FE522: Assignment 2

Lun Li

November 27, 2019

Question 1 Pricing American Option

We implement a re-combining binomial tree to price American Option. The following ingredients are required: initial stock price S_0, volatility σ, strike of the option K, expiry of option T and risk free rate r. Notice the σ here will not be explicit as we are developing a tree. Suppose the volatility is a lognormal volatility, then to be compatible with Black-Scholes SDE, we can set up $m\_u = x$ as upper size and down size $m\_d = y$ such $xy = 1$, in the meanwhile, to match the moments, we can solve for:

$$m\_u = \exp(\sigma * \sqrt(\delta)), \quad m\_d = 1/m\_u,$$
$$\mathbb{P}(\textit{upper}) = 0.5 + \frac{r\sqrt(\delta)}{2\sigma}, \quad \mathbb{P}(\textit{down}) = 1 - m\_p;$$

Here δ is the discretization size, $\delta = T/N$ with user sepcified number of stages N. The call pricing mechanism is given by the following dynamic programming equation(DPE):

$$\begin{cases} V(t_{i-1}) & = \max\left\{(S(t_i) - K), e^{r\delta}\mathbb{E}[V(t_{i+1})]\right\}, \\ V(t_N) & = (S(t_N) - K)^+ \end{cases} \tag{1}$$

where the expectation is the sum of two probability with their corresponding values.

To implement the algorithm, we have American Option class the equips with the following member functions:

- **Constructor:** initialize members, check their validity and calls the *forwardFeed* function to generate the lattice;
- **CheckConstraints:** It checks if all inputs are legitimate;
- **GetLattice:** it display the 2D vector with upper triangle occupied by stocke prices;
- **ForwardFeed:** It populates a 2D vector with forward price on the leaves according to up-down sizes.
- **BackWardPropogation:** It execute DPE ((1)) and derive anther value lattice $V(t_i)$ which has exactly the same shape as underlying price evolution lattice. The very first stage with one value $V(t_0)$ is the final value.

The results below is pricing an option with

optType = "call", spotPrice = 158.28, strikePrice = 172.5, interestRate= 0.001, volatility = 0.2153, timeToMaturity = 1.;

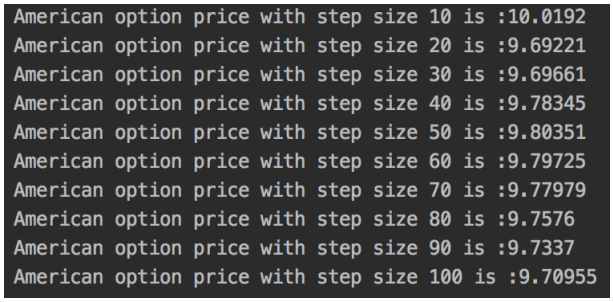

Figure 1: Convergence of Binomial Tree for American Option Pricing

Question 2 Option Pricer

We have a base class *Option* which implements common functions that derived class will have the same implementation. They are:

- **Constructor**: the option constructor initialize all member variables and check validity of parameters;
- **CheckConstraints**: check if all values are legitimate;
- **getDelta/Vega/Rho/Theta**: it implements a high value bump-reval;

$$\frac{dV}{dX} = \frac{V(X+\epsilon) - V(X)}{\epsilon} \tag{2}$$

  regardless the option type. Here V is calculated by calling *getPrice* function, X is the risk factor and ϵ is bump size.

- **Setters**: they are used to reset parameters, in this application, the risk calculation requires such functionalities.

Notice *getPrice()* function is set to be virtual, so the derived class can have its own implementaion of pricing analytics.

In main function, we can take advantage of polymorphism, declare a base class pointer, and specify it to American and European class repectively. Taking the same option as in *Question 1*, we have results below: Due to the extra optionality (arbitrary exercise time), the price of American option is greater than

Figure 2: American Option v.s. European Option

European option. In addition, American option is more sensitive to all risk factors than European one.

Question 3 and Question 4 Linked List

To have the node component accomodating American/European option, we set the member being a Option pointer type, this makes casting to different class convenient. The *PrintAll()* function implementation is quite straight forward.

- **Step 1:** Deal with empty list. If not empty, continue, otherwise, throw;

- **Step 2:** Define current pointer variable to hold current component, backward traverse the linked list until the head is hit;
- **Step 3:** Start from the head node, print option information node by node until it reached the end of the linked list.

In terms of implementation of *AddOrdered()*, we add to the constructor of link a functor below:

string setID(Option* ptr){ return ptr->m_exerciseType + "-" + ptr->m_optType + "-" + to_string(ptr->m_timeToMaturity) + "-" + to_string(ptr->m_strikePrice); }

This is more flexible, allowing user to define unique identification at their discretions. For this question, we define ID by (type, time to maturity, strike price), where all numeric numbers are converted to string, which has a natural order. This allows us to insert element in an ordered way. The code below is doing the insertion based on composition of uniqueID.

```
Link* Link::addOrdered(Link *n)
{
  if(n == nullptr) return this;
  if(this == nullptr) return n;

  // for non-corner case
  Link* cur = this;
  string id = n->uniqueID;
  if(id >= this->uniqueID){
    while(cur->next())
    {
      cur = cur->next();
      if(id <= cur->uniqueID) return cur->insert(n);
    }
    cur->setNext(n);
    n->setPre(cur);
    return n;
  }else{
    while(cur->previous())
    {
      cur = cur->previous();
      if(id >= cur->uniqueID) return cur->add(n);
    }
    cur->setPre(n);
    n->setNext(cur);
    return n;
  }
}
```

Figure 3: Add Ordered Option

To briefly explain, it firstly deal with corner cases, then we have two cases: one is current insertion is smaller than the link component interested, then it has to go previous until it find a slot where it is larger then the previous one but smaller then the next one, if it is larger than the link of interests, we go next until it is larger than the passing one and smaller than the next one. See results below with a mixture of American and Euroepan Option.

