Pricing Lookback Options using Tree Method

FE 620 - Pricing and Hedging

Group 5

FangChi Wu

FangYih Chan

Chen HaoYu

Table of Contents

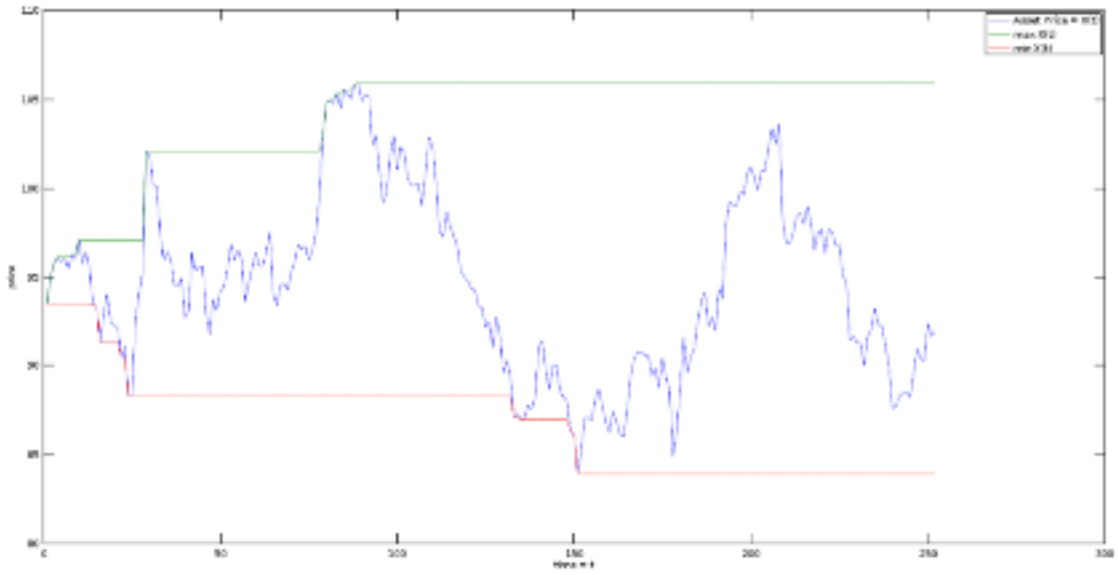
1. Introduction 3
2. Data Analysis 4
3. Methodology
    a. Binomial tree 5
    b. Non-recombined tree with lookback features 5
    c. Recombined tree with lookback features 6
    d. Trinomial tree extension 9
4. Theoretical formula 9
5. Valuation results Analysis 11
6. Delta hedging Analysis 13
7. Discussion & Conclusion 15
8. References 17
9. Python codes 18

Introduction

Lookback options are path dependent and exotic options that allow the holder to “look back” at the price of the underlying asset from the beginning to the expiration T to decide the optimal price to exercise. Therefore, the option holder will not have to worry for selling too early or holding too long as the lookback feature allows the option holder to buy at the lowest price for call option or sell at the highest price for put option respectively during the lifetime of the option (OptionsTradingpedia, 2020).

There are two types of lookback options: **Floating-Strike** lookback option and **Fixed-Strike** lookback options. Floating-Strike lookback option pays off in the profile the difference between the most favourable price throughout the lifetime and the price at expiration T, i.e. $max(S_T - S_{min}, 0)$ for call option or $max(S_{max} - S_T, 0)$ for put option. Whereas, Fixed-Strike lookback option pays off in profit the difference between the most favourable price throughout the lifetime and fixed strike price, i.e. $max(S_{max} - K, 0)$ for call option or $max(K - S_{min}, 0)$ for put option. With the advantage of potentially gaining higher profit, lookback option is therefore priced much more expensive/premium compared to plain vanilla option (OptionTradingpedia,2020)

Similar to vanilla option, lookback options come in both *European* style which allows holder to exercise at the expiration or *American* style that allows holders to have the flexibility to exercise at any time before and including the day of expiration to capture profit as soon as the stock price moves favorably.

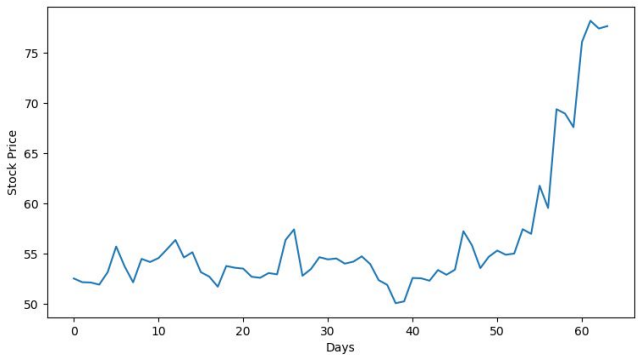


Data Analysis

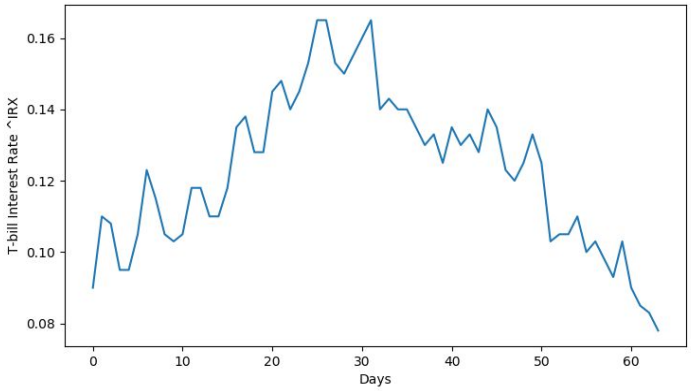
We will be using tree method to compute the option prices. In order to construct the binomial tree or trinomial tree, the parameters of the underlying stock are required. Therefore, data analysis on historical data was performed and we have chosen AMD as the stock for our analysis. AMD stock is a suitable candidate as it is a non-dividend paying stock and have actively traded option markets. 3 months (63 business days) recent historical data (from May 2020 to Aug 2020) were extracted from Yahoo Finance to compute the volatility of the stock:

$$\sigma = \frac{1}{\sqrt{\Delta t}}\sqrt{\frac{1}{N-1}\sum(u_i - \overline{u})^2}$$ where u_i = daily log return $\ln(S_{i+1}/S_i)$ and Δt= 1/252

The software *Python* programming which is also used to simulate our option pricing was used to pull the data directly using library '*panda_datareader'*. The volatility was found to be 58.39% and the stock price is plotted as below



The risk free interest rate is also determined by extracting the 13 week Treasury Bill ^IRX data from Yahoo Finance for a period of 3 months similarly to AMD stock price. The average rate was found to be 0.12 and the interest rate is plotted as below:



Methodology

Binomial tree

The binomial tree will be generated to present the diffusion process and each node (j,i) (i.e. *j*th node at *i*th time step) will have a stock price generated from its previous node with up probability of $p=(e^{(r-q)\Delta t}-d)/(u-d)$ or down probability of $1-p=(u-e^{(r-q)\Delta t})/(u-d)$ by an multiplier amount $u=e^{\sigma\sqrt{\Delta t}}$ or $d=e^{-\sigma\sqrt{\Delta t}}$ respectively. This binomial model approach is based on Cox, Ross and Rubinstein (Hull, 2015). This result is actually based on Black-Scholes equation, where the stock price at the end of the period of lognormal random variable (in the risk-neutral world)

$$S(t+\Delta t) = S(t) \cdot e^{\mu \Delta t + \sigma\varepsilon\sqrt{\Delta t}} \quad \text{where } \mu = r - \sigma^2/2;\ \varepsilon \text{ is a normal variable } N(0,1)$$

with mean $e^{r\Delta t}.S$ and variance $e^{2r\Delta t}(e^{\sigma^2.\Delta t}-1).S^2$ (London, 2005). In the binomial model, the price at the end of single period Δt is a random variable of 2 possible states *uS* and *dS* with probability *p* and *1-p* and must have the same mean and variance and thus satisfy the constraint set by 2 matching moments:

$$puS + (1-p)dS = e^{r\Delta t}.S$$

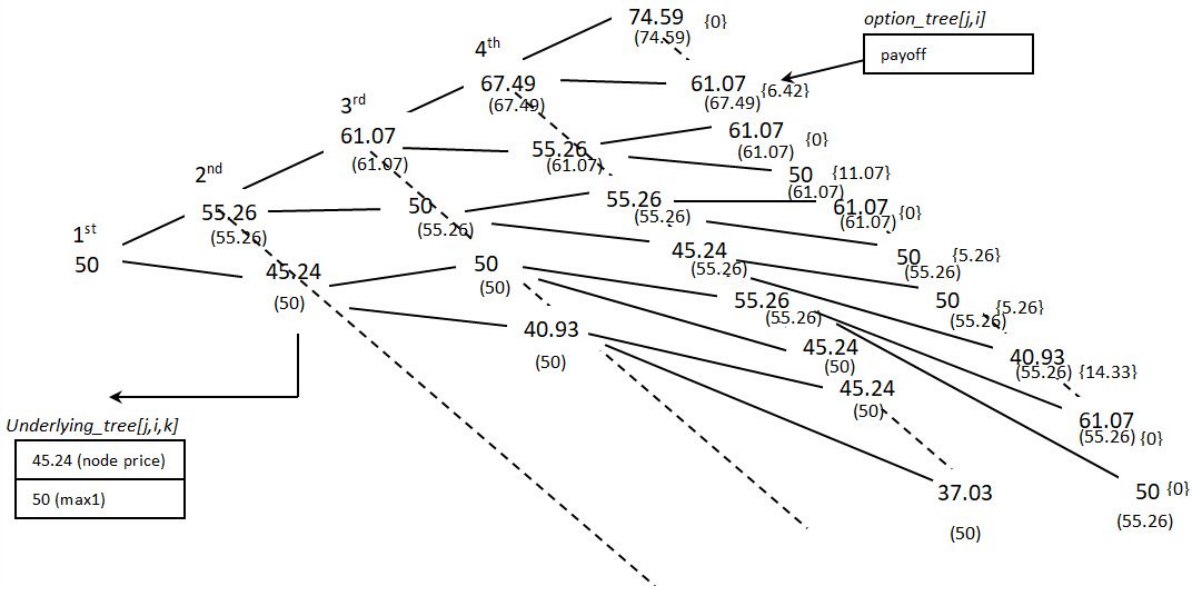
$$pu^2S^2 + (1-p)d^2S^2 = e^{(2r+\sigma^2).\Delta t}.S^2$$

The first moment constraint leads to $p= (e^{r\Delta t} - d)/(u-d)$ and substituting into the second moment constraint leads to $e^{r\Delta t}(u+d) - du = e^{(2r+\sigma^2).\Delta t}$. Cox, Ross and Rubinstein assumed $u= 1/d$ and this involved solving quadratic solutions which eventually leads to $u=e^{\sigma\sqrt{\Delta t}}$ and $d=e^{-\sigma\sqrt{\Delta t}}$ using first order Δt approximation.

