details as well.

```python
def calc_price_barrier_call_up_and_out(S_0, K, T, r, vol, B, cycles=1, base_iteration_steps=1000):
    """
    Function to compute the price for a Barrier Up And Out Call Option
    :param S_0: stk starting price
    :param K: Option Strike Price
    :param T: Option Life
    :param r: risk-free rate of interest
    :param vol: Volatility of underlying stk
    :param B: Barrier level
    :param cycles: number of epochs type iteration to perform
    :param base_iteration_steps: base number of iterations to perform in a cycle
    :return:
        Price of a Barrier Up And Out Call Option,
        Cycle length list of mean of the call option prices computed,
        Cycle length list of std dev of the call option prices computed,
    """

    months = int(T) * 12  # total months
    mcall_estimates = [None] * cycles
    mcall_std = [None] * cycles
    dt_step_value = 1 / 12

    # Pre-computing to save CPU time
    dt = np.array([0.0] * months)  # to store the time steps
    dt_S0_exp_r_vol = np.array([0.0] * months)  # to store the S_0*e^{(r - .5 * vol^2) * dt_i}, i being the index variable
    dt_sqrt_vol = np.array([0.0] * months)  # to store the e^{sqrt{dt_i} * vol}, i being the index variable

    for i in range(0, months):  # Compute the above declared arrays for all the months, which are the stepping function, because dt_i is gonna be same. Only Z would change.
        dt[i] = (i + 1) * dt_step_value
        dt_S0_exp_r_vol[i] = S_0 * np.exp((r - .5 * vol ** 2) * dt[i])
        dt_sqrt_vol[i] = np.exp(np.sqrt(dt[i]) * vol)

    exp_r_t = np.exp(-r * T)  # Compute the e^{-rT} value to be used for the calc_discounted_call_payoff
    # Pre-Compute Complete

    for simulation_cycle in range(1, cycles + 1):  # Going through the cycles
        simulation_limit = base_iteration_steps * simulation_cycle  # Calculating the number of sample sizes to use in this cycle
        random_store = norm.rvs(size=[simulation_limit, months])  # Generating normal random variables for the entire sample size population. 2-d array.
        barrier_call_option_price = np.array([0.0] * simulation_limit)  # Initializing array to use for storing the calculated Barrier Call Option Price

        for iteration in range(0, simulation_limit):  # Iterate through the sample size
            S_T = 0.0
            barrier_breached = False
            for month in range(0, months):  # Iterate through the months
                Z = random_store[iteration][month]  # Obtain the random value to be used for this iteration x month.
                S_T = calc_terminal_stock_price(dt_S0_exp_r_vol[month], dt_sqrt_vol[month], Z)  # Here all the pre-computed values will help in speeding up the compute. For more details, look at the PDF.
                if S_T > B:  # If stk price crosses barrier, then the iteration is null & void as the option expires.
                    barrier_breached = True
                    break
            barrier_call_option_price[iteration] = calc_discounted_call_payoff(S_T, K, exp_r_t) if not barrier_breached else 0.0
            # Compute the discounted call payoff for the S_T only if the barrier was unbreached, and store result.

        mcall_estimates[simulation_cycle - 1] = np.mean(barrier_call_option_price)  # Allot the mean of computed option price at cycle level
        mcall_std[simulation_cycle - 1] = np.std(barrier_call_option_price) / np.sqrt(simulation_limit)  # Allot the std deviation of computed option price at cycle level
    return np.mean(mcall_estimates), mcall_estimates, mcall_std
```

