財務演算法期末報告

台大財金所財工組碩二 R10723058

陳韋勳

Objection:

Price an arithmetic average call with the following payoff using the binomial tree model.

$$\text{Payoff}_{\tau} = \max\left(S_{\text{ave},\tau} - K, 0\right),$$

where $S_{ave,\tau}$ is the arithmetic average of stock prices calculated from the issue date until the current time point τ

Code Files:

1. *Arithmetic_Average_Call_Option_Pricing_by_BT_and_MT.py*
2. *Arithmetic_Average_Call_by_BT_with_Linearly_vs_Logarithmically_Equally_Spaced_Placement_Method.py*
3. *Arithmetic_Average_Call_by_BT_with_Different_FindingWays.py*

Methods:

1. Implement the binomial tree model to price both European and American arithmetic average calls.
   (*Arithmetic_Average_Call_Option_Pricing_by_BT_and_MT.py*)
   The algorithm of Hull and White (1993):
   (1) For any node(i, j), the maximum arithmetic average price is contributed by a price path starting with $i - j$ consecutive up movements followed by j consecutive down movements, and the minimum arithmetic average price can be calculated from a price path starting with j consecutive down movements followed by $i - j$ consecutive up movements.

```python
    def record_possible_Save(self):
        self.tree[0, 0].possibleSaveList.append(self.S_avet)
        for ithCol in range(1, self.n + 1):
            for jthRow in range(ithCol + 1):
                currentNode = self.tree[jthRow, ithCol]

                A_max = (self.S_avet * self.previous_n
                         + self.St * self.u * ((1 - self.u ** (ithCol - jthRow)) / (1 - self.u))
                         + self.St * (self.u ** (ithCol - jthRow)) * self.d * ((1 - self.d ** jthRow) / (1 - self.d)))
                A_max /= (self.previous_n + ithCol)

                A_min = (self.S_avet * self.previous_n
                         + self.St * self.d * ((1 - self.d ** jthRow) / (1 - self.d))
                         + self.St * (self.d ** jthRow) * self.u * ((1 - self.u ** (ithCol - jthRow)) / (1 - self.u)))
                A_min /= (self.previous_n + ithCol)
```

   (2) For each node, representative average prices are placed (logarithmically) equally spaced from the maximum to the minimum arithmetic average prices for each node via the following formula.

```python
if((jthRow == 0) or (jthRow == ithCol)):
    currentNode.possibleSaveList.append(A_max)
else:
    if(self.Savg_spaced_way == 'Linear'):
        for k in range(0, self.M + 1):
            A_i_j_k = ((self.M - k) / self.M) * A_max + (k / self.M) * A_min
            currentNode.possibleSaveList.append(A_i_j_k)
    elif(self.Savg_spaced_way == 'Logarithmically'):
        for k in range(0, self.M + 1):
            A_i_j_k = np.exp(((self.M - k) / self.M) * np.log(A_max) + (k / self.M) * np.log(A_min))
            currentNode.possibleSaveList.append(A_i_j_k)
```

(3) For each terminal node (n, j), decide the payoff for each representative average price $A(n, j, k)$.

```python
def calc_terminal_node_Payoff(self):
    lastColIndex = self.n
    for jthRow in range(lastColIndex + 1):
        # Map each element in possibleSmaxList to possibleCallList
        currentNode = self.tree[jthRow, lastColIndex]
        for i in range(len(currentNode.possibleSaveList)):
            currentNode.possibleCallList.append(max(currentNode.possibleSaveList[i] - self.K, 0))
```

