Binomial Tree Models

QF607 Numerical Methods

Zhenke Guan
zhenkeguan@smu.edu.sg

Pricing Financial Derivatives

Flow or Vanilla options that can be priced analytically

- Swap / Forward / Futures
  - Linear products, priced by "Law of One Price"

- European option - an option that may only be exercised on expiry date, priced by Black-Scholes analytic formulas

Pricing Financial Derivatives

Exotic options require numerical pricer

- **American option** - option that allows exercise any time prior to the expiry date
- **Barrier option** - if the spot price touches the pre-defined barrier in a given time window, the option holder obtains (Knock-In Option) or loses (Knock-Out Option) an underlying payoff
  - the underlying payoff can be a European option,
  - or just a constant rebate (touch option)
- **Asian option** - an option where the payoff is determined by the average underlying price over some pre-set period of time
  - average of price has lower volatility, therefore a cheaper option for hedging purpose,
  - less sensitive to price manipulation on the expiry date
- Target redemption forward, Bermudan callable structure, etc.

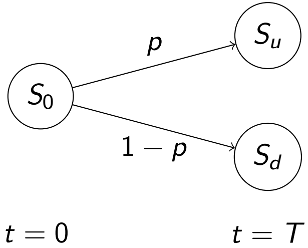
Binomial tree option pricing model

- A numerical method for the valuation of options.

- The model uses a discrete-time model of varying price over time of the underlying financial instrument.

- First proposed by Cox, Ross and Rubinstein in 1979 – the CRR binomial tree

One-Step Binomial Tree



- A single time step from 0 to T
- Two traded instruments in the market: stock S and a zero coupon bond with continuous yield r (risk free interest rate)
- Two possible states at time T: up and down
- What's the price of the call and put option at time 0?

Price by Replication

Consider an option with final payoff $V(S_T, T)$, we want to solve for it's present value $V(S_0, 0)$:

- Construct a portfolio of δ shares of the stock and $V(S_0, 0) - \delta S_0$ units of bond that grows at constant compounding rate r.

- At time T, we would like the portfolio to have value $V(S_T, T)$ no matter whether S_0 goes to the up state or the down state, so the below equations should hold

$$\begin{cases} \delta S_u + e^{rT}(V_0 - \delta S_0) = V_u \\ \delta S_d + e^{rT}(V_0 - \delta S_0) = V_d \end{cases} \tag{1}$$

Solution and Risk Neutral Probabilities

We know S_u, S_d, V_u and V_d in the equations, so two unknowns δ, V_0 and two equations we can solve

$$\begin{cases} \delta = \frac{V_u - V_d}{S_u - S_d} \\ V_0 = e^{-rT} \left(\underbrace{\frac{S_u - S_0 e^{rT}}{S_u - S_d}}_{q_d} V_d + \underbrace{\frac{S_0 e^{rT} - S_d}{S_u - S_d}}_{q_u} V_u \right) \end{cases} \tag{2}$$

We call q_u and q_d risk neutral probabilities, and the option's price is expectation of V_T under the risk neutral probability measure ($\mathbb{Q}$-measure):

$$V_0 = e^{-rT} \mathbb{E}_{\mathbb{Q}}[V_T] \tag{3}$$

Why Binomial Model?

Pros:

- Overly simplified, but surprisingly general after extensions
- More final states can be included with multiple steps
- Handle many payoffs and option types – the only assumption of the payoff V we make is that it depends on the terminal value of S_T
- Handle American options naturally
- Easy to implement

Cons:

- Difficult to handle path-dependent options

How Easy Is The Implementation

Inputs are:

- Current value of the underlying stock S_0
- Risk free interest rate r
- Up state S_u and down state S_d
- Option type: Call or Put for Now
- Option strike K and time to maturity T

Output: option price

One Step Binomial Tree Implementation

```python
from enum import Enum
import math
class PayoffType(str, Enum):
    Call = 'Call'
    Put = 'Put'

def oneStepBinomial(S:float, r:float, u:float, d:float, optType:PayoffType, K:
     float, T:float) -> float:
    p = (math.exp(r * T) - d) / (u-d)
    if optType == PayoffType.Call:
        return math.exp(-r*T) * (p*max(S*u-K, 0) + (1-p) * max(S*d-K, 0))
```

```
oneStepBinomial(S=100, r=0.01, u=1.2, d=0.8, optType=PayoffType.Call, K=105, T=1.0)
7.798504987524955
```

Let's recap the input of our simple one step binomial model:

- Option type, strike K and time to maturity T,
- Current value of the underlying stock S_0, risk free interest rate r
- Up state S_u and down state S_d
  — HOW can we possibly know S_u and S_d in reality?

But does it mean our binomial tree model is impractical? No, it is in fact a discretized version of Black-Scholes model

Black-Scholes Model

The Black-Scholes model says, under the risk neutral measure, the non-dividend paying stock price follows a log-normal process:

$$\frac{dS_t}{S_t} = rdt + \sigma dW_t \tag{4}$$

where

- σ is the volatility
- W_t is a standard Brownian motion
- r is the risk free interest rate

This is much closer to reality than the one step binomial model!

Black-Scholes Solution of the Stock Price

Applying Ito's lemma we can get the diffusion of $d\ln S_t$:

$$d\ln S_t = \frac{1}{S_t}dS_t - \frac{1}{2S_t^2}dS_t^2 = (r - \frac{1}{2}\sigma^2)dt + \sigma dW_t$$

So $\ln S_t$ is a drifted Brownian motion

$$\ln S_t = \ln S_0 + \int_0^t (r - \frac{1}{2}\sigma^2)ds + \int_0^t \sigma dW \tag{5}$$

$$= \ln S_0 + (r - \frac{1}{2}\sigma^2)t + \sigma W_t \tag{6}$$

And the solution of SDE (4) is

$$S_t = S_0 \exp\left((r - \frac{1}{2}\sigma^2)t + \sigma W_t\right) \tag{7}$$

Black-Scholes Formula

The present value ($t = 0$) of a call option with strike at K and expiry at T is

$$C(S_0, K, T) = e^{-rT}\mathbb{E}_{\mathbb{Q}}[(S_T - K)_+] = \ S_0 N(d_+) - Ke^{-rT}N(d_-) \qquad (8)$$

where

- r is the risk-free rate
- $d_{\pm} = \dfrac{ln\frac{S_0}{K} + (r \pm \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}$
- $N(\cdot)$ is the standard cumulative normal function