Non-recombined tree with lookback features

The tree will starts with initial stock price on the left and branch out to j^{th} row with upper or lower movement as it proceeds to the i^{th} time step to the right. For each node, the array *underlying_tree*[j,i,0] stores the tree node price while *underlying_tree*[j,i,1] stores the maximum (or minimum) value by picking the larger (or smaller) value between current i^{th} node price or previous $(i-1)^{th}$ node maximum (or minimum) value to facilitate lookback features as it propagates down the single path. For floating strike put option, the payoff at the terminal time node is calculated as $max(S_{max} – S_T,0)$ for both upper and lower branch and then discounted back using $e^{-r\Delta t}(p.\{upper\ payoff\} + (1-p).\{lower\ payoff\})$ and stored in array *option_tree*[j,i]. For floating strike call option, the payoff at terminal node is $max(S_T - S_{min},0)$ instead. For American option, the larger value between the calculated discounted payoff or immediate payoff from early exercise on that node (i.e. $S_{max} - S_t$) will be stored. This payoff procedures repeats for each i^{th} time steps by using the $(i+1)^{th}$ time step payoff and discounted backward until it reaches initial node where *option_tree*[0,0] will provide the option value. This option valuation differs for *fixing strike* put or call option only at the calculation of payoff at terminal time node where the payoff is

max(K – S_{min},0) or *max*(S_{max}-K,0) respectively. The problem with non-recombined tree is that the number of nodes at each *i*th time step **grows exponentially** (i.e. 2^i) and this pose issue for long computation time and limited memory resources (i.e. for i=20, 2^{20} is 1 million nodes)

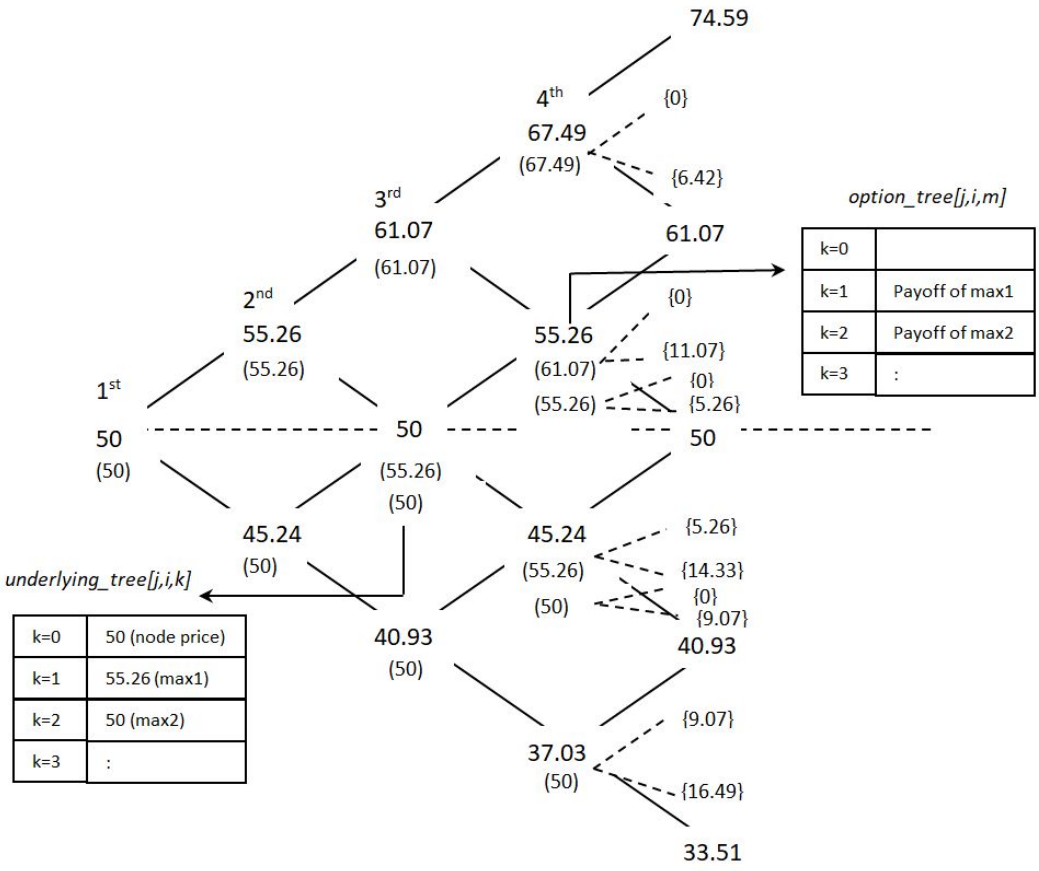


Recombined tree with lookback features

Recombined tree recombine tree nodes to reduce redundancy but the operation is more involving. In order to facilitate lookback features and record the paths, each node will store **a list of maximum or minimum values** depending whether it is a put option or call option. For *floating strike put option*, the maximum value will be stored since the payoff is *max*(S_{max} – S_T,0). Let's demonstrate a tree with N=4 time steps as below (assuming risk free rate r=0.1, sigma=0.4). The first node will start with initial stock price as the first maximum value in the maximum list since this is the largest stock values to-date. Proceeding to *i*th nodes on the right, if any maximum list values in the previous (*i-1*)th upper or lower nodes is ***greater or equal*** to *i*th node price, it is transferred to the *i*th node maximum list. Otherwise if none were transferred, it signify that the *i*th node price is higher (compared to *previous maximum values*) that the current node price will be the new maximum in the *i*th node list. The array *underlying_tree*[j,i,0] stores the (j,i) tree node price while *underlying_tree*[j,i,k] for k≥1 stores each maximum value of the list of node (j,i). The maximum values in the list of each node is sorted in descending order for consistency.

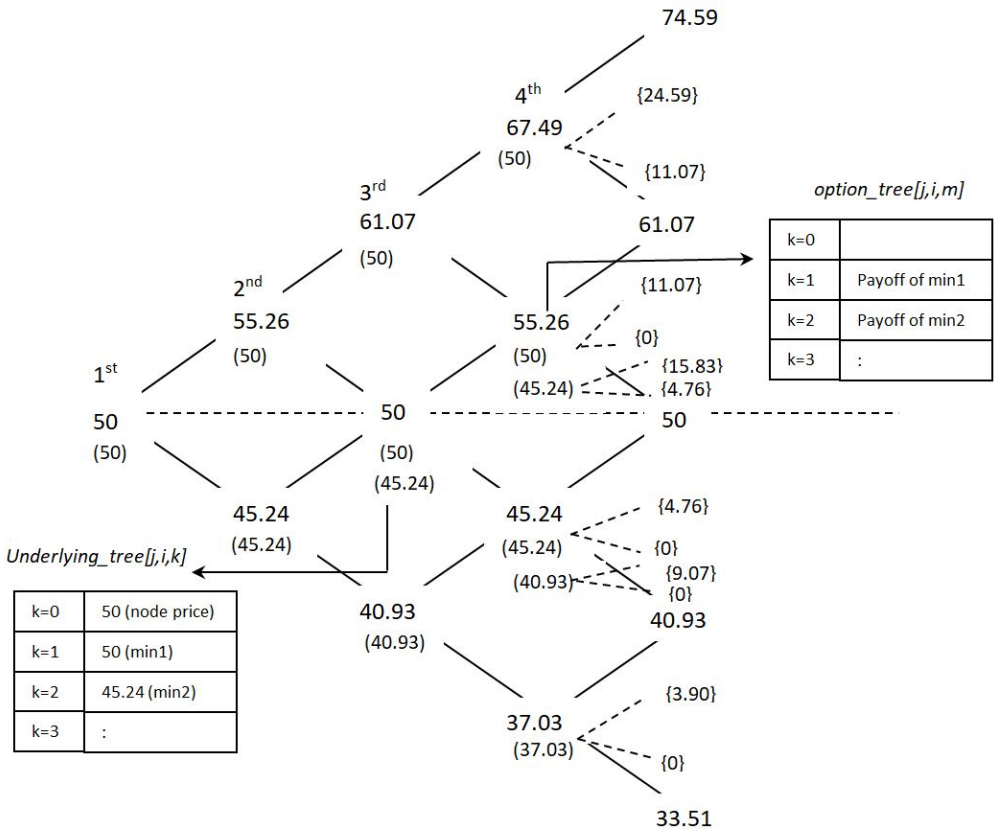
For **each *k*th maximum value** in the list for 4th time step, the payoff is calculated such that if $uS_{max,k} = S_T$ or $S_{max,k} = S_{max,k+1}$ then the upper branch put payoff is {$uS_{max,k}$-S_T=0} or {$S_{max,k+1}$-S_T} and lower branch put payoff is {$S_{max,k}$ - S_T}. The payoff for 4th time step on each *k*th maximum value of each node is calculated and discounted back using $e^{-r\Delta t}$(*p.{upper payoff}* + *(1-p).*{*lower payoff*}) and stored in *option_tree*[j,i,k] for k≥1. For American option, the calculated payoff is compared with the immediate payoff if the option is early exercised (i.e. $S_{max,k}$ - S_t),

whichever larger will be stored. The *k*th maximum value of 4th time step is then traced one step back by **finding a match in maximum values** of 3rd time step from previous (upper or lower) node ($uS_{prev\_max} = S_{cur\_max}$, or $S_{prev\_max} = S_{cur\_max}$). If a maximum values pair match is found (i.e. evidence of same path travelled) , the payoffs from 4th steps is used to calculate payoff for 3rd step in a similar way using $e^{-r\Delta t}(p.\{upper\ payoff\} + (1-p).\{lower\ payoff\})$. This repeats for each maximum value until backward traced to the initial point (first maximum value) in 1st time step to value the lookback option, i,e.*option_tree*[0,0,1] should provide the option value. Despite storing a list of maximum values, notice that the number of nodes at *i*th time step is only N, significant saving in computation time & resource compared to non-recombined tree. This tree is as shown below:



For *floating strike call option*, the **minimum value** will be stored since the payoff is $max(S_T - S_{min} -,0)$. The operation logic is quite similar but it has to be handled in a vertically flipped sense for minimum value. The first node will start off with initial stock price as the first minimum value in the minimum list (stored in *underlying_tree*[j,i,k]) and when proceeding to *i*th nodes, any minimum list values in the previous (*i-1*)th nodes will be transferred to *i*th node if it is ***smaller or equal*** to *i*th node price. The node price will become the new minimum in the *i*th node list if none were transferred. **For each *k*th minimum value** in the list for 4th time step, the payoff for lower branch is $\{S_T - dS_{min,k} = 0\}$ or $\{S_T - S_{min,k+1}\}$ if $dS_{min,k} = S_T$

or $S_{min,k} = S_{min,k+1}$ and for upper branch is $\{S_T - S_{min,k}\}$. The payoff for 4th time step is calculated and discounted back similarly using $e^{-r\Delta t}(p.\{upper\ payoff\} + (1-p).\{lower\ payoff\})$ and stored in *option_tree*[j,i,k]. For American option, the immediate payoff for early exercised is also considered (i.e. $S_t - S_{min,k}$) and whichever larger payoff was retained. The payoff from 4th steps is used to calculate payoff for 3rd step when *k*th minimum values of 4th time step **matches** that of 3rd time step (i.e. $dS_{prev\_min} = S_{cur\_min}$, or $S_{prev\_min} = S_{cur\_min}$). This repeats similarly for each minimum value traced backwardly until the initial first minimum value (i,e.*option_tree*[0,0,1]) that provides the option value.



For *fixed strike call option*, the maximum value will be stored since the payoff is $max(S_{max} - K, 0)$. Therefore operation logic is quite similar to the floating strike put option except that the payoff on the 4th time step is the difference between the corresponding S_{max} and strike value K instead of terminal stock price S_T. Whereas in the case of *fixed strike put option*, the minimum value will be stored since the payoff is $max(K - S_{min}, 0)$ and operation logic is similar to the floating strike call option except that the payoff on the 4th time step is difference between the strike value K (instead of S_T) and the corresponding S_{min}. The backward induction operation for discounting payoff until the initial maximum or minimum value by finding the match between previous and current maximum or minimum values pair is the same as described before. Payoff from early exercise at each node is considered for American lookback option.

Trinomial tree extension

The recombined binomial tree with lookback features has been extended to trinomial tree with exactly the same principal such as storing maximum or minimum values at each node propagated from previous upper/lower node, backward discounting through maximum or minimum value pair matching of previous and current node. The difference is that now there is an extra *middle node* branching out from/recombined to other nodes and the up and down movement multiplier are given by $u=e^{\sigma\sqrt{3\Delta t}}$, $d=e^{-\sigma\sqrt{3\Delta t}}$ with the probability of up, middle and down movement given by the following (Hull, 2015):

$$p_d = -\sqrt{\frac{\Delta t}{12\sigma^2}}\left(r-q-\frac{\sigma^2}{2}\right)+\frac{1}{6} \;\; ; \;\; p_m=\frac{2}{3} \;\; ; \;\; p_u = \sqrt{\frac{\Delta t}{12\sigma^2}}\left(r-q-\frac{\sigma^2}{2}\right)+\frac{1}{6}$$

The payoff backward discounting will use the equation $e^{-r\Delta t}(p_u\{\textit{upper payoff}\} + p_m\{\textit{middle payoff}\} (p_d.\{\textit{lower payoff}\})$ with extra p_m associated with *middle node* payoff. The trinomial tree results is used for further comparison analysis.

