FE545 Design, Patterns and Derivatives Pricing

Trees

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

03/19/2024

Overview

Price Options using Binomial Tree

- The Black-Scholes model uses continuous-time stochastic process methods that interfere with understanding the simple intuition underlying these models.
- We allow the possibility of variable interest rates and dividend rates, as well as variable but deterministic volatility. The stock price therefore follows the process

  $$dS_t = (r(t) - d(t))S_t dt + \sigma(t) S_t dW_t, \tag{1}$$

  with S_0 given.
- The value of a European option with expiry T is then

  $$e^{-rT}\mathbb{E}(C(S,T)).$$

  where $C(S,T)$ is the final pay-off.

- We will start instead with the binomial option pricing model of Cox, Ross, and Rubinstein, which captures all of the economics of the continuous time model but is simple to understand and program.
- The dynamics of the stock price under geometric Brownian motion are such that

$$S_t = S_0 e^{(r-d-\frac{1}{2}\sigma^2)t+\sigma W_t}, \tag{2}$$

  where μ and σ are constant, and dW_t is a Brownian motion.
- We wish to discretize W_t. We have N steps to get from 0 to T. Each time step is therefore of length T/N. Across step I, we need to approximate

$$W_{(I+1)T/N} - W_{IT/N} = \sqrt{\frac{T}{N}} N(0,1)$$

Price Options using Binomial Tree

- There is only one random variable taking precisely two values which has the same mean and variance as $N(0,1)$, and this variable takes $+/-1$ with probability $1/2$
- We therefore take a set of N independent random variables X_j with this distribution, and approximate $W_{lT/N}$ by

$$Y_l = \sqrt{\frac{T}{N}\sum_{j=1}^{l} X_j} \tag{3}$$

- The approximation for $S_{lT/N}$ is then

$$S_0 e^{(r-d-\frac{1}{2}\sigma^2)lT/N+\sigma Y_l} \tag{4}$$

- Note the crucial point here that since the value of S_t does not depend on the path of W_t but solely upon its value at time t, it is only the value of Y_l that matters not the value of each individual X_j.

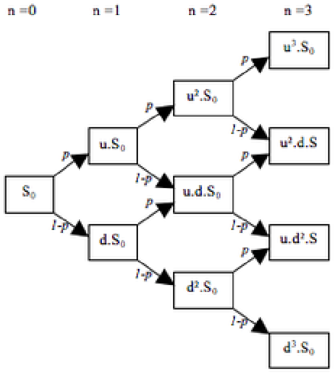
Price Options using Binomial Tree

- This is crucial because it means that our tree is recombining; it does not matter whether we go down then up, or up then down. This is not the case if we allow variable volatility, which is why we have assumed its constancy.
- The nature of martingale pricing means that the value at a given time-step is equal to the discounted value at the next time-step, thus if we let $S_{l,k}$ be the value of the stock at time Tl/N if Y_l is k, we have that

$$\begin{aligned} C(S_{l,k}, Tl/N) &= e^{-rT/N}\mathbb{E}(S_{l+1}(Y_{l+1}|Y_l = k)), \\ = \frac{1}{2}e^{-rT/N}(C(S_{l,k}&e^{(r-d-1/2\sigma^2)T/N+\sigma\sqrt{T/N}}, (l+1)T/N) \\ &+C(S_{l,k}e^{(r-d-1/2\sigma^2)T/N-\sigma\sqrt{T/N}}, (l+1)T/N)) \end{aligned} \quad (5)$$

- Note that we are not doing true martingale pricing in the discrete world in that we are not adjusting probabilities to make sure the assets are discrete martingales;

- The reason trees are introduced was that they are given an effective method for pricing American options. The analysis for a American option is similar except that the value at a point in the tree is the maximum of the exercise value at that point and the discounted expected value at the next time.
- The algorithm for pricing an American option is as follows:
  (i) Create an array of final spot values which are of the form
  $$S_0 e^{(r-d-1/2\sigma^2)T+\sigma\sqrt{T/N}j} \tag{6}$$
  where j ranges from $-N$ to N.
  (ii) For each of these spot values evaluate the pay-off and store it.
  (iii) At the previous time-slice compute the possible values of spot:
  $$S_0 e^{(r-d-1/2\sigma^2)(N-1)T/N+\sigma\sqrt{T/N}j} \tag{7}$$
  where j ranges from $-(N-1)$ to $N-1$.
  (iv) For each of these values of spot, compute the pay-off and take the maximum with the discounted pay-off of the two possible values of spot at the next time.
  (v) Repeat 3, 4 until time zero is reached.