And the price of a put option is

$$P(S_0, K, T) = e^{-rT}\mathbb{E}_{\mathbb{Q}}[(K - S_T)_+] = \ Ke^{-rT}N(-d_-) - S_0 N(-d_+) \quad (9)$$

Black-Scholes Formula Implementation

Input:

- option type, strike, current stock price, volatility, and interest rate

```
def cnorm(x):
    return (1.0 + math.erf(x / math.sqrt(2.0))) / 2.0
def bsPrice(S, r, vol, payoffType, K, T):
    fwd = S * math.exp(r * T)
    stdev = vol * math.sqrt(T)
    d1 = math.log(fwd / K) / stdev + stdev / 2
    d2 = d1 - stdev
    if payoffType == PayoffType.Call:
        return math.exp(-r * T) * (fwd * cnorm(d1) - cnorm(d2) * K)
    elif payoffType == PayoffType.Put:
        return math.exp(-r * T) * (K * cnorm(-d2) - cnorm(-d1) * fwd)
    else:
        raise Exception("not supported payoff type", payoffType)
# test ---
S, r, vol, K, T, u, d = 100, 0.01, 0.2, 105, 1.0, 1.2, 0.8
print("blackPrice: ", bsPrice(S, r, vol, T, K, PayoffType.Call))
print("oneStepTree: ", oneStepBinomial(S, r, u, d, PayoffType.Call, K, T))
```

Do they agree on the price? Why not?

Where is the Gap?

The inputs for the option are the same, how about the market inputs?

- One step binomial tree model:
  - current stock price S_0,
  - risk free interest rate r,
  - up state S_u and down state S_d
- Black-Schole model:
  - current stock price S_0,
  - risk free interest rate r,
  - volatility σ

The two models are making different assumption on the distribution of the stock price at T.

Possible to Build the Bridge?

Distributions of S_T

- Binomial tree: two state discrete distribution.
  - Risk neutral probability of S_u: $p = \frac{S_0 e^{rT} - S_d}{S_u - S_d}$
  - Risk neutral probability of S_d: $1 - p$
  - Mean of S_t: $S_0 e^{rT}$
  - Variance of S_t: $pS_u^2 + (1-p)S_d^2 - S_0^2 e^{2rT}$
- Black-Scholes: continuous log-normal distribution

$$S_t = S_0 e^{(r - \frac{1}{2}\sigma^2)t + \sigma W_t} \tag{10}$$

  - Mean of S_t: $S_0 e^{rT}$
  - Variance of S_t: $S_0^2 e^{2rt} e^{\sigma^2 t} - S_0^2 e^{2rt}$

They have the same mean, so can we match the variance?

Matching the Variance

Stock price is always positive, we can rewrite $S_u = uS_0, S_d = dS_0$ with $u > 0, d > 0$
The variance of the binomial tree becomes

$$S_0^2(pu^2 + (1-p)d^2) - S_0^2 e^{2rt} \tag{11}$$

To match the variance with Black-Sholes's, we need to satisfy:

$$e^{2rt+\sigma^2 t} = pu^2 + (1-p)d^2, \quad p = \frac{e^{rT} - d}{u - d} \tag{12}$$

One equation and two unknowns — Let us impose another constraint $d = \frac{1}{u}$: CRR binomial tree

Equation (12) now becomes

$$e^{2rT+\sigma^2 T} = \frac{e^{rT}u^2 - u + 1/u - e^{rT}(1/u)^2}{u - 1/u} = e^{rT}(u + 1/u) - 1 \tag{13}$$

$$e^{rT}u^2 - (e^{2rT+\sigma^2 T} + 1)u + e^{rT} = 0 \tag{14}$$

$$u^2 - (e^{rT+\sigma^2 T} + e^{-rT})u + 1 = 0 \tag{15}$$

Solving the quadratic equation (14) we have

$$u = \frac{b \pm \sqrt{b^2 - 4}}{2}, \quad \text{where} \quad b = e^{rT+\sigma^2 T} + e^{-rT} \tag{16}$$

The two solutions correspond to u and d, and we expect $u > 1$ since its the up state, so

$$u = \frac{b + \sqrt{b^2 - 4}}{2} \tag{17}$$

$$d = \frac{b - \sqrt{b^2 - 4}}{2} = \frac{1}{u} \tag{18}$$

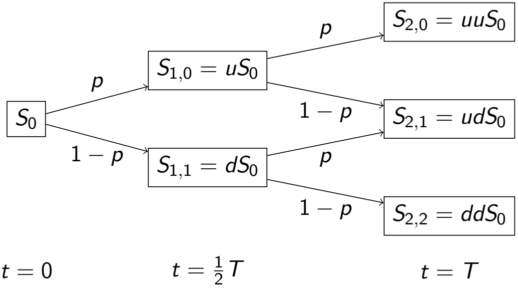
Now It Is Consistent

Now we can derive u and d from σ, making the input about the market the same for both pricers

```python
def oneStepBinomial2(S, r, vol, optType, K, T):
    b = math.exp(vol * vol * T+r*T) + math.exp(-r * T)
    u = (b + math.sqrt(b*b - 4)) / 2
    d = 1/u
    p = (math.exp(r * T) - d) / (u-d)
    if optType == PayoffType.Call:
        return math.exp(-r * T) * (p * max(S * u - K, 0) + (1-p) * max(S * d - K, 0))
# test ---
S,r,vol,K,T,u,d = 100, 0.01, 0.2, 105, 1.0, 1.2, 0.8
print("blackPrice: \t", bsPrice(S, r, vol, PayoffType.Call, K, T))
print("oneStepTree1: \t", oneStepBinomial(S, r, u, d, PayoffType.Call, K, T))
print("oneStepTree2: \t", oneStepBinomial2(S, r, vol, PayoffType.Call, K, T))
```