Figure 6: Functions declared for Calculating Actual Numerical compute for Barrier Up-Out pricing

```python
def calc_price_barrier_call_up_and_out_alternate_approach(S_0, K, T, r, vol, B, cycles=1, base_iteration_steps=1000):
    """
    Alternate Function to compute the price for a Barrier Up And Out Call Option. The difference of this method and the calc_price_barrier_call_up_and_out is explained in the PDF.
    :param S_0: stk starting price
    :param K: Option Strike Price
    :param T: Option Life
    :param r: risk-free rate of interest
    :param vol: Volatility of underlying stk
    :param B: Barrier level
    :param cycles: number of epochs type iteration to perform
    :param base_iteration_steps: base number of iterations to perform in a cycle
    :return:
        Price of a Barrier Up And Out Call Option,
        Cycle length list of mean of the call option prices computed,
        Cycle length list of std dev of the call option prices computed,
    """
    months = int(T) * 12  # total months
    n_steps = 12  # Define number of steps.
    mcall_estimates = [None] * cycles
    mcall_std = [None] * cycles

    # Pre-computing below vals to save CPU time
    dt = T / n_steps
    dt_sqrt = np.sqrt(dt)
    vol_dt_sqrt = vol * dt_sqrt
    exp_vol_dt_sqrt = np.exp(vol_dt_sqrt)
    r_vol_dt = (r - 0.5 * vol ** 2) * dt
    exp_r_vol_dt = np.exp(r_vol_dt)
    exp_r_T = np.exp(-r * T)

    for simulation_cycle in range(1, cycles + 1):
        simulation_limit = base_iteration_steps * simulation_cycle
        random_store = norm.rvs(size=[simulation_limit, months])
        calls = np.array([0.0] * simulation_limit)

        for iteration in range(0, simulation_limit):
            S_T = S_0
            barrier_breached = False
            for month in range(0, months):
                Z = random_store[iteration][month]
                S_T *= exp_r_vol_dt * exp_vol_dt_sqrt ** Z
                if S_T > B:
                    barrier_breached = True
                    break
            calls[iteration] = calc_discounted_call_payoff(S_T, K, exp_r_T) if not barrier_breached else 0.0

        mcall_estimates[simulation_cycle - 1] = np.mean(calls)
        mcall_std[simulation_cycle - 1] = np.std(calls) / np.sqrt(simulation_limit)
    return np.mean(mcall_estimates), mcall_estimates, mcall_std
```

Figure 7: Function declared for Calculating alternate Numerical compute for Barrier Up-Out pricing

```
cycles = 50

from datetime import datetime
dt1 = datetime.now()
barrier_up_out_price, call_estimates, call_std = calc_price_barrier_call_up_and_out(S_0=S_0, K=K, T=T, r=r, vol=vol, B=B_uo, cycles=cycles)
print(f"Price for the Barrier up-and-out Call: $ {barrier_up_out_price:.3f}")
delta = datetime.now() - dt1
print(f"Time taken for the compute: {delta}")

[8]  28.6s

Price for the Barrier up-and-out Call: $ 5.279
Time taken for the compute: 0:00:28.522509
```

Figure 8: Computing the Numerical Barrier Up-Out Option Price

Analytical Value compute part below

```
def calc_v(r, vol):
    return r - .5 * vol ** 2


def calc_d(a, b, v, vol, T):
    return (np.log(a / b) + v * T) / (vol * np.sqrt(T))


def calc_analytical_up_and_out_barrier(S0, r, T, vol, K, B):
    """Function to calculate the Analytical price of the Barrier Up and Out Option"""
    v = calc_v(r, vol)
    call_price_S_K = calc_price_analytical_eur_call_option(  # Computes C_BS (S, K)
        S0=S0,
        r=r,
        T=T,
        vol=vol,
        K=K
    )
    call_price_S_B = calc_price_analytical_eur_call_option(  # Computes C_BS (S, B)
        S0=S0,
        r=r,
        T=T,
        vol=vol,
        K=B
    )
    call_price_B2_S_K = calc_price_analytical_eur_call_option(  # Computes C_BS (B^2/S, K)
        S0=B ** 2 / S0,
        r=r,
        T=T,
        vol=vol,
        K=K
    )
    call_price_B2_S_B = calc_price_analytical_eur_call_option(  # Computes C_BS (B^2/S, B)
        S0=B ** 2 / S0,
        r=r,
        T=T,
        vol=vol,
        K=B
    )

    part_1 = call_price_S_K - call_price_S_B - (B - K) * np.exp(-r * T) * calc_cdf(calc_d(S0, B, v, vol, T))
    part_2 = (B / S0) ** (2 * v / (vol ** 2)) * (call_price_B2_S_K - call_price_B2_S_B - (B - K) * np.exp(-r * T) * calc_cdf(calc_d(B, S0, v, vol, T)))
    analytical_barrier_option_price = part_1 - part_2  # Brings together the parts shown in PDF, to compute the Analytical Barrier up-and-out price
    print(f"Barrier up and out at {B} price: {analytical_barrier_option_price} with a standard deviation of {vol} for time {T} and S0={S0} & K={K}")
    return analytical_barrier_option_price
```