Theoretical formula

The valuation formula for *floating strike lookback* put option (European) has been given as the following in a Black-Schole look kind of equation (Hull, 2015):

$$p_{fl} = S_{max}e^{-rT}\left[N(b_1)-\frac{\sigma^2}{2(r-q)}\,e^{Y_2}N(-b_3)\right]+S_0e^{-qT}\frac{\sigma^2}{2(r-q)}\,N(-b_2)-S_0e^{-qT}N(b_2)$$

Where

$$b_1=\frac{ln(S_{max}/S_0)+(-r+q+\sigma^2/2)T}{\sigma\sqrt{T}}\;;\;\; b_2=b_1-\sigma\sqrt{T}\;;\; b_3=\frac{ln(S_{max}/S_0)+(r-q-\sigma^2/2)T}{\sigma\sqrt{T}}\;;\; Y_2=\frac{2(r-q-\sigma^2/2)ln(S_{max}/S_0)}{\sigma^2}$$

and $S_{max}=S_0$ when pricing (originated) at t_0.

For *floating strike call option* (European) valuation formula is given by:

$$c_{fl} = S_0e^{-qT}N(a_1)\;-S_0\,e^{-qT}\frac{\sigma^2}{2(r-q)}\,N(-a_1)-S_{min}e^{-rT}\left[N(a_2)-\frac{\sigma^2}{2(r-q)}\,e^{Y_1}N(-a_3)\right]$$

$$a_1 = \frac{ln(S_0/S_{min})+(r-q+\sigma^2/2)T}{\sigma\sqrt{T}}\ ;\ \ a_2\ = a_1 - \sigma\sqrt{T}\ ;\ a_3\ = \frac{ln(S_0/S_{min})+(-r+q+\sigma^2/2)T}{\sigma\sqrt{T}}\ \ ;\ Y_1 = -\frac{2(r-q-\sigma^2/2)ln(S_0/S_{min})}{\sigma^2}$$

and $S_{min}=S_0$ when pricing (originated) at t_0.

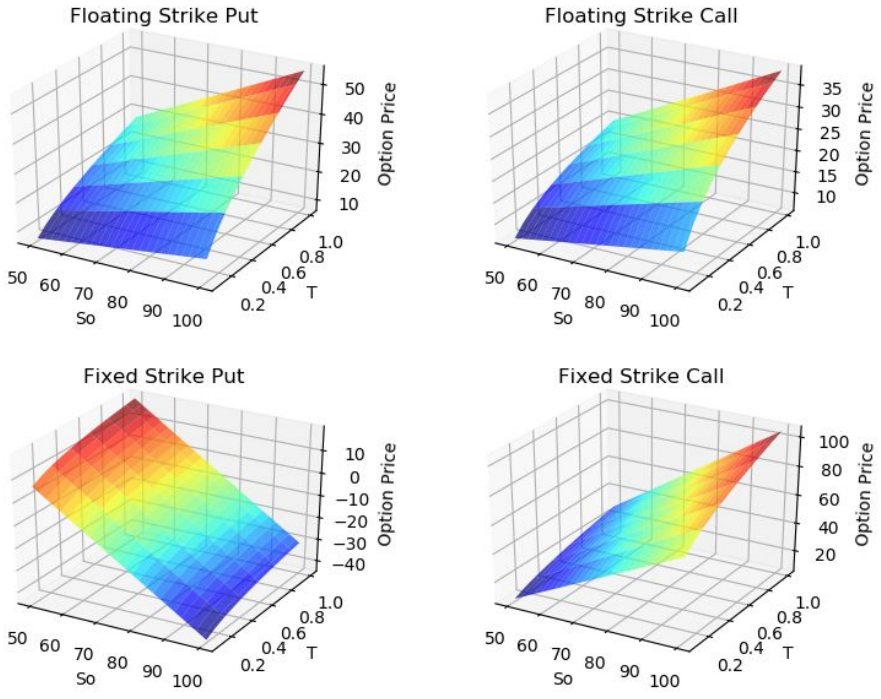
The equation floating lookbacks can be modified to price fixed lookback options via put call parity. John Hull showed that valuation for *fixed strike put option* and *fixed strike call option* are given respectively by:

$$c_{fix} = p_{fl}^{*} + S_0 e^{-qT} - Ke^{-rT}$$

$$p_{fix} = c_{fl}^{*} + Ke^{-rT} - S_0 e^{-qT}$$

where in each floating strike equivalent p_{fl}^{*} and p_{fix}^{*} the S_{max} and S_{min} is replaced by max(S_{max},K) and min(S_{min},K) respectively.

The theoretical valuation of lookback option prices are plotted as below (with K=50,r=0.0012,σ=0.58):



Valuation results analysis

Since lookback options are traded only in OTC market which is not easily accessible to the general public through the stock & exchanges, there is no available market price of lookback options for comparison. Hence, we can only perform results analysis with theoretical results from formula.

Based on the real parameters that were estimated previously from AMD stock price (volatility) and T-bill risk free interest rate from historical data, the following parameters are used to price our lookback options in the *Python* programming software.

S_o=K=50, σ=0.58, r =0.0012, T=0.25 (or 3month)

The underlying tree (non-recombined) vs. underlying tree (recombined binomial) when N=4 for *floating strike put option* is shown below:

```
array([[50.  , 57.8 , 66.82, 77.25, 89.3 ],
       [ 0.  , 43.25, 50.  , 57.8 , 66.82],
       [ 0.  ,  0.  , 50.  , 57.8 , 66.82],
       [ 0.  ,  0.  , 37.41, 43.25, 50.  ],
       [ 0.  ,  0.  ,  0.  , 57.8 , 66.82],
       [ 0.  ,  0.  ,  0.  , 43.25, 50.  ],
       [ 0.  ,  0.  ,  0.  , 43.25, 50.  ],
       [ 0.  ,  0.  ,  0.  , 32.36, 37.41],
       [ 0.  ,  0.  ,  0.  ,  0.  , 66.82],
       [ 0.  ,  0.  ,  0.  ,  0.  , 50.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  , 50.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  , 37.41],
       [ 0.  ,  0.  ,  0.  ,  0.  , 50.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  , 37.41],
       [ 0.  ,  0.  ,  0.  ,  0.  , 37.41],
       [ 0.  ,  0.  ,  0.  ,  0.  , 27.99]])
```

```
array([[50.  , 57.8 , 66.82, 77.25, 89.3 ],
       [ 0.  , 43.25, 50.  , 57.8 , 66.82],
       [ 0.  ,  0.  , 37.41, 43.25, 50.  ],
       [ 0.  ,  0.  ,  0.  , 32.36, 37.41],
       [ 0.  ,  0.  ,  0.  ,  0.  , 27.99]])
```

The lookback option tree for non-recombined and recombined binomial tree when N=4 shows that both the option price is 9.01 (i.e. value at left top most position). However, the theoretical floating strike put option based on Hull's formula was found to be 12.65.

```
array([[ 9.01,  8.66,  7.42,  5.59,  0.  ],
       [ 0.  ,  9.32,  9.73,  9.01, 10.43],
       [ 0.  ,  0.  ,  5.56,  4.18,  0.  ],
       [ 0.  ,  0.  , 12.58, 14.55, 16.82],
       [ 0.  ,  0.  ,  0.  ,  4.18,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  6.75,  7.8 ],
       [ 0.  ,  0.  ,  0.  ,  6.75,  7.8 ],
       [ 0.  ,  0.  ,  0.  , 17.64, 20.39],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  7.8 ],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  , 12.59],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  , 12.59],
       [ 0.  ,  0.  ,  0.  ,  0.  , 12.59],
       [ 0.  ,  0.  ,  0.  ,  0.  , 22.01]])
```

```
array([[ 9.01,  8.66,  7.42,  5.59,  0.  ],
       [ 0.  ,  9.32,  9.73,  9.01,  0.  ],
       [ 0.  ,  0.  , 12.58, 14.55,  0.  ],
       [ 0.  ,  0.  ,  0.  , 17.64,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ]])
```

As N is increased to N=20, both non-recombined tree and recombined binomial tree gives option value of 10.81 which is closer to theoretical value. The **non-recombined tree fails and can no longer go beyond N=20** due to *memory resource limitation error*. This is the downside of non-recombined tree. As N is increased further to N=50 or N=100 for recombined binomial tree, the option value becomes 11.45 and 11.78 respectively which is much closer to theoretical value.

The option pricing simulations results for all type of lookback option with N=100 and N=130 are shown below:

	Floating Strike Put Option (recomb. Binomial tree) ***European***	**Floating Strike Put Option** (recomb. Binomial tree) ***American***	**Floating Strike Put Option** (recomb. Trinomial tree) ***European***	**Floating Strike Put Option** (recomb. Trinomial tree) ***American***	Theoretical Floating Strike Put Option
N=100	11.78	11.78	11.18	11.18	12.65
N=130	11.89	11.89	11.34	11.34	
	Floating Strike Call Option (recomb. Binomial tree) ***European***	**Floating Strike Call Option** (recomb. Binomial tree) ***American***	**Floating Strike Call Option** (recomb. Trinomial tree) ***European***	**Floating Strike Call Option** (recomb. Trinomial tree) ***American***	Theoretical Floating Strike Call Option
N=100	10.01	10.01	9.61	9.61	10.56
N=130	10.08	10.08	9.72	9.72	
	Fixed Strike Put Option (recomb. Binomial tree) ***European***	**Fixed Strike Put Option** (recomb. Binomial tree) ***American***	**Fixed Strike Put Option** (recomb. Trinomial tree) ***European***	**Fixed Strike Put Option** (recomb. Trinomial tree) ***American***	Theoretical Fixed Strike Put Option
N=100	10.0	10.0	9.59	9.60	10.55
N=130	10.06	10.06	9.70	9.70	
	Fixed Strike Call Option (recomb. Binomial tree) **European**	**Fixed Strike Call Option** (recomb. Binomial tree) **American**	**Fixed Strike Call Option** (recomb. Trinomial tree) **European**	**Fixed Strike Call Option** (recomb. Trinomial tree) **American**	Theoretical Fixed Strike Call Option
N=100	11.79	11.79	11.19	11.19	12.67
N=130	11.91	11.91	11.36	11.36	

The table above shows that the European lookback binomial/trinomial tree option pricing simulation results are generally close to theoretical results. The percentage differences is small around 5%, ie. for example the (12.65-11.85)/12.65 = 6% for floating

strike put option and (12.65-11.89)/12.65 = 6% for fixed strike call option. Also, increasing N=100 to N=130 definitely improved the results overall at the expense of longer computation time.

The European fixed strike lookback call or put option is priced higher and more expensive (~ doubled) than the equivalent normal call or put option (i.e. equivalent normal call option=5.77 and normal put option=5.75) due to potential higher payoff having the flexibility of picking S_{max} or S_{min} instead of S_T. Similar to normal options, the American lookback options are also slightly higher or equal to the price of European lookback options but the difference is almost non-existent and could look like a good bargain/better choice compared to European lookback option considering that lookback is already expensive.

Surprisingly, trinomial tree results are generally smaller compared to binomial tree results and seems biased downward. For example, the binomial tree American floating put option price of 11.89 compared to American trinomial tree of 11.34. We can use trinomial tree European price as a *control variate* to improve trinomial tree American price, by comparing European option price with theoretical price and assume the same bias appears for both American and European options. The *improved* American trinomial price = trinomial American price + (theoretical price - trinomial European price), i.e. for floating put option improved American price = 11.34 + (12.65 - 11.34) = 12.65.

Delta hedging Analysis:

The *delta* of a stock option (one of the Greeks) is defined as the ratio of the change in price of stock option to the change in the price of the underlying stock. Let's look at delta of a floating strike lookback put option.

