FE545 - Homework #4

Author: Sid Bhatia

Date: April 6th, 2023

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

Professor: Steve Yang

Problem 1

Pricing a derivative security entails calculating the *expected discounted value* of its *payoff*. This reduces, in principle, to a problem of numerical integration; but in practice this calculation is often difficult for high-dimensional pricing problems.

Broadie and **Glasserman** (1997) proposed the method of the simulated tree to price American options, which can derive the upper and lower bounds for American options. This combination makes it possible to measure and control errors as the computational effort increases.

The main drawback of the random tree method is that its computational requirements grow exponentially in the number of exercise dates m, so the method is applicable only when m is small. Nevertheless, for problems with small m, it is very effective, and it also serves to illustrate a theme of managing sources of high and low bias.

High Estimator

As its name suggests, the random tree method is based on simulating a tree of paths of the underlying **Markov Chain** $X_0, X_1, \ldots, X_m$. Fix a branching parameter $b \geq 2$. From the initial state X_0, simulate b independent successor states $X_1^1, \ldots, X_1^B$ all having the **law** of X_1. For each X_1^i, simulate b independent successors $X_1^{i1}, \ldots X_1^{ib}$ from the **conditional law** of X_2 given $X_1 = X_1^i$. From each $X_2^{i_1 i_2}$, generate b successors $X_3^{i_1 i_2 1}, \ldots X_3^{i_1 i_2 b}$, and so on. We denote a *generic node* in the tree at time step i by $X_i^{j_1 j_2 \cdots j_i}$. The superscript indicates that this node is reached by following the j-th branch out of X_0, the j_2-th branch out of the next node, and so on.

Although it is not essential that the branching parameter remain fixed across time steps, this is a convenient simplification in discussing the method. From the random tree method we define high and low estimators at each node by backward induction.

We use **formulation 2**. Thus, $\hat{h}_i$ is the discounted payoff function at the i-th exercise date, and the discounted option value satisfies $\hat{V}_m = \hat{h}_m$,

$$\tilde{V}_i(x) = \max\left(\tilde{h}_i(x), \mathbb{E}^Q[\tilde{V}_{i+1}(X_{i+1}) \mid X_i = x]\right),\ i = 1, \ldots, m-1 \tag{3}$$

Write $\hat{V}_i^{j_i \cdots j_i}$ for the **value of the high estimator** at node $X_i^{j_i \cdots j_i}$. At the terminal nodes, we set

$$\hat{V}_i^{j_1 \cdots j_m} h_m(X_m^{j_1 \cdots j_m}). \tag{4}$$

Working backward, we then set

$$\hat{V}_i^{j_1 \cdots j_i} = \max\left(h_i(X_i^{j_1 \cdots j_i}),\ \frac{1}{b}\sum_{j=1}^{b} \hat{v}_{i+1}^{j_1 \cdots j_i j}\right). \tag{5}$$

Formulation 5 is based on successor nodes, so the estimator is unfairly peeking into the future in making its decision. To remove this source of bias, we need to separate the exercise decision from the value received upon continuation. A new estimator can be defined as follows:

At all terminal nodes, set the estimator equal to the payoff at that node:

$$\hat{v}_m^{j_1 \cdots j_m} = h_m(X_m^{j_1 \cdots j_m}). \tag{6}$$

At node $j_1, j_2, \ldots j_i$ at time step i, and for each $k = 1, \ldots, b$, set

$$\hat{v}_{ik}^{j_1 j_2 \cdots j_i} = \begin{cases} h_i(X_i^{j_1 j_2 \cdots j_i}) & \text{if } \frac{1}{b}\sum_{j=1}^{b} \hat{v}_{i+1}^{j_1 j_2 \cdots j_i j} \le h_i(X_i^{j_1 j_2 \cdots j_i}); \\ \hat{v}_{i+1}^{j_1 j_2 \cdots j_i k} & \text{otherwise} \end{cases} \tag{7}$$

Then set

$$\hat{v}_i^{j_1 \cdots j_i} = \frac{1}{b}\sum_{k=1}^{b} \hat{v}_{i,k}^{j_1 \cdots j_i} \tag{8}$$

Working backward, we then set

$$\hat{V}_i^{j_1 \cdots j_i} = \max\left(h_i(X_i^{j_1 \cdots j_1}), \frac{1}{b}\sum_{k=1}^{b} \hat{V}_{i,k}^{j_1 \cdots j_i})\right) \tag{9}$$

The high estimator of the option price at the current time and state is $\hat{v}_0$.