(4) Backward induction

```python
def backward_induction(self, type='European', searchWay='Sequential search'):
    p = self.p

    for ithCol in range(self.n - 1, -1, -1):
        for jthRow in range(ithCol + 1):
            # Process each node
            currentNode = self.tree[jthRow, ithCol]
            upperChildNode = self.tree[jthRow, ithCol + 1]
            lowerChildNode = self.tree[jthRow + 1, ithCol + 1]
            call_upperChildNode = -1
            call_lowerChildNode = -1

            # Match every element
            for k in range(len(currentNode.possibleSaveList)):
                # The upper child
                Au = (((ithCol + self.previous_n) * currentNode.possibleSaveList[k] + upperChildNode.sPrice)
                      / (self.previous_n + ithCol + 1))

                j = 0
                if(searchWay == 'Sequential search'):
                    while(upperChildNode.possibleSaveList[j] - Au > 10**-8):
                        j += 1
                elif(searchWay == 'Binary search'):
                    j = self.binary_search(upperChildNode.possibleSaveList, Au, 0, len(upperChildNode.possibleSaveList) - 1)
                elif(searchWay == 'Linear interpolation search'):
                    j = self.linear_interpolation_search(upperChildNode.possibleSaveList, Au, 0, len(upperChildNode.possibleSaveList) - 1)

                if(abs(upperChildNode.possibleSaveList[j] - Au) <= 10**-8):
                    call_upperChildNode = upperChildNode.possibleCallList[j]
                elif(upperChildNode.possibleSaveList[j] < Au):
                    # Inner interploation
                    Wu = (upperChildNode.possibleSaveList[j - 1] - Au) / (upperChildNode.possibleSaveList[j - 1] - upperChildNode.possibleSaveList[j])
                    call_upperChildNode = Wu * (upperChildNode.possibleCallList[j]) + (1 - Wu) * (upperChildNode.possibleCallList[j - 1])
```

```python
                # The lower child
                Ad = (((ithCol + self.previous_n) * currentNode.possibleSaveList[k] + lowerChildNode.sPrice)
                      / (self.previous_n + ithCol + 1))

                j = 0
                if(searchWay == 'Sequential search'):
                    while(lowerChildNode.possibleSaveList[j] - Ad > 10**-8):
                        j += 1
                elif(searchWay == 'Binary search'):
                    j = self.binary_search(lowerChildNode.possibleSaveList, Ad, 0, len(lowerChildNode.possibleSaveList) - 1)
                elif(searchWay == 'Linear interpolation search'):
                    j = self.linear_interpolation_search(lowerChildNode.possibleSaveList, Ad, 0, len(lowerChildNode.possibleSaveList) - 1)

                if(abs(lowerChildNode.possibleSaveList[j] - Ad) <= 10**-8):
                    call_lowerChildNode = lowerChildNode.possibleCallList[j]
                elif(lowerChildNode.possibleSaveList[j] < Ad):
                    # Inner interploation
                    Wd = (lowerChildNode.possibleSaveList[j - 1] - Ad) / (lowerChildNode.possibleSaveList[j - 1] - lowerChildNode.possibleSaveList[j])
                    call_lowerChildNode = Wd *  (lowerChildNode.possibleCallList[j])+ (1 - Wd) * (lowerChildNode.possibleCallList[j - 1])

                currentNodeCall = np.exp((-1) * self.r * (self.deltaT / self.n)) * (p * call_upperChildNode + (1 - p) * call_lowerChildNode)

                if(type == 'American'):
                    # Early exercise or not
                    if(currentNodeCall < (currentNode.possibleSaveList[k] - self.K)):
                        currentNodeCall = (currentNode.possibleSaveList[k] - self.K)
                currentNode.possibleCallList.append(currentNodeCall)
```

2. Implement the Monte Carlo simulation to price European arithmetic average calls. (*Arithmetic_Average_Call_Option_Pricing_by_BT_and_MT.py*)

(1) Inputs: St, K, r, q, σ, t, $T-t$, M, n, $S_{ave,t}$, number of simulations, number of repetitions.