There is another *floating strike lookback* put option valuation formula given by Shreve (2004):

$$V(t,x,y) = \left(1 + \frac{\sigma^2}{2r}\right) xN(\delta_+(T,z)) + e^{-rT} yN(-\delta_-(T,z)) - \frac{\sigma^2}{2r} e^{-rT} z^{2r/\sigma^2} xN(-\delta_-(T,z)) - x$$

where z=x/y, x=S(t), y=S_{max}(t), S_{max}=S_0 or z=1 when pricing (originated) at t_0 and

$$\delta_\pm(T,k) = \frac{1}{\sigma\sqrt{T}} \left[log(k) + (r \pm \tfrac{1}{2}\sigma^2)T \right]$$

The above equation is actually the same and consistent with the floating strike lookback valuation formula found in Hull mentioned previously.

Using definition of delta, it is calculated by differentiating the option price in the above equation with stock price, x. Hence, it was found that

$$\Delta = V_x(t,x,y) = \left(1+\frac{\sigma^2}{2r}\right)N(\delta_+(T,z)) + (1-\frac{\sigma^2}{2r})e^{-rT}z^{2r/\sigma^2}N(-\delta_-(T,z^{-1})) - 1$$

The above is the *theoretical delta equation* for *floating strike put option*.

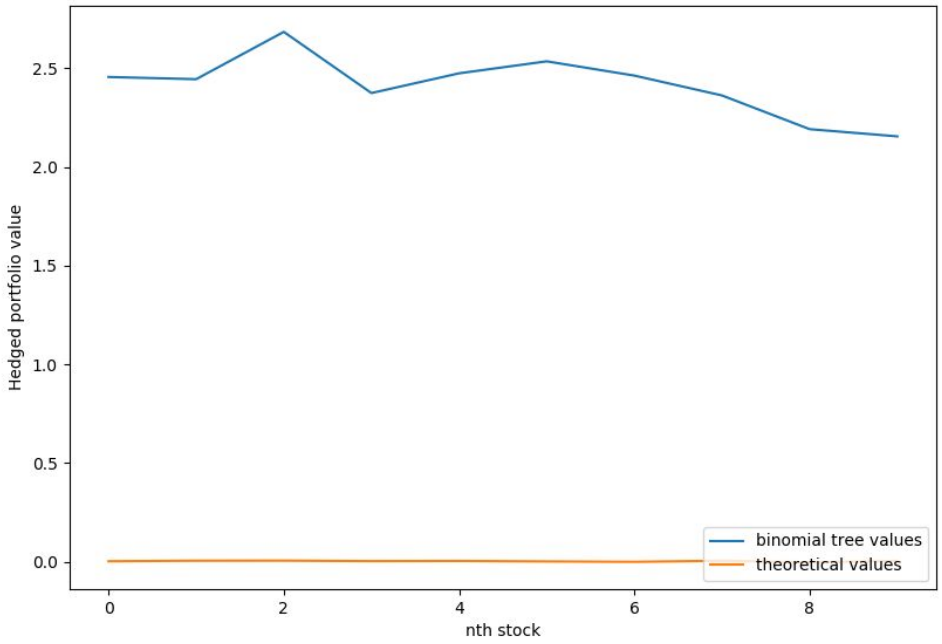
Delta is also the number of units of the stock one should hold for each option in order to create a riskless or delta neutral portfolio/position, known as *delta hedging*. Thus, a delta-hedged portfolio should have a relatively constant value P - Δ. S where P is the option price and S is the stock price. This is because as stock price increase to S+δS, option price will increase by P+δP, but since Δ=δP/δS and hence -Δ.δS=-δP will offset +δP. If delta is positive, hedging for a long position in an option involves maintaining a short position of stock and vice-versa.

In order to test out the delta-hedged portfolio (eg. for floating strike put option, T=3M), 10 daily stock prices have been generated using GBM Black Schole as below. The option prices and delta of binomial tree is based on S_o=50, K=50, r=0.0012,σ=0.58, N=100. Delta of binomial tree is determined from the value difference between 2 closest adjacent nodes to S_o = (P(S+ΔS)-P(S))/ΔS.

Stock Price from BS, S	Option Price, P from binomial tree	Delta from binomial tree, Δ	Theoretical Option Price	Theoretical Delta,Δ	Hedge portfolio, P- Δ.S (bin. Tree value)	Hedge portfolio, P- Δ.S (theoretical)
51.33	12.10	(12.38-11.82)/(52.84-49.86)=0.1879	12.99	0.253	2.4551	0.00351
52.07	12.27	(12.56-11.99)/(53.60-50.58)=0.1887	13.18	0.253	2.444	0.00629
54.44	12.84	(13.14-12.55)/(56.04-52.88)=0.1867	13.78	0.253	2.6835	0.00668
51.09	12.04	(12.32-11.76)/(52.59-49.63)=0.1892	12.93	0.253	2.3738	0.00423
49.98	11.78	(12.05-11.51)/(51.45-48.55)=0.1862	12.65	0.253	2.4737	0.00506
51.61	12.17	(12.45-11.89)/(53.13-50.13)=0.1867	13.06	0.253	2.5344	0.00267
50.63	11.93	(12.21-11.66)/(52.12-49.18)=0.1870	12.81	0.253	2.4622	0.00061
48.79	11.50	(11.77-11.24)/(50.23-47.40)=0.1873	12.35	0.253	2.3616	0.00613
45.74	10.79	(11.04-10.54)/(47.09-44.43)=0.1880	11.57	0.253	2.191	-0.00222

45.71	10.78	(11.03-10.53)/(47.05-44.40)=0.1887	11.57	0.253	2.1545	0.00537

The delta calculated from binomial tree (0.187) appeared to be smaller than the theoretical value (0.253), but can be improved with increased N (eg. N=150 yield delta of =0.189) at the expense of longer computation time. The hedged portfolio value is relative constant as seen in the table and the plot below as well. Hedged portfolio value calculated from binomial tree seems higher than theoretical value due to smaller delta. Delta hedged creates a delta neutral position for a relatively short period of time only and the position has to be adjusted/rebalanced periodically.



Discussion & Conclusion:

The tree method for pricing lookback option looks satisfactory and relatively good when compared with the theoretical valuation formula, with some percentage difference of around 5%. Increasing the number of nodes may increase the accuracy of the option pricing but comes with the expense of longer computation. Although the non-recombined tree method failed for larger N values due to memory resource limitation, it served as the foundation to compare simulation results and generate further ideas in recombined tree. The recombined binomial tree method was extended to trinomial tree with the initial idea that it may increase the accuracy, but the simulation results proved that this is not the case (the

results were slightly biased lower although close to binomial tree results). Perhaps the extra middle path/node of trinomial tree did not contribute much considering that travelling through the middle path does not change the maximum or minimum values retained previously in the computation of lookback option. The delta hedged portfolio result also showed a relatively flat curve and hence riskless profile achieved. Much of the work of this project were focused on generating the algorithm in programming and a few failed attempts were made with recombined tree but resolved eventually. The extension to trinomial tree also consumed much time and it was not initially planned for. In order to get accuracy neared to theoretical value, other approach besides binomial/trinomial tree could be used such as finite difference method or Monte Carlo Simulation could be attempted for future works. Other binomial tree method such as Jarrow-Rudd method can also be attempted besides Cox-Ross-Rubinstein method. It would be interesting to compare the results with these other methods as the current binomial/trinomial tree method hits a ceiling in terms of accuracy. Other delta formulas such as for lookback fixed strike could be researched and more comparison results could be made as well in hedging analysis part. We could also attempt on the exotic pricing tools found on bloomberg that may aid in other comparisons.

References:

Hull, John C. (2015). *Options, Futures and other derivatives, 9th ed. US: Pearson*

London, Justin (2005). *Modeling Derivatives in C++, 1st ed. John Wiley & Sons.*

OptionsTradingpedia (2020). *Exotic Options: Lookback Options*. http://www.optiontradingpedia.com/lookback_options.htm (Accessed: 9 July 2020)

Semantic (2018) . *Analytic Solutions and Monte Carlo Simulation for Lookback Options II. Pricing Lookback Options with the Binomial Tree III. Finite Difference Method for Path Dependent Options* https://www.semanticscholar.org/paper/I.-Analytic-Solutions-and-Monte-Carlo-Simulation/ed0324a36d4bd0e1cd3e395ea8bc0f4a6d3078dc (Accessed: 1 Aug 2020)

Sharoy, Augustine S. (2016). *Pricing and Hedging Lookback Options Using Black-Scholes in Borsa Instanbul [Masters Thesis], Middle East Tecnical University*

Shreve, Steven E. (2004). *Stochastic Calculus for Finance II (Continuous Time), 1st ed. Springer Finance*.

Python code:

Example usage:

option1 = lookback_option(call_or_put=0, maturity = 0.25, spot_price = 50, sigma = 0.58, risk_free_rate = 0.0012, strike_price = 50, dividends = 0)

option1.floating_underlying_price_tree(4)[:,:,0] #non-recombined binomial tree

option1.floating_EU_option_tree(4) #non-recombined binomial tree

option1.floating_underlying_price_tree_0(50)[:,:,0] #recombined binomial tree, N=50

option1.floating_US_option_tree_0(50) [:,:,1] #recombined binomial tree, N=50

option1.fixed_underlying_price_tree_1(100)[0,:,0] #recombined trinomial tree, N=100

