CS 476 A2

Q1

Jeongseop Yi (Patrick), j22yi

Q1a)

In [ ]:
```
# import libraries
import numpy as np
import pandas as pd
from scipy import stats
```

In [ ]:
```
# payoff function for put
def put_payoff_func(S, K):
    return max(K - S, 0)

# payoff function for put
def call_payoff_func(S, K):
    return max(S - K, 0)

# create a binomial lattice with n steps
def binomial_lattice(n, sigma, r, T, S0, K, payoff_func):
    # create the arrays
    stock_array = np.zeros((n+1))
    option_array = np.zeros((n+1))
    new_option_array = np.zeros((n+1))
    # the value u, d, and q
    u = np.exp(sigma*np.sqrt(T/n) + (r - 0.5*(sigma**2))*T/n)
    d = np.exp(-sigma*np.sqrt(T/n) + (r - 0.5*(sigma**2))*T/n)
    q = 1/2
    for j in range(0, n+1):
        # calculate the stock price at each node
        stock_array[j] = S0 * (u**j) * (d**(n-j))
        # calculate the option price at each node
        option_array[j] = payoff_func(stock_array[j], K)

    # loop through the binomial lattice backwards
    for i in range(n, 0, -1):
        # get lagged option array for calculation
        option_array_r = np.roll(option_array, -1)
        # set the last value to nan
        option_array_r[-1] = np.nan
        # calculate the new option array
        new_option_array = (np.exp(-r * T / n) * q *
                            (option_array_r + option_array))

        # set the array to the new array
        option_array = new_option_array.copy()
        # reset the new row
        new_option_array = np.zeros((n+1))
```

```python
    return option_array
```

In [ ]:
```python
def blsprice2(S0, K, r, T, sigma):
    ''' Valuation of European option in BSM model Analytical formula.
    '''
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = (np.log(S0 / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    # if optionType == 'call':
    C_value = (S0 * stats.norm.cdf(d1, 0.0, 1.0) -
               K * np.exp(-r * T) * stats.norm.cdf(d2, 0.0, 1.0))
    # elif optionType == 'put':
    P_value = (K * np.exp(-r * T) * stats.norm.cdf(-d2, 0.0, 1.0)
               - S0 * stats.norm.cdf(-d1, 0.0, 1.0))
    return (C_value, P_value)
```

In [ ]:
```python
# set the parameters
sigma = 0.2
r = 0.03
T = 1
S0 = 10
K = 10

print("The value of (call, put) options from blsprice: ",
      blsprice2(S0, K, r, T, sigma))

# create a dataframe to store the convergence results
convergence_test = {'n': []}
for i in range(11):
    convergence_test['n'].append(2 ** i * 20)

convergence_test = pd.DataFrame(convergence_test)

# calculate the option price for each n
convergence_test['Dt'] = T / convergence_test['n']
# value of the put option
convergence_test['Put'] = convergence_test['n'].apply(
    lambda x: binomial_lattice(x, sigma, r, T, S0, K, put_payoff_func)[0])
# difference between the option values
convergence_test['Put_Change'] = convergence_test['Put'].diff()
# ratio of the change
convergence_test['Put_Ratio'] = convergence_test['Put_Change'].shift() \
    / convergence_test['Put_Change']
# value of the call option
convergence_test['Call'] = convergence_test['n'].apply(
    lambda x: binomial_lattice(x, sigma, r, T, S0, K, call_payoff_func)[0])
# Change between the option values
convergence_test['Call_Change'] = convergence_test['Call'].diff()
# ratio of the change
convergence_test['Call_Ratio'] = convergence_test['Call_Change'].shift() \
    / convergence_test['Call_Change']

# show the results
display(convergence_test)
```

```
The value of (call, put) options from blsprice:  (0.9413403383853014, 0.6457956738703832)
```

