CS 476 A2

Q2

Jeongseop Yi (Patrick), j22yi

Q2a)

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

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)))
                 / (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)))

# payoff function for dividends for given stock price array
def div_payoff_func(S, D0, p):
    return np.array(list(map(lambda x: max(p * x, D0), S)))

# create a binomial lattice with dividends
def binomial_lattice_div(n, sigma, r, T, S0, K, D0, p, div,
                         payoff_func, div_payoff_func):
    # change the dividend array in terms of time steps
```

```
    div = np.array(div) * n

    # create the arrays for the stock and option values
    stock_array = np.zeros((n+1))
    new_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):
        # set the element to the sum of the two elements above it
        stock_array[j] = S0 * (u**j) * (d**(n-j))
        option_array[j] = payoff_func(stock_array[j], K, sigma, n, T)

    # loop through the lattice backwards
    for i in range(n - 1, -1, -1):
        # get lagged stock and option array for calculation
        stock_array_r = np.roll(stock_array, -1)
        option_array_r = np.roll(option_array, -1)
        # set the last value to nan
        option_array_r[-1] = np.nan
        stock_array_r[-1] = np.nan
        # calculate the new stock and option array
        new_stock_array = (np.exp(-r * T / n) * q *
                           (stock_array_r + stock_array))
        new_option_array = (np.exp(-r * T / n) * q *
                            (option_array_r + option_array))
        # set the arrays to the new arrays
        stock_array = new_stock_array.copy()
        option_array = new_option_array.copy()
        # reset the new row
        new_stock_array = np.zeros((n+1))
        new_option_array = np.zeros((n+1))

        # check if current time is dividend time
        # check if i is in between div and div + 1
        if (np.any(np.logical_and(i >= div, i < div + 1))):
            # interpolate the option array to the stock array for the dividend
            option_array = \
                np.interp(stock_array - div_payoff_func(stock_array, D0, p),
                          stock_array, option_array).copy()
    # return the array
    return option_array
```

In [ ]:

```
# set the parameters for q2a
sigma = 0.2
r = 0.03
T = 1
S0 = 10
K = 10
D0 = 0.3
```

```
p = 0.003
div = np.array([T/2])

# create a dataframe to store the convergence results
conv_test_smooth_div = {'n' : []}
for i in range(12):
    conv_test_smooth_div['n'].append(2 ** i * 20)

conv_test_smooth_div = pd.DataFrame(conv_test_smooth_div)

# calculate the value of the option for each n
# Dt = T / n
conv_test_smooth_div['Dt'] = T / conv_test_smooth_div['n']
# value of the put option with the smooth payoff function
conv_test_smooth_div['Put_Value'] = conv_test_smooth_div['n'].apply(
    lambda x: binomial_lattice_div(x, sigma, r, T, S0, K, D0, p, div,
                                   put_payoff_func_smooth, div_payoff_func)[0])
# calculate the change in value
conv_test_smooth_div['Put_Change'] = conv_test_smooth_div['Put_Value'].diff()
# calculate the ratio of the change in value
conv_test_smooth_div['Put_Ratio'] = conv_test_smooth_div['Put_Change'].shift(1) \
    / conv_test_smooth_div['Put_Change']
# value of the call option with the smooth payoff function
conv_test_smooth_div['Call_Value'] = conv_test_smooth_div['n'].apply(
    lambda x: binomial_lattice_div(x, sigma, r, T, S0, K, D0, p, div,
                                   call_payoff_func_smooth, div_payoff_func)[0])
# calculate the change in value
conv_test_smooth_div['Call_Change'] = conv_test_smooth_div['Call_Value'].diff()
# calculate the ratio of the change in value
conv_test_smooth_div['Call_Ratio'] = conv_test_smooth_div['Call_Change'].shift(1) \
    / conv_test_smooth_div['Call_Change']

# display the dataframe
display(conv_test_smooth_div)
```

	n	Dt	Put_Value	Put_Change	Put_Ratio	Call_Value	Call_Change	Call_Ratio
0	20	0.050000	0.808951	NaN	NaN	0.812419	NaN	NaN
1	40	0.025000	0.798157	-0.010793	NaN	0.799753	-0.012666	NaN
2	80	0.012500	0.790926	-0.007232	1.492498	0.791729	-0.008024	1.578505
3	160	0.006250	0.786549	-0.004377	1.652254	0.786956	-0.004773	1.681175
4	320	0.003125	0.786359	-0.000190	23.030164	0.786568	-0.000388	12.299278
5	640	0.001563	0.785417	-0.000941	0.201861	0.785527	-0.001040	0.372961
6	1280	0.000781	0.785207	-0.000210	4.477767	0.785268	-0.000260	4.005580
7	2560	0.000391	0.785031	-0.000176	1.196470	0.785067	-0.000200	1.295673
8	5120	0.000195	0.784950	-0.000081	2.159054	0.784973	-0.000094	2.138063
9	10240	0.000098	0.784910	-0.000040	2.035533	0.784927	-0.000046	2.030771
10	20480	0.000049	0.784890	-0.000020	1.992230	0.784904	-0.000023	1.993268
11	40960	0.000024	0.784880	-0.000010	2.000579	0.784892	-0.000012	2.000505

As the absolute value of the changes in both call and put options decreases, the option values for both call and put options are converging to some value close to 0.784880 for put option and 0.784892 for the call option. As n increases, the ratio also converges to 2, which indicates that the option converges linearly. The ratio fluctuates a bit for small n values, as we use interpolation to guess the option value for $S(t_d^-)$. It is more likely to use interpolated option value, where the stock price $S(t_d^-) - D$ does not exist in $S(t_d)$, when n is small.

Q2b)

In [ ]:
```
# parameters for Q2b
D0_b = 0.1
p_b = 0.001
div_b = [0.25, 0.5, 0.75]