In other words, the high estimator is simply the result of applying **ordinary dynamic programming** to the *random tree*, assigning equal weight to each branch. Its calculation is illustrated in Figure 1 with $h_i(x) = (x - 100)_+$.

A simple induction argument demonstrates that the high estimator is **indeed biased high** at every node, in the sense that

$$\mathbf{E}^Q[\hat{V}_i^{j_1 \cdots j_i} \mid X_i^{j_1 \cdots j_i}] = \mathbf{E}[V_i^{j_i \cdots j_i}] \tag{10}$$

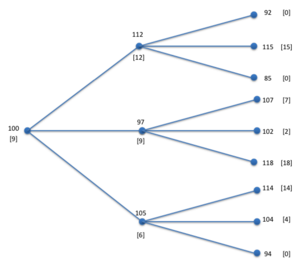

Figure 1: High Estimator K = 100

The low estimator is defined as follows. At all terminal nodes, set the estimator equal to the payoff at that node:

$$\hat{v}_m^{j_1 j_2 \cdots j_m} = h_m(X_m^{j_1 j_2 \cdots j_m}) \tag{11}$$

At each node $j_1, j_2, \ldots, j_i$ at time step i, and for each $k = 1, \ldots b$, set

$$\hat{v}_{ik}^{j_1 j_2 \cdots j_i} = \begin{cases} h_i(X_i^{j_1 j_2 \cdots j_i}) & \text{if } \frac{1}{b-1}\sum_{j=1; j\neq k}^{b} \hat{v}_{i+1}^{j_1 j_2 \cdots j_i j} \leq h_i(X_i^{j_1 j_2 \cdots j_i}); \\ \hat{v}_{i+1}^{j_1 j_2 \cdots j_i k} & \text{otherwise} \end{cases} \tag{12}$$

We then set

$$\hat{v}_i^{j_1 j_2 \cdots j_i} = \frac{1}{b} \sum_{k=1}^{b} \hat{v}_{ik}^{j_1 j_2 \cdots j_i}. \tag{13}$$

The low estimator of the option price at the current time and state is $\hat{v}_o$.

Assume a random tree for pricing American call option is given in Figure 1. Please use the random tree in Figure 1 to calculate the high and low estimate of the American call option price with a **strike price** $K = 50$. Please show your steps for both the high estimator and low estimator.

Note: You do not need to write C++ program to get the price, but rather manually calculating the estimators following the algorithms introduced in class.

In [ ]:
```python
import math

def compute_terminal_nodes(S0, K, r, sigma, T, n):
    # Time delta per step
    dt = T / n
    # Compute up and down factors
    u = math.exp(sigma * math.sqrt(dt))
    d = 1 / u
    # Compute risk-neutral probability
    p = (math.exp(r * dt) - d) / (u - d)
    # Initialize terminal nodes list
    terminal_nodes = []

    # Calculate terminal nodes using the binomial model formula
    for i in range(n+1):
        # The stock price at the ith terminal node
        ST = S0 * (u ** i) * (d ** (n - i))
        # Payoff of a call option at the ith terminal node
        payoff = max(ST - K, 0)
        terminal_nodes.append(payoff)

    return terminal_nodes

# Test parameters
S0 = 50  # Current stock price
K = 50    # Strike price
r = 0.05  # Risk-free interest rate
sigma = 0.2  # Volatility
T = 1     # Time to expiration in years
n = 3     # Number of time steps

# Compute terminal nodes
terminal_nodes = compute_terminal_nodes(S0, K, r, sigma, T, n)
terminal_nodes
```

```
Out[ ]: [0, 0, 6.120045122283379, 20.699122904025828]
```

```
In [ ]: # Calculate the value at a non-terminal node
def calc_node_value(i, path, terminal_payoffs, n, r, dt, is_high_estimator=True):
    if i == n:
        return terminal_payoffs[len(path)]

    # Discount factor for one period
    discount_factor = math.exp(-r * dt)

    successors = [calc_node_value(i + 1, path + (j,), terminal_payoffs, n, r, dt, i
    h_i = max(S0 * (path.count(1) - path.count(0)) - K, 0)

    if is_high_estimator:
        # High estimator does not need discounting as we are taking expectations
        return max(h_i, sum(successors) / 2)
    else:  # Low estimator
        # Low estimator should be discounted back one period
        discounted_successors = [v * discount_factor for v in successors]
        continuation_value = sum(discounted_successors) - max(discounted_successors
        continuation_value /= (len(successors) - 1)
        return max(h_i, continuation_value)

# Calculate high estimator
high_estimator = calc_node_value(0, tuple(), terminal_nodes, n, r, T/n, True)

# Calculate low estimator
low_estimator = calc_node_value(0, tuple(), terminal_nodes, n, r, T/n, False)

(low_estimator, high_estimator)
```

```
Out[ ]: (19.689614767666036, 28.02434217801937)
```

