In [ ]:
```
import numpy as np
import pandas as pd
from scipy.stats import norm
import scipy.interpolate as spi
import scipy.sparse as sp
import scipy.linalg as sla
from scipy.sparse.linalg import inv
from scipy.sparse.linalg import spsolve
import scipy.stats as stats
import math


np.random.seed(1031)
```

We learn about a put option with stock price S = 100, strike price K = 100, interest rate of 0.025, dividend rate of 0, maturity of 0.642 years and volatility of 0.2, which has the following characteristics:

Properties	Symbol	Value
Stock price	S	100
exercise price	K	100
continuous interest rate	r	0.025
continuous dividend rate	q	0
volatility	σ	0.2
years to maturity	T	0.642

In [ ]:
```
#init_param
(S, K, r, q, T, sigma, option_type) = (100, 100, 0.025, 0.00, 231 / 360, 0.2, 'p
(Smin, Smax, Ns, Nt) = (0, 4*np.maximum(S,K), 200, 200)

class init_option():

    def __init__(self, S, K, r, q, T, sigma, option_type, Smin, Smax, Ns, Nt):
        self.S = S
        self.K = K
        self.r = r
        self.q = q
        self.T = T
        self.sigma = sigma
        self.option_type = option_type
        self.is_call = (option_type[0].lower()=='c')
        self.omega = 1 if self.is_call else -1
        self.Smin = Smin
        self.Smax = Smax
        self.Ns = int(Ns)
        self.Nt = int(Nt)
        self.dS = (Smax-Smin)/Ns * 1.0
        self.dt = T/Nt*1.0
        self.Svec = np.linspace(Smin, Smax, self.Ns+1)
        self.Tvec = np.linspace(0, T, self.Nt+1)
        self.grid = np.zeros(shape=(self.Ns+1, self.Nt+1))
```

```
    def _set_terminal_condition_(self):
        self.grid[:, -1] = np.maximum(self.omega*(self.Svec - self.K), 0)

    def _set_boundary_condition_(self):
        tau = self.Tvec[-1] - self.Tvec;
        DFq = np.exp(-q*tau)
        DFr = np.exp(-r*tau)

        self.grid[0,  :] = np.maximum(self.omega*(self.Svec[0]*DFq - self.K*DFr)
        self.grid[-1, :] = np.maximum(self.omega*(self.Svec[-1]*DFq - self.K*DFr

    def _set_coefficient__(self):
        drift = (self.r-self.q)*self.Svec[1:-1]/self.dS
        diffusion_square = (self.sigma*self.Svec[1:-1]/self.dS)**2

        self.l = 0.5*(diffusion_square - drift)
        self.c = -diffusion_square - self.r
        self.u = 0.5*(diffusion_square + drift)

    def _solve_(self):
        pass

    def _interpolate_(self):
        tck = spi.splrep( self.Svec, self.grid[:,0], k=3 )
        return spi.splev( self.S, tck )
        #return np.interp(self.S, self.Svec, self.grid[:,0])

    def price(self):
        self._set_terminal_condition_()
        self._set_boundary_condition_()
        self._set_coefficient__()
        self._set_matrix_()
        self._solve_()
        return self._interpolate_()
```

Part a Using crank nicolson to price a European put vanilla

In [ ]:
```
class CrankNicolsonEu(init_option):

    theta = 0.5

    def _set_matrix_(self):
        self.A = sp.diags([self.l[1:], self.c, self.u[:-1]], [-1, 0, 1],  format
        self.I = sp.eye(self.Ns-1)
        self.M1 = self.I + (1-self.theta)*self.dt*self.A
        self.M2 = self.I - self.theta*self.dt*self.A

    def _solve_(self):
        _, M_lower, M_upper = sla.lu(self.M2.toarray())
        for j in reversed(np.arange(self.Nt)):

            U = self.M1.dot(self.grid[1:-1, j+1])

            U[0] += self.theta*self.l[0]*self.dt*self.grid[0, j] \
                 + (1-self.theta)*self.l[0]*self.dt*self.grid[0, j+1]
            U[-1] += self.theta*self.u[-1]*self.dt*self.grid[-1, j] \
                  + (1-self.theta)*self.u[-1]*self.dt*self.grid[-1, j+1]
```

```
            Ux = sla.solve_triangular( M_lower, U, lower=True )
            self.grid[1:-1, j] = sla.solve_triangular( M_upper, Ux, lower=False
```

