C++ in Quantitative Finance part #1

Submitted by: Zimin Luo 417124

Objective

The objective of this project is to use Monte Carlo simulations to approximate the price of the up-and-out put option with barrier. The effect of different barrier levels and number of path generated are also briefly analyzed.

Characteristics of the Option

Up-and-out option is the option that starts at a price that's lower than the barrier price and terminates whenever the option price reaches the barrier price or maturity. Due to the inclusion of the barrier, the European option becomes path-dependent, hence the entire path needs to be generated.

The payoff of a **PUT UP-AND-OUT** barrier option is:

$$\Phi_t = \max(X - S_T, 0) \quad \text{if} \quad \max_{0 \le t \le T}(S_t) \le L$$

where:

- Φ_t is the payoff at time t
- X is the strike price
- S_T is the price at maturity
- T is time to maturity
- L is the barrier

The following assumptions are made:

- prices of the underlying instrument follow the log-normal distribution
- distribution parameters μ and σ are constant,
- no transaction costs and no taxes,
- it is possible to purchase or sell any amount of stock or options or their fractions at any given time
- underlying instrument does not pay dividend
- risk-free arbitrage is not possible
- trading is continuous
- traders can borrow and invest their capital at the risk-free interest rate
- risk-free interest rate r is constant

Code

Parameters

The following notation and default values are used for the purpose of this report:

- `spot`: price of the underlying at the moment of option pricing: $S_0 = 145$
- `strike`: strike price: $K = 150$
- `vol`: annualized voltality rate: $\sigma = 25\%$
- `r`: annualized risk-free rate: $r = 5\%$
- `expiry`: time to maturity: $t = 1$ (one year)

Barrier level and the number of path to be generated are defined by the user.

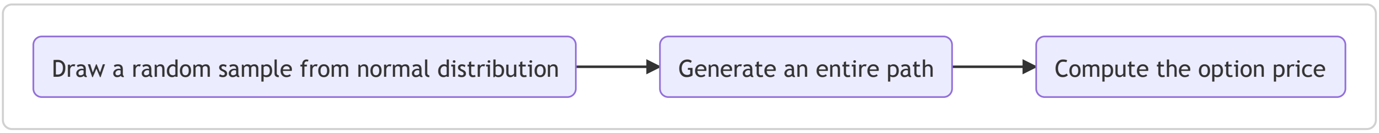
File Structure

There are total five `cpp` files in the directory:

1. `myMonteCarloProject.cpp`: contains the main function
2. `myEuroOptionBarrier.cpp`: contains the class and methods of `EuroOptionBarrier`
3. `myEuroOptionBarrier.h`: the header file for `myEuroOptionBarrier.cpp`
4. `Random1.cpp`: draws a random value from the normal distribution via Box-Muller transform
5. `Random1.h`: the header file for `Random1.h`

The Algorithm

The overall procedure is described in the following flowchart:



First, draw a random value x from the normal distribution $x \sim N(0, 1)$

```
// Random1.cpp

double getOneGaussianByBoxMueller(){
    double result;

    double x;
    double y;

    double sizeSquared;
    do {
        x = 2.0*rand()/static_cast<double>(RAND_MAX)-1;
        y = 2.0*rand()/static_cast<double>(RAND_MAX)-1;
        sizeSquared = x*x + y*y;
    }
    while
        ( sizeSquared >= 1.0);

    result = x*sqrt(-2*log(sizeSquared)/sizeSquared);

    return result;

}
```

Second, generate the entire path based on the following formula, where $N(0,1)$ is the random value that we obtained from the first step.

$$S_t = S_0 e^{(r-\frac{1}{2}\sigma^2)T/n+\sigma\sqrt{T/n}N(0,1)}$$

where:

- `thisDrift` is $(r-\frac{1}{2}\sigma^2)T/n$
- `cumShocks` is the cumulative sum of $(r-\frac{1}{2}\sigma^2)T/n+\sigma\sqrt{T/n}N(0,1)$
- `thisPath` appends every $S_0 e^{\text{cumShocks}}$ to the end

```cpp
// EuroBarrierOption.cpp

void EuroBarrierOption::generatePath(){
    // declaration of variables
    double thisDrift = (r * expiry - 0.5 * vol * vol * expiry) / double(nInt);
    double cumShocks = 0;

    // clear the possible `thisPath` stored in memory
    thisPath.clear();

    for (int i = 0; i < nInt; i++) {
        cumShocks += (thisDrift + vol * sqrt(expiry / double(nInt)) *
        getOneGaussianByBoxMueller());
        // add to the end of the path
        thisPath.push_back(spot * exp(cumShocks));

    }
};
```

The European UP-AND-OUT PUT option is computed in the following

```cpp
// EuroBarrierOption.cpp

double EuroBarrierOption::getEuroBarrierUNOPutPrice(int nReps){
    // declaration of variables
    double rollingSum = 0.0;
    double thisLast = 0.0;
    // creates a loop that repeats for `nRep` times
    for (int i=0; i<nReps; i++) {
        // generates a path
        generatePath();
        // get the maximum value in the path
        double thisMax = *max_element(thisPath.begin(), thisPath.end());
        // get the last value in the path
        thisLast = thisPath[thisPath.size() - 1];
        // if the last price < strike price, and
        // the maximum value in the path is < the barrier price
        // the payoff is strike price - last price
        // otherwise the payoff is 0
        rollingSum += (( thisLast < strike ) && ( thisMax < barrier )) ?
        ( strike - thisLast ) : 0;
    }
    // calculates the average price
    // and multiplies it by $e^{-rT}$
    return exp(-r * expiry) * rollingSum / double(nReps);

}
```

In the main file `myMonteCarloProject.cpp`, a new instance of the class `EuroBarrierOption` is created and the method `getEuroBarrierUNOPutPrice` is called to calculate the option price.