Problem 2

The Black-Scholes formula is sufficiently complicated that there is no analytic inverse function, and therefore this inversion must be carried out numerically. Our objective is to use Newton-Ralphson method and the programming techniques to design a solver in a reusable fashion.

When we have a well-behaved function with an analytic derivative, then Newton-Ralphson can be used to find inverse values of the Black-Scholes formula. The idea of Newton-Raphson is that we pretend the function is linear and look for the solution where the linear function predicts it to be. Thus, we take a standing point x_0, and approximate f by

$$g_0(x) = f(x_0) + f(x - x_0)f'(x_0) \cdot \epsilon \tag{14}$$

We have that $g_0(x) = 0$ iff

$$x = \frac{y - f(x_0)}{f'(x_0)} + x_0 \tag{15}$$

We therefore take this value as our new guess x_1. We repeat until we find that $f(x_n)$ is within ϵ of y.

In this problem, you will use a **pointer to a member function** which is similar in syntax and idea to a function pointer, but it is restricted to methods of a single class. The difference in syntax is that the class name with a :: must be attached to the * when it is declared.

Thus, to declare function pointers called `Value` and `Derivative` which must point to methods of the class T, we have `double (T::*Derivative)(double) double` and `double (T::*Value) (double) double`. The function `Value(Derivative)` is a const member function which takes in a double as an argument and outputs a double as return value. If we have an object of class T called `TheObject` and y is a double, then the function pointed to can be invoked by either `The Object.*Value(y)` or `TheObject.*Derivative(y)`.

Use the classes introduced in Lecture 09 and finish the main.cpp file for European Call Option with the following parameters:

- $T = 1$
- $S_0 = 50$
- $K = 50$
- $r = 0.05$
- $q = 0.08$ (Dividend)
- $P = 4.23$ (Option Price)
- $\sigma_g = 0.23$ (Vol Estimate)

- Tolerance $= 0.0001$

Use the **NewtonRalphso** class from the sample as the starting point and find implied volatility for the given option price of 4.23.

```
// main.cpp

#include <iostream>
#include "BSCallClass.h"
#include "BlackScholesFormulas.h"
#include "NewtonRaphso.h"
#include "BSCallTwo.h"

using namespace std;

int main()
{
    double Expiry = 1.0;
    double Strike = 50.0;
    double Spot = 50.0;
    double r = 0.05;
    double d = 0.08; // This is the continuous dividend yield
    double Price = 4.23;

    double start; // This will be the starting guess for the implied
volatility
    cout << "Enter start volatility guess: ";
    cin >> start;

    double tolerance;
    cout << "Enter Tolerance for Newton-Raphson: ";
    cin >> tolerance;

    // Create function object for a vanilla call with all given parameters
    BSCallTwo theCall(r, d, Expiry, Spot, Strike);

    // Use Newton-Raphson method to find the implied volatility
    double vol = NewtonRaphson<BSCallTwo, &BSCallTwo::Price,
&BSCallTwo::Vega>(Price, start, tolerance, theCall);

    cout << "\nUsing Newton-Raphson method:\n";
    cout << "Implied vol is: " << vol << endl;

    // Check the price using BlackScholesCall with the implied vol
    double PriceTwo = BlackScholesCall(Spot, Strike, r, d, vol, Expiry);

    cout << "Option price by implied volatility is: " << PriceTwo << endl;

    return 0;
}
```

```
Enter start volatility guess: 0.23
Enter Tolerance for Newton-Raphson: 0.0001

Using Newton-Raphson method:
Implied vol is: 0.262937
Option price by implied volatility is: 4.23005
```