Bringing It Closer

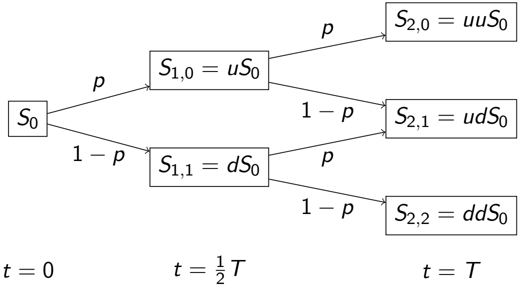
- Now we have first moment and second moment matched, have we closed the gap between the two pricers now? Sort of, but we are still subject to discretization errors – our one step tree is very coarse
- Natural way to extend our model is to make more steps and more states



Multi-Step Binomial Tree

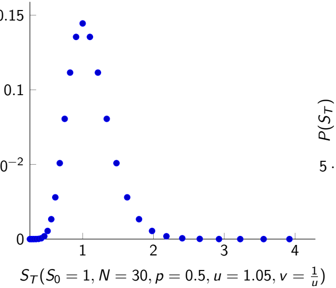
- Fortunately, our tree is **recombining**: the two intermediate states collapse to the same final state because $ud = du$. Otherwise the number of nodes will grow exponentially and will soon become unmanageable.
- The probability associated with each final state:

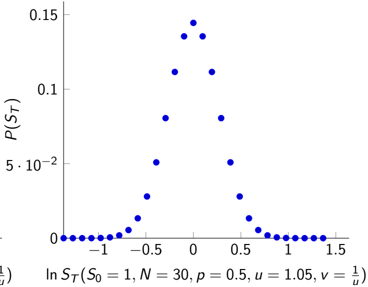
$$P(S_{2,0}) = p^2, P(S_{2,1}) = 2p(1-p), P(S_{2,2}) = (1-p)^2$$



Multi-Step Binomial Tree

- Adding one time step will give us one more final state
- The probability of the state $S_{N,i}$ is $P(S_{N+1,i}) = \binom{N}{i} p^i (1-p)^{N-i}$
- And the probability density function of S_T converges to a log-normal distribution:





- So, we can match as close as we want the Black-Scholes distribution by matching the first and second moments, and increasing the number of time steps

N-Step Binomial Tree — First Moment

$$
\begin{tikzcd}
 & & \boxed{\mathbb{E}[S_T|S_{2,0}] = u^2 S_0} \arrow[dl, "p"'] \\
 & \boxed{\mathbb{E}[S_T|S_{1,0}] = uS_0(pu + (1-p)d)} \arrow[dl, "p"'] & \\
\boxed{\mathbb{E}[S_T|S_{0,0}]} & & \boxed{\mathbb{E}[S_T|S_{2,1}] = udS_0} \arrow[ul, "1-p"] \arrow[dl, "p"'] \\
 & \boxed{\mathbb{E}[S_T|S_{1,0}] = dS_0(pu + (1-p)d)} \arrow[ul, "1-p"] & \\
 & & \boxed{\mathbb{E}[S_T|S_{2,2}] = d^2 S_0} \arrow[ul, "1-p"] \\
t = 0 & t = \frac{1}{2}T & t = T
\end{tikzcd}
$$

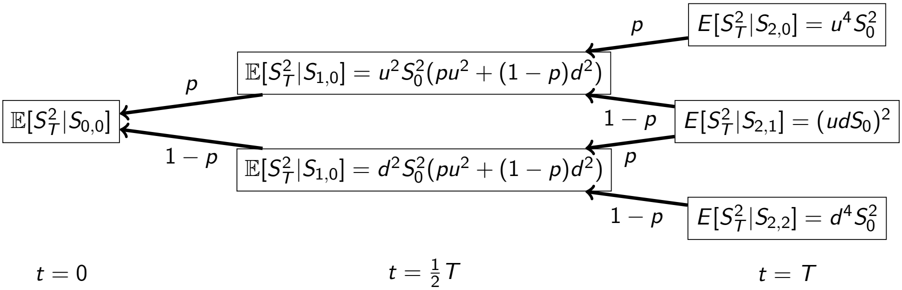
For two step model

$$\mathbb{E}[S_T|S_0] = S_0(pu + (1-p)d)^2 \tag{19}$$

By induction for N step model

$$\mathbb{E}[S_T|S_0] = S_0 \times (pu + (1-p)d)^N \tag{20}$$

N-Step Binomial Tree — Second Moment



For two step model

$$\mathbb{E}[S_T^2|S_0] = S_0^2 (pu^2 + (1-p)d^2)^2 \tag{21}$$

By induction for N step model

$$\mathbb{E}[S_T^2|S_0] = S_0^2 \times (pu^2 + (1-p)d^2)^N \tag{22}$$

N-Step Binomial Tree — Matching With Black-Scholes

We split the N step in equal space, so each time step is $\Delta_t = \dfrac{T}{N}$

- To match the mean we just need the probability p to be risk neutral:

$$p = \frac{S_0 e^{r\Delta_t} - dS_0}{uS_0 - dS_0} = \frac{e^{r\Delta_t} - d}{u - d} \tag{23}$$

  then

$$\mathbb{E}[S_t|S_0] = S_0(pu + (1-p)d)^N = S_0 e^{r\Delta_t N} = S_0 e^{rT} \tag{24}$$

- Similarly, we just need to replace T by Δ_t in our one step binomial model so as to match the second moment:

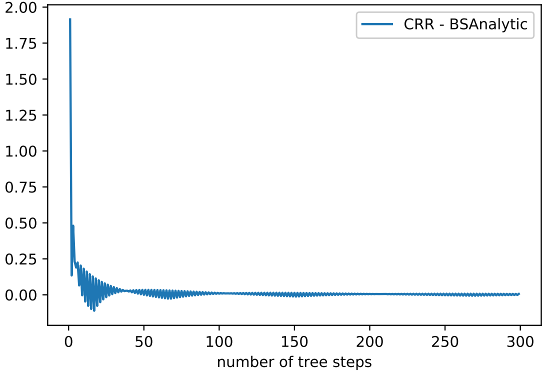
$$u = \frac{e^{(2r+\sigma^2)\Delta_t} + 1 + \sqrt{(e^{(2r+\sigma^2)\Delta_t} + 1)^2 - 4e^{2r\Delta_t}}}{2e^{r\Delta_t}}, \quad d = \frac{1}{u} \tag{25}$$