Figure 9: Functions declared for Analytical compute of Barrier Up-Out Options

```
analytic_barrier_up_out_price = calc_analytical_up_and_out_barrier(S_0, r, T, vol, K, B_uo)
```

[10] ✓ 0.1s

```
Barrier up and out at 150.0 price: 5.312942876953661 with a standard deviation of 0.3 for time 1.0 and S0=100.0 & K=100.0
```

```
plot_estimated_and_analytical_values(
    analytical_value=analytic_barrier_up_out_price,
    estimated_values=call_estimates,
    estimated_values_std=call_std,
    cycles=cycles
)
```

[11] ✓ 0.5s

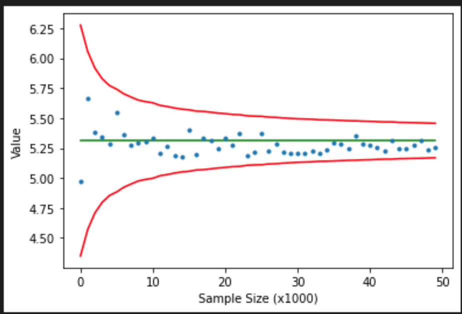


Figure 10: Computing & Displaying the plot of Barrier Up-Out priced using the proper method

```
from datetime import datetime
dt1 = datetime.now()
barrier_up_out_price_alternate, call_estimates_alternate, call_std_alternate = calc_price_barrier_call_up_and_out_alternate_approach(S_0=S_0, K=K, T=T, r=r, vol=vol, B=B_uo, cycles=cycles)
print(f"Price for the Barrier up-and-out Call using alternate approach: $ {barrier_up_out_price_alternate:.3f}")
delta = datetime.now() - dt1
print(f"Time taken for the compute: {delta}")
```

[12] ✓ 26.3s

```
Price for the Barrier up-and-out Call using alternate approach: $ 6.698
Time taken for the compute: 0:00:26.140598
```

```
plot_estimated_and_analytical_values(
    analytical_value=analytic_barrier_up_out_price,
    estimated_values=call_estimates_alternate,
    estimated_values_std=call_std_alternate,
    cycles=cycles
)
```

[13] ✓ 0.5s

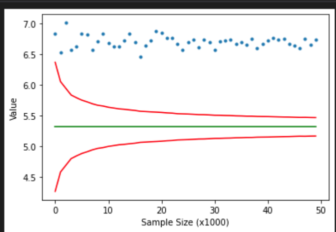


Figure 11: Computing & Displaying the plot of Barrier Up-Out priced using the alternate method

Here we see the difference between the 2 methods used for the Up-and-out compute.
The first method, shown in 6, uses increments of dt i.e. $\frac{1}{12}$ in groups.
So the 12 times the S_T is calculated, the dt changes from $\frac{1}{12}$ to $\frac{2}{12}$ until $\frac{12}{12}$.
Using this method, the Numerical price turned out to be in-bounds with the Analytical price, as shown in 10.

Whereas, the second (alternate) compute method, shown in 7, uses iterative multiplication of S_t until months exhaustion.
It seems correct but there turns out to be a slight difference in the actual dt being used.

dt = $\frac{1}{12}$. $S_T = S_0$;. Per-iteration, S_T takes it's previous S_T as S_0 and recomputes the S_T.
But what happens there is, $S_T = S_0 e^{(r-\frac{\sigma^2}{2})dt+\sigma\sqrt{dt}Z}$
$\Rightarrow S_{T2} = S_T * (e^{(r-\frac{\sigma^2}{2})dt+\sigma\sqrt{dt}Z})$
$\Rightarrow S_{T2} = S_0 * (e^{(r-\frac{\sigma^2}{2})dt+\sigma\sqrt{dt}Z})^2$
$\Rightarrow S_{T2} = S_0 * (e^{(r-\frac{\sigma^2}{2})2dt+2\sigma\sqrt{dt}Z})$
$\Rightarrow S_{T2} = S_0 * (e^{(r-\frac{\sigma^2}{2})2dt+\sigma\sqrt{4dt}Z})$
This is where the mismatch happens in terms of using this approach. If it's re-fed, then it becomes 4dt, whereas it should have been 2dt? Maybe that's the reason why the graph of 11 is distant from the bounds of analytical output.

$\therefore$ Figure 10 shows the optimal Numerical compute of the Barrier up-and-out call option & thus concludes the Barrier Up-and-Out Call Option Pricing!

