Name : NGUEPONWOUO Joseph-Marie

Project : Finite Difference Method for a European Option

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import norm
import time
import numpy.linalg as lng
from scipy.sparse import csr_matrix as sparse
from scipy.sparse.linalg import spsolve
```

1. The Euler Forward Scheme

$$\frac{U_j^{n+1}-U_j^n}{\Delta t}+\frac{\sigma^2}{2}s_j^2\frac{-U_{j-1}^n+2U_j^n-U_{j+1}^n}{h^2}-rs_j\frac{U_{j+1}^n-U_{j-1}^n}{2h}+rU_j^n=0$$
$n=0,\dots N-1; j=1,\dots,I$

$U_0^n=v_l(t_n)=Ke^{-rt_n}-S_{min}; n=0,\dots,N$

$U_{I+1}=v_r(t_n)=0; n=0,\dots,N$

$U_j^0=\varphi(s_j)=(K-s_j)_+; j=1,\dots,I$

2. Programming Explicit Euler Scheme

2.2. Programming indications

The class "PutOption" below contains all the attributes and methods useful to solve the questions related to this project.

In [15]:
```
class PutOption:
    def __init__(self, r = 0.1, sigma = 0.2, K = 100, T = 1, Smin = 0, Sm
        self.r = r
        self.sigma = sigma
        self.K = K
        self.T = T
        self.Smin = Smin
        self.Smax = Smax
        self.I = I
        self.N = N
        self.scheme = scheme
        self.dt = T / N
        self.h = (Smax - Smin) / (I + 1)
        self.s = Smin + self.h * np.arange(1, I + 1)
```

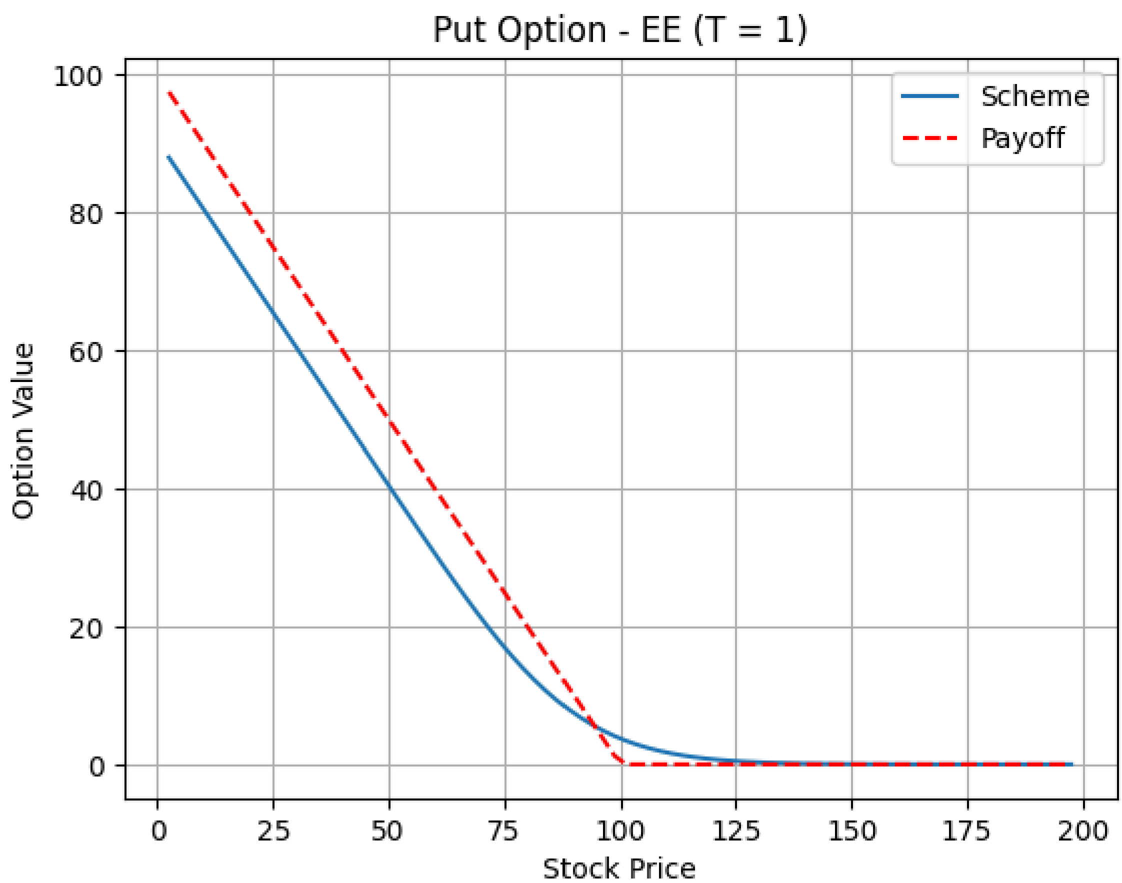
```python
        self.U = self.phi(self.s)
        self.alpha = sigma**2 * self.s**2 / (2 * self.h**2)
        self.beta = r * self.s / (2 * self.h)

    def phi(self, s):
        return np.maximum(self.K - s, 0).reshape(self.I, 1)

    def u_left(self, t):
        return self.K * np.exp(-self.r * t) - self.Smin

    def u_right(self, t):
        return 0

    def construct_A(self):
        A = np.zeros((self.I, self.I))
        for j in range(0, self.I):
          A[j, j] = 2 * self.alpha[j] + self.r
          if j-1 >= 0:
            A[j, j-1] = - self.alpha[j] + self.beta[j]
          if j+1 < self.I:
            A[j, j+1] = - self.alpha[j] - self.beta[j]
        return A

    def construct_A_sparse(self):
        row = []
        col = []
        data = []

        for i in range(self.I):
            if i > 0:  # Lower diagonal
                row.append(i)
                col.append(i - 1)
                data.append(-self.alpha[i] + self.beta[i])
            # Main diagonal
            row.append(i)
            col.append(i)
            data.append(2 * self.alpha[i] + self.r)
            if i < self.I - 1:  # Upper diagonal
                row.append(i)
                col.append(i + 1)
                data.append(-self.alpha[i] - self.beta[i])

        return sparse((data, (row, col)), shape=(self.I, self.I))

    def q(self, t):
        y = np.zeros((self.I, 1))
        y[0] = (-self.alpha[0] + self.beta[0]) * self.u_left(t)
        y[-1] = (-self.alpha[-1] - self.beta[-1]) * self.u_right(t)
        return y

    def explicit_euler(self):
        A = self.construct_A()
        for n in range(self.N):
            t = n * self.dt
            self.U = self.U - self.dt * (A @ self.U + self.q(t))
        return self.U

    def implicit_euler(self):
        A = self.construct_A()
        Id = np.identity(self.I)
```