N-Step Binomial Tree Implementation

```python
def crrBinomial(S, r, vol, payoffType, K, T, n):
    t = T / n
    b = math.exp(vol * vol * t+r*t) + math.exp(-r * t)
    u = (b + math.sqrt(b*b - 4)) / 2
    p = (math.exp(r * t) - (1/u)) / (u - 1/u)
    # set up the last time slice, there are n+1 nodes at the last time slice
    payoffDict = {
        PayoffType.Call: lambda s: max(s-K, 0),
        PayoffType.Put: lambda s: max(K-s, 0),
    }
    vs = [payoffDict[payoffType]( S * u**(n-i-i)) for i in range(n+1)]
    # iterate backward
    for i in range(n-1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i+1):
            vs[j] = math.exp(-r * t) * (vs[j] * p + vs[j+1] * (1-p))
    return vs[0]
# test ---
S, r, vol, K, T = 100, 0.01, 0.2, 105, 1.0
print("blackPrice: \t", bsPrice(S, r, vol, T, K, PayoffType.Call))
print("crrNStepTree: \t", crrBinomial(S, r, vol, PayoffType.Call, K, T, 300))
```

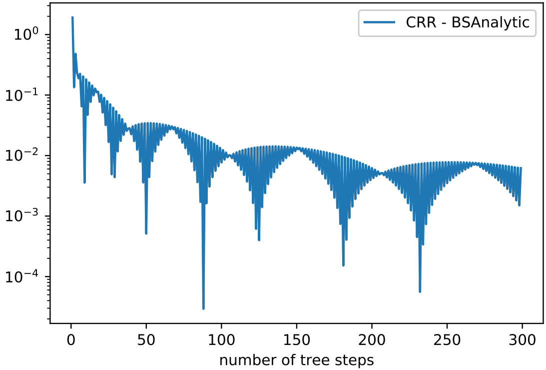
Difference between CRR tree and BS analytic

```python
import matplotlib.pyplot as plt
n = 300
S, r, vol, K, T = 100, 0.01, 0.2, 105, 1.0
bsPrc = bsPrice(S, r, vol, PayoffType.Call, K, T)
crrErrs = [(crrBinomial(S,r,vol,PayoffType.Call,K,T,i) - bsPrc) for i in range(1, n)]
plt.plot(range(1, n), crrErrs, label = "CRR - BSAnalytic")
plt.xlabel('number of tree steps')
plt.legend()
plt.savefig('../figs/crrError.eps', format='eps')
```



Easier to look at error on log scale

```
import matplotlib.pyplot as plt
n = 300
S, r, vol, K, T = 100, 0.01, 0.2, 105, 1.0
bsPrc = bsPrice(S, r, vol, PayoffType.Call, K, T)
crrErrs = [abs(crrBinomial(S,r,vol,PayoffType.Call,K,T,i) - bsPrc) for i in range(1, n)] # abs error here
plt.plot(range(1, n), crrErrs, label = "CRR - BSAnalytic")
plt.xlabel('number of tree steps')
plt.yscale('log')  # plot on log scale
plt.legend()
plt.savefig('../figs/crrLogError.eps', format='eps')
```



What's the Next Step

- European call/put and digital options are not all — we have plenty of other products we would like to trade
  - European option with generic payoff
  - American options
  - Barrier options
  - And maybe more — you never know how the industry evolves but you should be ready for changes
- Not all of them have analytic formulae, but our binomial tree pricer can handle much more
- We would like the tree pricer to be implemented elegantly so that it is
  - easy to maintain
  - easy to extend

American Option

- At any point in time, or any node of the tree, we know the continuation value of the product — through calculating the conditional expectation
- American product allows the option holder to exercise the option any time
- This translates to the choice to make at any node of the tree: continue holding the option or take the intrinsic value
  - Continue holding the option — the option is worth its conditional expectation
  - Exercise now — the option is worth its intrinsic value
  - Optimal exercise strategy is to exercise when intrinsic value is worth more than the continuation value — taking the max

American Binomial Tree Pricer — A Trivial Extension

```python

def crrBinomialAmer(S, r, vol, payoffType, K, T, n):
    ... # set up tree
    ... # set up last time slice payoff
    # iterate backward
    for i in range(n-1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i+1):
            # here we deal with american early exercise:
            continuation = math.exp(-r * t) * (vs[j] * p + vs[j+1] * (1-p))
            vs[j] = max(continuation, payoffDict[payoffType](S * u**(i-j-j)))
    return vs[0]
```

Only change to the European pricer `crrBinomial()`:

- At each iteration, `max` is taken between
  - continuation value — not exercise, and
  - payoff value — exercise immediately

Problems?

- Trivial extension is OK for experimenting the algorithms

- Not serious code for production usage
  - Copy-and-paste should be avoided in general

  - Both "'crrBinomial"' and "'crrBinomialAmer"' do not leave too much room for payoff extension — what if I want to price a digital option or call spread?

First Step Extension

Encapsulate arguments that belong to an trade-able `instrument`

- For European style options:
  - expiry T
  - strike K
  - payoffType
- For American style options:
  - everything of an European option
  - early exercise feature
- So let us define classes for the trading instruments

European option class:

```python
class EuropeanOption():
    def __init__(self, expiry, strike, payoffType):
        self.expiry = expiry
        self.strike = strike
        self.payoffType = payoffType
    def payoff(self, S):
        if self.payoffType == PayoffType.Call:
            return max(S - self.strike, 0)
        elif self.payoffType == PayoffType.Put:
            return max(self.strike - S, 0)
        else:
            raise Exception("payoffType not supported: ", self.payoffType)
```

American option class:

```python
class AmericanOption():
    def __init__(self, expiry, strike, payoffType):
        self.expiry = expiry
        self.strike = strike
        self.payoffType = payoffType
    def payoff(self, S):
        if self.payoffType == PayoffType.Call:
            return max(S - self.strike, 0)
        elif self.payoffType == PayoffType.Put:
            return max(self.strike - S, 0)
        else:
            raise Exception("payoffType not supported: ", self.payoffType)
```

