Implied Volatility

Kannan Singaravelu, CQF

1 Implied Volatility

Implied volatility (IV) is one of the most important parameter in options pricing. IV is determined by the current market price of option contracts on a particular underlying asset. IV is commonly represented as a percentage that indicates the annualized expected one standard deviation range for the underlying asset implied from the option prices.

IV σ_{imp} is the volatility value σ that makes the Black Scholes value of the option equal to the traded price of the option. In the Black-Scholes model, volatility is the only parameter that can't be directly observed. All other parameters can be determined through market data and this parameter is determined by a numerical optimization technique given the Black-Scholes model.

Import Required Libraries

```
[ ]: # Import Libraries
     from numpy import *
     from datetime import datetime
     from tabulate import tabulate
```

2 Implied Volatility Formulation

The Black–Scholes equation describes the price of the option over time as

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

Solving the above equation, we know that the value of a call option for a non-dividend paying stock is:

$$C = SN(d_1) - Ke^{-rt}N(d_2)$$

and, the corresponding put option price is:

$$P = Ke^{-rt}N(-d_2) - SN(-d_1)$$

where,

$$d_1 = \frac{1}{\sigma\sqrt{t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right]$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

$$N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} \mathrm{e}^{-\frac{1}{2}x^2}dx$$

S is the spot price of the underlying asset K is the strike price r is the annualized continuous compounded risk free rate σ is the volatility of returns of the underlying asset t is time to maturity (expressed in years) $N(x)$ is the standard normal cumulative distribution

We can look at the call and put equation as a function of the volatility parameter σ. Finding implied volatility thus requires solving the nonlinear problem $f(x) = 0$ where $x = \sigma$ given a starting estimate.

For call options we have,

$$f(x) = SN(d_1) - Ke^{-rt}N(d_2) - C$$

and for puts,

$$f(x) = Ke^{-rt}N(-d_2) - SN(-d_1) - P$$

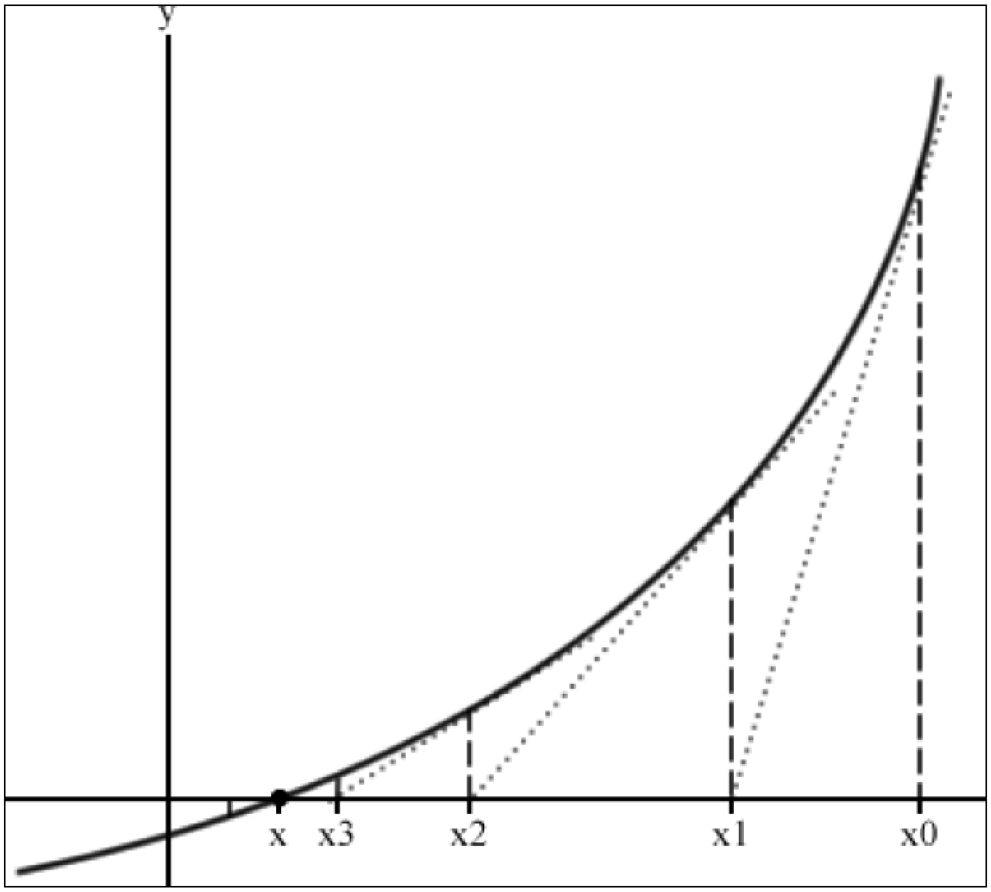
To solve the function when $f(x) = 0$, numerical precedures like Bisection or Newton's method are employed.

2.1 Newton Method

The Newton-Raphson method uses an iterative procedure to solve for a root using informaiton about the derivative of a function. The first-order derivation f' of the function f represents the tangent line and the approximation to the next value of x is given as

$$x_1 = x - \frac{f(x)}{f'(x)}$$

The tangent line intersects the x axis and x_1 that produces $y = 0$. The iterative process is repeated until a maximum iterations are reached or the difference between x_1 and x are within acceptable values.



This method requires to specify initial guess and expect the function to be differentiable. Newton approaches are extremely fast as the rate of convergence is quadractic. The downside to this approach is that it does not guarantee convergence if there are more than one root or when it reaches a local extremum.