```
        B = Id + self.dt * A
        for n in range(self.N):
            t = (n+1) * self.dt
            self.U = lng.solve(B, self.U - self.dt * self.q(t))
        return self.U

    def crank_nicolson(self):
       A = self.construct_A()
       Id = np.identity(self.I)
       B = Id + (self.dt / 2) * A
       C = Id - (self.dt / 2) * A
       for n in range(self.N):
           t = (n + 1) * self.dt
           self.U = lng.solve(B, C @ self.U - (self.dt / 2) * (self.q(t -
       return self.U

    def black_scholes(self, S, t):
        d1 = (np.log(S / self.K) + (self.r + 0.5 * self.sigma**2) * (self
        d2 = d1 - self.sigma * np.sqrt(self.T - t)
        return self.K * np.exp(-self.r * (self.T - t)) * norm.cdf(-d2) -


    def interpolate_value(self, s_bar):
        for i in range(len(self.s) - 1):
            if self.s[i] <= s_bar <= self.s[i + 1]:
                s_i, s_ip1 = self.s[i], self.s[i + 1]
                if self.scheme == 'EE':
                    temp = self.explicit_euler()
                elif self.scheme == 'IE':
                    temp = self.implicit_euler()
                elif self.scheme == 'CN':
                    temp = self.crank_nicolson()
                else:
                    raise ValueError("Invalid scheme. Choose 'EE', 'IE',
                U_i, U_ip1 = temp[i], temp[i + 1]
                interpolated_value = ((s_ip1 - s_bar) / self.h) * U_i + (
                return interpolated_value[-1]
        raise ValueError("s_bar is out of bounds.")

    def compute_error_table(self, Sval):
        alphas = []
        errex = []
        h_vals = []
        Uvals = []
        alphas = [0.0000, 0.00]
        tab = []
        for I_val in [10, 20, 40, 80, 160, 320]:
            N_val = (I_val ** 2) // 10
            #N_val = I_val
            dt_val = self.T / N_val
            h_val = (self.Smax - self.Smin) / (I_val + 1)
            s_val = self.Smin + h_val * np.arange(1, I_val + 1)

            # Update parameters
            self.dt = dt_val
            self.h = h_val
            self.s = s_val
            self.I = I_val
            self.N = N_val
            self.alpha = self.sigma**2 * self.s**2 / (2 * self.h**2)
```

```
            self.beta = self.r * self.s / (2 * self.h)
            self.U = self.phi(self.s)
            exact = self.black_scholes(Sval, 0)


            start_time = time.time()
            U_ = self.interpolate_value(Sval)
            end_time = time.time()
            h_vals.append(h_val)
            Uvals.append(U_)
            errex.append(abs(U_ - exact))
            tab.append([I_val, N_val, U_, abs(U_ - exact), end_time - sta

        tab[0][3] = 0
        errors = [abs(Uvals[j+1] - Uvals[j]) for j in range(len(Uvals)-1)
        errors.insert(0, 0)
        orders = []
        for k in range(2, len(Uvals)):
            alpha_k = (np.log(abs(Uvals[k-2]- Uvals[k-1]) / abs(Uvals[k]
            alphas.append(alpha_k)
            beta_k = alpha_k / 2
            orders.append([k, alpha_k, beta_k])
        tab = np.insert(tab, 3, errors, axis=1)
        tab = np.insert(tab, 4, alphas, axis=1)
        tab = np.array(tab)
        df = pd.DataFrame(tab, columns=['I', 'N', 'U(s)', 'error', 'alpha
        return df

    def run(self):
        if self.scheme == 'EE':
            self.explicit_euler()
        elif self.scheme == 'IE':
            self.implicit_euler()
        elif self.scheme == 'CN':
            self.crank_nicolson()

        # Plot results
        plt.plot(self.s, self.U, label = 'Scheme')
        plt.plot(self.s, [max(self.K - p, 0) for p in self.s], 'r--', lab
        plt.xlabel('Stock Price')
        plt.ylabel('Option Value')
        plt.title(f'Put Option - {self.scheme} (T = {self.T})')
        plt.legend()
        plt.grid()
        plt.show()
```

- Figure of Scheme for "EE"

In [83]:
```
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=80,
option.run()
```



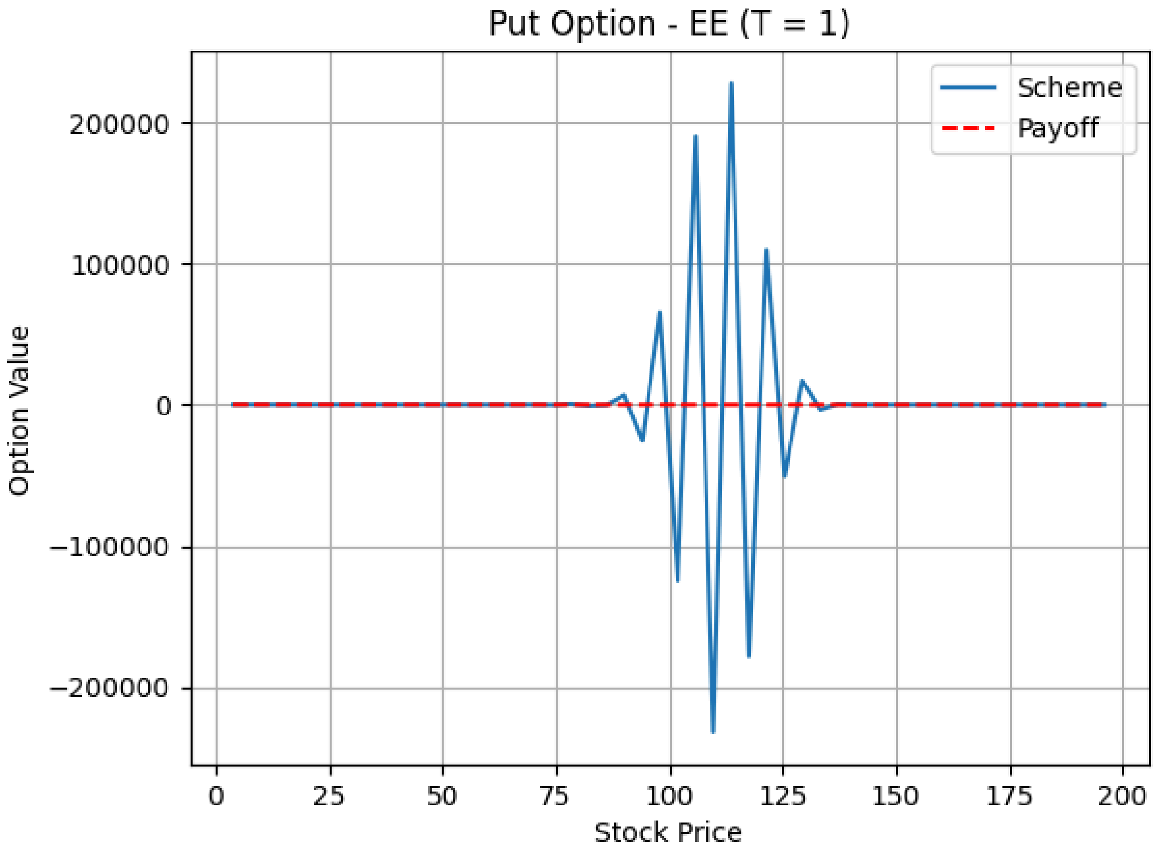
d) Program the matrix A of size $I * I$

In [5]:
```
def construct_A(self):
  A = np.zeros((self.I, self.I))
  for j in range(0, self.I):
    A[j, j] = 2 * self.alpha[j] + self.r
    if j-1 >= 0:
      A[j, j-1] = - self.alpha[j] + self.beta[j]
    if j+1 < self.I:
      A[j, j+1] = - self.alpha[j] - self.beta[j]
  return A
```

3. First numerical tests

a) Test the Euler Forward Scheme (EE)

In [6]:
```
# N = 10, I = 50
N = 10
I = 50

