Computational Finance

AMSTERDAM BUSINESS SCHOOL

Business

Binomial Trees

Setup and Notation

- Consider a market containing three assets: a risk-free bond with price $B_t = e^{rt}$, a stock S_t, and a (European style) derivative C_t with maturity T and payoff $C_T(S_T)$ that we wish to price.

- Split the time interval $[0, T]$ into N parts of length $\delta t = T/N$ and let $t_i = i\delta t, i = 0, \ldots, N$, so $t_0 = 0$ and $t_N = T$.

- Write $\{B_i, S_i, C_i, i = 0, \ldots, N\}$ for asset prices at time $t_i = i\delta t$. E.g., $C_1 \equiv C_{\delta t}, C_N \equiv C_T$, and $B_i = e^{r\, i\delta t}$.

- The stock price S_i either moves up to $S_{i+1}(u)$ or down to $S_{i+1}(d)$. Usually $S_{i+1}(u) = S_i u$ and $S_{i+1}(d) = S_i d$ for fixed u and d, often $u = 1/d$.

The One-Period Case: $N = 1$.

- To find C_0, construct a replicating portfolio $V_t = \phi S_t + \psi B_t$ in such a way that
$$V_T(u) = \phi S_0 u + \psi B_0 e^{rT} = C(S_0 u) =: c_u,$$
$$V_T(d) = \phi S_0 d + \psi B_0 e^{rT} = C(S_0 d) =: c_d.$$

- Solving for ϕ and ψB_0 yields
$$\phi = \frac{c_u - c_d}{S_0 u - S_0 d}, \quad \psi B_0 = e^{-rT}\left(c_u - \frac{c_u - c_d}{S_0 u - S_0 d} S_0 u\right).$$

- ϕ is known as the *hedge ratio*, or *delta* of the derivative.

- Therefore,

$$\begin{aligned} V_0 &= \phi S_0 + \psi B_0 \\ &= \frac{c_u - c_d}{u-d} + e^{-rT}\left(c_u - \frac{c_u - c_d}{u-d}u\right) \\ &= e^{-rT}\left(c_u \frac{e^{rT}-d}{u-d} + c_d \frac{u-e^{rT}}{u-d}\right) \\ &= e^{-rT}\left(c_u p + c_d[1-p]\right). \end{aligned}$$

- In the absence of arbitrage, we must have $C_0 = V_0$, and hence
$$C_0 = e^{-rT}\left(c_u p + c_d[1-p]\right).$$

- Interpretation: $p \in [0,1]$, so p is a probability and C_0 is an expectation.
- p and $1-p$ are known as *risk-neutral* probabilities. We collect these in the *risk-neutral probability measure* $\mathbb{Q}$, so that $\mathbb{Q}[u] = 1 - \mathbb{Q}[d] = p$.
- We write

$$C_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}[C_T] = e^{-rT}\left(c_u p + c_d[1-p]\right).$$

- The probabilities are called risk-neutral because if these were the true probabilities, all assets would earn the risk-free rate. E.g.,

$$\mathbb{E}^{\mathbb{Q}}[S_T] = S_0 e^{rT},$$

which you should verify.

- Note that we do *not* assume that $p = \mathbb{P}[u]$. The actual probability $\mathbb{P}[u]$ is *irrelevant* for the value C_0 of the derivative (as long as it is not zero or one).

The N-Period Case

- Next, consider a two-period model ($N = 2$):

$$
\begin{array}{ccccc}
t=0 & & t=\delta t & & t=T=2\delta t\\
i=0 & & i=1 & & i=N=2\\
 & & & & S_0uu\\
 & & & \nearrow & \\
 & & S_0u & & \\
 & \nearrow & & \searrow & \\
S_0 & & & & S_0ud = S_0du\\
 & \searrow & & \nearrow & \\
 & & S_0d & & \\
 & & & \searrow & \\
 & & & & S_0dd
\end{array}
$$