In [ ]:

```
# (th-1, alpha, epsilon) = (0.5, 1.5, 1e-6)
euro_opt = CrankNicolsonEu(S, K, r, q, T, sigma, option_type, Smin, Smax, Ns, Nt
parta_answer=euro_opt.price()
print(parta_answer.round(4))
```

```
5.5574
```

Part b Using crank nicolson to price an American put vanilla

In [ ]:

```
class CrankNicolsonAm(init_option):

    def __init__(self, S, K, r, q, T, sigma, option_type, Smin, Smax, Ns, Nt, th
        super().__init__(S, K, r, q, T, sigma, option_type, Smin, Smax, Ns, Nt)
        self.theta = theta
        self.lbd = lbd
        self.epsilon = epsilon
        self.max_iter = 10*Nt

    def _set_matrix_(self):
        self.A = sp.diags([self.l[1:], self.c, self.u[:-1]], [-1, 0, 1], format=
        self.I = sp.eye(self.Ns-1)

    def _solve_(self):
        (theta, dt) = (self.theta, self.dt)
        payoff = self.grid[1:-1, -1]
        pastval = payoff.copy()
        G = payoff.copy()

        for j in reversed(np.arange(self.Nt)):
            counter = 0
            noBreak = 1
            newval = pastval.copy()

            while noBreak:
                counter += 1
                oldval = newval.copy()
                D = sp.diags( (G > (1-theta)*pastval + theta*newval).astype(int)
                z = (self.I + (1-theta)*dt*(self.A - self.lbd*D))*pastval + dt*s

                z[0] += theta*self.l[0]*dt*self.grid[0, j] \
                 + (1-theta)*self.l[0]*dt*self.grid[0, j+1]
                z[-1] += theta*self.u[-1]*dt*self.grid[-1, j] \
                  + (1-theta)*self.u[-1]*dt*self.grid[-1, j+1]

                M = self.I - theta*dt*(self.A - self.lbd*D)
                newval = spsolve(M,z)

                noBreak = CrankNicolsonAm.trigger( oldval, newval, self.epsilon,

            pastval = newval.copy()
            self.grid[1:-1, j] = pastval

    @staticmethod
    def trigger( oldval, newval, tol, counter, maxIteration ):
```

```
        noBreak = 1
        if np.max( np.abs(newval-oldval)/np.maximum(1,np.abs(newval)) ) <= tol:
            noBreak = 0
        elif counter > maxIteration:
            print('The results may not converge.')
            noBreak = 0
        return noBreak
```

In [ ]:

```
(theta, lbd, epsilon) = (0.5, 1e6, 1e-6)
amer_opt = CrankNicolsonAm(S, K, r, q, T, sigma, option_type, Smin, Smax, Ns, Nt
partb_answer=amer_opt.price()
print(partb_answer.round(4))
```

```
5.6921
```

The Black-Scholes-Merton PDE:

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

Approximation of derivative terms:

$$\frac{\partial V}{\partial t} \approx \frac{V_i^{n+1} - V_i^n}{\Delta t}$$

$$\frac{\partial V}{\partial S} \approx \frac{V_{i+1}^n - V_{i-1}^n}{2\Delta S}$$

$$\frac{\partial^2 V}{\partial S^2} \approx \frac{V_{i+1}^n - 2V_i^n + V_{i-1}^n}{\Delta S^2}$$