In this way we lift out the code that deals with payoff from our binomial tree pricer, so that it is more orthogonal to the trade being priced:

```python
def crrBinomial(S, r, vol, trade, n):
    t = trade.expiry / n
    b = math.exp(vol * vol * t+r*t) + math.exp(-r * t)
    u = (b + math.sqrt(b*b - 4)) / 2
    p = (math.exp(r * t) - (1/u)) / (u - 1/u)
    # d = 1 / u
    # set up the last time slice, there are n+1 nodes at the last time slice
    vs = [trade.payoff( S * u**(n-i-i)) for i in range(n+1)]
    # iterate backward
    for i in range(n-1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i+1):
            vs[j] = math.exp(-r * t) * (vs[j] * p + vs[j+1] * (1-p))
    return vs[0]
```

- The only requirement from `crrBinomial` to `trade` is the `expiry` and `payoff` function, everything else is generic

Dealing with American Option

In order not to ```copy paste```, our first attempt is to add a boolean flag ```isAmer``` to the signature of ```crrBinomial```

```
def crrBinomial(S, r, vol, trade, isAmer, n):
    ...
    for i in range(n-1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i+1):
            vs[j] = math.exp(-r * t) * (vs[j] * p + vs[j+1] * (1-p))
            if isAmer:
               vs[j] = max(vs[j], trade.payoff(S * u**(i-j-j)))
    return vs[0]
```

- It does avoid ```copy paste```, but...
- It puts back attributes that belong to the tradeable to the pricing model, consider these two function calls

```
crrBinomial(S = 100, r = 0.01, vol = 0.2, trade = EuropeanOption(expiry=1, strike=105, payoffType =
      PayoffType.Call), isAmer = False, n = 1000)
crrBinomial(S = 100, r = 0.01, vol = 0.2, trade = AmericanOption(expiry=1, strike=105, payoffType =
      PayoffType.Call), isAmer = True, n = 1000)
```

- without looking at the code of ```crrBinomial``` it's hard to tell which one is pricing an American option and which one is European option.

Second Attempt

What if we require a function `isAmer() -> bool` from `trade`

```python
def crrBinomial(S, r, vol, trade, n):
    ...
    for i in range(n-1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i+1):
            vs[j] = math.exp(-r * t) * (vs[j] * p + vs[j+1] * (1-p))
            if trade.isAmer():
               vs[j] = max(vs[j], trade.payoff(S * u**(i-j-j)))
    return vs[0]
```

Better that the attribute of the trade is provided by the trade, but

- the pricer is restricted to American trade and non-American trade
- Exercise strategy should be associated with the product, not the pricer

Look At Our Tree Algorithm Again

Algorithm 1 succeed = PlaceQ (prevQ, i)

```
1: Set up the tree and parameters
2: Initialize the last time slice with final payoff
3: for k = N − 1 to 0 do
4:     for i = 0 to k do
5:         Calculate the continuation value (discounted expectation)
6:         Given the information of the tree node, calculate the option value
           at Node(k, i)
7:     end for
8: end for
```

- Step 5 belongs to the tree pricer
- Step 6 is fully determined by the product:
  - **Input**: stock price S, continuation value V, current time t
  - **Output**: current option value

Pricing American Products With Tree

- It makes sense to encapsulate step 6 in the tree algorithm into the trade-able classes

```
class AmericanOption():
    def __init__(self, expiry, strike, payoffType):
        self.expiry = expiry
        self.strike = strike
        self.payoffType = payoffType
    def payoff(self, S):
        if self.payoffType == PayoffType.Call:
            return max(S - self.strike, 0)
        elif self.payoffType == PayoffType.Put:
            return max(self.strike - S, 0)
        else:
            raise Exception("payoffType not supported: ", self.payoffType)
    # step 6 in tree algo, exercise logic
    def valueAtNode(self, t, S, continuation):
        return max(self.payoff(S), continuation)
```

- For European option, `valueAtNode` just pass on the continuation value

```
class EuropeanOption():
    def __init__(self, expiry, strike, payoffType):
        self.expiry = expiry
        self.strike = strike
        self.payoffType = payoffType
    def payoff(self, S):
        if self.payoffType == PayoffType.Call:
            return max(S - self.strike, 0)
        elif self.payoffType == PayoffType.Put:
            return max(self.strike - S, 0)
        else:
            raise Exception("payoffType not supported: ", self.payoffType)

    def valueAtNode(self, t, S, continuation):
        return continuation
```

Pricing American Products With Tree

The tree pricer now becomes

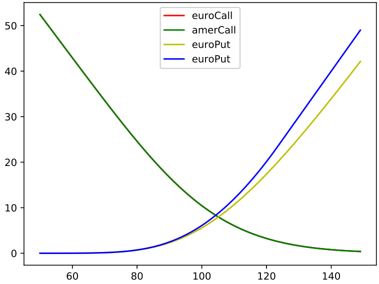
```
def crrBinomialG(S, r, vol, trade, n):
    t = trade.expiry / n
    b = math.exp(vol * vol * t+r*t) + math.exp(-r * t)
    u = (b + math.sqrt(b*b - 4)) / 2
    p = (math.exp(r * t) - (1/u)) / (u - 1/u)
    # d = 1 / u
    # set up the last time slice, there are n+1 nodes at the last time slice
    vs = [trade.payoff( S * u**(n-i-i)) for i in range(n+1)]
    # iterate backward
    for i in range(n-1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i+1):
            nodeS = S * u**(i-j-j)
            continuation = math.exp(-r * t) * (vs[j] * p + vs[j+1] * (1-p))
            vs[j] = trade.valueAtNode(t*i, nodeS, continuation)
    return vs[0]
```

There is no more ambiguity pricing European and American options:

```
euroPrc.append(crrBinomialG(S, r, vol, EuropeanOption(expiry, strike, PayoffType.Call), 300))
amerPrc.append(crrBinomialG(S, r, vol, AmericanOption(expiry, strike, PayoffType.Call), 300))
```