6. Price Barrier Up-and-In Call Option

As in the problem statement hint, the Barrier Up-and-in call option price can be calculated as follows:
$P_C = P_{B_{UO}} + P_{B_{UI}}$
$\Rightarrow P_{B_{UI}} = P_C - P_{B_{UO}}$

And as we have computed both, P_C - Call Price, & $P_{B_{UO}}$ - Barrier Up-and-Out Price, we can easily obtain $P_{B_{UI}}$ - Barrier Up-and-In Price. As shown in below snippets:

- The Analytical price of Barrier up-in Call Option turned out to be **$10.398**
- The Numerical (Monte-Carlo) price of Barrier up-in Call Option turned out to be **$10.399**

Figure 12: Pricing the Barrier Up-and-In Call option using Analytical and Numerical Methods

7. Price Barrier Up-and-out Call Option price for multiple strike prices

In the below code snippets, the entire 50 cycles of a total of 1,275,000 simulations are run through for the 6 varying Strike Prices.
$\therefore$ A total of 7,650,000 simulations are run for this question!
Below table has the same data as shown in the Pandas dataframe of 14.

Strike price	Barrier up-and-out price
85	10.048
90	8.259
95	6.627
100	5.279
105	4.067
110	3.058
115	2.224

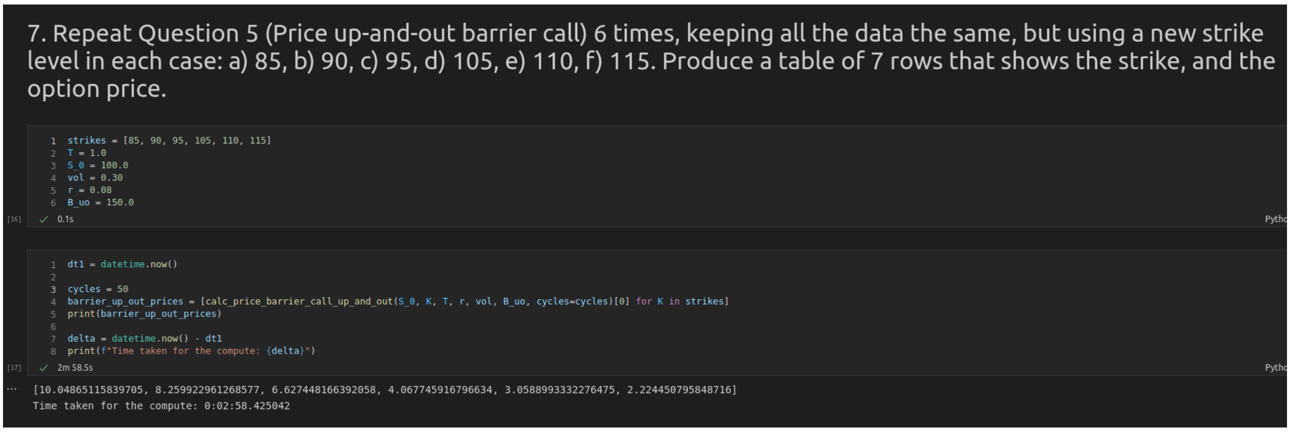


Figure 13: Running pricing of the Barrier Up-and-In Call option using Numerical Method for varying Strike Prices