$$p = \frac{e^{rt/n} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t/n}}$$

$$d = e^{-\sigma\sqrt{t/n}}$$

Figure: Cox, John C., Stephen A. Ross, and Mark Rubinstein (1979) "Option Pricing: A Simplified Approach" Journal of Financial Economics 7, 229-263

- We could do a barrier option or an American option that could only be exercised within certain period of time. For a knock-out barrier option, the procedure would be the same as for the European option, except that the value at a point in the tree would be zero if it lay behind the barrier.
- For an American option with limited early exercise the procedure would be the same again, except that we would only take the maximum at times at which early exercise was allowed.
- Note that in our formulation, we have not used any no-arbitrage arguments. The reason is that we have implicitly assumed that the no-arbitrage arguments have been done in the continuous case before any discretatization has been carried out.
- This means that the completeness property of a binomial tree, i.e. that it generates a unique no-arbitrage price, is not relevant.

The Design

- Here are some concepts that our discussion has thrown up:
  1. the discretization;
  2. the final pay-off of the option;
  3. the rule for deciding the value of an option at a point in the tree given spot and the discounted future value of the option.
- We have an obvious orthogonalization: we have a tree class which handles the discretization, and a second class which deals with the final pay-off and the rule at previous times.
- There are a number of ways we could approach the second class. We could inherit from *PayOffBridge* since we could view our class as adding structure to an existing class.
- Another approach might be simply to define a second class to do the calculation rule, and plug both the final pay-off and the calculation rule into the tree.
- Ultimately, the pay-off is an aspect of the option, and it therefore makes more sense to define it as data member of the class which can be referenced via a final pay-off method.

The Design

- Therefore we define options on trees via an abstract base class which has three defining methods:
  i *FinalTime* indicates the expiry time of the option;
  ii *FinalPayOff* gives the final pay-off a function of spot;
  iii *PreFinalValue* gives the value at a point in the tree as a function of spot, time and the discounted future value of the option.
- Note by defining the option class in this fashion, we have not allowed it to know anything about interest rates nor the process of the underlying.
- This means it can be used in any tree-like structure provided the structure is always in a position to let it know its own discounted future value.
- For Monte Carlo, the history is easy, but the discounted future value of an option is hard; while on a tree the discounted future value is easy, but the history is hard.

The TreeProducts Class

TreeProducts.h

```
#ifndef TREE_PRODUCTS_H
#define TREE_PRODUCTS_H

class TreeProducts
{
public:

    TreeProducts(double FinalTime_);
    virtual double FinalPayOff(double Spot) const=0;
    virtual double PreFinalValue(double Spot,
                                 double Time,
                                 double DiscountedFutureValue) const=0;
    virtual ~TreeProducts(){}
    virtual TreeProducts* clone() const=0;
    double GetFinalTime() const;

private:
    double FinalTime;

};
```

The TreeProducts Class

TreeProducts.cpp

```
#include <TreeProducts.h>

TreeProducts::TreeProducts(double FinalTime_)
: FinalTime(FinalTime_)
{
}

double TreeProducts::GetFinalTime() const
{
    return FinalTime;
}
```

The TreeAmerica Class

TreeAmerica.h

```
#include "TreeProducts.h"
#include "PayOffBridge.h"

class TreeAmerican : public TreeProducts
{

public:

    TreeAmerican(double FinalTime,
                 const PayOffBridge& ThePayOff_);

    virtual TreeProducts* clone() const;
    virtual double FinalPayOff(double Spot) const;
    virtual double PreFinalValue(double Spot,
                                 double Time,
                                 double DiscountedFutureValue) const;
    virtual ~TreeAmerican(){}

private:

    PayOffBridge ThePayOff;

};
```

The TreeEuropean Class

TreeEuropean.h

```
#include "TreeProducts.h"
#include "PayOffBridge.h"

class TreeEuropean : public TreeProducts
{

public:

    TreeEuropean(double FinalTime,
                 const PayOffBridge& ThePayOff_);

    virtual TreeProducts* clone() const;
    virtual double FinalPayOff(double Spot) const;
    virtual double PreFinalValue(double Spot,
                                 double Time,
                                 double DiscountedFutureValue) const;
    virtual ~TreeEuropean(){}

private:

    PayOffBridge ThePayOff;

};
```