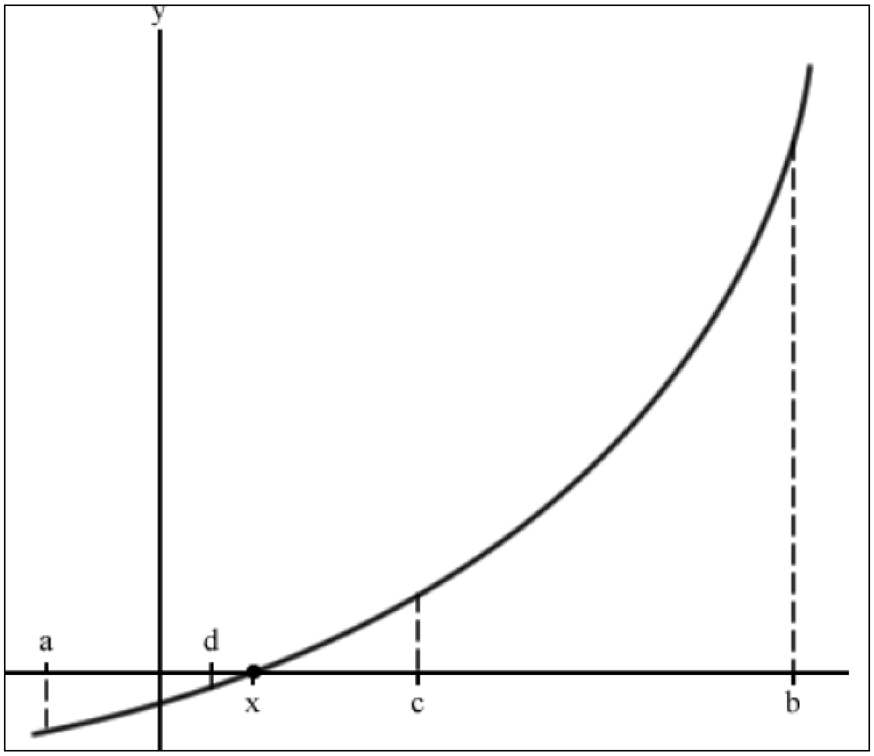
[ ]:
```
def newton_iv(className, spot, strike, rate, dte, callprice=None, putprice=None):

    x0 = 1                                          # initial guess
    h = 0.001                                       # step size
    tolerance = 1e-7                                # 7-digit accuracy is desired
    epsilon = 1e-14                                 # do not divide by a number␣
↪smaller than this, some kind of error / floor
    maxiter = 200                                   # maximum number of iterations␣
↪to execute

    # function whose root we are trying to find
    # f(x) = Black Scholes Call price - Market Price - defining the f(x) here
    if callprice:
        f = lambda x: eval(className)(spot, strike, rate, dte, x).callPrice -␣
↪callprice
    if putprice:
        f = lambda x: eval(className)(spot, strike, rate, dte, x).putPrice -␣
↪putprice

    for i in range(maxiter):
        y = f(x0)                                   # starting with initial guess
        yprime = (f(x0+h) - f(x0-h))/(2*h)          # central difference, the␣
↪derivative of the function

        if abs(yprime)<epsilon:                     # stop if the denominator is too␣
↪small
            break
        x1 = x0 - y/yprime                          # perform Newton's computation

        if (abs(x1-x0) <= tolerance*abs(x1)):       # stop when the result is within␣
↪the desired tolerance
            break
        x0=x1                                       # update x0 to start the process␣
↪again

    return x1                                       # x1 is a solution within␣
↪tolerance and maximum number of iterations
```

[ ]:
```
# newton iv
newton_iv('BS',100,100,0.02,1,callprice=8)
```

2.2 Bisection Method

The bisection method is considered to be one of the simplest and robust root finding algorithm.

Suppose, we know the two points of an interval a and b, where $a < b$ and $f(a) < 0$ and $f(b) > 0$ lie along the continuous function and the mid-point of this interval $c = \frac{a+b}{2}$, then we can evaluate the value as $f(c)$.



Iteratively, we replace c as either a or b, thereby shortening the interval to find the root. If $f(c) = 0$ or within acceptable value, we have a root. Bisection methods are stable and guarantee to converge. As it does not require knowledge of the derivative, it takes more computational time.

```
[ ]: # Bisection Method
def bisection_iv(className, spot, strike, rate, dte, callprice=None,␣
 ↪putprice=None, high=500.0, low=0.0):

    # this is market price
    if callprice:
        price = callprice
    if putprice and not callprice:
        price = putprice

    tolerance = 1e-7

    for i in range(1000):
        mid = (high + low) / 2              # c= (a+b)/2
        if mid < tolerance:
            mid = tolerance

        if callprice:
            estimate = eval(className)(spot, strike, rate, dte, mid).callPrice #␣
 ↪Blackscholes price
        if putprice:
            estimate = eval(className)(spot, strike, rate, dte, mid).putPrice

        if round(estimate,6) == price:
            break
        elif estimate > price:
            high = mid                       # replace c with b | b = c
        elif estimate < price:
            low = mid                        # replace c with a | a = c

    return mid
```

```
[ ]: # bisection iv
bisection_iv('BS',100,100,0.02,1,callprice=8)
```

3 BS Implied Volatility

Let's now update our Blackscholes' class to incorporate implied volatility.

```
[ ]: # Import updated blackscholes object
     from blackscholes import BS
```

```
[ ]: # Initialize option
     option = BS(100,100,0.05,1,0.2, callprice=8)

     header = ['Option Price', 'Delta', 'Gamma', 'Theta', 'Vega', 'Rho', 'IV']
     table = [[option.callPrice, option.callDelta, option.gamma, option.callTheta,␣
      ↪option.vega, option.callRho, option.impvol]]

     print(tabulate(table,header))
```

4 References

- Python Resources

Python Labs by Kannan Singaravelu.