option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.run()
```



Remark : We can see that for some values for N, we have the value of U^n that explodes. So this scheme is not always numerically stable.

b) Let's look at the amplification of $B := I_d - \Delta t A$.

Check that the coefficients of B are not all positive and that they may have a modulus greater than 1.

In [7]:
```
# N = 10 , I = 50
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=50,
B = np.identity(option.I) - option.dt * option.construct_A()
print(B)
```

```
[[ 9.860e-01  7.000e-03  0.000e+00 ...  0.000e+00  0.000e+00  0.000e+00]
 [-2.000e-03  9.740e-01  1.800e-02 ...  0.000e+00  0.000e+00  0.000e+00]
 [ 0.000e+00  3.000e-03  9.540e-01 ...  0.000e+00  0.000e+00  0.000e+00]
 ...
 [ 0.000e+00  0.000e+00  0.000e+00 ... -8.226e+00  4.848e+00  0.000e+00]
 [ 0.000e+00  0.000e+00  0.000e+00 ...  4.557e+00 -8.614e+00  5.047e+00]
 [ 0.000e+00  0.000e+00  0.000e+00 ...  0.000e+00  4.750e+00 -9.010e+00]]
```

- Compute the norm $||B||_\infty$

In [8]:
```
print("Norm 1 of B:", np.linalg.norm(option.construct_A(), np.inf))
```

```
Norm 1 of B: 192.18000000000006
```

- Compute the norm $||B||_2$

In [9]:
```
print("Norm 2 of B:", np.linalg.norm(option.construct_A(), 2))
```

```
Norm 2 of B: 174.73949111980585
```

- Let's now check that for $N = I = 10$, the coefficients of B are almost all positive and smaller than 1.

In [10]:
```
N = I = 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
B = np.identity(option.I) - option.dt * option.construct_A()
print(B)
```

```
[[ 0.986  0.007  0.     0.     0.     0.     0.     0.     0.     0.    ]
 [-0.002  0.974  0.018  0.     0.     0.     0.     0.     0.     0.    ]
 [ 0.     0.003  0.954  0.033  0.     0.     0.     0.     0.     0.    ]
 [ 0.     0.     0.012  0.926  0.052  0.     0.     0.     0.     0.    ]
 [ 0.     0.     0.     0.025  0.89   0.075  0.     0.     0.     0.    ]
 [ 0.     0.     0.     0.     0.042  0.846  0.102  0.     0.     0.    ]
 [ 0.     0.     0.     0.     0.     0.063  0.794  0.133  0.     0.    ]
 [ 0.     0.     0.     0.     0.     0.     0.088  0.734  0.168  0.    ]
 [ 0.     0.     0.     0.     0.     0.     0.     0.117  0.666  0.207]
 [ 0.     0.     0.     0.     0.     0.     0.     0.     0.15   0.59 ]]
```

Remark : We can notice from the matrix B above that all coefficients of B are positive and almost smaller than 1.

c) For the same previous values $(N, I) = (10, 10)$ or $(10, 50)$, compute the CFL number defined here as:

$$\mu := \frac{\Delta t}{h^2}\sigma^2 S_{max}^2$$

In [11]:
```
# I = N = 10
I = N = 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
mu = (option.dt / (option.h ** 2)) * (option.sigma ** 2) * (option.Smax *
print("CFL number for (N, I) = (10, 10):", mu)
```

```
CFL number for (N, I) = (10, 10): 0.484
```

In [12]:
```
# N = 10 ; I = 50
N = 10
I = 50
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
mu = (option.dt / (option.h ** 2)) * (option.sigma ** 2) * (option.Smax *
print("CFL number for (N, I) = (10, 50):", mu)
```

```
CFL number for (N, I) = (10, 50): 10.404000000000003
```

Remark: From the computations above, we noticed that the oscillation problem occurs for $(10, 50)$ but not for $(10, 10)$. Then, we can conclude that there is no stability problem when μ is sufficiently small. This can also be seen by taking Δt very small (N very large) compared to h^2.

d) Compute the P_1- interpolated value at $\bar{s} = 90$. If $\bar{s} \in [s_i, s_{i+1}]$, then this interpolated value can be obtained by using the affine (P_1) approximation.

In [13]:
```
s_bar = 90
N = 10
I = 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
print("P1 - interpolated value:", option.interpolate_value(s_bar))
```