	n	Dt	Put	Put_Change	Put_Ratio	Call	Call_Change	Call_Ratio
0	20	0.050000	0.643605	NaN	NaN	0.939083	NaN	NaN
1	40	0.025000	0.646015	0.002411	NaN	0.941527	0.002444	NaN
2	80	0.012500	0.646696	0.000680	3.543240	0.942224	0.000697	3.506337
3	160	0.006250	0.646662	-0.000034	-20.288474	0.942199	-0.000025	-27.656104
4	320	0.003125	0.646381	-0.000281	0.119266	0.941922	-0.000277	0.090984
5	640	0.001563	0.646054	-0.000327	0.858698	0.941596	-0.000325	0.851391
6	1280	0.000781	0.645758	-0.000296	1.107077	0.941301	-0.000295	1.103922
7	2560	0.000391	0.645837	0.000079	-3.728976	0.941381	0.000080	-3.691605
8	5120	0.000195	0.645808	-0.000029	-2.723792	0.941352	-0.000029	-2.766415
9	10240	0.000098	0.645814	0.000006	-4.619267	0.941359	0.000006	-4.485322
10	20480	0.000049	0.645805	-0.000010	-0.650867	0.941349	-0.000010	-0.668805

As n increases, the value for put and call option clearly converges to the blsprice for the each option respectively, which is 0.6457957 and 0.9413403. However, we cannot determine whether the value converges linearly or quadradically as the ratio fluctuates randomly as in the table above.

The ratio is 4 when the convergence is quadratic.

The proof is as follows:

$$
\begin{aligned}
&\lim_{\Delta\to 0}\frac{V_0^{tree}((\Delta t)/2)-V_0^{tree}(\Delta t)}{V_0^{tree}((\Delta t)/4)-V_0^{tree}((\Delta t)/2)}\\
&=\lim_{\Delta\to 0}\frac{V_0^{exact}+\alpha(\Delta t/2)^2+o((\Delta t/2)^2)-V_0^{exact}-\alpha(\Delta t)^2-o((\Delta t)^2)}{V_0^{exact}+\alpha(\Delta t/4)^2+o((\Delta t/4)^2)-V_0^{exact}-\alpha((\Delta t/2))^2-o((\Delta t/2)^2)}\\
&=\lim_{\Delta\to 0}\frac{\alpha(\Delta t/2)^2-\alpha(\Delta t)^2+o((\Delta t)^2)}{\alpha(\Delta t/4)^2-\alpha((\Delta t/2))^2+o((\Delta t)^2)}\\
&=\lim_{\Delta\to 0}\frac{\alpha(\Delta t/2)^2-\alpha(\Delta t)^2+o((\Delta t)^2)}{1/4(\alpha(\Delta t/2)^2-\alpha(\Delta t)^2+o((\Delta t)^2))}\\
&=\lim_{\Delta\to 0}4\cdot\frac{\alpha(\Delta t/2)^2-\alpha(\Delta t)^2+o((\Delta t)^2)}{\alpha(\Delta t/2)^2-\alpha(\Delta t)^2+o((\Delta t)^2)}\\
&=4
\end{aligned}
$$

Q1b)

In [ ]:

```
# smooth payoff function for call
def call_payoff_func_smooth(S, K, sigma, n, T):
    # case 1: S * exp(-sigma * np.sqrt(T/n)) > K
    if (S * np.exp(-sigma * np.sqrt(T/n)) > K):
        return ((S * (np.exp(sigma * np.sqrt(T/n))
                 - np.exp(-sigma * np.sqrt(T/n)))
```

```python
                 / (2 * sigma * np.sqrt(T/n))) - K)
    # case 2: S * exp(sigma * np.sqrt(T/n)) < K
    elif (S * np.exp(sigma * np.sqrt(T/n)) < K):
        return 0
    # case 3: between the two values
    else:
        return ((S * (np.exp(sigma * np.sqrt(T/n)) - (K / S)) -
                K * (sigma * np.sqrt(T/n) - np.log(K / S)))
                / (2 * sigma * np.sqrt(T/n)))

# Smooth payoff function for put
def put_payoff_func_smooth(S, K, sigma, n, T):
    # case 1: S * exp(-sigma * np.sqrt(T/n)) > K
    if (S * np.exp(-sigma * np.sqrt(T/n)) > K):
        return 0
    # case 2: S * exp(sigma * np.sqrt(T/n)) < K
    elif (S * np.exp(sigma * np.sqrt(T/n)) < K):
        return (K - (S * (np.exp(sigma * np.sqrt(T/n)) -
                          np.exp(-sigma * np.sqrt(T/n)))
                     / (2 * sigma * np.sqrt(T/n))))
    # case 3: between the two values
    else:
        return ((K * (np.log(K / S) + sigma * np.sqrt(T/n))
                 - S * ((K / S) - np.exp(-sigma * np.sqrt(T/n))))
                / (2 * sigma * np.sqrt(T/n)))

# create a binomial lattice with n steps with smooth payoff function
def binomial_lattice_smooth(n, sigma, r, T, S0, K, payoff_func_smooth):
    # create the arrays
    stock_array = np.zeros((n+1))
    option_array = np.zeros((n+1))
    new_option_array = np.zeros((n+1))
    # u and d
    u = np.exp(sigma*np.sqrt(T/n) + (r - 0.5*(sigma**2))*T/n)
    d = np.exp(-sigma*np.sqrt(T/n) + (r - 0.5*(sigma**2))*T/n)
    q = 1/2
    # time N
    for j in range(0, n+1):
        # calculate the stock price at each node
        stock_array[j] = S0 * (u**j) * (d**(n-j))
        # calculate the option price at each node
        option_array[j] = payoff_func_smooth(stock_array[j], K, sigma, n, T)

    # loop through the binomial lattice backwards
    for i in range(n, 0, -1):
        # get lagged option array for calculation
        option_array_r = np.roll(option_array, -1)
        # set the last value to nan
        option_array_r[-1] = np.nan
        # calculate the new option array
        new_option_array = np.exp(-r * T / n) * 0.5 * \
            (option_array_r + option_array)

        # set the row to the new row
        option_array = new_option_array.copy()
        # reset the new row
```

```python
        new_option_array = np.zeros((n+1))

    # return the array
    return option_array
```