Testing Binomial Tree Pricer with American Option

```python
euroPrc, amerPrc = [],[]
S, r, vol = 100, 0.05, 0.2
ks = range(50, 150)
for k in ks:
    euroPrc.append(crrBinomialG(S, r, vol, EuropeanOption(1, float(k), PayoffType.Call), 300))
    amerPrc.append(crrBinomialG(S, r, vol, AmericanOption(1, float(k), PayoffType.Call), 300))
plt.plot(ks, euroPrc, 'r', label='euroCall')
plt.plot(ks, amerPrc, 'g', label='amerCall')
euroPrc, amerPrc = [], []
for k in ks:
    euroPrc.append(crrBinomialG(S, r, vol, EuropeanOption(1, float(k), PayoffType.Put), 300))
    amerPrc.append(crrBinomialG(S, r, vol, AmericanOption(1, float(k), PayoffType.Put), 300))
plt.plot(ks, euroPrc, 'y', label='euroPut')
plt.plot(ks, amerPrc, 'b', label='amerPut')
plt.legend()
```



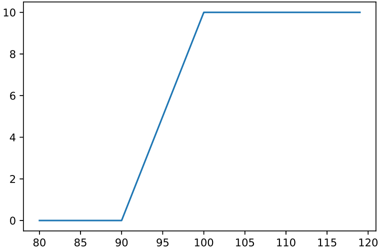
Generalize the Payoff Function

Now the CRR binomial tree pricer is generic, and if we want to price European / American exercise style option with any payoff function, we just need to create a tradeable

```
class EuropeanPayoff():
    def __init__(self, expiry, payoffFun):
        self.expiry = expiry
        self.payoffFun = payoffFun
    def payoff(self, S):
        return self.payoffFun(S)
    def valueAtNode(self, t, S, continuation):
        return continuation

class AmericanPayoff():
    def __init__(self, expiry, payoffFun):
        self.expiry = expiry
        self.payoffFun = payoffFun
    def payoff(self, S):
        return self.payoffFun(S)
    def valueAtNode(self, t, S, continuation):
        return max(self.payoff(S), continuation)
```

Example: Pricing a Call Spread

```
S, r, vol = 100, 0.05, 0.2
callSpread = lambda S: min(max(S-90, 0), 10)
plt.plot(range(80, 120), [callSpread(i) for i in range(80, 120)] )
plt.show()
print("Euro callspread: ", crrBinomialG(S, r, vol, EuropeanPayoff(1, callSpread), 300))
print("Amer callspread: ", crrBinomialG(S, r, vol, AmericanPayoff(1, callSpread), 300))
```




```
Euro callspread:  6.259190489574921
Amer callspread:  10.0
```

Practice: plot a spot ladder of Euro/Amer call spread prices, explain what you see.

Pricing Barrier Options

Now let us extend the CRR binomial pricer to Barrier Options

- A barrier option defines
  - one or two barrier levels (up or/and down) and
  - a pre-defined time window (normally from now to option expiry)

  such that if the spot price touches the barrier in the given time window, the option holder obtain (**Knock-In**) or lose (**Knock-Out**) a **underlying payoff**

- The underlying payoff can be
  - a European option
  - or just a constant rebate (normally called touch option)

Pricing Barrier Options

- Knock-Out (KO) barrier option can be priced naturally with tree
  - At each point of the tree, if the spot price triggers the KO, then the option is worth 0, otherwise it is worth its continuation value
  - The continuation value (discounted future expectation) does not contribute to the nodes triggering the KO
- Knock-In (KI) barrier option is not so natural for pricing with tree:
  - Not-yet knocked in does not mean the value is 0
  - Requires an auxiliary variable to price with tree
  - KIKO parity: KI + KO = Vanilla static replication
- We consider knock-out barrier option for now

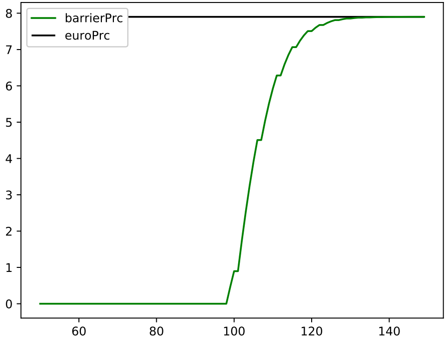
Barrier Option Class

```python
class BarrierOption():
    def __init__(self, downBarrier, upBarrier, barrierStart, barrierEnd,
     underlyingOption):
        self.underlyingOption = underlyingOption
        self.barrierStart = barrierStart
        self.barrierEnd = barrierEnd
        self.downBarrier = downBarrier
        self.upBarrier = upBarrier
        self.expiry = underlyingOption.expiry
    def payoff(self, S):
        return self.underlyingOption.payoff(S)
    def valueAtNode(self, t, S, continuation):
        if t > self.barrierStart and t < self.barrierEnd:
            if self.upBarrier != None and S > self.upBarrier:
                return 0
            elif self.downBarrier != None and S < self.downBarrier:
                return 0
        return continuation
```

Now we can enjoy the fruit of our generic CRR biniomial pricer — no change needed to the pricer

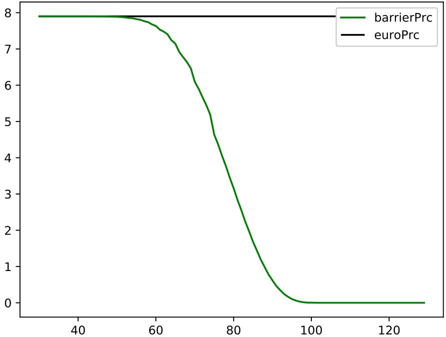
Testing Barrier Options — Up Barriers

```python
# varying up barrier
S, r, vol, K = 100, 0.05, 0.2, 105
eurOpt = EuropeanOption(1, k, PayoffType.Put)
euroPrc = crrBinomialG(S, r, vol, eurOpt, 300)
barrierPrc, ks = [], range(50, 150)
for barrierLevel in ks:
    prc = crrBinomialG(S, r, vol, BarrierOption(barrierStart = 0, barrierEnd = 1.0, downBarrier = None,
      upBarrier = barrierLevel, underlyingOption = eurOpt), n = 300)
    barrierPrc.append(prc)
plt.hlines(euroPrc, ks[0], ks[-1], label = 'euroPrc')
plt.plot(ks, barrierPrc, 'g', label='barrierPrc')
plt.xlabel('up barrier level'); plt.legend(); plt.savefig('../figs/upKO.eps', format='eps')
```