```
P1 - interpolated value: 6.296946349254133
```

e) Numerical order of the scheme (TABLES FOR $N = I$ AND $N = I^2/10$)

In [10]:
```
# N = I
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=10,
option.compute_error_table(80)
```

Out[10]:

	I	N	U(s)	error	alpha	errex	tc
0	10.0	10.0	1.425509e+01	0.000000e+00	0.000000	0.000000e+00	0.0002
1	20.0	20.0	1.351532e+01	7.397757e-01	0.000000	2.416537e-01	0.0005
2	40.0	40.0	1.332033e+01	1.949874e-01	1.992995	4.666637e-02	0.0014
3	80.0	80.0	-1.061474e+04	1.062806e+04	-16.017683	1.062801e+04	0.0018
4	160.0	160.0	-8.121408e+69	8.121408e+69	-220.831897	8.121408e+69	0.0110
5	320.0	320.0	NaN	NaN	NaN	NaN	0.0153

Remark : We can easily notice from the table above the stability issue characterised by very large values for the payoff (even negative).

In [16]:
```
# N = (I ** 2) / 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=10,
option.compute_error_table(80)
```

Out[16]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	10.0	14.255092	0.000000	0.000000	0.000000	0.000292
1	20.0	40.0	13.547634	0.707459	0.000000	0.273971	0.000961
2	40.0	160.0	13.345106	0.202528	1.869520	0.071443	0.003827
3	80.0	640.0	13.291930	0.053175	1.964063	0.018267	0.009169
4	160.0	2560.0	13.278284	0.013646	1.979940	0.004621	0.071863
5	320.0	10240.0	13.274825	0.003459	1.989035	0.001162	0.471527

4. Implicit Euler (IE) Scheme

Demonstration of the IE scheme :

From the "IE" scheme, we have :

$$\frac{U_j^{n+1}-U_j^n}{\Delta t}+\frac{\sigma^2}{2}s_j^2\frac{-U_{j-1}^{n+1}+2U_j^{n+1}-U_{j+1}^{n+1}}{h^2}-rs_j\frac{U_{j+1}^{n+1}-U_{j-1}^{n+1}}{2h}+rU_j^{n+1}=0$$
$$n=0,\dots,N-1$$
$$j=1,\dots,I$$

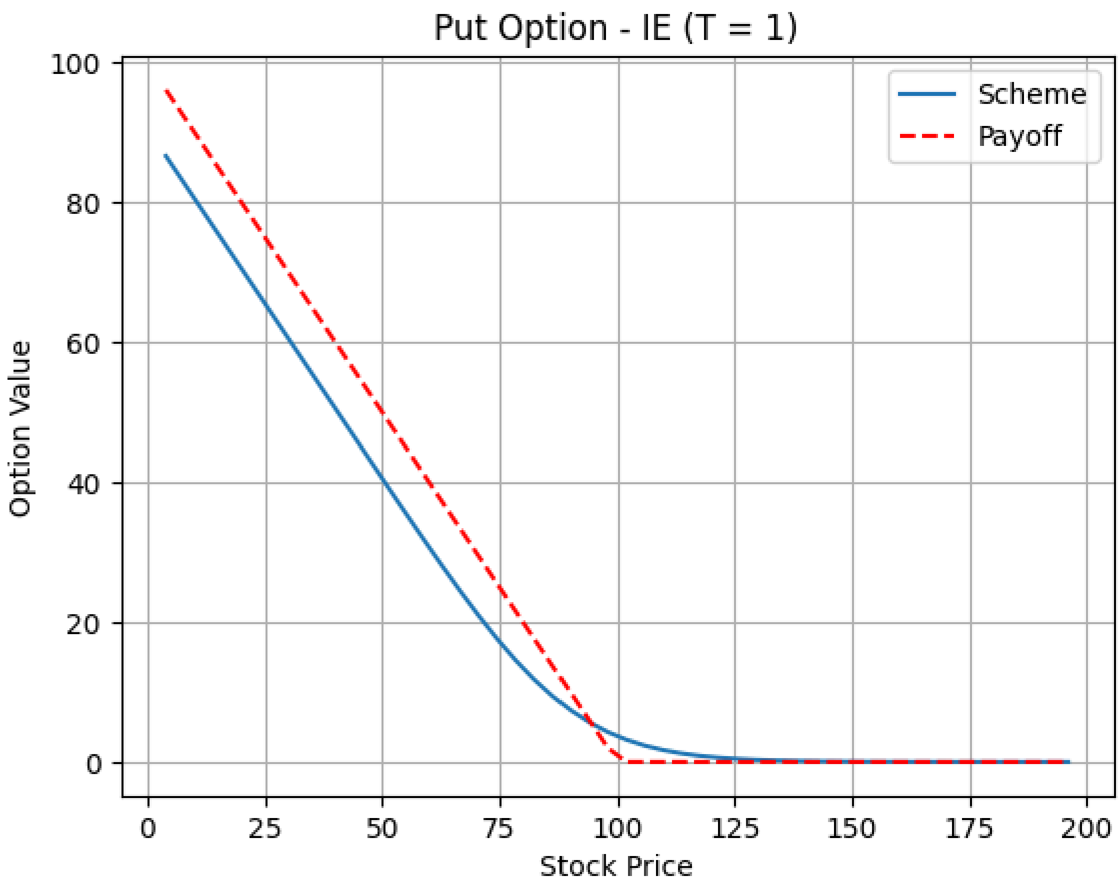
Apart from this term, $\frac{U_j^{n+1}-U_j^n}{\Delta t}$, we identify the same scheme as in the "EE" scheme with U^{n+1} in place of U^n. Hence, we have:

$$\frac{U_j^{n+1}-U_j^n}{\Delta t}+AU^{n+1}+q(t_{n+1})=0$$
$$n=0,\dots,N-1$$
$$U^0=(\varphi(s_i))_{1\le i\le I}$$

a) Check that with the IE scheme, there is no more stability problems

- Figure of scheme for "IE"

In [84]:
```
# I = 50, N = 10
I = 50
N = 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.run()
```



b) Computation Table

- N = I

In [9]:
```
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.compute_error_table(80)
```

Out[9]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	10.0	14.448406	0.000000	0.000000	0.000000	0.000393