Crank-Nicolson method:

$$V_i^{n+1} - V_i^n = \frac{\Delta t}{2}\left[\left(\frac{\sigma^2 S_i^2(V_{i+1}^n - 2V_i^n + V_{i-1}^n)}{\Delta S^2} + rS_i\frac{V_{i+1}^n - V_{i-1}^n}{2\Delta S} - rV_i^n\right) + \left(\frac{\sigma^2 S_i^2}{}\right.\right.$$

Boundary conditions for an up-and-out call option:

$$V = \begin{cases} \max(S-K,0), & \text{if } S < B \\ 0, & \text{if } S \geq B \end{cases}$$

In [ ]:

```
def barrier_option_price(S, K, r, q, T, sigma, H):
    # Calculate d1 and d2 parameters
    d1 = (np.log(S / K) + (r - q + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    # Calculate lambda and x1 through y parameters
    lambda_ = (r - q + sigma ** 2) / (sigma ** 2)
    x1 = np.log(S / H) / (sigma * np.sqrt(T)) + lambda_ * sigma * np.sqrt(T)
    y1 = np.log(H / S) / (sigma * np.sqrt(T)) + lambda_ * sigma * np.sqrt(T)
    x2 = np.log(S / H) / (sigma * np.sqrt(T)) - lambda_ * sigma * np.sqrt(T)
    y2 = np.log(H**2 / (S * K)) / (sigma * np.sqrt(T)) + lambda_ * sigma * np.sc
    z = np.log(H / S) / (sigma * np.sqrt(T)) - lambda_ * sigma * np.sqrt(T)

    # Calculate the price based on the type of barrier option
```

```
    price = K * np.exp(-r * T) * (stats.norm.cdf(-y1) - stats.norm.cdf(-y2)) - S
    price += -1 * S * np.exp(-q * T) * (H / S)**(2 * lambda_) * (stats.norm.cdf(
    price -= -1 * K * np.exp(-r * T) * (H / S)**(2 * lambda_ - 2) * (stats.norm.

    return price

# Test the function

H = 80       # Barrier level


partc_answer = barrier_option_price(S, K, r, q, T, sigma, H)
print("Barrier Option Price:", partc_answer.round(4))
```

```
Barrier Option Price: 38.8974
```

According to the **BS** formula, the analytical solution of the Euclidean option is

$$V = e^{-rT} \cdot E\left[[\omega\left(S_T - K\right)]^+\right] = \omega \cdot \left[e^{-qT} S_0 \Phi(\omega \cdot d_+) - e^{-rT} K \Phi(\omega \cdot d_-)\right]$$

where

$$d_\pm = \frac{1}{\sigma\sqrt{T}} \ln\left(\frac{S_0 e^{(r-q)T}}{K}\right) \pm \frac{\sigma\sqrt{T}}{2}$$

The call option corresponds to $\omega = 1$ and the put option corresponds to $\omega = -1$.

In [ ]:
```
def blackscholes( S0=100, K=100, r=0.025, q=0.00, T=231 / 360, sigma=0.2, omega=
    discount = np.exp(-r*T)
    forward = S0*np.exp((r-q)*T)
    moneyness = np.log(forward/K)
    vol_sqrt_T = sigma*np.sqrt(T)

    d1 = moneyness / vol_sqrt_T + 0.5*vol_sqrt_T
    d2 = d1 - vol_sqrt_T

    V = omega * discount * (forward*norm.cdf(omega*d1) - K*norm.cdf(omega*d2))
    return V

partd_answer = blackscholes(S, K, r, q,T, sigma,-1)
print("Using the BS model the option price is ",partd_answer.round(4))
print("the part a and c answer are",parta_answer.round(4), partc_answer.round(4)
```

```
Using the BS model the option price is  5.5697
the part a and c answer are 5.5574 38.8974
```

To calculate the potential stock price movements, we use:

- Up: $S_u = Su$
- Down: $S_d = Sd$
- Flat: $S_m = Sm$

For a put option, the exercise value at each node is calculated as:

$$\text{Exercise Value} = \max(K - S, 0)$$

The discounted expected future value at each node (assuming risk-neutral probabilities p, q, and $1-p-q$ for up, down, and flat movements respectively) is given by:

$$\text{Expected Future Value} = e^{-r\Delta t}\left[pV_{\text{up}} + qV_{\text{down}} + (1-p-q)V_{\text{flat}}\right]$$

where V_{up}, V_{down}, and V_{flat} are the option values at the next time step for up, down, and flat movements respectively.