option1.fixed_EU_option_tree_1(100) [100,:,1] #recombined trinomial tree, N=100

Lookback.py

```
import numpy as np
from scipy.stats import norm

class lookback_option:
    '''define a class to store common option data and function definition'''
    ## if call_or_put == 1, means call option, else, put option.

    def __init__(self, call_or_put, maturity, spot_price, sigma, risk_free_rate, strike_price, dividends):
        self.I = call_or_put
        self.T = maturity
        self.S0 = spot_price
        self.sigma = sigma
        self.r = risk_free_rate
        self.K = strike_price
        self.q = dividends

    ## ------------------------------------ BT floating ---------------------------------------- ##
    ## unlike the underlying tree created before, this time we generate the tree by (N+1) lists
    ## i(0, N) list has 2**i nodes. each pair nodes are generated by last one node in last layer.
    ## the nodes which have the same underlying price are not merge to one nodes (important)
    ## each node has two value to store, the currenct price and the max spot price before.
    def floating_underlying_price_tree(self, N):
        '''input: the option and the number of periods'''
        '''output: the binomial tree of the underlying price (N+1 * 2**(N) * 2 array)'''

        ## parameter initializing
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        underlying_tree = np.zeros([2**N,N+1,2])


        underlying_tree[0,0,0] = self.S0
        underlying_tree[0,0,1] = self.S0


        if self.I == 0 :
```

```
        for i in range(1,N+1):
            for j in range(2**i):
                if j % 2 == 0:
                    underlying_tree[j,i,0] = round(underlying_tree[int(j/2),i-1,0]*u,2)
                    underlying_tree[j,i,1] = max(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])
                else:
                    underlying_tree[j,i,0] = round(underlying_tree[int((j-1)/2),i-1,0]*d,2)
                    underlying_tree[j,i,1] = underlying_tree[int((j-1)/2),i-1,1]

    else:
        for i in range(1,N+1):
            for j in range(2**i):
                if j % 2 == 0:
                    underlying_tree[j,i,0] = round(underlying_tree[int(j/2),i-1,0]*u,2)
                    underlying_tree[j,i,1] = min(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])
                else:
                    underlying_tree[j,i,0] = round(underlying_tree[int((j-1)/2),i-1,0]*d,2)
                    underlying_tree[j,i,1] = min(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])


    return underlying_tree

# <--- TRIAL
def floating_underlying_price_tree_0(self, N):

    ## parameter initializing
    dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
    d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
    print('p ',p,'u ',u,'dt ',dt)

    # compute list size
    list_size=int(np.floor(N)/2+1)
    temp_list=np.zeros(list_size)
    print('list size',list_size)

    # underlying_tree_0[j,i,0] index 0 store underlying price
    # underlying tree_0[j,i,1:] index 1 onwards store max or min
    underlying_tree_0= np.zeros([N+1,N+1,list_size+1])
    underlying_tree_0[0,0,0] = self.S0
    underlying_tree_0[0,0,1]= self.S0    # first max or min list item
    underlying_tree_0[0,0,2:]=0

    # check if element exist in array
    def checkif_exist(array_,element):
        exist=0
        for n in range(len(array_)):
            if array_[n]==element:
                exist=1
        return exist

    # check the next available index of empty slots in array
    def fnext_index(array_):
        index_=-1
        for n in range(len(array_)):
            if array_[n]==0:
                index_=n
                break
        return index_

    # put option, max list
    if self.I == 0:
        for i in range(1,N+1):
            for j in range(i+1):
                # generate underlying price for tree
                if j==0:
                    underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j),i-1,0]*u,2)
```

```
                else:
                    underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j-1),i-1,0]*d,2)
                # if not lowest branch
                if j!=i:
                    cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                    cnt_old=cnt
                    # add max list items from prev lower left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i-1,k]!=0 and underlying_tree_0[j,i-1,k]>=underlying_tree_0[j,i,0]:
                            if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j,i-1,k])==0:
                                underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i-1,k]
                                cnt+=1
                    # add cur price to list if none was added from prev lower left list (i.e. highest price)
                    if cnt==cnt_old:
                        underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i,0]
                        cnt+=1
                # if not highest branch
                if j>=1:
                    cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                    # add max list items from prev upper left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j-1,i-1,k]!=0 and
underlying_tree_0[j-1,i-1,k]>=underlying_tree_0[j,i,0]:
                            if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j-1,i-1,k])==0:
                                underlying_tree_0[j,i,cnt]=underlying_tree_0[j-1,i-1,k]
                                cnt+=1
        # call option, min list
        else:
            for i in range(1,N+1):
                for j in range(i+1):
                    # generate underlying price for tree
                    if j==0:
                        underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j),i-1,0]*u,2)
                    else:
                        underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j-1),i-1,0]*d,2)
                    # if not lowest branch
                    if j!=i:
                        cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                        # add min list items from prev lower left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i-1,k]!=0 and underlying_tree_0[j,i-1,k]<=underlying_tree_0[j,i,0]:
                                if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j,i-1,k])==0:
                                    underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i-1,k]
                                    cnt+=1
                    # if not highest branch
                    if j>=1:
                        cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                        cnt_old=cnt
                        # add min list items from prev upper left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j-1,i-1,k]!=0 and
underlying_tree_0[j-1,i-1,k]<=underlying_tree_0[j,i,0]:
                                if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j-1,i-1,k])==0:
                                    underlying_tree_0[j,i,cnt]=underlying_tree_0[j-1,i-1,k]
                                    cnt+=1
                        # add cur price to list if none was added from prev upper left list (i.e. lowest price)
                        if cnt==cnt_old and cnt==1:
                            underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i,0]
                            cnt+=1
        # sort the list at each node from high to low
        for i in range(1,N+1):
                for j in range(i+1):
                    temp_list=sorted(underlying_tree_0[j,i,1:],reverse=True)
                    underlying_tree_0[j,i,1:]=temp_list

        return underlying_tree_0
```

```
    #--->

    # <--- TRIAL
    def floating_underlying_price_tree_1(self, N):

        ## parameter initializing
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(3*self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        p_d = - (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) + 1/6
        p_m = 2/3; p_u = (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) +
1/6

        print('p ',p,'u ',u,'dt ',dt)
        print('p_u',p_u,'p_m',p_m,'p_d',p_d)

        # compute list size
        list_size=int(np.floor(N)/2+1)
        temp_list=np.zeros(list_size)
        print('list size',list_size)

        # underlying_tree_0[j,i,0] index 0 store underlying price
        # underlying tree_0[j,i,1:] index 1 onwards store max or min
        underlying_tree_1= np.zeros([2*N+1,N+1,list_size+1])
        underlying_tree_1[N,:,0] = self.S0
        underlying_tree_1[N,0,1]= self.S0    # first max or min list item
        underlying_tree_1[0,0,2:]=0

        # check if element exist in array
        def checkif_exist(array_,element):
            exist=0
            for n in range(len(array_)):
                if array_[n]==element:
                    exist=1
            return exist

        # check the next available index of empty slots in array
        def fnext_index(array_):
            index_=-1
            for n in range(len(array_)):
                if array_[n]==0:
                    index_=n
                    break
            return index_

        # generate underlying price for tree first
        for i in range(1,N+1):
            for j in range(1,i+1):
                underlying_tree_1[N-j,i,0] = round(underlying_tree_1[N+1-j, i-1,0]*u,2)
                underlying_tree_1[N+j,i,0] = round(underlying_tree_1[N-1+j, i-1,0]*d,2)

        # put option, max list
        if self.I == 0:
            for i in range(1,N+1):
                for j in range(N-i,N+i+1):
                    cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                    cnt_old=cnt
                    # if not lowest branch and 2nd lowest branch
                    if j<N+i-1:
                        cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                        # add max list items from prev lower left branch if >= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j+1,i-1,k]!=0 and
underlying_tree_1[j+1,i-1,k]>=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j+1,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j+1,i-1,k]
                                    cnt+=1
```

```
                # if not the lowest branch or highest branch
                if j!=N+i or j!=N-i:
                    # add max list items from prev middle left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j,i-1,k]!=0 and underlying_tree_1[j,i-1,k]>=underlying_tree_1[j,i,0]:
                            if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j,i-1,k])==0:
                                underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i-1,k]
                                cnt+=1
                # if not highest branch and 2nd highest branch
                if j>N-i+1:
                    # add max list items from prev upper left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j-1,i-1,k]!=0 and
underlying_tree_1[j-1,i-1,k]>=underlying_tree_1[j,i,0]:
                            if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j-1,i-1,k])==0:
                                underlying_tree_1[j,i,cnt]=underlying_tree_1[j-1,i-1,k]
                                cnt+=1
                # add cur price to list if none was added from prev nodes list (i.e. highest price)
                if cnt==cnt_old:
                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i,0]
                    cnt+=1
#
        else:
            for i in range(1,N+1):
                for j in range(N-i,N+i+1):
                    cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                    cnt_old=cnt
                    # if not lowest branch and 2nd lowest branch
                    if j<N+i-1:
                        cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                        # add min list items from prev lower left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j+1,i-1,k]!=0 and
underlying_tree_1[j+1,i-1,k]<=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j+1,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j+1,i-1,k]
                                    cnt+=1
                    # if not the lowest branch or highest branch
                    if j!=N+i or j!=N-i:
                        # add min list items from prev middle left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i-1,k]!=0 and underlying_tree_1[j,i-1,k]<=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i-1,k]
                                    cnt+=1
                    # if not highest branch and 2nd highest branch
                    if j>N-i+1:
                        # add min list items from prev upper left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j-1,i-1,k]!=0 and
underlying_tree_1[j-1,i-1,k]<=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j-1,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j-1,i-1,k]
                                    cnt+=1
                    # add cur price to list if none was added from prev nodes list (i.e. lowest price)
                    if cnt==cnt_old and cnt==1:
                        underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i,0]
                        cnt+=1
        # sort the list at each node from high to low
        for i in range(1,N+1):
            for j in range(N-i,N+i+1):
                temp_list=sorted(underlying_tree_1[j,i,1:],reverse=True)
                underlying_tree_1[j,i,1:]=temp_list

        return underlying_tree_1
    #--->
```

```
    def floating_EU_option_tree(self, N):
        underlying_tree = self.floating_underlying_price_tree(N)
        option_tree = np.zeros([2**N, N+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        option_tree[:,N] = np.maximum(np.zeros(2**N),(1-2*self.I)* (underlying_tree[:,N,1] -
underlying_tree[:,N,0]))
        for i in np.arange(N-1,-1,-1):
            for j in np.arange(0,2**i):
                option_tree[j,i]=np.exp(-self.r*dt)*(p*option_tree[2*j,i+1]+(1-p)*option_tree[2*j+1,i+1])

        return np.round(option_tree,2)

    # <--- TRIAL
    def floating_EU_option_tree_0(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_0 = self.floating_underlying_price_tree_0(N)
        option_tree_0 = np.zeros([N+1, N+1,list_size+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        # put option, max list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                        up_payoff=underlying_tree_0[j,i+1,m]-underlying_tree_0[j,i+1,0]
                                    if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                        up_payoff=round(underlying_tree_0[j,i,0]*u,2)-underlying_tree_0[j,i+1,0]
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=underlying_tree_0[j,i,k]-underlying_tree_0[j+1,i+1,0]
                                # store cur i payoff in option_tree_0
                                option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                    # tree nodes before N-1
                    else:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then extract i+1 payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                        up_payoff=option_tree_0[j,i+1,m]
                                    if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                        up_payoff=option_tree_0[j,i+1,m]
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=option_tree_0[j+1,i+1,m]
                                # store cur i payoff in option_tree_0
                                option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
        # call option, min list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
                    if i==N-1:
```

```
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then calc. payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=underlying_tree_0[j+1,i+1,0]-underlying_tree_0[j+1,i+1,m]
                                if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    down_payoff=underlying_tree_0[j+1,i+1,0]-round(underlying_tree_0[j,i,0]*d,2)
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=underlying_tree_0[j,i+1,0]-underlying_tree_0[j,i,k]
                            # store cur i payoff in option_tree_0
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                                if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=option_tree_0[j,i+1,m]
                            # store cur i payoff in option_tree_0
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)

        return np.round(option_tree_0,2)
    # <--- TRIAL
    def floating_EU_option_tree_1(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_1 = self.floating_underlying_price_tree_1(N)
        option_tree_1 = np.zeros([2*N+1, N+1,list_size+1])

        #dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        #d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(3*self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        p_d = - (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) + 1/6
        p_m = 2/3; p_u = (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) +
1/6

        # put option, max list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=underlying_tree_1[j-1,i+1,m]-underlying_tree_1[j-1,i+1,0]
                                    if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        up_payoff=round(underlying_tree_1[j,i,0]*u,2)-underlying_tree_1[j-1,i+1,0]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=underlying_tree_1[j,i+1,m]-underlying_tree_1[j,i+1,0]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=underlying_tree_1[j,i,k]-underlying_tree_1[j+1,i+1,0]
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                    # tree nodes before N-1
```

```
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
                                if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                    mid_payoff=option_tree_1[j,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                    down_payoff=option_tree_1[j+1,i+1,m]
                            # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
        # call option, min list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=underlying_tree_1[j+1,i+1,0]-underlying_tree_1[j+1,i+1,m]
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=underlying_tree_1[j+1,i+1,0]-round(underlying_tree_1[j,i,0]*d,2)
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=underlying_tree_1[j,i+1,0]-underlying_tree_1[j,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=underlying_tree_1[j-1,i+1,0]-underlying_tree_1[j,i,k]
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                    # tree nodes before N-1
                    else:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then extract i+1 payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=option_tree_1[j+1,i+1,m]
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=option_tree_1[j+1,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=option_tree_1[j,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=option_tree_1[j-1,i+1,m]
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
        return np.round(option_tree_1,2)
    #--->


    #--->