- This stock price tree is *recombinant*: an up move followed by a down move leads to the same value as a down move followed by an up move. This is a consequence of u and d being fixed and independent of the price.
- Advantage: the number of nodes remains managable ($N+1$ at the Nth step, rather than 2^N).
- This leads to a derivative price tree that is also recombinant. Given a recombinant stock price tree, this follows from the fact that C_N only depends on S_N.
- Path-dependent derivatives where $C_N = C(S_i, i \leq N)$ may lead to non-recombinant trees.

$$
\begin{array}{ccccc}
 & & & & C_N(uu) \\
 & & & \nearrow & \\
 & & C_1(u) & & \\
 & \nearrow & & \searrow & \\
C_0 & & & & C_N(ud) = C_N(du) \\
 & \searrow & & \nearrow & \\
 & & C_1(d) & & \\
 & & & \searrow & \\
 & & & & C_N(dd)
\end{array}
$$

- Only the payoffs $C_N(uu)$, $C_N(ud)$ and $C_N(dd)$ are known, and we wish to obtain $C_0, C_1(u)$ and $C_1(d)$.
- At time $t = \delta t$ (after one step), we know whether the stock has gone up or down.
- If it has gone up, then only the branch from $C_1(u)$ to $C_N(uu)$ or $C(ud)$ is relevant.
- Since this is just a binary model, we can price $C_1(u)$ (and $C_1(d)$) by no-arbitrage:

$$C_1(u) = e^{-r\,\delta t}\left[C_N(uu)p + C_N(ud)(1-p)\right] = e^{-r\,\delta t}\mathbb{E}^{\mathbb{Q}}\left[C_N|S_1 = S_0 u\right],$$
$$C_1(d) = e^{-r\,\delta t}\left[C_N(du)p + C_N(dd)(1-p)\right] = e^{-r\,\delta t}\mathbb{E}^{\mathbb{Q}}\left[C_N|S_1 = S_0 d\right].$$

- Recall that $p = \dfrac{e^{r\,\delta t} - d}{u-d}$; in general the risk-neutral probability might depend on S_1, but in this case it doesn't, because r, u and d are the same at each step.

- The values $C_1(u)$ and $C_1(d)$ are the market prices (under the no-arbitrage condition), so the derivative can be sold at this price at time $t = \delta t$, depending on whether the stock goes up or down.
- Therefore, at time $t = 0$ we know that the two possible payoffs in the next period are $C_1(u)$ and $C_1(d)$, and so

$$\begin{aligned} C_0 &= e^{-r\,\delta t}\left[C_1(u)p + C_1(d)(1-p)\right] \\ &= e^{-rT}\Big[C_N(uu)p^2 + C_N(ud)[p(1-p)+(1-p)p] \\ &\qquad + C_N(dd)(1-p)^2\Big] \\ &= e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[C_N\right]. \end{aligned}$$

- In the N-period case, denote by $\mathcal{F}_t$ the information at time t, i.e., whether the stock went up or down at each $s \leq t$. Then, at each step in the tree,
$$C_t = e^{-r\delta t}\mathbb{E}^{\mathbb{Q}}[C_{t+\delta t}|\mathcal{F}_t].$$
- Starting at C_T, this can be solved backwards until one arrives at the price at $t = 0$.
- At every step in the tree, we have that
$$C_t = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}[C_T|\mathcal{F}_t],$$
and in particular
$$C_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}[C_T].$$
- This is known as the *risk neutral pricing formula*: the price of an attainable European claim equals the expected discounted payoff, but where expectations are under a set of risk-neutral probabilities $\mathbb{Q}$.