1	20.0	20.0	13.642159	0.806247	0.000000	0.368496	0.001068
2	40.0	40.0	13.386145	0.256015	1.714605	0.112482	0.009551
3	80.0	80.0	13.310502	0.075642	1.790659	0.036839	0.015228
4	160.0	160.0	13.287066	0.023436	1.705703	0.013403	0.095420
5	320.0	320.0	13.279089	0.007977	1.561785	0.005426	0.939561

- N = I / 10

In [14]:
```
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.compute_error_table(80)
```

Out[14]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	1.0	15.115616	0.000000	0.000000	0.000000	0.000202
1	20.0	2.0	14.135687	0.979929	0.000000	0.862024	0.000332
2	40.0	4.0	13.667824	0.467863	1.105006	0.394161	0.000515
3	80.0	8.0	13.456693	0.211131	1.168633	0.183030	0.001723
4	160.0	16.0	13.360985	0.095708	1.151718	0.087322	0.017526
5	320.0	32.0	13.316197	0.044788	1.100467	0.042534	0.094994

5. Crank Nicolson Scheme

a) Program the Crank-Nicolson scheme (CN)

- Write the CN scheme in vector form :

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \frac{1}{2}\left(\frac{\sigma^2}{2} s_j^2 \frac{-U_{j-1}^{n+1} + 2U_j^{n+1} - U_{j+1}^{n+1}}{h^2} - r s_j \frac{U_{j+1}^{n+1} - U_{j-1}^{n+1}}{2h} + rU_j^{n+1}\right)$$

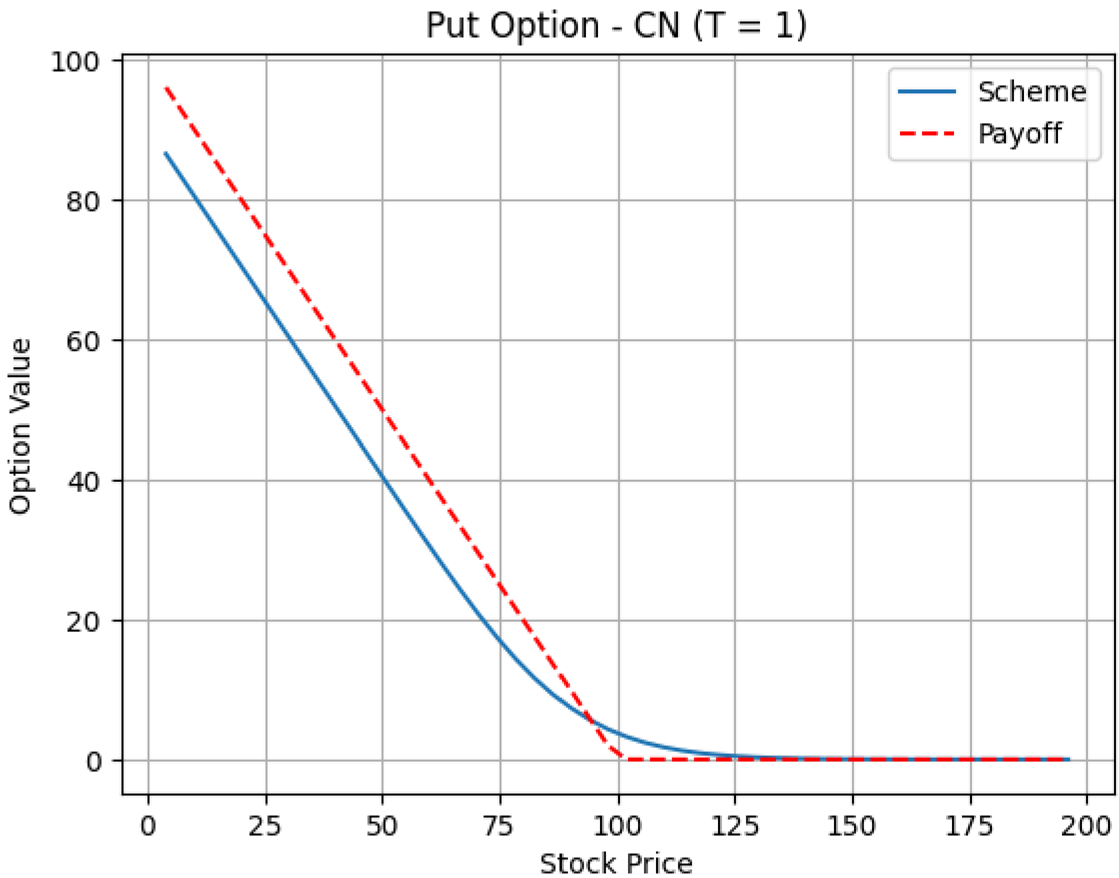
$$n = 0, \dots, N -$$
$$j = 1, \dots, I$$

From the "EE" and "IE" schemes, we can see that the vector form of the "CN" scheme is given by :

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + \frac{1}{2}\left(AU^{n+1} + q(t_{n+1})\right) + \frac{1}{2}\left(AU^n + q(t_n)\right) = 0$$
$$\implies \left(Id + \frac{1}{2}\Delta t A\right) U^{n+1} = (Id - \frac{1}{2}\Delta t A)U^n - \frac{1}{2}\Delta t \left(q(t_n) + q(t_{n+1})\right)$$

- Figure of scheme for "CN"

In [85]:
```
# I = 50, N = 10
I = 50
N = 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.run()
```



b) Draw the corresponding table with $I = N$ and $N = I/10$.

The numerical order should be clear for $N = I$.

In [8]:
```
# N = I = 10
N = I = 10
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.compute_error_table(80)
```

Out[8]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	10.0	14.353451	0.000000	0.000000	0.000000	0.000686
1	20.0	20.0	13.578892	0.774559	0.000000	0.305229	0.002233
2	40.0	40.0	13.353140	0.225752	1.842696	0.079477	0.007711
3	80.0	80.0	13.293944	0.059195	1.965990	0.020281	0.014035
4	160.0	160.0	13.278788	0.015156	1.983267	0.005125	0.097610
5	320.0	320.0	13.274951	0.003837	1.990918	0.001288	3.011112

- $N = I/10$

In [6]:

```
option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
option.compute_error_table(80)
```

Out[6]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	1.0	14.230461	0.000000	0.000000	0.000000	0.000306
1	20.0	2.0	13.506181	0.724281	0.000000	0.232518	0.004908
2	40.0	4.0	13.334783	0.171397	2.154094	0.061120	0.000674
3	80.0	8.0	13.288798	0.045985	1.932315	0.015135	0.002072
4	160.0	16.0	13.277492	0.011307	2.042244	0.003829	0.010620
5	320.0	32.0	13.274627	0.002865	1.989305	0.000964	0.091959

Discussion on numerical tests :

We can say that both schemes, in case of stability, we have a good convergence of the solution of the PDE as N and I increase. This can be noticed from $errex$ which tends to 0 as N and I increase. The different schemes are then consistent and coherent with theoretical analysis.

Complement 1: Programmation of the Black Scholes formula done in the class "PutOption" above.

In [86]:

```
def black_scholes(self, S, t):
    d1 = (np.log(S / self.K) + (self.r + 0.5 * self.sigma**2) * (self.T -
    d2 = d1 - self.sigma * np.sqrt(self.T - t)
    return self.K * np.exp(-self.r * (self.T - t)) * norm.cdf(-d2) - S *
```

Complement 2: Program the sparse matrices.

In [20]:

```
class PutOption:
    def __init__(self, r = 0.1, sigma = 0.2, K = 100, T = 1, Smin = 0, Sm
```

```
        self.r = r
        self.sigma = sigma
        self.K = K
        self.T = T
        self.Smin = Smin
        self.Smax = Smax
        self.I = I
        self.N = N
        self.scheme = scheme
        self.dt = T / N
        self.h = (Smax - Smin) / (I + 1)
        self.s = Smin + self.h * np.arange(1, I + 1)
        self.U = self.phi(self.s)
        self.alpha = sigma**2 * self.s**2 / (2 * self.h**2)
        self.beta = r * self.s / (2 * self.h)

    def phi(self, s):
        return np.maximum(self.K - s, 0).reshape(self.I, 1)

    def u_left(self, t):
        return self.K * np.exp(-self.r * t) - self.Smin

    def u_right(self, t):
        return 0

    def construct_A(self):
        A = np.zeros((self.I, self.I))
        for j in range(0, self.I):
          A[j, j] = 2 * self.alpha[j] + self.r
          if j-1 >= 0:
            A[j, j-1] = - self.alpha[j] + self.beta[j]
          if j+1 < self.I:
            A[j, j+1] = - self.alpha[j] - self.beta[j]
        return A

    def construct_A_sparse(self):
        row = []
        col = []
        data = []

        for i in range(self.I):
            if i > 0:
                row.append(i)
                col.append(i - 1)
                data.append(-self.alpha[i] + self.beta[i])

            row.append(i)
            col.append(i)
            data.append(2 * self.alpha[i] + self.r)
            if i < self.I - 1:
                row.append(i)
                col.append(i + 1)
                data.append(-self.alpha[i] - self.beta[i])

        return sparse((data, (row, col)), shape=(self.I, self.I))

    def q(self, t):
        y = np.zeros((self.I, 1))
        y[0] = (-self.alpha[0] + self.beta[0]) * self.u_left(t)
        y[-1] = (-self.alpha[-1] - self.beta[-1]) * self.u_right(t)
```

```
        return y

    def explicit_euler(self):
        A = self.construct_A_sparse()
        for n in range(self.N):
            t = n * self.dt
            self.U = self.U - self.dt * (A @ self.U + self.q(t))
        return self.U

    def implicit_euler(self):
        A = self.construct_A_sparse()
        Id = np.identity(self.I)
        B = Id + self.dt * A
        for n in range(self.N):
            t = (n+1) * self.dt
            self.U = (spsolve (B, self.U - self.dt * self.q(t))).reshape(
        return self.U

    def crank_nicolson(self):
       A = self.construct_A_sparse()
       Id = np.identity(self.I)
       B = Id + (self.dt / 2) * A
       C = Id - (self.dt / 2) * A
       for n in range(self.N):
           t = (n + 1) * self.dt
           self.U = (spsolve(B, C @ self.U - (self.dt / 2) * (self.q(t -
       return self.U

    def black_scholes(self, S, t):
        d1 = (np.log(S / self.K) + (self.r + 0.5 * self.sigma**2) * (self
        d2 = d1 - self.sigma * np.sqrt(self.T - t)
        return self.K * np.exp(-self.r * (self.T - t)) * norm.cdf(-d2) -


    def interpolate_value(self, s_bar):
        for i in range(len(self.s) - 1):
            if self.s[i] <= s_bar <= self.s[i + 1]:
                s_i, s_ip1 = self.s[i], self.s[i + 1]
                if self.scheme == 'EE':
                    temp = self.explicit_euler()
                elif self.scheme == 'IE':
                    temp = self.implicit_euler()
                elif self.scheme == 'CN':
                    temp = self.crank_nicolson()
                else:
                    raise ValueError("Invalid scheme. Choose 'EE', 'IE',
                U_i, U_ip1 = temp[i], temp[i + 1]
                interpolated_value = ((s_ip1 - s_bar) / self.h) * U_i + (
                return interpolated_value[-1]
        raise ValueError("s_bar is out of bounds.")

    def compute_error_table(self, Sval):
        alphas = []
        errex = []
        h_vals = []
        Uvals = []
        alphas = [0.0000, 0.00]
        tab = []
        for I_val in [10, 20, 40, 80, 160, 320]:
            N_val = (I_val**2) // 10
```

```python
            #N_val = I_val
            dt_val = self.T / N_val
            h_val = (self.Smax - self.Smin) / (I_val + 1)
            s_val = self.Smin + h_val * np.arange(1, I_val + 1)

            # Update parameters
            self.dt = dt_val
            self.h = h_val
            self.s = s_val
            self.I = I_val
            self.N = N_val
            self.alpha = self.sigma**2 * self.s**2 / (2 * self.h**2)
            self.beta = self.r * self.s / (2 * self.h)
            self.U = self.phi(self.s)
            exact = self.black_scholes(Sval, 0)


            start_time = time.time()
            U_ = self.interpolate_value(Sval)
            end_time = time.time()
            h_vals.append(h_val)
            Uvals.append(U_)
            errex.append(abs(U_ - exact))
            tab.append([I_val, N_val, U_, abs(U_ - exact), end_time - sta

        tab[0][3] = 0
        errors = [abs(Uvals[j+1] - Uvals[j]) for j in range(len(Uvals)-1)
        errors.insert(0, 0)
        orders = []
        for k in range(2, len(Uvals)):
            alpha_k = (np.log(abs(Uvals[k-2]- Uvals[k-1]) / abs(Uvals[k]
            alphas.append(alpha_k)
            beta_k = alpha_k / 2
            orders.append([k, alpha_k, beta_k])
        tab = np.insert(tab, 3, errors, axis=1)
        tab = np.insert(tab, 4, alphas, axis=1)
        tab = np.array(tab)
        df = pd.DataFrame(tab, columns=['I', 'N', 'U(s)', 'error', 'alpha
        return df

    def run(self):
        if self.scheme == 'EE':
            self.explicit_euler()
        elif self.scheme == 'IE':
            self.implicit_euler()
        elif self.scheme == 'CN':
            self.crank_nicolson()