The value of the put option at each node is then the maximum of the exercise value and the discounted expected future value:

$$V = \max\left(K - S, e^{-r\Delta t}\left[pV_{\text{up}} + qV_{\text{down}} + (1-p-q)V_{\text{flat}}\right]\right)$$

In the trinomial model, the probabilities p, q, and $1-p-q$ and the factors u, d, and m are determined using the risk-free rate, the volatility of the stock, and the time step size.

In [ ]:
```
def american_trinominal_model(S, T, K, r, q, sigma, call):
    dt = 1. / 360
    N = int(T / dt)
    mu = r - q - (sigma ** 2) / 2.0
    smax = 2 * abs(mu) * dt ** .5
    smax = max(smax, sigma * (2 ** .5))
    if smax == 0:
        return -9999
    M = int(5 * (N ** .5))
    C_ = np.empty(2 * M + 1, dtype=np.float64)
    pC_ = np.empty(2 * M + 1, dtype=np.float64)
    S_ = np.empty(2 * M + 1, dtype=np.float64, )
    p = float(0.5 * (sigma ** 2)) / (smax ** 2)
    p_u = p + 0.5 * mu * dt ** .5 / float(smax)
    p_m = 1 - 2 * p
    p_d = p - 0.5 * mu * dt ** .5 / float(smax)
    D = 1.0 / (1 + r * dt)
    E = math.exp(smax * dt ** .5)

    for j in range(0, len(S_)):
        if j == 0:
            S_[j] = S * math.exp(-M * smax * dt ** .5)
        else:
            S_[j] = S_[j - 1] * E
        if call == True:
            C_[j] = max(S_[j] - K, 0)
        else:
            C_[j] = max(K - S_[j], 0)

    for k in range(0, N):
        for j in range(1, 2 * M):
            pC_[j] = (p_u * C_[j + 1] + p_m * C_[j] + p_d * C_[j - 1]) * D
        pC_[0] = 2 * pC_[1] - pC_[2]
        pC_[2 * M] = 2 * pC_[2 * M - 1] - pC_[2 * M - 2]

        for n in range(0, 2 * M + 1):
            if call == True:
                C_[n] = max(pC_[n], max(S_[n] - K, 0))
            else:
                C_[n] = max(pC_[n], max(K - S_[n], 0))
    ret = C_[M]
```

```
    return ret

partd_answer2=american_trinominal_model(S=S, K=K, r=r, q=q, T=T, sigma=sigma, ca
print("Using the trinominal model the price is",partd_answer2.round(4))
print("The part b price is ", partb_answer.round(4))
```

```
Using the trinominal model the price is 5.7037
The part b price is  5.6921
```

e part

Use a Monte Carlo pricer to reconcile with price from part a.

In [ ]:

```
def monte_carlo_european_put(S0, K, r, sigma, T, num_simulations):
    np.random.seed(42)  # For reproducibility
    num_timesteps = 10
    dt = T / num_timesteps

    total_payoff = 0
    for i in range(num_simulations):
        S = S0
        for j in range(num_timesteps):
            z = np.random.standard_normal()
            S *= np.exp((r - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z)

        total_payoff += max(K - S, 0)

    price = np.exp(-r * T) * total_payoff / num_simulations
    return price

# Parameters from part (a)

num_simulations = 10000
monte_carlo_price = monte_carlo_european_put(S, K, r, sigma, T, num_simulations)

print("Monte Carlo European Put price:", monte_carlo_price.round(4))
```

```
Monte Carlo European Put price: 5.6675
```