- It is worth noting that the hedging strategy is dynamic: let ϕ_{i+1} and ψ_{i+1} denote the number of shares and cash bonds held from t_i till t_{i+1}.
- The single-period binary model implies
$$\phi_{i+1} = \frac{C_{i+1}(u) - C_{i+1}(d)}{S_{i+1}(u) - S_{i+1}(d)}.$$
- Between t_i and t_{i+1}, the value changes from V_i to $\phi_{i+1}S_{i+1} + \psi_{i+1}B_{i+1}$, after which rebalancing occurs.
- The strategy is *replicating*: after N steps, the value is $V_N = \phi_N S_N + \psi_N B_N = C_N$.
- It can also be verified to be *self-financing*:
$$V_i = \phi_i S_i + \psi_i B_i = \phi_{i+1}S_i + \psi_{i+1}B_i,$$
which may be rewritten as
$$V_{i+1} - V_i = \phi_{i+1}(S_{i+1} - S_i) + \psi_{i+1}(B_{i+1} - B_i).$$
- Thus, a dynamic strategy allows us to hedge against more than two states at time T with only two assets.

Martingales and the FTAP

- A sequence of random variables such as $\{S_i\}_{i \geq 0}$ is called a *stochastic process*.
- Observe that under $\mathbb{Q}$,
$$\mathbb{E}^{\mathbb{Q}}\left[S_{i+1}|\mathcal{F}_i\right] = S_i\big(up + d(1-p)\big) = S_i e^{r\delta t}.$$
- Define the *discounted stock price process* $\tilde{S}_i = S_i e^{-ir\delta t}$. Then
$$\mathbb{E}^{\mathbb{Q}}\left[\tilde{S}_{i+1}|\mathcal{F}_i\right] = S_i e^{r\delta t} e^{-(i+1)r\delta t} = S_i e^{-ir\delta t} = \tilde{S}_i.$$
This is the defining property of a *martingale*. Hence, the risk-neutral measure is also called a *martingale measure*.
- $\mathbb{Q}$ and $\mathbb{P}$ are *equivalent* if $\mathbb{Q}[A] = 0 \Longleftrightarrow \mathbb{P}[A] = 0$.
- *Fundamental Theorem of Asset Pricing*: if (and only if) the market is arbitrage free, then there exists an equivalent martingale measure $\mathbb{Q}$ under which discounted stock prices are martingales, and the risk neutral pricing formula holds. $\mathbb{Q}$ is unique if the market is complete.

Tree Calibration

- We are given S_0, T (measured in years), and the function $C_T = C(S_T)$; for a European call, $C(S_T) = \max\{(S_T - K), 0\}$.
- We have to choose the number N of steps, and hence $\delta t = T/N$. This involves a trade-off between computational burden and accuracy.
- $r = \log(1 + R)$, where R is the current value (per annum) of a suitable risk-free interest rate (e.g. LIBOR) over the holding period of the option.
- u and d are chosen to match the stock price volatility: under $\mathbb{Q}$,

$$R_{i+1} \equiv \log(S_{i+1}/S_i) = \begin{cases} \log u & \text{with probability } p, \\ \log d = -\log u & \text{with probability } 1 - p. \end{cases}$$

- Thus,

$$\mathbb{E}^{\mathbb{Q}}[R_{i+1}] = 2p - 1 \quad \text{and}$$

$$\sigma^2 \delta t := \mathrm{var}^{\mathbb{Q}}(R_{i+1}) = (\log u)^2 \left[1 - (2p-1)^2\right] \approx (\log u)^2.$$

- Hence we choose

$$u = e^{\sigma\sqrt{\delta t}}, \qquad d = 1/u = e^{-\sigma\sqrt{\delta t}}.$$

- Possible estimates for σ:
  - Annualized historical volatility (see last week):

$$\sigma = \sqrt{252}\sigma_{t,HIST}$$

  - Implied volatility: the value of σ that equates model price and market price (see later).

Binomial Trees in Python

- We will look at several Python implementations and compare their speed.
- The first implementation is a "loopy" version that could be written in a similar way in most imperative programming languages.