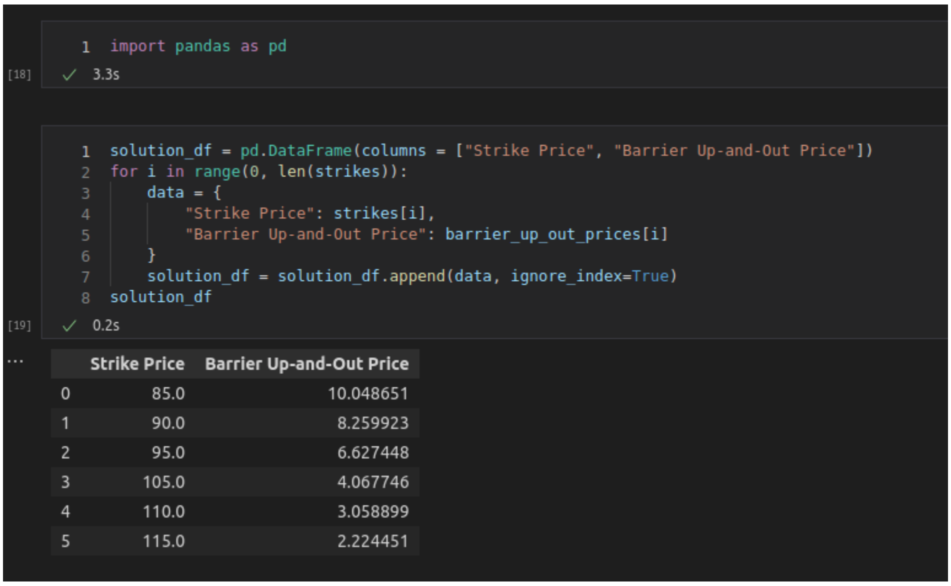


Figure 14: Displaying the Barrier Up-and-In Call option prices against the Strike Prices

Q.8 and Q.9 Procedural summary

We have already discussed the pricing of an up and out barrier option. This part combines that pricing to the firm itself based on the firm's ability to keep the agreement or if they can't then by how much are they able to meet their obligation. We focus on calculation CVA (Credit value adjustment) which can be seen as the value associated 'default/credit risk'

Steps followed are as follows:

a) Instantiating the required input in variable names
b) Instantiating the output variables in form of lists
c) Defining necessary functions and Correlation matrix
d) Running simulations
e) Making necessary plots for Default free option price, CVA and Default adjusted option price

Instantiating the required input in variable names

In [1]:
```
#i/p variables

#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
```

Instantiating the output variables in form of lists

In [2]:
```
#import necessary packages and set seed value
import numpy as np
np.random.seed(0)

#o/p arrays
eu_uao_call_mean = [None]*50
eu_uao_call_stderror = [None]*50

cva_mean = [None]*50
cva_stderror = [None]*50

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

Defining necessary functions and Correlation matrix

i. Terminal stock value, terminal Call payoff and discounted call payoff

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

Here, Z follows standard normal distribution.

```
In [3]: #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)
```

$$Terminal\ Call\ Payoff = (S_T - K)^+$$

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

$$Discounted\ call\ payoff\ =\ e^{-r(T-t)} * Terminal\ call\ payoff$$

```
#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)
```

ii. Analytical price of vanilla call option

$$C = S_0 \varnothing (d_1) - Ke^{-r(T-t)} \varnothing (d_2)$$

d1 and d2 are as defined in the code.

```
In [5]: #Black scholes price of vanilla call option
def bs_call(current_stock_price, strike, time_to_maturity, risk_free_rate, volatility):

    from scipy import stats

    d1 = (np.log(current_stock_price/strike) + (risk_free_rate + volatility**2/2)*time_to_maturity)
                                                              /(volatility*np.sqrt(time_to_maturity))

    d2 = d1 - volatility * np.sqrt(time_to_maturity)

    return current_stock_price * stats.norm.cdf(d1) - strike * np.exp(-risk_free_rate*time_to_maturity) * stats.norm.cdf(d2)
```

iii. Stock price path generator: This function generates the value of stock price for every period of time till expiration of option. (every month in this case, so, 12 months in 1 year)

```python
In [4]: #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
```

iv. Correlation matrix

Correlation matrix

```python
In [7]: corr = np.array([[1, corr_stock_cp],[corr_stock_cp, 1]])
```

Running Simulations for size ranging from 1000 to 50000 with difference of 1000 each.

```python
In [9]: 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

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

```python
#CVA estimate
norm_matrix = norm.rvs(size = np.array([2, paths]))
corr_norm_matrix = np.matmul(np.linalg.cholesky(corr), 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_mean[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] = eu_uao_call_mean[simulation-1] - cva_mean[simulation-1]
default_adj_call_val_stderror[simulation-1] = np.sqrt((eu_uao_call_stderror[simulation-1])**2 +
                                                      (cva_stderror[simulation-1])**2)

print('Running simulation', simulation, '=>\n', 'Call Value:', eu_uao_call_mean[simulation-1].round(3),'\n',
      'CVA:', cva_mean[simulation-1].round(3),'\n', 'Default-adj Call Value:',
      default_adj_call_val[simulation-1].round(3))
```

Explanation for 1st Simulation

Basically, we calculate a call price estimate and CVA estimate and the difference between the two gives us default adjusted call price estimate.