(2) Outputs: Option values for both methods and 95% confidence interval for Monte Carlo simulation.

```python
def monteCarlo_calc():
    '''
    Calculate call and put pirce by Monte Carlo simulation.
        Parameters:
            paramsList: list
        Outputs:
            c: float -> call price
            p: float -> put price
    '''
    print('Monte Carlo simulation')

    # St, K, r, q, σ, t, T-t, n, S_avet, simCnt, repCnt
    paramsList = [float(i) for i in input('# St, K, r, q, σ, t, T-t, n, S_avet, simCnt, repCnt = ').split()]
    St, K, r, q, sigma, t, deltaT, n, S_avet, simCnt, repCnt = paramsList
    n, simCnt, repCnt = int(n), int(simCnt), int(repCnt)
    previous_n_to_timet = n * (deltaT + t) / deltaT - n + 1
    total_n = previous_n_to_timet + n

    def drawSample(St, K, r, q, sigma, t, deltaT, n, S_avet, simCnt):
        mean = np.log(St) + ((r - q - sigma ** 2 / 2)*(deltaT / n))
        std = sigma*(((deltaT) / n) ** 0.5)
        lnSt_list = np.random.normal(loc=mean, scale=std, size=int(simCnt))

        St_sum_list = np.add(np.multiply(S_avet, previous_n_to_timet), np.exp(lnSt_list))
        for i in range(n-1):
            mean = lnSt_list + ((r - q - sigma ** 2 / 2) * (deltaT / n))
            lnSt_list = np.random.normal(loc=mean, scale=std)
            St_sum_list = np.add(np.exp(lnSt_list), St_sum_list)

        Save_list = St_sum_list / total_n
        payoff_list = np.maximum(Save_list - K, 0) * np.exp(-r * deltaT)
        return np.mean(payoff_list)

    resultList = []
    for i in range(int(repCnt)):
        p = drawSample(St, K, r, q, sigma, t, deltaT, n, S_avet, simCnt)
        resultList.append(p)

    print(f'  Average call price upper bound: {np.round(np.mean(resultList) + 2*np.std(resultList), 4)}')
    print(f'  Average call priceclower bound: {np.round(np.mean(resultList) - 2*np.std(resultList), 4)}', '\n')
```

Results:

1. *Arithmetic_Average_Call_Option_Pricing_by_BT_and_MT.py* 裡面有 binomial tree method & Monte Carlo simulation 的程式碼。
   按執行後，依序輸入 50 50 0.1 0.05 0.8 0 0.25 100 100 50
   會得到$t = 0$時的 European & American average call price，如下圖：

```
# St, K, r, q, σ, t, T-t, M, n, S_avet = 50 50 0.1 0.05 0.8 0 0.25 100 100 50
Binomial Tree method
European Average Call price:  4.7354
American Average Call price:  5.4146
```

2. 接著再輸入 50 50 0.1 0.05 0.8 0.25 0.25 100 100 50，
   會得到$t = 0.25$時的 European & American average call price，如下圖：

```
# St, K, r, q, σ, t, T-t, M, n, S_avet = 50 50 0.1 0.05 0.8 0.25 0.25 100 100 50
Binomial Tree method
European Average Call price:  2.3795
American Average Call price:  2.5079
```

3. 接著再輸入 50 50 0.1 0.05 0.8 0 0.25 100 50 10000 20，
   會得到 Monte Carlo simulation 的上下界，如下圖：

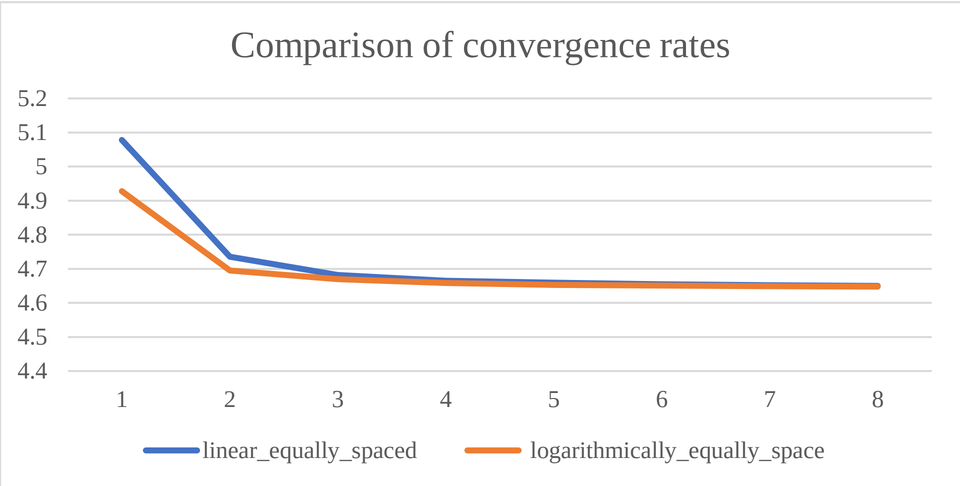
```
Monte Carlo simulation
# St, K, r, q, σ, t, T-t, n, S_avet, simCnt, repCnt = 50 50 0.1 0.05 0.8 0 0.25 100 50 10000 20
  Average call price upper bound: 4.7968
  Average call priceclower bound: 4.5167
```