    def floating_US_option_tree(self, N):
        underlying_tree = self.floating_underlying_price_tree(N)

        option_tree = np.zeros([2**N, N+1])
```

```
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        option_tree[:,N] = np.maximum(np.zeros(2**N), (1-2*self.I)*(underlying_tree[:,N,1] -
underlying_tree[:,N,0]))
        for i in np.arange(N-1,-1,-1):
            for j in np.arange(0,2**i):
                option_tree[j,i]=np.exp(-self.r*dt)*(p*option_tree[2*j,i+1]+(1-p)*option_tree[2*j+1,i+1])

            option_tree[0:2**i,i]= np.maximum(option_tree[0:2**i, i], (1-2*self.I)*(underlying_tree[0:2**i,i,1] -
underlying_tree[0:2**i,i,0]))

        return np.round(option_tree,2)

    # <--- TRIAL
    def floating_US_option_tree_0(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_0 = self.floating_underlying_price_tree_0(N)
        option_tree_0 = np.zeros([N+1, N+1,list_size+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        # put option, max list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                        up_payoff=underlying_tree_0[j,i+1,m]-underlying_tree_0[j,i+1,0]
                                    if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                        up_payoff=round(underlying_tree_0[j,i,0]*u,2)-underlying_tree_0[j,i+1,0]
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=underlying_tree_0[j,i,k]-underlying_tree_0[j+1,i+1,0]
                                # store cur i payoff in option_tree_0, check larger payoff if early exercise
                                option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)

option_tree_0[j,i,k]=max(underlying_tree_0[j,i,k]-underlying_tree_0[j,i,0],option_tree_0[j,i,k])
                    # tree nodes before N-1
                    else:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then extract i+1 payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                        up_payoff=option_tree_0[j,i+1,m]
                                    if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                        up_payoff=option_tree_0[j,i+1,m]
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=option_tree_0[j+1,i+1,m]
                                # store cur i payoff in option_tree_0, check larger payoff if early exercise
                                option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)

option_tree_0[j,i,k]=max(underlying_tree_0[j,i,k]-underlying_tree_0[j,i,0],option_tree_0[j,i,k])
        # call option, min list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
```

```
                if i==N-1:
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then calc. payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=underlying_tree_0[j+1,i+1,0]-underlying_tree_0[j+1,i+1,m]
                                if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    down_payoff=underlying_tree_0[j+1,i+1,0]-round(underlying_tree_0[j,i,0]*d,2)
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=underlying_tree_0[j,i+1,0]-underlying_tree_0[j,i,k]
                            # store cur i payoff in option_tree_0, check larger payoff if early exercise
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)

option_tree_0[j,i,k]=max(underlying_tree_0[j,i,0]-underlying_tree_0[j,i,k],option_tree_0[j,i,k])
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                                if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=option_tree_0[j,i+1,m]
                            # store cur i payoff in option_tree_0, check larger payoff if early exercise
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)

option_tree_0[j,i,k]=max(underlying_tree_0[j,i,0]-underlying_tree_0[j,i,k],option_tree_0[j,i,k])

        return np.round(option_tree_0,2)
    #--->

    # <--- TRIAL
    def floating_US_option_tree_1(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_1 = self.floating_underlying_price_tree_1(N)
        option_tree_1 = np.zeros([2*N+1, N+1,list_size+1])

        #dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        #d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(3*self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        p_d = - (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) + 1/6
        p_m = 2/3; p_u = (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) +
1/6

        # put option, max list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=underlying_tree_1[j-1,i+1,m]-underlying_tree_1[j-1,i+1,0]
                                    if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        up_payoff=round(underlying_tree_1[j,i,0]*u,2)-underlying_tree_1[j-1,i+1,0]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
```

```
                            mid_payoff=underlying_tree_1[j,i+1,m]-underlying_tree_1[j,i+1,0]
                        if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                            down_payoff=underlying_tree_1[j,i,k]-underlying_tree_1[j+1,i+1,0]
                    # store cur i payoff in option_tree_1, check larger payoff if early exercise

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)

option_tree_1[j,i,k]=max(underlying_tree_1[j,i,k]-underlying_tree_1[j,i,0],option_tree_1[j,i,k])
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
                                if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                    mid_payoff=option_tree_1[j,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                    down_payoff=option_tree_1[j+1,i+1,m]
                            # store cur i payoff in option_tree_1, check larger payoff if early exercise

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)

option_tree_1[j,i,k]=max(underlying_tree_1[j,i,k]-underlying_tree_1[j,i,0],option_tree_1[j,i,k])
        # call option, min list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=underlying_tree_1[j+1,i+1,0]-underlying_tree_1[j+1,i+1,m]
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=underlying_tree_1[j+1,i+1,0]-round(underlying_tree_1[j,i,0]*d,2)
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=underlying_tree_1[j,i+1,0]-underlying_tree_1[j,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=underlying_tree_1[j-1,i+1,0]-underlying_tree_1[j,i,k]
                                # store cur i payoff in option_tree_1, check larger payoff if early exercise

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)

option_tree_1[j,i,k]=max(underlying_tree_1[j,i,0]-underlying_tree_1[j,i,k],option_tree_1[j,i,k])
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                    down_payoff=option_tree_1[j+1,i+1,m]
                                if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                    down_payoff=option_tree_1[j+1,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                    mid_payoff=option_tree_1[j,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
```

```
                    # store cur i payoff in option_tree_1, check larger payoff if early exercise
option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
option_tree_1[j,i,k]=max(underlying_tree_1[j,i,0]-underlying_tree_1[j,i,k],option_tree_1[j,i,k])
        return np.round(option_tree_1,2)
    #--->

    ## ------------------------------------ BT floating ------------------------------------ ##

    ## ------------------------------------ BT fixed ------------------------------------ ##
    def fixed_underlying_price_tree(self, N):
        '''input: the option and the number of periods'''
        '''output: the binomial tree of the underlying price (N+1 * 2**(N) * 2 array)'''

        ## parameter initializing
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        underlying_tree = np.zeros([2**N,N+1,2])

        underlying_tree[0,0,0] = self.S0
        underlying_tree[0,0,1] = self.S0

        if self.I == 0:
            ## put option ##
            for i in range(1,N+1):
                for j in range(2**i):
                    if j % 2 == 0:
                        underlying_tree[j,i,0] = round(underlying_tree[int(j/2),i-1,0]*u,2)
                        underlying_tree[j,i,1] = min(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])
                    else:
                        underlying_tree[j,i,0] = round(underlying_tree[int((j-1)/2),i-1,0]*d,2)
                        underlying_tree[j,i,1] = min(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])

        else:
            for i in range(1,N+1):
                for j in range(2**i):
                    if j % 2 == 0:
                        underlying_tree[j,i,0] = round(underlying_tree[int(j/2),i-1,0]*u,2)
                        underlying_tree[j,i,1] = max(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])
                    else:
                        underlying_tree[j,i,0] = round(underlying_tree[int((j-1)/2),i-1,0]*d,2)
                        underlying_tree[j,i,1] = max(underlying_tree[j,i,0], underlying_tree[int(j/2),i-1,1])

        return underlying_tree

    # <--- TRIAL
    def fixed_underlying_price_tree_0(self, N):

        ## parameter initializing
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        print('p ',p,'u ',u,'dt ',dt)

        # compute list size
        list_size=int(np.floor(N)/2+1)
        temp_list=np.zeros(list_size)
        print('list size',list_size)

        # underlying_tree_0[j,i,0] index 0 store underlying price
        # underlying tree_0[j,i,1:] index 1 onwards store max or min
        underlying_tree_0= np.zeros([N+1,N+1,list_size+1])
        underlying_tree_0[0,0,0] = self.S0
```

```
        underlying_tree_0[0,0,1]= self.S0    # first max or min list item
        underlying_tree_0[0,0,2:]=0

        # check if element exist in array
        def checkif_exist(array_,element):
            exist=0
            for n in range(len(array_)):
                if array_[n]==element:
                    exist=1
            return exist

        # check the next available index of empty slots in array
        def fnext_index(array_):
            index_=-1
            for n in range(len(array_)):
                if array_[n]==0:
                    index_=n
                    break
            return index_

        # put option, min list
        if self.I == 0:
            for i in range(1,N+1):
                for j in range(i+1):
                    # generate underlying price for tree
                    if j==0:
                        underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j),i-1,0]*u,2)
                    else:
                        underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j-1),i-1,0]*d,2)
                    # if not lowest branch
                    if j!=i:
                        cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                        # add min list items from prev lower left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i-1,k]!=0 and underlying_tree_0[j,i-1,k]<=underlying_tree_0[j,i,0]:
                                if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j,i-1,k])==0:
                                    underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i-1,k]
                                    cnt+=1
                    # if not highest branch
                    if j>=1:
                        cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                        cnt_old=cnt
                        # add min list items from prev upper left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j-1,i-1,k]!=0 and
underlying_tree_0[j-1,i-1,k]<=underlying_tree_0[j,i,0]:
                                if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j-1,i-1,k])==0:
                                    underlying_tree_0[j,i,cnt]=underlying_tree_0[j-1,i-1,k]
                                    cnt+=1
                        # add cur price to list if none was added from prev upper left list (i.e. lowest price)
                        if cnt==cnt_old and cnt==1:
                            underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i,0]
                            cnt+=1
        # call option, max list
        else:
            for i in range(1,N+1):
                for j in range(i+1):
                    # generate underlying price for tree
                    if j==0:
                        underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j),i-1,0]*u,2)
                    else:
                        underlying_tree_0[j,i,0] = round(underlying_tree_0[int(j-1),i-1,0]*d,2)
                    # if not lowest branch
                    if j!=i:
                        cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                        cnt_old=cnt
```

```
                    # add max list items from prev lower left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i-1,k]!=0 and underlying_tree_0[j,i-1,k]>=underlying_tree_0[j,i,0]:
                            if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j,i-1,k])==0:
                                underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i-1,k]
                                cnt+=1
                    # add cur price to list if none was added from prev lower left list (i.e. highest price)
                    if cnt==cnt_old:
                        underlying_tree_0[j,i,cnt]=underlying_tree_0[j,i,0]
                        cnt+=1
                # if not highest branch
                if j>=1:
                    cnt=fnext_index(underlying_tree_0[j,i,1:])+1
                    # add max list items from prev upper left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j-1,i-1,k]!=0 and
underlying_tree_0[j-1,i-1,k]>=underlying_tree_0[j,i,0]:
                            if checkif_exist(underlying_tree_0[j,i,1:],underlying_tree_0[j-1,i-1,k])==0:
                                underlying_tree_0[j,i,cnt]=underlying_tree_0[j-1,i-1,k]
                                cnt+=1

        # sort the list at each node from high to low
        for i in range(1,N+1):
            for j in range(i+1):
                temp_list=sorted(underlying_tree_0[j,i,1:],reverse=True)
                underlying_tree_0[j,i,1:]=temp_list

        return underlying_tree_0
    #--->

    # <--- TRIAL
    def fixed_underlying_price_tree_1(self, N):

        ## parameter initializing
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(3*self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        p_d = - (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) + 1/6
        p_m = 2/3; p_u = (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) +
1/6

        print('p ',p,'u ',u,'dt ',dt)
        print('p_u',p_u,'p_m',p_m,'p_d',p_d)

        # compute list size
        list_size=int(np.floor(N)/2+1)
        temp_list=np.zeros(list_size)
        print('list size',list_size)

        # underlying_tree_0[j,i,0] index 0 store underlying price
        # underlying tree_0[j,i,1:] index 1 onwards store max or min
        underlying_tree_1= np.zeros([2*N+1,N+1,list_size+1])
        underlying_tree_1[N,:,0] = self.S0
        underlying_tree_1[N,0,1]= self.S0    # first max or min list item
        underlying_tree_1[0,0,2:]=0

        # check if element exist in array
        def checkif_exist(array_,element):
            exist=0
            for n in range(len(array_)):
                if array_[n]==element:
                    exist=1
            return exist

        # check the next available index of empty slots in array
        def fnext_index(array_):
            index_=-1
```

```
            for n in range(len(array_)):
                if array_[n]==0:
                    index_=n
                    break
            return index_

                # generate underlying price for tree first
        for i in range(1,N+1):
            for j in range(1,i+1):
                underlying_tree_1[N-j,i,0] = round(underlying_tree_1[N+1-j, i-1,0]*u,2)
                underlying_tree_1[N+j,i,0] = round(underlying_tree_1[N-1+j, i-1,0]*d,2)

        # put option, min list
        if self.l == 0:
            for i in range(1,N+1):
                for j in range(N-i,N+i+1):
                    cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                    cnt_old=cnt
                    # if not lowest branch and 2nd lowest branch
                    if j<N+i-1:
                        cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                        # add min list items from prev lower left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j+1,i-1,k]!=0 and
underlying_tree_1[j+1,i-1,k]<=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j+1,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j+1,i-1,k]
                                    cnt+=1
                    # if not the lowest branch or highest branch
                    if j!=N+i or j!=N-i:
                        # add min list items from prev middle left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i-1,k]!=0 and underlying_tree_1[j,i-1,k]<=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i-1,k]
                                    cnt+=1
                    # if not highest branch and 2nd highest branch
                    if j>N-i+1:
                        # add min list items from prev upper left branch if <= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j-1,i-1,k]!=0 and
underlying_tree_1[j-1,i-1,k]<=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j-1,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j-1,i-1,k]
                                    cnt+=1
                    # add cur price to list if none was added from prev upper left list (i.e. lowest price)
                    if cnt==cnt_old and cnt==1:
                        underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i,0]
                        cnt+=1
        # call option, max list
        else:
            for i in range(1,N+1):
                for j in range(N-i,N+i+1):
                    cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                    cnt_old=cnt
                    # if not lowest branch and 2nd lowest branch
                    if j<N+i-1:
                        cnt=fnext_index(underlying_tree_1[j,i,1:])+1
                        # add max list items from prev lower left branch if >= cur price
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j+1,i-1,k]!=0 and
underlying_tree_1[j+1,i-1,k]>=underlying_tree_1[j,i,0]:
                                if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j+1,i-1,k])==0:
                                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j+1,i-1,k]
                                    cnt+=1
                    # if not the lowest branch or highest branch
```

```
                if j!=N+i or j!=N-i:
                    # add max list items from prev middle left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j,i-1,k]!=0 and underlying_tree_1[j,i-1,k]>=underlying_tree_1[j,i,0]:
                            if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j,i-1,k])==0:
                                underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i-1,k]
                                cnt+=1
                # if not highest branch and 2nd highest branch