a) Call price estimation
   Here, we generate 1000 stock price paths with the previously defined function and then check whether the stock price breaches the barrier of the up and out option. If the barrier is breached, then the option value for that price path is 0 else it is the same as any call option. Hence we get 1000 terminal values for the up-and-out call option. Then the average and standard error is calculated from those values where the barrier wasn't breached and stored in o/p list defined at the start of the process.
   The process for all simulation is similar except of the change in number of price paths generated.
b) CVA estimation
   Here, we use cholesky's decomposition on the previously defined correlation matrix and then matrix multiply it with a random variable (Standard Normal distribution) matrix of shape (2, no_of_paths_generated) to get correlated Standard Normal random variable.
   We then used above vectors in the matrix to calculate terminal value of both stock and firm.
   The we use the terminal stock value to calculate option payoff which is then be used to calculate 'amount_lost' mean and standard error of which is our CVA estimate and standard error o/p list defined at the start of the process.

The difference of above two lists gives us default adjusted values for option price. The corresponding standard error values is square root of sum of individual variances of Call value and CVA estimate. Head of output is shown below and complete output of above procedures is attached as a '.csv' attachment.

Out[11]:

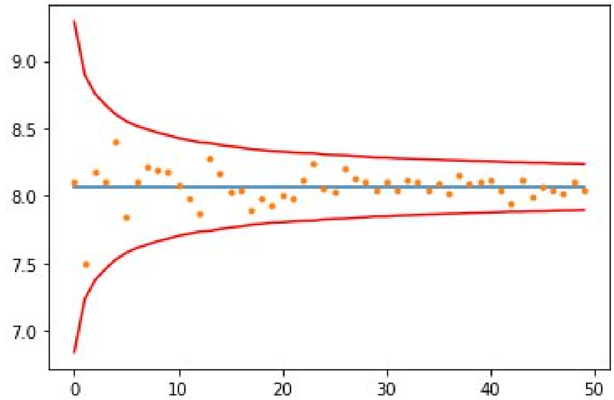
Simulation No.	Default-free UAO Call Value	CVA Estimate	Default-adjusted UAO Call Value
1	8.101	1.654	6.447
2	7.502	1.956	5.546
3	8.177	1.963	6.215
4	8.103	1.873	6.230
5	8.400	2.094	6.306

Making necessary plots for Default free option price, CVA and Default adjusted option price

1)

In [14]:
```
#import required library
import matplotlib.pyplot as plt
%matplotlib inline
```

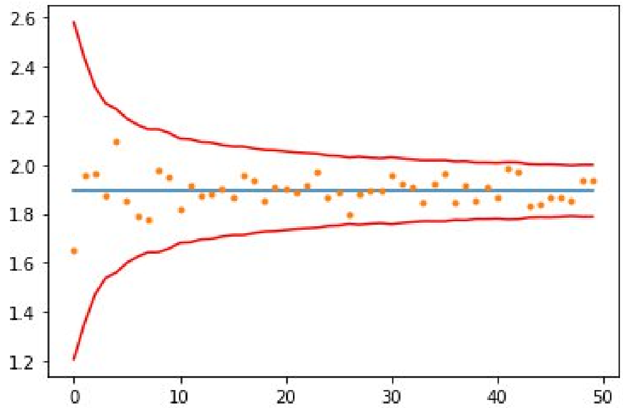
In [15]:
```
#Default free uao eu call option
plt.plot([sum(eu_uao_call_mean)/len(eu_uao_call_mean)]*50)
plt.plot(eu_uao_call_mean, '.')
plt.plot(sum(eu_uao_call_mean)/len(eu_uao_call_mean) +
         np.array(eu_uao_call_stderror) * 3, 'r')
plt.plot(sum(eu_uao_call_mean)/len(eu_uao_call_mean) -
         np.array(eu_uao_call_stderror) * 3, 'r');
```



2)

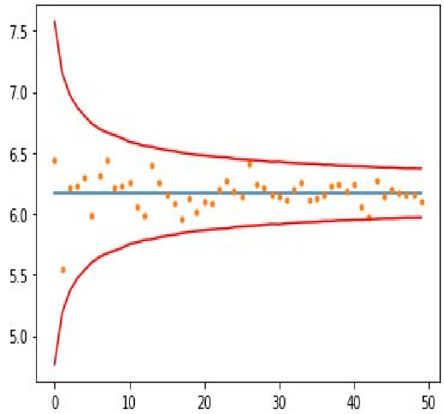
In [16]:
```
#CVA estimates
plt.plot([sum(cva_mean)/len(cva_mean)]*50)
plt.plot(cva_mean, '.')
plt.plot(sum(cva_mean)/len(cva_mean) + np.array(cva_stderror) * 3, 'r')
plt.plot(sum(cva_mean)/len(cva_mean) - np.array(cva_stderror) * 3, 'r');
```