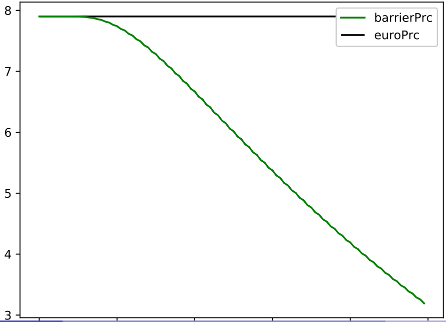
Testing Barrier Options — Down Barriers

```python
# varying down barrier
S, r, vol, K = 100, 0.05, 0.2, 105
eurOpt = EuropeanOption(1, k, PayoffType.Put)
euroPrc = crrBinomialG(S, r, vol, eurOpt, 300)
barrierPrc, ks = [], range(30, 130)
for barrierLevel in ks:
    prc = crrBinomialG(S, r, vol, BarrierOption(barrierStart = 0, barrierEnd = 1.0, downBarrier =
      barrierLevel, upBarrier = None, underlyingOption = eurOpt), n = 300)
    barrierPrc.append(prc)
plt.hlines(euroPrc, ks[0], ks[-1], label = 'euroPrc')
plt.plot(ks, barrierPrc, 'g', label='barrierPrc')
plt.xlabel('down barrier level'); plt.legend(); plt.savefig('../figs/downKO.eps', format='eps')
```



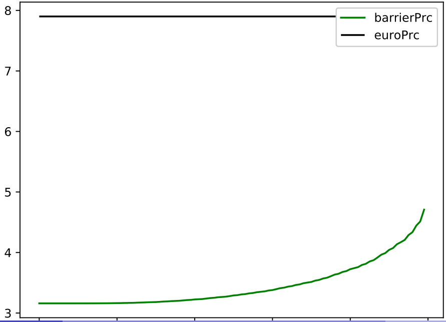
Testing Barrier Options — Window Barriers

```python
# varying barrier window, barrier end
S, r, vol, k = 100, 0.05, 0.2, 105
eurOpt = EuropeanOption(1, k, PayoffType.Put)
euroPrc = crrBinomialG(S, r, vol, eurOpt, 300)
barrierPrc = []
ks = range(0, 100)
for t in ks:
    prc = crrBinomialG(S, r, vol, BarrierOption(barrierStart = 0, barrierEnd = t / 100.0, downBarrier =
      80, upBarrier = 150, underlyingOption = eurOpt), n = 300)
    barrierPrc.append(prc)
plt.hlines(euroPrc, ks[0], ks[-1], label = 'euroPrc')
plt.plot(ks, barrierPrc, 'g', label='barrierPrc')
plt.legend(); plt.xlabel('window end'); plt.savefig('../figs/winBarrier.eps', format='eps')
```



Testing Barrier Options — Window Barriers

```
# varying barrier window, barrier start
S, r, vol, k = 100, 0.05, 0.2, 105
eurOpt = EuropeanOption(1, k, PayoffType.Put)
euroPrc = crrBinomialG(S, r, vol, eurOpt, 300)
barrierPrc = []
ks = range(0, 100)
for t in ks:
    prc = crrBinomialG(S, r, vol, BarrierOption(barrierStart = t/100.0, barrierEnd = 1.0, downBarrier =
      80, upBarrier = 150, underlyingOption = eurOpt), n = 300)
    barrierPrc.append(prc)
plt.hlines(euroPrc, ks[0], ks[-1], label = 'euroPrc')
plt.plot(ks, barrierPrc, 'g', label='barrierPrc')
plt.legend(); plt.xlabel('window start'); plt.savefig('../figs/winBarrierStart.eps', format='eps')
```



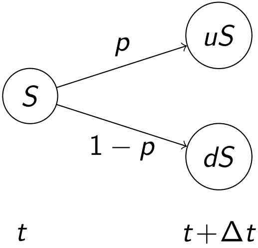
More On Binomial Tree Models

- Recall that we use CRR binomial tree model [1] where the additional constraint $u = \frac{1}{d}$ was imposed to restrict the solution space when trying to match the second moment

$$e^{2r\Delta t + \sigma^2 \Delta t} = pu^2 + (1-p)d^2 \tag{26}$$

- Recall also that the first moment is matched by setting

$$p = \frac{e^{r\Delta t} - d}{u - d} \tag{27}$$



Alternative Binomial Tree Models

Alternative binomial tree models are built by imposing different constriants

- Jarrow-Rudd binomial tree
  - Equal probability variation (not risk neutral): $p = \frac{1}{2}$
  - Risk neutral variation
- Tian binomial model: match the first three moments
- ...

M. Joshi presented a comparison of 11 different tree models: [2]

Jarrow-Rudd Models

Jarrow-Rudd Equal Probability Model (JREQ)

$$\begin{cases} p &= \frac{1}{2} \\ u &= e^{(r-\frac{\sigma^2}{2})\Delta t+\sigma\sqrt{\Delta t}} \\ d &= e^{(r-\frac{\sigma^2}{2})\Delta t-\sigma\sqrt{\Delta t}} \end{cases} \tag{28}$$

- Not risk neutral since $\frac{e^{r\Delta t}-d}{u-d} \neq \frac{1}{2}$

Jarrow-Rudd Risk Neutral Model (JRRN)

$$\begin{cases} p &= \frac{e^{r\Delta t}-d}{u-d} \\ u &= e^{(r-\frac{\sigma^2}{2})\Delta t+\sigma\sqrt{\Delta t}} \\ d &= e^{(r-\frac{\sigma^2}{2})\Delta t-\sigma\sqrt{\Delta t}} \end{cases} \tag{29}$$

- Use the same u and d as JREQ model, but replace the probability by risk neutral probability.