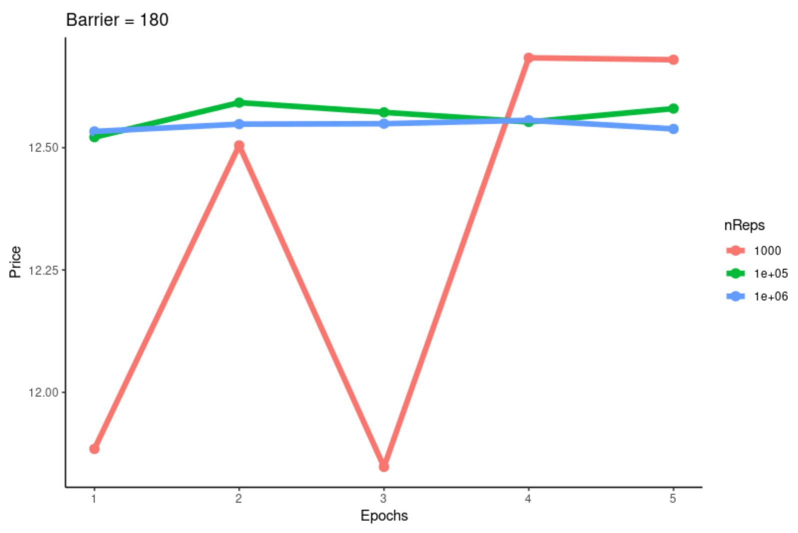
```
// myMonteCarloProject.cpp

int main(){
    int nInt = 252; // number of intervals: assume 252 trading days in a year
    double Strike = 150.0; // strike price
    double Spot = 145.0; // price of the underlying instrument
    double Vol = 0.25; // volatility
    double Rfr = 0.05; // risk-free rate
    double Expiry = 1.0; // time to maturity (1 = one year)
    double Barrier; // barrier level → to be defined by user
    int nReps; // number of paths generated → to be defined by user

    // set the seed for reproducible results
    srand( time(NULL) );

    // ask user to set the barrier level and nReps
    cout << "Enter the barrier level: ";
    cin >> Barrier;

    cout << "Enter the number of paths to be generated: ";
    cin >> nReps;

    // create a new instance of the EuroBarrierOption class named myEuro
    EuroBarrierOption myEuro(nInt, Strike, Spot, Vol, Rfr, Expiry, Barrier);

    // call the up-and-out Put method to get option price
    double uno = myEuro.getEuroBarrierUNOPutPrice(nReps);

    // display the result
    cout << "The up-and-out put option price is: "<< uno << endl;


    return 0;
}
```

Results

Constant barrier and increasing nReps

- When the `nReps = 1000` and `barrier=180` the results: 11.8843, 12.5046, 11.8476, 12.6837, 12.6797

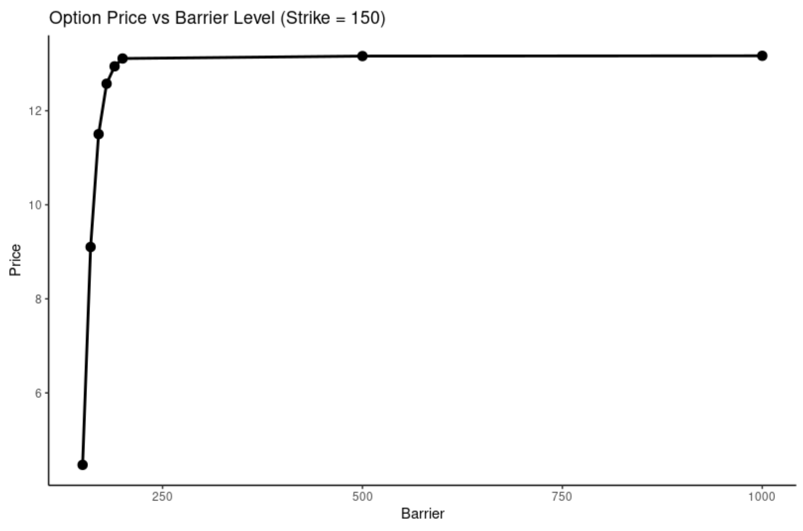
- When the `nReps = 100000` and `barrier=180` the results: 12.5215, 12.5921, 12.5722, 12.5526, 12.5799
- When the `nReps = 1000000` and `barrier=180` the results: 12.5332, 12.5481, 12.549, 12.556, 12.5383



Constant nReps and varying barrier

Let `nReps = 1000000`:

- `barrier = 150`: 4.47014
- `barrier = 160`: 9.10273
- `barrier = 170`: 11.5022
- `barrier = 180`: 12.5739
- `barrier = 190`: 12.943
- `barrier = 200`: 13.1098
- `barrier = 500`: 13.1614
- `barrier = 1000`: 13.1674



From the above result, I have the following observations:

1. As the number of paths generated increases, the results become less volatile. Comparing the three results in the line plot, it is clear to see that when `nReps` is only 1000 (red line), the approximation varies between

11 and 13; the blue and green lines are much smoother, meaning that their values fluctuate within a much smaller interval. This is inline with my expectations because of the law of large numbers.

2. As the barrier level increases, the price of the option also increases. However, when the barrier level reaches a certain value, its impact on the option price diminishes and the option price converges. This is also inline with my expectation as the higher the barrier level is, the less chance the path will reach it.

> Honor Code
> *In accordance with the Honor Code, I certify that my answers here are my own work, and I did not make my solutions available to anyone else.*