In [1]:

```
import numpy as np
def calltree(S0, K, T, r, sigma, N):
    """
    European call price based on an N-step binomial tree.
    """
    deltaT = T/float(N)
    u=np.exp(sigma * np.sqrt(deltaT))
    d=1/u
    p=(np.exp(r*deltaT) - d)/(u-d)
    C=np.zeros([N+1,N+1])
    S=np.zeros([N+1,N+1])
    piu=np.exp(-r*deltaT)*p
    pid=np.exp(-r*deltaT)*(1-p)
    for i in xrange(N+1):
        for j in xrange(i, N+1):
            S[i,j]=S0*u**j*d**(2*i)
    for i in xrange(N+1):
        C[i,N]=max(0, S[i,N]-K)
    for j in xrange(N-1,-1,-1):
        for i in xrange(j+1):
            C[i,j] = piu * C[i,j+1] + pid * C[i+1,j+1]
    return  C[0,0]
```

- Let's see if it works:

In [2]:
```
S0=50.;K=50.;T=5.0/12;r=.1;sigma=.4;N=500;
calltree(S0, K, T, r, sigma, N)
```

Out[2]:
```
6.1139619792052535
```

- Great. Now let's look at the speed:

In [3]:
```
%timeit calltree(S0, K, T, r, sigma, N) #ipython magic for timing things
```

```
10 loops, best of 3: 178 ms per loop
```

- Loops tend to be slow in Python. It is often preferable to write code in a *vectorized* style.
- This means calling NumPy `ufuncs` on entire vectors of data, so that the looping happens inside NumPy, i.e., in compiled C code (which means it's fast).

In [4]:

```
def calltree_numpy(S0, K, T, r, sigma, N):
    """
    European call price based on an N-step binomial tree.
    """
    deltaT = T/float(N)
    u=np.exp(sigma * np.sqrt(deltaT))
    d=1/u
    p=(np.exp(r*deltaT) - d)/(u-d)
    piu=np.exp(-r*deltaT)*p
    pid=np.exp(-r*deltaT)*(1-p)
    C=np.zeros([N+1,N+1])
    S=S0*u**np.arange(N+1)*d**(2*np.arange(N+1)[:, np.newaxis])
    S=np.triu(S) #keep only upper trinagular part
    C[:,N]=np.maximum(0, S[:,N]-K) #note maximum in place of max
    for j in xrange(N-1,-1,-1):
        C[:j+1,j] = piu * C[:j+1,j+1] + pid * C[1:j+2,j+1]
    return  C[0,0]
```

- Let's verify that both implementations give the same answer.
- We'll use NumPy's `allclose` function, which tests if all elements of an array are close to zero.

In [5]:
```
np.allclose(calltree(S0, K, T, r, sigma, N), calltree_numpy(S0, K, T, r, sigma, N))
```

Out[5]:
```
True
```

- Now let's time it:

In [6]:
```
%timeit calltree_numpy(S0, K, T, r, sigma, N)
```

```
100 loops, best of 3: 4.41 ms per loop
```

- A third option is to use Numba (user guide (http://numba.pydata.org/numba-doc/latest/index.html)).
- Numba implements a *just in time compiler*. It can compile certain (array-heavy) code to native machine code.
- If Numba is able to compile your code, then the speed is often comparable to C.
- All we need to do is import the package, and then add a *decorator* to our function.
- Other than that, the code is exactly the same as our first attempt.

In [7]:

```python
from numba import jit
@jit
def calltree_numba(S0, K, T, r, sigma, N):
    """
    European call price based on an N-step binomial tree.
    """
    deltaT = T/float(N)
    u=np.exp(sigma * np.sqrt(deltaT))
    d=1/u
    p=(np.exp(r*deltaT) - d)/(u-d)
    C=np.zeros([N+1,N+1])
    S=np.zeros([N+1,N+1])
    piu=np.exp(-r*deltaT)*p
    pid=np.exp(-r*deltaT)*(1-p)
    for i in xrange(N+1):
        for j in xrange(i, N+1):
            S[i,j]=S0*u**j*d**(2*i)
    for i in xrange(N+1):
        C[i,N]=max(0, S[i,N]-K)
    for j in xrange(N-1,-1,-1):
        for i in xrange(j+1):
            C[i,j] = piu * C[i,j+1] + pid * C[i+1,j+1]
    return  C[0,0]
```