# convergence test for Q2b
conv_test_smooth_div_b = {'n' : []}
for i in range(15):
    conv_test_smooth_div_b['n'].append(2 ** i * 20)

# Dataframe for Q2b
conv_test_smooth_div_b = pd.DataFrame(conv_test_smooth_div_b)

# calculate the value of the put option for each n
conv_test_smooth_div_b['Dt'] = T / conv_test_smooth_div_b['n']
conv_test_smooth_div_b['Put_Value'] = conv_test_smooth_div_b['n'].apply(
    lambda x: binomial_lattice_div(x, sigma, r, T, S0, K, D0_b, p_b, div_b,
                                   put_payoff_func_smooth, div_payoff_func)[0])
conv_test_smooth_div_b['Put_Change'] = conv_test_smooth_div_b['Put_Value'].diff()
conv_test_smooth_div_b['Put_Ratio'] = conv_test_smooth_div_b['Put_Change'].shift(1) \
    / conv_test_smooth_div_b['Put_Change']
```

```
# calculate the value of the call option for each n
conv_test_smooth_div_b['Call_Value'] = conv_test_smooth_div_b['n'].apply(
    lambda x: binomial_lattice_div(x, sigma, r, T, S0, K, D0_b, p_b, div_b,
                                   call_payoff_func_smooth, div_payoff_func)[0])
conv_test_smooth_div_b['Call_Change'] = conv_test_smooth_div_b['Call_Value'].diff()
conv_test_smooth_div_b['Call_Ratio'] = conv_test_smooth_div_b['Call_Change'].shift(1) \
    / conv_test_smooth_div_b['Call_Change']

display(conv_test_smooth_div_b)
```

	n	Dt	Put_Value	Put_Change	Put_Ratio	Call_Value	Call_Change	Call_Ratio
0	20	0.050000	0.811693	NaN	NaN	0.818142	NaN	NaN
1	40	0.025000	0.803776	-0.007917	NaN	0.805463	-0.012679	NaN
2	80	0.012500	0.796759	-0.007017	1.128316	0.797557	-0.007906	1.603739
3	160	0.006250	0.792038	-0.004721	1.486360	0.792440	-0.005117	1.545077
4	320	0.003125	0.788866	-0.003173	1.487993	0.789069	-0.003371	1.518103
5	640	0.001563	0.786710	-0.002156	1.471571	0.786814	-0.002255	1.494773
6	1280	0.000781	0.785268	-0.001441	1.495909	0.785323	-0.001491	1.512649
7	2560	0.000391	0.785199	-0.000069	20.771309	0.785229	-0.000094	15.835726
8	5120	0.000195	0.784975	-0.000224	0.309973	0.784993	-0.000236	0.398516
9	10240	0.000098	0.784910	-0.000065	3.457767	0.784922	-0.000071	3.330582
10	20480	0.000049	0.784852	-0.000059	1.103283	0.784860	-0.000062	1.148198
11	40960	0.000024	0.784826	-0.000026	2.253496	0.784833	-0.000028	2.239281
12	81920	0.000012	0.784815	-0.000011	2.334803	0.784821	-0.000012	2.313078
13	163840	0.000006	0.784809	-0.000006	1.960329	0.784815	-0.000006	1.962876
14	327680	0.000003	0.784806	-0.000003	1.999371	0.784812	-0.000003	1.999293

As the absolute value of the changes in both call and put options decreases, the option values for both call and put options are converging to some value close to 0.784826 for put option and 0.784833 for the call option. As n increases, the ratio also converges to 2, which indicates that the option converges linearly. The ratio fluctuates for small n values, as we use interpolation to guess the option value for $S(t_d^-)$. It is more likely to use interpolated option value, where the stock price $S(t_d^-) - D$ does not exist in $S(t_d)$, when n is small. Especially in b), as there exist more dividend payout periods, the error from the interpolations may be bigger affecting ratio values than the ratio values in a).

In comparison to a), there is very minimal difference in the option values for both put and call. The difference between in a) and b) is that in a), the dividend is paid once with $D_0 = 0.3$ and $\rho = 0.3\%$ and in b) the dividend is paid three times with $D_0 = 0.1$ and $\rho = 0.1\%$. The main difference between a) and b) is whether the dividend is paid once or three times with the same total dividend at time 0. As the total dividend at time 0 in both a) and b) equals, the timing of the dividend payout should not create a difference

for the put and call option values. Therefore, the option values in both a) and b) must converge to the same value, and we have a corresponding result in both tables. The minimal difference between a) and b) may derive from calculation and interpolation errors.