Out[16]:
```
[<matplotlib.lines.Line2D at 0x1bc9b847c70>]
```



3)

In [18]:
```
#default adj european call option
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');
```



Question 10. (ANSWER)

Before diving deep into explaining the difference between default-free value of the option and the Credit Valuation Adjustment (CVA), we will start by defining some basic concepts and giving a brief history that led to the introduction of Credit Valuation Adjustment.

Definitions

Exposure: By assuming a zero recovery, exposure is essentially the loss in case of counterparty default.

Potential future exposure (PFE) is a concept used in finance to define the exposure of a creditor over a given period of time, computed at a defined confidence level. It is a metric of credit risk.

Credit Valuation Adjustment (CVA) is a method of valuing financial derivatives to accommodate potential credit events, including default. At the portfolio or contract level, the CVA is defined as the difference between the risk-free valuation and the valuation that takes into account the probability of default. Calculating the CVA requires prior modeling of default risk.

Default risk: When a party lends money to another party, there is always a risk that the other party will not pay back. This risk applies to bonds. This is called default risk. This risk is naturally a function of the quality of the issuer; it’s in that case that bonds issued by governments offer less profit as their default risk is close to zero.

Default-free value of the option: is the price of the option with zero default risk.

History of CVA

At the time of the 2008 financial crisis, the new regulations focused on a new goal, which was to protect against counterparty risk. For a long time, financial institutions neglected this risk because the amounts invested in their portfolios were relatively small and the probability of default by counterparties was low. Over time, however, the extent of this risk has increased, and its inclusion in the valuation of financial instruments has become a critical factor. The adjustment in value that is made to a financial product to take into account the counterparty risk is called the Credit Value Adjustment (CVA).

Difference between default-free value of the option and the Credit Valuation Adjustment

For our first group work submission, we considered an UAO European Call Option with option maturity T = 1 year. From computations, the default-free value of the UAO call option is 3.5. In the existence of a closed-form analytical solution for the option, the value of the option is computed using the analytical formula and is also called the fair value of the Up-And-Out call. On the other hand, in taking into account the default risk, Credit Valuation Adjustment gives the amount that the holder risks losing in case of default.

Simulations showed that the value of the option could go below the value of the debt, meaning that default risk is not to be neglected and the issuer risks defaulting. As a one-sided derivative, if the option issuer defaults, we suffer a full loss of the fair value of the UAO European Call Option. With the computed CVA, we can hedge against the credit risk of the counterparty by paying the CVA. We remove the default risk to obtain a market value of the derivative or the value of the derivative containing the credit risk of the counterparty. Accordingly, we would anticipate that the default-free value of the option would exceed the value of the option containing the counterparty credit risk - the market value. Additionally, the market value of the derivative is expected to exceed the amount we would use to hedge against the counterparty's credit risk - CVA.

Done by

Part	Creator
1-3	Christian
4-7	Vivek
8-9	Harshil
10	Christian
Compilation	Vivek

References

1. Alavian, Shahram, et al. *Credit Valuation Adjustment (CVA)*. 2014
2. Niklas Westermark, "Barrier Option Pricing" https://www.math.kth.se/matstat/seminarier/reports/K-exjobb09/090601a.pdf
3. Mitchell, Cory. “Up-and-out Option Definition.” *Investopedia*, Investopedia, 8 Feb. 2022, https://www.investopedia.com/terms/u/up-and-outoption.asp#:~:text=What%20Is%20an%20Up%2Dand,level%2C%20called%20the%20barrier%20price. Accessed April 15, 2022
4. S.M. Levitan, et al. *Discrete Closed-Form Solutions for Barrier Options*. May 30, 2003, https://econrsa.org/system/files/publications/working_papers_interest/wp29_interest.pdf
5. Chen, James. “Vanilla Option Definition.” *Investopedia*, Investopedia, 7 Feb. 2022, https://www.investopedia.com/terms/v/vanillaoption.asp#:~:text=Vanilla%20Option%20Features&text=If%20the%20strike%20price%20is,for%20it%20to%20be%20exercised. Accessed April 17, 2022