        # Plot results
        plt.plot(self.s, self.U, label = 'Scheme')
        plt.plot(self.s, [max(self.K - p, 0) for p in self.s], 'r--', lab
        plt.xlabel('Stock Price')
        plt.ylabel('Option Value')
        plt.title(f'Put Option - {self.scheme} (T = {self.T})')
        plt.legend()
        plt.grid()
        plt.show()
```

- Improvement of execution time for "EE" ($N = I^2/10$)

```
In [21]: option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
         option.compute_error_table(80)
```

Out[21]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	10.0	14.255092	0.000000	0.000000	0.000000	0.001168
1	20.0	40.0	13.547634	0.707459	0.000000	0.273971	0.012897
2	40.0	160.0	13.345106	0.202528	1.869520	0.071443	0.014936
3	80.0	640.0	13.291930	0.053175	1.964063	0.018267	0.057537
4	160.0	2560.0	13.278284	0.013646	1.979940	0.004621	0.132647
5	320.0	10240.0	13.274825	0.003459	1.989035	0.001162	0.272238

- Improvement of execution time for "IE" ($N = I/10$)

```
In [18]: I = N = 10
         option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
         option.compute_error_table(80)
```

Out[18]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	1.0	15.115616	0.000000	0.000000	0.000000	0.019329
1	20.0	2.0	14.135687	0.979929	0.000000	0.862024	0.001873
2	40.0	4.0	13.667824	0.467863	1.105006	0.394161	0.002576
3	80.0	8.0	13.456693	0.211131	1.168633	0.183030	0.008545
4	160.0	16.0	13.360985	0.095708	1.151718	0.087322	0.015194
5	320.0	32.0	13.316197	0.044788	1.100467	0.042534	0.069265

- Improvement of execution time for "CN" ($N = I/10$)

```
In [19]: option = PutOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N
         option.compute_error_table(80)
```

Out[19]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	1.0	14.230461	0.000000	0.000000	0.000000	0.002608
1	20.0	2.0	13.506181	0.724281	0.000000	0.232518	0.004488
2	40.0	4.0	13.334783	0.171397	2.154094	0.061120	0.002958
3	80.0	8.0	13.288798	0.045985	1.932315	0.015135	0.011206
4	160.0	16.0	13.277492	0.011307	2.042244	0.003829	0.026732
5	320.0	32.0	13.274627	0.002865	1.989305	0.000964	0.117092

Conclusion : Compared to the values obtained before, we can notice a good improvement of execution time by using sparse matrices.

Complement 3: Program the Call option

a) Adequate left and right boundary conditions for the call option

$$v(t, S_{min}) = v_l(t) = 0, t \in (0, T)$$
$$v(t, S_{max}) = v_r(t) = S_{max} - Ke^{-rt}, t \in (0, T)$$
$$v(0, s) = \varphi(s) = (s - K)_+, s \in (S_{min}, S_{max})$$

b) PDE with boundary conditions for the call option

$$\frac{\partial c}{\partial t} + rs\frac{\partial c}{\partial s} + \frac{\sigma^2}{2}s^2\frac{\partial^2 c}{\partial s^2} - rc = 0$$
$$v(t, S_{min}) = v_l(t) = 0$$
$$v(t, S_{max}) = v_r(t) = S_{max} - Ke^{-rt}$$
$$v(0, s) = \varphi(s) = (s - K)_+, s \in (S_{min}, S_{max})$$

c) Program and test the Implicit Euler Scheme for the call option

In [25]:
```
class CallOption:
    def __init__(self, r = 0.1, sigma = 0.2, K = 100, T = 1, Smin = 0, Sm
        self.r = r
        self.sigma = sigma
        self.K = K
        self.T = T
        self.Smin = Smin
        self.Smax = Smax
        self.I = I
        self.N = N
        self.scheme = scheme
        self.dt = T / N
        self.h = (Smax - Smin) / (I + 1)
        self.s = Smin + self.h * np.arange(1, I + 1)
        self.U = self.phi(self.s)
        self.alpha = sigma**2 * self.s**2 / (2 * self.h**2)
        self.beta = r * self.s / (2 * self.h)

    def phi(self, s):
        return np.maximum(s - self.K, 0).reshape(self.I, 1)

    def u_left(self, t):
        return 0

    def u_right(self, t):
        return self.Smax - self.K * np.exp(-self.r * t)