- Check that it gives the right answer:

In [8]:
```
np.allclose(calltree(S0, K, T, r, sigma, N), calltree_numba(S0, K, T, r, sigma, N))
```

Out[8]:
```
True
```

- The moment of truth:

In [9]:
```
%timeit calltree_numba(S0, K, T, r, sigma, N)
```

```
100 loops, best of 3: 4.27 ms per loop
```

- Not bad at all. We essentially match our NumPy implementation.
- There's one more thing we might try: what if we JIT-compile the vectorized version?
- Instead of writing out the whole function again, we'll use an alternative way to invoke the JIT compiler:

In [10]:
```
calltree_numpy_numba=jit(calltree_numpy)
np.allclose(calltree(S0, K, T, r, sigma, N), calltree_numpy_numba(S0, K, T, r, sigma, N))
```

Out[10]:
```
True
```

In [11]:
```
%timeit calltree_numpy_numba(S0, K, T, r, sigma, N)
```

```
1000 loops, best of 3: 1.08 ms per loop
```

- Wow. That's three times as fast as the pure NumPy version, and 150 times as fast as our naive implementation.
- Looking at the absolute timings, the improvements may seem small, but keep in mind that you may need to call these functions many many times.
- Other tools for compiling Python to native code include Cython (http://cython.org/) and Pythran (https://pythonhosted.org/pythran/).

A Closed Form for European Options

- The price of a European option
$$C_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[\max(S_T - K), 0\right]$$
depends only on S_T, so there is no need to use a tree explicitly to evaluate it.
- Let k denote the number of up moves of the stock , so that $N-k$ is the number of down moves. Then
$$S_T = S_0 u^k d^{N-k} = S_0 u^{2k-N},$$
where under $\mathbb{Q}$, $k \sim \mathrm{Bin}(N, p)$, with pmf $f(k; N, p) = \binom{N}{k} p^k (1-p)^{N-k}$. Thus
$$C_0 = e^{-rT}\sum_{k=0}^{N} f(k; N, p)\max(S_0 u^k d^{N-k} - K, 0).$$

- Let a denote the minimum number of up moves so that $S_T > K$, i.e., the smallest integer greater than $N/2 + \log(K/S_0)/(2\log u)$. Then
$$C_0 = e^{-rT} \sum_{k=a}^{N} f(k; N, p) \left[S_0 u^k d^{N-k} - K\right].$$
- The second term is $[1 - F(a-1; N, p)]e^{-rT} K = \bar{F}(a-1; N, p)e^{-rT} K$, where F is the binomial cdf and $\bar{F}$ is the survivor function.
- Let $p_* = e^{-r\delta t} pu$. The first term is
$$e^{-rT} S_0 \sum_{k=a}^{N} \binom{N}{k} p^k (1-p)^{N-k} u^k d^{N-k} = S_0$$
$$\sum_{k=a}^{N} \binom{N}{k} p_*^k (1-p_*)^{N-k}.$$

- Putting things together,

$$\begin{aligned} C_0 &= S_0 \bar{F}(a-1; N, p_*) - \bar{F}(a-1; N, p) e^{-rT} K \\ &= S_0 \mathbb{Q}^*(S_T > K) - \mathbb{Q}(S_T > K) e^{-rT} K \end{aligned}$$

- You will be implementing this in a homework exercise.