By using template, we can parameterize LinkedList developed so it is more general, that is, the content each component carry can be "arbitrary"(see Figure **??**). We did one more step abstract to allow printAll function take in a functor, that functor decides what is to be displayed from the each content. This way, we almost completely rip off the concreteness.

```
This option is of type AmericanOption-call, has spot price 100, strike price 100, risk free rate 0.01, volatility 0.2, and time to maturity 1.
This option is of type AmericanOption-call, has spot price 100, strike price 101, risk free rate 0.01, volatility 0.3, and time to maturity 2.
This option is of type AmericanOption-call, has spot price 100, strike price 102, risk free rate 0.01, volatility 0.4, and time to maturity 3.
This option is of type AmericanOption-call, has spot price 100, strike price 103, risk free rate 0.01, volatility 0.5, and time to maturity 4.
This option is of type EuropeanOption-call, has spot price 100, strike price 103, risk free rate 0.01, volatility 0.5, and time to maturity 4.
This option is of type EuropeanOption-call, has spot price 100, strike price 104, risk free rate 0.01, volatility 0.5, and time to maturity 4.
```

Figure 4: Display Option List

Question 5 Char Manipulation

(a) Write a function, char* strdup(const char*), that copies a C-style string into memory it allocates on the free store. This is done by using c memory allocation function *malloc*.

```
char* strdup(char const * cStr)
{
  char *temp = (char*)malloc( size: lenOfChar(cStr) + 1);

  char* ptr = temp;
  while(*cStr != '\0')
  {
    *ptr = *cStr;
    ptr++; cStr++;
  }
  *ptr = '\0';
  return temp;
}
```

Figure 5: Meomry Allocation

(b) Write a function, char* findx(const char* s, const char* x), that finds the first occurrence of the C-style string x in s. This is done by looping through characters and compareStr function is the subroutine to compare string, which is again done by looping through chars(see code submitted).

```
// find first occurence of the C-stye string x in s
const char* findx(const char* s, const char* x)
{
  const char* cur = s;
  while(*cur != '\0')
  {
    if(compareStr(cur, x))
      break;
    else
      cur++;
  }
  return cur;
}
```

Figure 6: Char First Occurence

(c) Write a function, int strcmp(const char* s1, const char* s2), that compares C-style strings. Let it return a negative number if s1 is lexicographically before s2, zero if s1 equals s2, and a positive number if s1 is lexicographically after s2. This utilizes char iterator..

```
int strCmp(const char *a, const char *b)
{
  for(; *a && *b && *a == *b; ++a, ++b)
    ;
  if (*b == *a)
    return 0;
  else if(*b > *a)
    return -1;
  else
    return 1;
}
```

Figure 7: Char First Occurence

Question 6 (Bonus) Barrier Option

The Barrier class is also derived from base class, but the pricing is done by simulation. Namely, we assume again the lognromal process of underlying price:

$$dS_t = rS_t dt + \sigma S_t dW_t, \ S_0 = S \tag{3}$$

Let us partition time to maturity T into N equal size pieces, $\delta = T/N$. Using Euler scheme, we can write the following recursive form:

$$S_{t+1} = S_t + rS_t\delta + \sigma S_t\sqrt{\delta}Z, \ S_0 = S \tag{4}$$

where $Z \sim N(0,1)$ is standard normal. The member function *simulatePath()* is evolving underlying price according to (4), while combined with barrier conditions, for which we support the following four types:

- **up-and-out**: if the process S_t reach some upper bound U, we get zero payoff;
- **down-and-out**: if the process S_t decreased to some lower bound L, we get zero payoff;
- **up-and-in**: only if the process S_t reach some upper bound U, the option can be activated;
- **down-and-in**: if the process S_t decreased to some lower bound L, the option can be activated.

In *getPrice()* function, we simulate M number of times paths. Then by strong law of large number, we can approximate option pricing expectation by averaging payoff on each path, i.e.,

$$V(t_0) = P(0,t_N)\mathbb{E}[H(S_\cdot)] = \frac{1}{N}\sum_{i=1}^{N} H(S_\cdot^i) \tag{5}$$

where H is the payoff functional that can possibly depend on the whole path $S_\cdot$. Below is the results we ran on the same option as *Question 2*, with up-and-out Barrier 200.

```
Barrier option price is :8.93687, Delta is :1.05546, Vega is :24.1926, Theta is :2.76314, Rho is :158.093
```

Figure 8: Barrier Option Results