Sami BEN AMMAR
Pierre DEVELTER
Pierre-Adrien VERGEAU

Project Report : Fix oscillations in binomial tree engines

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

I. Project description

A widespread method for computing option prices numerically is to use binomial trees. For an option of maturity *T* and *n* timesteps, a tree containing n level is built, where the ith level represent the state of the option and the underlying at time *T*n/i*. An assumption is made than at each time step, the underlying price can go up by *u* or down by *d*.

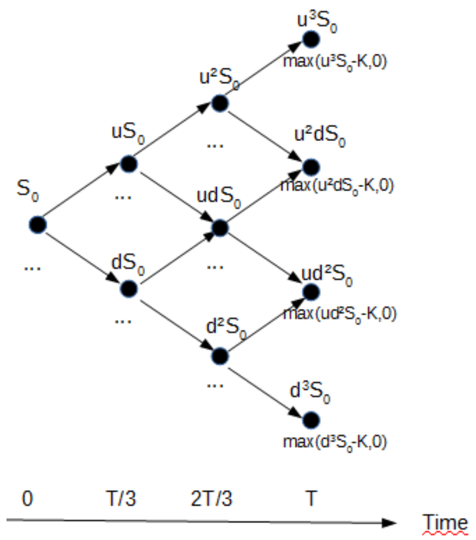

Figure 1 : Example of a Binomial tree with n=4

At the last level, the option prices can easily be computed as max(*P*-*K*,0), where *P* is the price of the underlying and *K* is the strike.

It is actually possible to go back a level in computing the option price using a no-arbitrage argument : it is shown that
$f = e^{-rT}[pf_u + (1-p)f_d]$ with $p=(e^{rT}-d)/(u-d)$ where f is the price of the option at the current node, and f_u and f_d are the prices of the option at following up and down nodes.

Repeting this calculation several times enable pricing the option at current time.

The choice of parameters *u*, *d* and the probability of going up or down at each step depends on the volatility, the risk-free interest rate, the time step and the chosen model.

This is a serviceable way to compute option prices. However, it suffers a problem : as the time step increases, the option prices oscillate around a value, which makes it longer to reach a certain precision.

Step	Value	
	360	12,67449686
	361	12,67902568
	362	12,685347
	363	12,68132474
	364	12,67514563
	365	12,68159961
	366	12,68515567
	367	12,68884421
	368	12,68505241
	369	12,68210949

Table 1 : Oscillation of Cox-Ross-Rubinstein tree for different number of time steps, with parameters S=100, q=0.0, r=0.03, sigma=0.2, k=110.0, T=1

Our goal during this project was to fix that by making the oscillations stop when option prices are computed using Quantlib's `BinomialVanillaEngine` class. To do this, we implemented a solution found by Chung and Shackleton in [1], which consists in setting the option price to the analytical values Black-Scholes values of a European option at the *n-1*th node. We then studied some characteristics of this solution, such as the number of steps needed to attain convergence and the time taken for the computation to be made.

II. Implementation details

To implement this new method, we modified the `BinomialVanillaEngine` class. It now has a boolean attribute named `no_osc`, which decides whether or not the new method is to be applied. It is set when calling the constructor.

If `no_osc` is set to true, then we rollback to the second last step and computes the analytical BS formula for its nodes. The Black-Scholes formula is computed thanks to a function named bsformula, which is implemented in another file, binomialformula.cpp, which we have included.

```
if (no_osc_){
    option.rollback(grid[timeSteps_-1]);

    //bool btype = true; // true = CALL
    double dividendyield = q;

    bool btype = (payoff->optionType() == Option::Call);


    for(int i=0;i<=timeSteps_-1;++i){
        double bs_value = bsformula(lattice->underlying(timeSteps_-1, i), payoff->strike(), maturity-grid[timeSteps_-1], r, v, btype, dividendy
        option.values()[i] = bs_value;
    }
}
```

Figure 2 : the changes made to the `BinomialVanillaEngine` *class*

```
double bsformula(double S, double K, double T,
                 double r, double sigma, bool call, double q) {
    double d1 = (1/(sigma*sqrt(T))) * (log(S/K) + (r-q+sigma*sigma/2)*T);
    double d2 = d1 - sigma*sqrt(T);

    double result;
    if (call) {
        result = N(d1)*S*exp(-q*T) - N(d2)*K*exp(-r*T);
    } else {
        result = N(-d2)*K*exp(-r*T) -N(-d1)*S*exp(-q*T);
    }
    return result;
}
```

Figure 3 : The Black-Scholes Formula Computation

III. Experiments and result

At first, it was made sure that the pricing scheme was working as intended. We checked that with our implementation, the oscillations disappeared, for different kind of trees . They all compute the parameters *u* and *d* in different ways :

Step	Value, no_osc = False	Value, no_osc = True
360	12,67449686	12,68312372
361	12,67902568	12,68306574
362	12,685347	12,6831157
363	12,68132474	12,68305852
364	12,67514563	12,68310799
365	12,68159961	12,68305395
366	12,68515567	12,68309911
367	12,68884421	12,68304846
368	12,68505241	12,68308896
369	12,68210949	12,68304378

Table 2 : Cox-Ross-Rubinstein tree for different number of time steps, with previous parameters

Step	Value, no_osc = False	Value, no_osc = True
360	12,67894614	12,68313769
361	12,67541614	12,68318579
362	12,68187914	12,68313385
363	12,68519331	12,68317575
364	12,67882015	12,683132
365	12,68508975	12,68316703
366	12,68233233	12,68313012
367	12,67619844	12,68315764
368	12,68254214	12,6831268
369	12,68486484	12,68314718

Table 3 : Jarrow-Rudd tree for different number of time steps, with previous parameters

Table 3 : Jarrow-Rudd tree for different number of time steps, with previous parameters

Step	Value, no_osc = False	Value, no_osc = True
360	12,67640824	12,68277667
361	12,67794436	12,68276507
362	12,68421308	12,68276217
363	12,6828218	12,68275281
364	12,67633949	12,68275421
365	12,68276124	12,68274622
366	12,68425773	12,68275035
367	12,67809943	12,6827444
368	12,68428511	12,68275054
369	12,68265345	12,68274606

Table 4 : Tian tree for different number of time steps, with previous parameters

We can see that the implemented princing scheme doesn't completely remove the oscillations, but it greatly reduces them for all the tree types : for the Cox-Ross-Rubinstein and Jarrow-Rudd models, the amplitude of oscillations is reduced by a hundred approximately, and the decrease is even greater for the Tian model.

This makes the option prices converge more quickly in terms of steps needed : to reach a precision of 0.0001 it originally took 387 steps with a Cox-Ross-Rubinstein model. Now it only takes 71 steps, which represents a 81,7 % decrease in the number of steps needed.

It does come at a small computational cost however : for 1000 steps, the former method took approximately 0,02159 seconds to compute. It now takes 0,02168 seconds, which is a little less than 1 more microsecond.