Time Complexity Analysis:

1. Compare the convergence rates of the linearly and logarithmically equally-spaced placement methods (*Arithmetic_Average_Call_by_BT_with_Linearly_vs_Logarithmically_Equally_Spaced_Placement_Method.py*)
執行後得到數據如下圖:

6	Option Value\ M	50	100	150	200	250	300	350	400
7	Linearly equally-spaced	5.0781	4.7354	4.6821	4.6652	4.659	4.6543	4.6514	4.6498
8	logarithmically equally-spaced	4.9278	4.6949	4.6699	4.6586	4.6531	4.6505	4.6488	4.6478

將結果繪出如圖:



得到 Logarithmically equally space method 會收斂比較快的結論。

2. Compare the computational time of the following three methods to locate the positions of A_u and A_d.
(*Arithmetic_Average_Call_by_BT_with_Different_FindingWays.py*, 需要引入 *Arithmetic_Average_Call_Option_Pricing_by_BT_and_MT.py*，因為 binary search & linear interpolation method 都在這份.py 檔中，如下圖)

```python

    def binary_search(self, array, target, low, high):
        '''
        Using binary search to find the index that target will larger or equal to array[index]
        and smaller than array[index - 1].
        '''
        if(high >= low):
            mid = low + (high - low) // 2

            if((abs(array[mid] - target) < 10 ** -8) or ((array[mid] < target) and (array[mid - 1] > target))):
                return mid

            elif(array[mid] < target):
                return self.binary_search(array, target, low, mid - 1)
            elif(array[mid] > target):
                return self.binary_search(array, target, mid + 1, high)

    def linear_interpolation_search(self, array, target, low, high):
        '''
        Using linear interpolation search to find the index that target will larger or equal to array[index]
        and smaller than array[index - 1].
        '''
        if high >= low:
            if high == low:
                return low

            index = int(((array[low] - target) * high + (target - array[high]) * low) / (array[low] - array[high]))

            counts = 0
            while(~((abs(array[index] - target) < 10 ** -8) or ((array[index] < target) and (array[index - 1] > target)))):
                if((abs(array[index] - target) < 10 ** -8) or ((array[index] < target) and (array[index - 1] > target))):
                    return index
                else:
                    if(counts < 0):
                        counts -= 1
                    else:
                        counts += 1
                    counts = -counts
                    index = index + counts
            return index
```

在 Arithmetic_Average_Call_by_BT_with_Different_FindingWays.py 的執行檔中按下執行後，輸入 50 50 0.1 0.05 0.8 0 0.25 500 100 50，會得到各個方法在 $M = 500$時的計算時間，如圖：

```
# St, K, r, q, σ, t, T-t, M, n, S_avet = 50 50 0.1 0.05 0.8 0 0.25 500 100 50
Binomial Tree method
--- Sequential search: 272.821049451828 seconds ---

Binomial Tree method
--- Binary search: 239.4256603717804 seconds ---

Binomial Tree method
--- Linear interpolation search: 204.05728435516357 seconds ---
```

得到 Linear interpolation search 快過 Binary search 再快過 Sequential search 的結論。

Reference:

http://homepage.ntu.edu.tw/~jryanwang/