                if j>N-i+1:
                    # add max list items from prev upper left branch if >= cur price
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j-1,i-1,k]!=0 and
underlying_tree_1[j-1,i-1,k]>=underlying_tree_1[j,i,0]:
                            if checkif_exist(underlying_tree_1[j,i,1:],underlying_tree_1[j-1,i-1,k])==0:
                                underlying_tree_1[j,i,cnt]=underlying_tree_1[j-1,i-1,k]
                                cnt+=1
                # add cur price to list if none was added from prev nodes list (i.e. highest price)
                if cnt==cnt_old:
                    underlying_tree_1[j,i,cnt]=underlying_tree_1[j,i,0]
                    cnt+=1

        # sort the list at each node from high to low
        for i in range(1,N+1):
            for j in range(N-i,N+i+1):
                temp_list=sorted(underlying_tree_1[j,i,1:],reverse=True)
                underlying_tree_1[j,i,1:]=temp_list

        return underlying_tree_1
    #--->

    def fixed_EU_option_tree(self,N):
        underlying_tree = self.fixed_underlying_price_tree(N)
        option_tree = np.zeros([2**N, N+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        option_tree[:,N] = np.maximum(np.zeros(2**N),(1-2*self.I)* (self.K - underlying_tree[:,N,1]))
        for i in np.arange(N-1,-1,-1):
            for j in np.arange(0,2**i):
                option_tree[j,i]=np.exp(-self.r*dt)*(p*option_tree[2*j,i+1]+(1-p)*option_tree[2*j+1,i+1])

        return np.round(option_tree,2)

    # <--- TRIAL
    def fixed_EU_option_tree_0(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_0 = self.fixed_underlying_price_tree_0(N)
        option_tree_0 = np.zeros([N+1, N+1,list_size+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        # put option, min list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=max(0,self.K-underlying_tree_0[j+1,i+1,m])
```

```
                            if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                down_payoff=max(0,self.K-round(underlying_tree_0[j,i,0]*d,2))
                            if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                up_payoff=max(0,self.K-underlying_tree_0[j,i,k])
                        # store cur i payoff in option_tree_0
                        option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                                if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=option_tree_0[j,i+1,m]
                            # store cur i payoff in option_tree_0
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
        # call option, max list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                        up_payoff=max(0,underlying_tree_0[j,i+1,m]-self.K)
                                    if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                        up_payoff=max(0,round(underlying_tree_0[j,i,0]*u,2)-self.K)
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=max(0,underlying_tree_0[j,i,k]-self.K)
                                # store cur i payoff in option_tree_0
                                option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                    # tree nodes before N-1
                    else:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then extract i+1 payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                        up_payoff=option_tree_0[j,i+1,m]
                                    if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                        up_payoff=option_tree_0[j,i+1,m]
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=option_tree_0[j+1,i+1,m]
                                # store cur i payoff in option_tree_0
                                option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)

        return np.round(option_tree_0,2)
    #--->

    # <--- TRIAL
    def fixed_EU_option_tree_1(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_1 = self.fixed_underlying_price_tree_1(N)
        option_tree_1 = np.zeros([2*N+1, N+1,list_size*2])

        #dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
```

```
        #d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(3*self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        p_d = - (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) + 1/6
        p_m = 2/3; p_u = (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) +
1/6

        # put option, min list
        if self.l==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=max(0,self.K-underlying_tree_1[j+1,i+1,m])
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=max(0,self.K-round(underlying_tree_1[j,i,0]*d,2))
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=max(0,self.K-underlying_tree_1[j,i+1,m])
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=max(0,self.K-underlying_tree_1[j,i,k])
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                    # tree nodes before N-1
                    else:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then extract i+1 payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=option_tree_1[j+1,i+1,m]
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=option_tree_1[j+1,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=option_tree_1[j,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=option_tree_1[j-1,i+1,m]
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)

        # call option, max list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=max(0,underlying_tree_1[j-1,i+1,m]-self.K)
                                    if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        up_payoff=max(0,round(underlying_tree_1[j,i,0]*u,2)-self.K)
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=max(0,underlying_tree_1[j,i+1,m]-self.K)
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=max(0,underlying_tree_1[j,i,k]-self.K)
```

```
                        # store cur i payoff in option_tree_1
option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_1[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
                                if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                    up_payoff=option_tree_1[j-1,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                    mid_payoff=option_tree_1[j,i+1,m]
                                if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                    down_payoff=option_tree_1[j+1,i+1,m]
                            # store cur i payoff in option_tree_1
option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)

        return np.round(option_tree_1,2)
    #--->


    def fixed_US_option_tree(self, N):
        underlying_tree = self.fixed_underlying_price_tree(N)

        option_tree = np.zeros([2**N, N+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        option_tree[:,N] = np.maximum(np.zeros(2**N), (1-2*self.I)*(self.K - underlying_tree[:,N,1]))
        for i in np.arange(N-1,-1,-1):
            for j in np.arange(0,2**i):
                option_tree[j,i]=np.exp(-self.r*dt)*(p*option_tree[2*j,i+1]+(1-p)*option_tree[2*j+1,i+1])

            option_tree[0:2**i,i]= np.maximum(option_tree[0:2**i, i],
(1-2*self.I)*(self.K-underlying_tree[0:2**i,i,0]))

        return np.round(option_tree,2)

    # <--- TRIAL
    def fixed_US_option_tree_0(self, N):
        list_size=int(np.floor(N)/2+1)
        underlying_tree_0 = self.fixed_underlying_price_tree_0(N)
        option_tree_0 = np.zeros([N+1, N+1,list_size+1])

        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)

        # put option, min list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(0,i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                        down_payoff=max(0,self.K-underlying_tree_0[j+1,i+1,m])
                                    if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
```

```
                            down_payoff=max(0,self.K-round(underlying_tree_0[j,i,0]*d,2))
                        if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                            up_payoff=max(0,self.K-underlying_tree_0[j,i,k])
                    # store cur i payoff in option_tree_0, check larger payoff if early exercise
                    option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                    option_tree_0[j,i,k]=max(self.K-underlying_tree_0[j,i,k],option_tree_0[j,i,k])
            # tree nodes before N-1
            else:
                for k in range(1,list_size+1):
                    if underlying_tree_0[j,i,k]!=0:
                        # compare i list with i+1 list, match found then extract i+1 payoff
                        for m in range(1,list_size+1):
                            if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                down_payoff=option_tree_0[j+1,i+1,m]
                            if round(underlying_tree_0[j,i,k]*d,2)==underlying_tree_0[j+1,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                down_payoff=option_tree_0[j+1,i+1,m]
                            if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                up_payoff=option_tree_0[j,i+1,m]
                        # store cur i payoff in option_tree_0, check larger payoff if early exercise
                        option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                        option_tree_0[j,i,k]=max(self.K-underlying_tree_0[j,i,k],option_tree_0[j,i,k])
    # call option, max list
    else:
        for i in np.arange(N-1,-1,-1):
            for j in np.arange(0,i+1):
                # N-1 tree nodes
                if i==N-1:
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then calc. payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=max(0,underlying_tree_0[j,i+1,m]-self.K)
                                if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    up_payoff=max(0,round(underlying_tree_0[j,i,0]*u,2)-self.K)
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=max(0,underlying_tree_0[j,i,k]-self.K)
                            # store cur i payoff in option_tree_0, check larger payoff if early exercise
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                            option_tree_0[j,i,k]=max(underlying_tree_0[j,i,k]-self.K,option_tree_0[j,i,k])
                # tree nodes before N-1
                else:
                    for k in range(1,list_size+1):
                        if underlying_tree_0[j,i,k]!=0:
                            # compare i list with i+1 list, match found then extract i+1 payoff
                            for m in range(1,list_size+1):
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j,i+1,m]:
                                    up_payoff=option_tree_0[j,i+1,m]
                                if round(underlying_tree_0[j,i,k]*u,2)==underlying_tree_0[j,i+1,m] and
underlying_tree_0[j,i,k]==underlying_tree_0[j,i,0]:
                                    up_payoff=option_tree_0[j,i+1,m]
                                if underlying_tree_0[j,i,k]==underlying_tree_0[j+1,i+1,m]:
                                    down_payoff=option_tree_0[j+1,i+1,m]
                            # store cur i payoff in option_tree_0, check larger payoff if early exercise
                            option_tree_0[j,i,k]=np.exp(-self.r*dt)*((p*up_payoff)+(1-p)*down_payoff)
                            option_tree_0[j,i,k]=max(underlying_tree_0[j,i,k]-self.K,option_tree_0[j,i,k])
    return np.round(option_tree_0,2)
#--->


# <--- TRIAL
def fixed_US_option_tree_1(self, N):
    list_size=int(np.floor(N)/2+1)
    underlying_tree_1 = self.fixed_underlying_price_tree_1(N)
```

```
        option_tree_1 = np.zeros([2*N+1, N+1,list_size*2])

        #dt, u = self.T/N, np.exp(self.sigma*np.sqrt(self.T/N))
        #d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        dt, u = self.T/N, np.exp(self.sigma*np.sqrt(3*self.T/N))
        d = 1/u; p = (np.exp((self.r-self.q)*dt)-d)/(u-d)
        p_d = - (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) + 1/6
        p_m = 2/3; p_u = (np.sqrt(dt/(12*self.sigma*self.sigma)))*(self.r-self.q-0.5*self.sigma*self.sigma) +
1/6

        # put option, min list
        if self.I==0:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=max(0,self.K-underlying_tree_1[j+1,i+1,m])
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=max(0,self.K-round(underlying_tree_1[j,i,0]*d,2))
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=max(0,self.K-underlying_tree_1[j,i+1,m])
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=max(0,self.K-underlying_tree_1[j,i,k])
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                                option_tree_1[j,i,k]=max(self.K-underlying_tree_1[j,i,k],option_tree_1[j,i,k])
                    # tree nodes before N-1
                    else:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then extract i+1 payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                        down_payoff=option_tree_1[j+1,i+1,m]
                                    if round(underlying_tree_1[j,i,k]*d,2)==underlying_tree_1[j+1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                        down_payoff=option_tree_1[j+1,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                        mid_payoff=option_tree_1[j,i+1,m]
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=option_tree_1[j-1,i+1,m]
                                # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                                option_tree_1[j,i,k]=max(self.K-underlying_tree_1[j,i,k],option_tree_1[j,i,k])

        # call option, max list
        else:
            for i in np.arange(N-1,-1,-1):
                for j in np.arange(N-i,N+i+1):
                    # N-1 tree nodes
                    if i==N-1:
                        for k in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]!=0:
                                # compare i list with i+1 list, match found then calc. payoff
                                for m in range(1,list_size+1):
                                    if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                        up_payoff=max(0,underlying_tree_1[j-1,i+1,m]-self.K)
                                    if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
```

```
                            up_payoff=max(0,round(underlying_tree_1[j,i,0]*u,2)-self.K)
                        if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                            mid_payoff=max(0,underlying_tree_1[j,i+1,m]-self.K)
                        if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                            down_payoff=max(0,underlying_tree_1[j,i,k]-self.K)
                    # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                    option_tree_1[j,i,k]=max(underlying_tree_1[j,i,k]-self.K,option_tree_1[j,i,k])
            # tree nodes before N-1
            else:
                for k in range(1,list_size+1):
                    if underlying_tree_1[j,i,k]!=0:
                        # compare i list with i+1 list, match found then extract i+1 payoff
                        for m in range(1,list_size+1):
                            if underlying_tree_1[j,i,k]==underlying_tree_1[j-1,i+1,m]:
                                up_payoff=option_tree_1[j-1,i+1,m]
                            if round(underlying_tree_1[j,i,k]*u,2)==underlying_tree_1[j-1,i+1,m] and
underlying_tree_1[j,i,k]==underlying_tree_1[j,i,0]:
                                up_payoff=option_tree_1[j-1,i+1,m]
                            if underlying_tree_1[j,i,k]==underlying_tree_1[j,i+1,m]:
                                mid_payoff=option_tree_1[j,i+1,m]
                            if underlying_tree_1[j,i,k]==underlying_tree_1[j+1,i+1,m]:
                                down_payoff=option_tree_1[j+1,i+1,m]
                        # store cur i payoff in option_tree_1

option_tree_1[j,i,k]=np.exp(-self.r*dt)*(p_u*up_payoff+p_m*mid_payoff+p_d*down_payoff)
                        option_tree_1[j,i,k]=max(underlying_tree_1[j,i,k]-self.K,option_tree_1[j,i,k])

        return np.round(option_tree_1,2)
    #--->


    ## ---------------------------------------- BT fixed ---------------------------------------- ##

    ## ---------------------------------------- BS floating ---------------------------------------- ##
    def floating_BS_call_option_tree(self, Smin):
        N = norm.cdf

        ## if the lookback option in our case has just been originated, so Smin = S0
        a1 = (np.log(self.S0/Smin) + (self.r - self.q + self.sigma**2/2)*self.T)/(self.sigma*np.sqrt(self.T))
        a2 = a1 - self.sigma*np.sqrt(self.T)
        a3 = (np.log(self.S0/Smin) + (-self.r + self.q + self.sigma**2/2)*self.T)/(self.sigma*np.sqrt(self.T))
        Y1 = -(2*(self.r - self.q- self.sigma**2/2)*np.log(self.S0/Smin))/self.sigma**2

        c = self.S0*np.exp(-self.q*self.T)*N(a1)-
self.S0*np.exp(-self.q*self.T)*(self.sigma**2/(2*(self.r-self.q)))*N(-a1)-Smin*np.exp(-self.r*self.T)*(N(a2)-(s
elf.sigma**2/(2*(self.r-self.q)))*np.exp(Y1)*N(-a3))
        return c

    def floating_BS_put_option_tree(self, Smax):
        N = norm.cdf

        ## the lookback option in our case has just been originated, so Smax = S0
        b1 = (np.log(Smax/self.S0) + (-self.r + self.q + self.sigma**2/2)*self.T)/(self.sigma*np.sqrt(self.T))
        b2 = b1 - self.sigma*np.sqrt(self.T)
        b3 = (np.log(Smax/self.S0) + (self.r - self.q - self.sigma**2/2)*self.T)/(self.sigma*np.sqrt(self.T))
        Y2 = (2*(self.r - self.q- self.sigma**2/2)*np.log(Smax/self.S0))/self.sigma**2

        p = -self.S0*np.exp(-self.q*self.T)*N(b2)+
self.S0*np.exp(-self.q*self.T)*(self.sigma**2/(2*(self.r-self.q)))*N(-b2) +
Smax*np.exp(-self.r*self.T)*(N(b1)-(self.sigma**2/(2*(self.r-self.q)))*np.exp(Y2)*N(-b3))
        return p
    ## ---------------------------------------- BS floating ---------------------------------------- ##

    ## ---------------------------------------- BS fixed ---------------------------------------- ##
    def fixed_BS_call_option_tree(self, Smax):
```

```
        N = norm.cdf

        SMAX = max(Smax, self.K)
        p = self.floating_BS_put_option_tree(SMAX)

        return p+self.S0*np.exp(-self.q*self.T)-self.K*np.exp(-self.r*self.T)

    def fixed_BS_put_option_tree(self, Smin):
        N = norm.cdf

        SMIN = min(Smin, self.K)
        c = self.floating_BS_call_option_tree(SMIN)

        return c+self.K*np.exp(-self.r*self.T)-self.S0*np.exp(-self.q*self.T)

    def cal_BS_put_option_delta(self, Smax):
        N = norm.cdf
        c1 = (np.log(self.S0/Smax) + (self.r + self.sigma**2/2)*self.T)/(self.sigma*np.sqrt(self.T))
        c2 = (np.log(Smax/self.S0) + (self.r - self.sigma**2/2)*self.T)/(self.sigma*np.sqrt(self.T))
        delta =
(1+self.sigma**2/(2*self.r))*N(c1)+(1-self.sigma**2/(2*self.r))*np.exp(-self.r*self.T)*(self.S0/Smax)**(-2*sel
f.r/self.sigma**2)*N(-c2) - 1

        return delta

    def BS_generate_stockprice(self,So,n):
        St=So
        dt, u_ = 1/252, self.r-self.sigma**2/2
        for i in range(1,n+1):
            phi=np.random.normal(u_*dt,(self.sigma)*np.sqrt(dt),1)
            St=St*np.exp(phi)
            print(St)
            option
```