Tian's Model

Matching the first three moments:

$$\begin{cases} pu + (1-p)d = e^{r\Delta t} \\ pu^2 + (1-p)d^2 = e^{2r\Delta t + \sigma^2 \Delta t} \\ pu^3 + (1-p)d^3 = e^{3r\Delta t + 3\sigma^2 \Delta t} \end{cases} \tag{30}$$

This leads to the parameters:

$$\begin{cases} p &= \frac{e^{r\Delta t} - d}{u-d} \\ u &= 0.5 e^{r\Delta t} v (v + 1 + \sqrt{v^2 + 2v - 3}) \\ d &= 0.5 e^{r\Delta t} v (v + 1 - \sqrt{v^2 + 2v - 3}) \\ v &= e^{\sigma^2 \Delta t} \end{cases} \tag{31}$$

Note that all these models calibrate model parameters p, u, d to the drift r and volatility σ, the binomial tree algorithm does not change — another sign of generalization.

Our Generic Binomial Tree Pricer

```python
def binomialPricer(S, r, vol, trade, n, calib):
    t = trade.expiry / n
    (u, d, p) = calib(r, vol, t)
    # set up the last time slice, there are n+1 nodes at the last time slice
    vs = [trade.payoff(S * u ** (n - i) * d ** i) for i in range(n + 1)]
    # iterate backward
    for i in range(n - 1, -1, -1):
        # calculate the value of each node at time slide i, there are i nodes
        for j in range(i + 1):
            nodeS = S * u ** (i - j) * d ** j
            continuation = math.exp(-r * t) * (vs[j] * p + vs[j + 1] * (1 - p))
            vs[j] = trade.valueAtNode(t * i, nodeS, continuation)
    return vs[0]
```

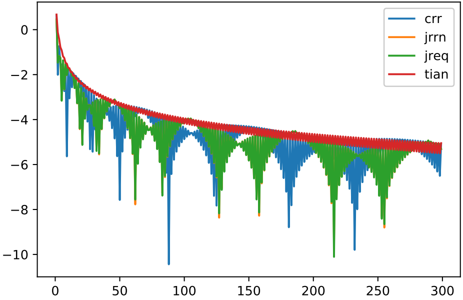
- The tree price expects `u, d, p` from the model
- Each model is responsible for the `calib` function:

Binommial Tree Models Calib

```python
def crrCalib(r, vol, t):
    b = math.exp(vol * vol * t + r * t) + math.exp(-r * t)
    u = (b + math.sqrt(b * b - 4)) / 2
    p = (math.exp(r * t) - (1 / u)) / (u - 1 / u)
    return (u, 1/u, p)

def jrrnCalib(r, vol, t):
    u = math.exp((r - vol * vol / 2) * t + vol * math.sqrt(t))
    d = math.exp((r - vol * vol / 2) * t - vol * math.sqrt(t))
    p = (math.exp(r * t) - d) / (u - d)
    return (u, d, p)

def jreqCalib(r, vol, t):
    u = math.exp((r - vol * vol / 2) * t + vol * math.sqrt(t))
    d = math.exp((r - vol * vol / 2) * t - vol * math.sqrt(t))
    return (u, d, 1/2)

def tianCalib(r, vol, t):
    v = math.exp(vol * vol * t)
    u = 0.5 * math.exp(r * t) * v * (v + 1 + math.sqrt(v*v + 2*v - 3))
    d = 0.5 * math.exp(r * t) * v * (v + 1 - math.sqrt(v*v + 2*v - 3))
    p = (math.exp(r * t) - d) / (u - d)
    return (u, d, p)
```

Test Binomial Models

```
opt = EuropeanOption(1, 105, PayoffType.Call)
S, r, vol, n = 100, 0.01, 0.2, 300
bsprc = bsPrice(S, r, vol, opt.payoffType, opt.strike, opt.expiry)
crrErrs = [math.log(abs(binomialPricer(S, r, vol, opt, i, crrCalib) - bsprc)) for i in range(1, n)]
jrrnErrs = [math.log(abs(binomialPricer(S, r, vol, opt, i, jrrnCalib) - bsprc)) for i in range(1, n)]
jreqErrs = [math.log(abs(binomialPricer(S, r, vol, opt, i, jreqCalib) - bsprc)) for i in range(1, n)]
tianErrs = [math.log(abs(binomialPricer(S, r, vol, opt, i, tianCalib) - bsprc)) for i in range(1, n)]
plt.plot(range(1, n), crrErrs, label = "crr")
plt.plot(range(1, n), jrrnErrs, label = "jrrn")
plt.plot(range(1, n), jreqErrs, label = "jreq")
plt.plot(range(1, n), tianErrs, label = "tian")
plt.legend(); plt.savefig('../figs/btrees.eps', format='eps')
```



OOP in Python

- Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
- Classes allow you to create user-defined data structures. Classes define functions called methods, which identify the behaviors and actions that an object created from the class can perform with its data.A class is a blueprint for how to define something.
- In Python, you define a class by using the **class** keyword followed by a name and a colon. Then you use **init** to declare which attributes each instance of the class should have:

```python
class EuropeanPayoff():
    def __init__(self, expiry, payoffFun):
        self.expiry = expiry
        self.payoffFun = payoffFun
    def payoff(self, S):
        return self.payoffFun(S)
    def valueAtNode(self, t, S, continuation):
        return continuation
```

OOP in Python - Class

- Every time you create a new EuropeanPayoff object, init sets the initial state of the object by assigning the values of the object's properties. That is, init initializes each new instance of the class. You can give init any number of parameters, but the first parameter will always be a variable called self. When you create a new class instance, then Python automatically passes the instance to the self parameter in init so that Python can define the new attributes on the object.
- Attributes created in init are called instance attributes. An instance attribute's value is specific to a particular instance of the class. All EuropeanPayoff objects have a Maturity, but the value for the Maturity attribute will vary depending on the EuropeanPayoff instance.
- On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of init).

Instantiate a Class in Python3

Creating a new object from a class is called instantiating a class. You can create a new object by typing the name of the class, followed by opening and closing parentheses

```
callSpread =  lambda S: min(max(S - 90, 0), 10)
  trade = EuropeanPayoff(1, callSpread)

```

After you create the EuropeanPayoff instances, you can access their instance attributes using dot notation:

```
    trade.expiry

```

References

- John C. Cox, Stephen A. Ross, and Mark Rubinstein. *Option pricing: A simplified approach.*,Journal of Financial Economics, 7(3):229-263, 1979.
- Mark Joshi *The convergence of binomial trees for pricing the American put*, Risk Net, 2009