The Black-Scholes Formula as Continuous Time Limit

- Let's consider what happens if we let $N \to \infty$.
- First, a first-order Taylor expansion, together with l'Hopital's rule, can be used to show that, for small δt,

$$p \approx \frac{1}{2}\left(1 + \sqrt{\delta t}\,\frac{r - \frac{1}{2}\sigma^2}{\sigma}\right).$$

- Similarly,

$$p^* \approx \frac{1}{2}\left(1 + \sqrt{\delta t}\,\frac{r + \frac{1}{2}\sigma^2}{\sigma}\right).$$

- Next, Let $X_T \equiv \log S_T$. Then

$$X_T = \log S_0 + \sum_{i=1}^{N} R_i = \log S_0 + \sigma\sqrt{\delta t}\,(2k - N),$$

because R_i is either $\log u$ or $\log d = -\log u$.

- As $k \sim \mathrm{Bin}(N, p)$, we have $\mathbb{E}^{\mathbb{Q}}[k] = Np$, $\mathrm{Var}^{\mathbb{Q}}[k] = Np(1-p)$, and

$$\mathbb{E}^{\mathbb{Q}}[X_T] = \log S_0 + \sigma\sqrt{\delta t}\,N(2p-1) \to \log S_0 + (r - \frac{1}{2}\sigma^2)T$$

$$\mathrm{Var}^{\mathbb{Q}}[X_T] = \sigma^2\delta t 4Np(1-p) \to \sigma^2 T.$$

- Finally, as $N \to \infty$, the distribution of X_T tends to a normal. This follows from the *central limit theorem* and the fact that X_T is the sum of N i.i.d. terms.

- Thus, as $N \to \infty$,

$$\mathbb{Q}(S_T > K) = \mathbb{Q}(X_T > \log K) = \mathbb{Q}\left(\frac{X_T - \mathbb{E}^{\mathbb{Q}}[X_T]}{\sqrt{\mathrm{Var}^{\mathbb{Q}}[X_T]}} > \frac{\log K - \mathbb{E}^{\mathbb{Q}}[X_T]}{\sqrt{\mathrm{Var}^{\mathbb{Q}}[X_T]}} \right)$$

$$= 1 - \Phi\left(\frac{\log K - \mathbb{E}^{\mathbb{Q}}[X_T]}{\sqrt{\mathrm{Var}^{\mathbb{Q}}[X_T]}} \right) =: 1 - \Phi(-d_2) = \Phi(d_2),$$

where Φ is the standard normal cdf and

$$d_2 \equiv \frac{\mathbb{E}^{\mathbb{Q}}[X_T] - \log K}{\sqrt{\mathrm{Var}^{\mathbb{Q}}[X_T]}} = \frac{\log(S_0/K) + (r - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}.$$

- The same argument can be used to show that as $N \to \infty$,
$$\mathbb{Q}^*(S_T > K) = \Phi(d_1),$$
where
$$d_1 \equiv d_2 + \sigma\sqrt{T} = \frac{\log(S_0/K) + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}.$$

- In summary, we have derived the *Black-Scholes formula*
$$\begin{aligned} C_0 &= S_0\Phi(d_1) - e^{-rT}K\Phi(d_2) \\ &=: BS(S_0, K, T, r, \sigma). \end{aligned}$$

- Implementation in Python:

In [12]:
```python
from scipy.stats import norm
def blackscholes(S0, K, T, r, sigma):
    """
    Price of a European call in the Black-Scholes model.
    """
    d1=(np.log(S0)-np.log(K)+(r+sigma**2/2)*T)/(sigma*np.sqrt(T))
    d2=d1-sigma*np.sqrt(T)
    return S0*norm.cdf(d1)-np.exp(-r*T)*K*norm.cdf(d2)
```

In [13]:
```python
calltree(S0, K, T, r, sigma, 500), blackscholes(S0, K, T, r, sigma)
```

Out[13]:
```
(6.1139619792052535, 6.116508129330871)
```

- Note that as written, the function can operate on arrays of strikes:

In [14]:
```python
Ks=np.linspace(K/2., 2.*K, 5)
blackscholes(S0, Ks, T, r, sigma)
```

Out[14]:
```
array([ 26.0260491 ,   9.77944137,   2.00056039,   0.27962697,   0.0331146 ])
```

American Options

- Unlike a European call, an American call with price C_t^{Am} can be exercised at any time before it matures. When exercised at $t \leq T$, it pays $\max(S_t - K, 0)$. Hence the call will be exercised early if at time t, $S_t - K > C_t^{Am}$.
- Recall that the price of a European call is at least as large as its *intrinsic value*: $C_t \geq \max(S_t - K, 0)$.
- As $C_t^{Am} \geq C_t$, an American call is therefore never exercised early (in the absence of dividends).
- There is no closed-form expression for the price of an American put option, so numerical methods are needed. Binomial trees are a popular choice.