DataAnalysisPlot.py

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
from pandas_datareader import data as pdr
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import Axes3D

#<---- data extraction for parameter estimation

AMD_daily_price = pdr.get_data_yahoo("AMD", start="2020-05-03", end="2020-08-03", interval =
'd').dropna()['Adj Close']
AMD_daily_price = np.array(AMD_daily_price)
n= len(AMD_daily_price)
AMD_daily_return = np.array([np.log(AMD_daily_price[i+1]/AMD_daily_price[i]) for i in range(n-1)])
sigma=AMD_daily_return.std()*252**0.5

IRX_daily_rate = pdr.get_data_yahoo("^IRX", start="2020-05-03", end="2020-08-03", interval =
'd').dropna()['Adj Close']
IRX_daily_rate = np.array(IRX_daily_rate)
r=IRX_daily_rate.mean()

plt.plot(AMD_daily_price)
plt.xlabel('Days')
plt.ylabel('Stock Price')

plt.plot(IRX_daily_rate)
plt.xlabel('Days')
plt.ylabel('T-bill Interest Rate ^IRX')
```

```
#<--- 3D plot of theoretical Lookback Option Prices

def floating_put(T,S0,sigma,K,r,q):
    N = norm.cdf
    Smax=S0

    ## the lookback option in our case has just been originated, so Smax = S0
    b1 = (np.log(Smax/S0) + (-r + q + sigma**2/2)*T)/(sigma*np.sqrt(T))
    b2 = b1 - sigma*np.sqrt(T)
    b3 = (np.log(Smax/S0) + (r - q - sigma**2/2)*T)/(sigma*np.sqrt(T))
    Y2 = (2*(r - q- sigma**2/2)*np.log(Smax/S0))/sigma**2

    p = -S0*np.exp(-q*T)*N(b2)+ S0*np.exp(-q*T)*(sigma**2/(2*(r-q)))*N(-b2) +
Smax*np.exp(-r*T)*(N(b1)-(sigma**2/(2*(r-q)))*np.exp(Y2)*N(-b3))
    return p

def floating_call(T,S0,sigma,K,r,q):
    N = norm.cdf
    Smin=S0

    ## if the lookback option in our case has just been originated, so Smin = S0
    a1 = (np.log(S0/Smin) + (r - q + sigma**2/2)*T)/(sigma*np.sqrt(T))
    a2 = a1 - sigma*np.sqrt(T)
    a3 = (np.log(S0/Smin) + (-r + q + sigma**2/2)*T)/(sigma*np.sqrt(T))
    Y1 = -(2*(r - q- sigma**2/2)*np.log(S0/Smin))/sigma**2

    c = S0*np.exp(-q*T)*N(a1)-
S0*np.exp(-q*T)*(sigma**2/(2*(r-q)))*N(-a1)-Smin*np.exp(-r*T)*(N(a2)-(sigma**2/(2*(r-q)))*np.exp(Y1)*N(-
a3))
    return c

def fixed_call(T,S0,sigma,K,r,q):
    N = norm.cdf
    Smax=S0

    SMAX = np.maximum(Smax, K)
    p = floating_put(T,SMAX,sigma,K,r,q)

    return p+S0*np.exp(-q*T)-K*np.exp(-r*T)

def fixed_put(T,S0,sigma,K,r,q):
    N = norm.cdf
    Smin=S0

    SMIN = np.minimum(Smin, K)
    c = floating_call(T,SMIN,sigma,K,r,q)

    return c+K*np.exp(-r*T)-S0*np.exp(-q*T)


sigma=0.58
r=0.0012
S0=50
K=50
q=0

T=np.linspace(1,12,12)/12
S=np.linspace(S0,2*S0,S0)
Km=np.linspace(K,K,S0)
x,y =np.meshgrid(S,T)

fig=plt.figure(figsize=(8,6))
ax1=fig.add_subplot(221,projection='3d')
ax1.plot_surface(x,y,floating_put(y,x,sigma,K,r,q),rstride=2,cstride=2,cmap=cm.jet,alpha=0.7,linewidth=0.
25)
```

```
ax1.set_xlabel('So')
ax1.set_ylabel('T')
ax1.set_zlabel('Option Price')
ax1.set_title('Floating Strike Put')

ax2=fig.add_subplot(222,projection='3d')
ax2.plot_surface(x,y,floating_call(y,x,sigma,K,r,q),rstride=2,cstride=2,cmap=cm.jet,alpha=0.7,linewidth=0.
25)
ax2.set_xlabel('So')
ax2.set_ylabel('T')
ax2.set_zlabel('Option Price')
ax2.set_title('Floating Strike Call')

ax3=fig.add_subplot(223,projection='3d')
ax3.plot_surface(x,y,fixed_put(y,x,sigma,K,r,q),rstride=2,cstride=2,cmap=cm.jet,alpha=0.7,linewidth=0.25
)
ax3.set_xlabel('So')
ax3.set_ylabel('T')
ax3.set_zlabel('Option Price')
ax3.set_title('Fixed Strike Put')

ax4=fig.add_subplot(224,projection='3d')
ax4.plot_surface(x,y,fixed_call(y,x,sigma,K,r,q),rstride=2,cstride=2,cmap=cm.jet,alpha=0.7,linewidth=0.25
)
ax4.set_xlabel('So')
ax4.set_ylabel('T')
ax4.set_zlabel('Option Price')
ax4.set_title('Fixed Strike Call')

#<---- Hedging Analysis Results Plot

x=[51.33,52.07,54.44,51.09,49.98,51.61,50.63,48.79,45.74,45.71]
y_tree=[2.4551,2.444,2.6835,2.3738,2.4737,2.5344,2.4622,2.3616,2.191,2.1545]
y_theor=[0.00351,0.00629,0.00668,0.00423,0.00506,0.00267,0.00061,0.00613,-0.00222,0.00537]
plt.plot(y_tree,label='binomial tree values')
plt.plot(y_theor, label='theoretical values')
plt.xlabel('nth stock')
plt.ylabel('Hedged portfolio value')
plt.legend(loc='lower right')
```