In [ ]:

```python
# create a dataframe to store the convergence results
convergence_test_smooth = {'n': []}
for i in range(11):
    convergence_test_smooth['n'].append(2 ** i * 20)

convergence_test_smooth = pd.DataFrame(convergence_test_smooth)

# calculate the option price for each n
# calculate Dt
convergence_test_smooth['Dt'] = T / convergence_test_smooth['n']
# value of the put option
convergence_test_smooth['Put_Value'] = convergence_test_smooth['n'].apply(
    lambda x: binomial_lattice_smooth(x, sigma, r, T, S0, K, put_payoff_func_smooth)[0])
# Change between the option values
convergence_test_smooth['Put_Change'] = convergence_test_smooth['Put_Value'].diff()
# ratio of the change
convergence_test_smooth['Put_Ratio'] = convergence_test_smooth['Put_Change'].shift(1) \
    / convergence_test_smooth['Put_Change']

# value of the call option
convergence_test_smooth['Call_Value'] = convergence_test_smooth['n'].apply(
    lambda x: binomial_lattice_smooth(x, sigma, r, T, S0, K, call_payoff_func_smooth)[0])
# Change between the option values
convergence_test_smooth['Call_Change'] = convergence_test_smooth['Call_Value'].diff()
# ratio of the change
convergence_test_smooth['Call_Ratio'] = convergence_test_smooth['Call_Change'].shift(1) \
    / convergence_test_smooth['Call_Change']

# show the results
display(convergence_test_smooth)
```

	n	Dt	Put_Value	Put_Change	Put_Ratio	Call_Value	Call_Change	Call_Ratio
0	20	0.050000	0.653878	NaN	NaN	0.952690	NaN	NaN
1	40	0.025000	0.649880	-0.003997	NaN	0.947059	-0.005631	NaN
2	80	0.012500	0.647849	-0.002031	1.968050	0.944211	-0.002848	1.977258
3	160	0.006250	0.646825	-0.001024	1.983720	0.942778	-0.001432	1.988384
4	320	0.003125	0.646311	-0.000514	1.991785	0.942060	-0.000718	1.994132
5	640	0.001563	0.646054	-0.000258	1.994850	0.941700	-0.000360	1.996317
6	1280	0.000781	0.645925	-0.000129	1.996454	0.941520	-0.000180	1.997462
7	2560	0.000391	0.645860	-0.000064	2.002006	0.941430	-0.000090	2.001439
8	5120	0.000195	0.645828	-0.000032	2.002092	0.941385	-0.000045	2.001499
9	10240	0.000098	0.645812	-0.000016	1.997844	0.941363	-0.000022	1.998456
10	20480	0.000049	0.645804	-0.000008	2.000374	0.941352	-0.000011	2.000267

The ratios with the non-smooth payoff function were fluctuating, so we could not determine whether the option value converges linearly or quadradically. However, using the smooth payoff function, the ratios are close to 2 as in the table above. Now, we can determine that the option value converges linearly.