- This works as follows:
  - At step N, the price of the put is $P_N^{Am} = \max(K - S_N, 0)$, just like for a European put.
  - At step $N-1$, the *continuation value* of the option is $e^{-r\delta t}\mathbb{E}^{\mathbb{Q}}[P_N^{Am}]$. Early exercise yields $K - S_{N-1}$, so
  $$P_{N-1}^{Am} = \max(e^{-r\delta t}\mathbb{E}^{\mathbb{Q}}[P_N^{Am}|\mathcal{F}_{N-1}], K - S_{N-1}).$$
  - This is iterated backwards until P_0^{Am}.
- The implementation is part of the homework exercise.

Implied Volatility

- The *implied volatility* (IV, σ_I) of an option is that value of σ which equates the BS model price to the observed market price C_0^{obs}, i.e., it solves
$$C_0^{obs} = BS(S_0, K, T, r, \sigma_I).$$

- If the BS assumptions were correct, then any option traded on the asset should have the same IV, which should in turn equal historical volatility.

- In practice, options with different strikes K and hence *moneyness* K/S_0 have different IVs: *volatility smile* or *smirk/skew*. Also, options with different times to maturity have different IVs: *volatility term structure*.

- These phenomena are evidence of a failure of the assumptions of the Black-Scholes model, most importantly that of a constant volatility σ.

- In practice, the BS formula is used as follows: the implied volatility is computed for options that are already traded in the market, for different strikes and maturities. This leads to the *IV surface*.
- When a new option is issued, the implied volatility corresponding to its strike and time to maturity is determined by interpolation on the surface. The BS formula then gives the corresponding price.
- Mathematically, the IV is the *root* (or *zero*) of the function

$$f(\sigma_I) = BS(S_0, K, T, r, \sigma_I) - C_0^{obs}.$$

- In Python, root finding can be done via SciPy's `brentq` function. In its simplest form, it takes 3 arguments: the unary function $f(\cdot)$, a lower bound L, and an upper bound U, such that $[L, U]$ contains exactly one root of f.

- Tehranchi (2016) (https://arxiv.org/abs/1512.06812) shows that

$$-\Phi^{-1}\left(\frac{S_0-C_0^{obs}}{2\min(S_0,e^{-rT}K)}\right)\le\frac{\sqrt{T}}{2}\sigma_I\le-\Phi^{-1}\left(\frac{S_0-C_0^{obs}}{S_0+e^{-rT}K)}\right)$$

- It remains to transform our objective function into a unary function (function of a single argument), via a process known as *partial function application*. Anonymous functions are a convenient way of achieving this.

In [15]:

```
from scipy.optimize import brentq
def impvol(S0, K, T, r, C_obs):
    L=-2*norm.ppf((S0-C_obs)/(2.0*min(S0, np.exp(-r*T)*K)))/np.sqrt(T)
    U=-2*norm.ppf((S0-C_obs)/(S0+np.exp(-r*T)*K))/np.sqrt(T)
    f=lambda s: blackscholes(S0, K, T, r, s)-C_obs #partial application: f(s)=BS(S0
, K, T, r, s)-C_obs
    return brentq(f, L, U)
```

In [16]:

```
C_obs=6.0 #for illustration
IV=impvol(S0, K, T, r, C_obs); (IV, blackscholes(S0, K, T, r, IV))
```

Out[16]:

```
(0.39056035816043205, 6.0)
```