The TreeAmerican Class

TreeAmerican.cpp

```
#include <TreeAmerican.h>
#include <minmax.h>

TreeAmerican::TreeAmerican(double FinalTime,
                           const PayOffBridge& ThePayOff_)
                 : TreeProducts(FinalTime),
                   ThePayOff(ThePayOff_)
{
}

TreeProducts* TreeAmerican::clone() const
{
     return new TreeAmerican(*this);
}

double TreeAmerican::FinalPayOff(double Spot) const
{
    return ThePayOff(Spot);
}

double TreeAmerican::PreFinalValue(double Spot,
                                 double , // Borland compiler doesnt like unuse
                                 double DiscountedFutureValue) const
```

The TreeEuropean Class

TreeEuropean.cpp

```
#include <TreeEuropean.h>
#include <minmax.h>

TreeEuropean::TreeEuropean(double FinalTime,
                           const PayOffBridge& ThePayOff_)
                  : TreeProducts(FinalTime),
                    ThePayOff(ThePayOff_)
{
}

double TreeEuropean::FinalPayOff(double Spot) const
{
    return ThePayOff(Spot);
}

double TreeEuropean::PreFinalValue(double, //Spot, Borland compiler
                                 double, //Time,   doesnt like unused named var
                                 double DiscountedFutureValue) const
{
    return DiscountedFutureValue;
}

TreeProducts* TreeEuropean::clone() const
```

A Tree Class

- We design the tree to work in three pieces.
  (a) The constructor does little except store the parameters.
  (b) The *BuildTree* method actually makes the tree.
  (c) The method *GetThePrice* to calculate the price.
- Our design is such that we can price multiple products with the same expiry; we call the method multiple times and only build the tree once.
- As we wish to be able to do this, we store the entire tree. Note this is not necessary since for any given time slice, we only need the next time slice and so we could easily save a lot of memory.

A Tree Class

BinomialTree.h

```
#include <TreeProducts.h>
#include <vector>
#include <Parameters.h>
#include <Arrays.h>


class SimpleBinomialTree
{

public:
 SimpleBinomialTree(double Spot_,
                       const Parameters& r_,
                       const Parameters& d_,
                       double Volatility_,
                       unsigned long Steps,
                       double Time);

 double GetThePrice(const TreeProducts& TheProduct);

protected:

    void BuildTree();
```

A Tree Class

BinomialTree.cpp

```
void SimpleBinomialTree::BuildTree()
{
    TreeBuilt = true;
    TheTree.resize(Steps+1);

    double InitialLogSpot = log(Spot);

    for (unsigned long i=0; i <=Steps; i++)
    {

        TheTree[i].resize(i+1);

        double thisTime = (i*Time)/Steps;

        double movedLogSpot =
                            InitialLogSpot+ r.Integral(0.0, thisTime)
                                          - d.Integral(0.0, thisTime);

        movedLogSpot -= 0.5*Volatility*Volatility*thisTime;

        double sd = Volatility*sqrt(Time/Steps);

        for (long j = -static_cast<long>(i), k=0; j <= static_cast<long>(i); j=
```

A Tree Class

BinomialTree.cpp

```
double SimpleBinomialTree::GetThePrice(const TreeProducts& TheProduct)
{
    if (!TreeBuilt)
        BuildTree();

    if (TheProduct.GetFinalTime() != Time)
        throw("mismatched product in SimpleBinomialTree");

    for (long j = -static_cast<long>(Steps), k=0; j <=static_cast<long>( Steps)
        TheTree[Steps][k].second = TheProduct.FinalPayOff(TheTree[Steps][k].fir

    for (unsigned long i=1; i <= Steps; i++)
    {
        unsigned long index = Steps-i;
        double ThisTime = index*Time/Steps;

        for (long j = -static_cast<long>(index), k=0; j <= static_cast<long>(in
        {
            double Spot = TheTree[index][k].first;
            double futureDiscountedValue =
                            0.5*Discounts[index]*(TheTree[index+1][k].second+Th
            TheTree[index][k].second = TheProduct.PreFinalValue(Spot,ThisTime,f
        }
```

Breakout Exercise: Create a Tree American Option Pricer

- Use the classes introduced in class and finish the TreeMain1.cpp file to price American Option with the following parameters:
  - Expiry: 1
  - Spot: 50
  - Strike: 50
  - Risk free rate: 0.05
  - Dividend: 0.08
  - Vol: 0.3
  - NumberOfPeriods: 50
- Use the SimpleBinomialTree class from the sample as the base.

Key Points

- Tree pricing is based on the discretization of a Brownian motion
- Trees are a natural way to price American options
- On a tree, knowledge of discounted future values is natural but knowing about the past is not.
- We can re-use the pay-off class when defining products on trees.
- By having a separate class encapsulating the definition of a derivative on a tree, we can re-use the products for more general structures.
- European options can be used as controls for American options.