    def construct_A(self):
        A = np.zeros((self.I, self.I))
        for j in range(0, self.I):
          A[j, j] = 2 * self.alpha[j] + self.r
          if j-1 >= 0:
```

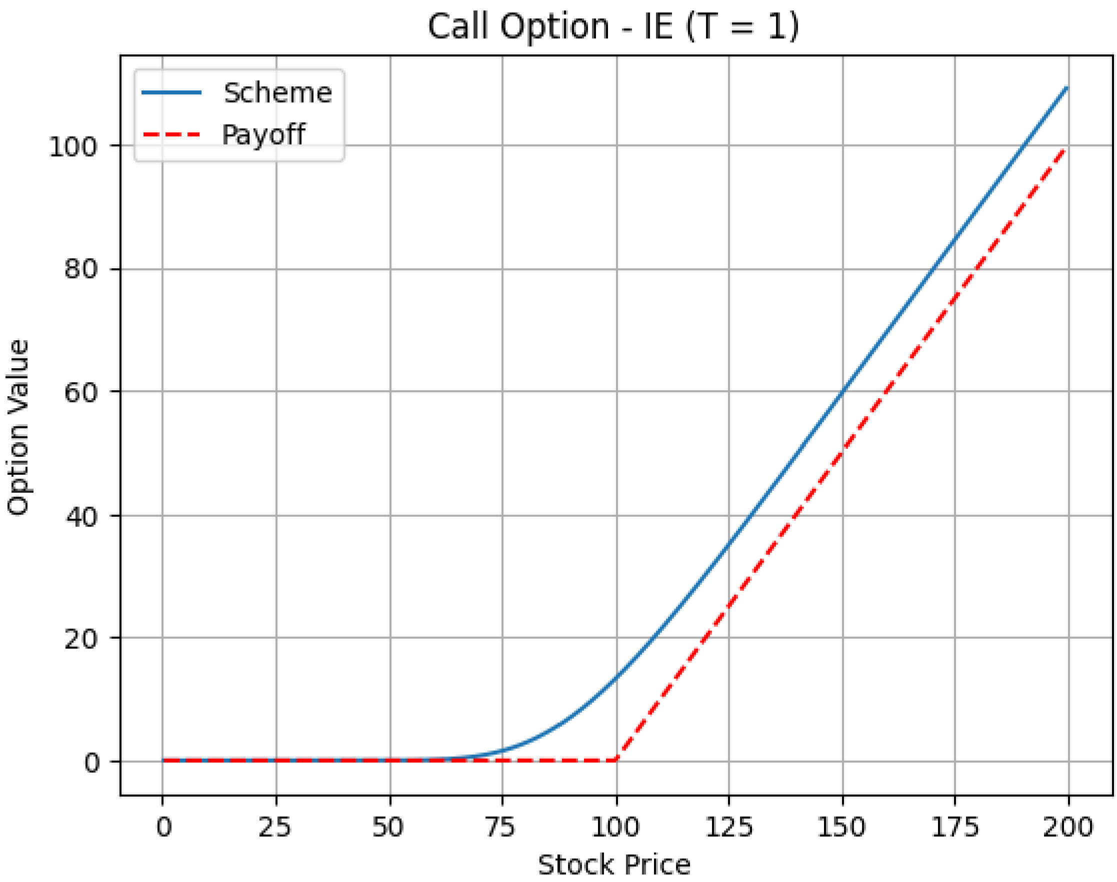
```
                A[j, j-1] = - self.alpha[j] + self.beta[j]
              if j+1 < self.I:
                A[j, j+1] = - self.alpha[j] - self.beta[j]
            return A

        def q(self, t):
            y = np.zeros((self.I, 1))
            y[0] = (-self.alpha[0] + self.beta[0]) * self.u_left(t)
            y[-1] = (-self.alpha[-1] - self.beta[-1]) * self.u_right(t)
            return y

        def explicit_euler(self):
            A = self.construct_A()
            for n in range(self.N):
                t = n * self.dt
                self.U = self.U - self.dt * (A @ self.U + self.q(t))
            return self.U

        def implicit_euler(self):
            A = self.construct_A()
            Id = np.identity(self.I)
            B = Id + self.dt * A
            for n in range(self.N):
                t = (n+1) * self.dt
                self.U = lng.solve(B, self.U - self.dt * self.q(t))
            return self.U

        def crank_nicolson(self):
           A = self.construct_A()
           Id = np.identity(self.I)
           B = Id + (self.dt / 2) * A
           C = Id - (self.dt / 2) * A
           for n in range(self.N):
               t = (n + 1) * self.dt
               self.U = lng.solve(B, C @ self.U - (self.dt / 2) * (self.q(t -
           return self.U

        def black_scholes(self, S, t):
            d1 = (np.log(S / self.K) + (self.r + 0.5 * self.sigma**2) * (self
            d2 = d1 - self.sigma * np.sqrt(self.T - t)
            return S * norm.cdf(d1) - self.K * np.exp(-self.r * (self.T - t))

        def interpolate_value(self, s_bar):
            for i in range(len(self.s) - 1):
                if self.s[i] <= s_bar <= self.s[i + 1]:
                    s_i, s_ip1 = self.s[i], self.s[i + 1]
                    if self.scheme == 'EE':
                        temp = self.explicit_euler()
                    elif self.scheme == 'IE':
                        temp = self.implicit_euler()
                    elif self.scheme == 'CN':
                        temp = self.crank_nicolson()
                    else:
                        raise ValueError("Invalid scheme. Choose 'EE', 'IE',
                    U_i, U_ip1 = temp[i], temp[i + 1]
                    interpolated_value = ((s_ip1 - s_bar) / self.h) * U_i + (
                    return interpolated_value[-1]
            raise ValueError("s_bar is out of bounds.")

        def compute_error_table(self, Sval):
```

```
        alphas = []
        errex = []
        h_vals = []
        Uvals = []
        alphas = [0.0000, 0.00]
        tab = []
        for I_val in [10, 20, 40, 80, 160, 320]:
            N_val = (I_val ) // 10
            dt_val = self.T / N_val
            h_val = (self.Smax - self.Smin) / (I_val + 1)
            s_val = self.Smin + h_val * np.arange(1, I_val + 1)

            # Update parameters
            self.dt = dt_val
            self.h = h_val
            self.s = s_val
            self.I = I_val
            self.N = N_val
            self.alpha = self.sigma**2 * self.s**2 / (2 * self.h**2)
            self.beta = self.r * self.s / (2 * self.h)
            self.U = self.phi(self.s)
            exact = self.black_scholes(Sval, 0)


            start_time = time.time()
            U = self.interpolate_value(Sval)
            end_time = time.time()
            h_vals.append(h_val)
            Uvals.append(U)
            errex.append(abs(U-exact))
            tab.append([I_val, N_val, U, abs(U-exact), end_time - start_t

        errors = [abs(Uvals[j+1] - Uvals[j]) for j in range(len(Uvals)-1)
        errors.insert(0, 0)
        tab[0][3] = 0
        orders = []
        for k in range(2, len(Uvals)):
            alpha_k = (np.log(abs(Uvals[k-2]- Uvals[k-1]) / abs(Uvals[k]
            alphas.append(alpha_k)
            beta_k = alpha_k / 2
            orders.append([k, alpha_k, beta_k])

        tab = np.insert(tab, 3, errors, axis=1)
        tab = np.insert(tab, 4, alphas, axis=1)
        tab = np.array(tab)
        df = pd.DataFrame(tab, columns=['I', 'N', 'U(s)', 'error', 'alpha
        return df

    def run(self):
        if self.scheme == 'EE':
            self.explicit_euler()
        elif self.scheme == 'IE':
            self.implicit_euler()
        elif self.scheme == 'CN':
            self.crank_nicolson()

        # Plot results
        plt.plot(self.s, self.U, label = 'Scheme')
        plt.plot(self.s, [max(p - self.K, 0) for p in self.s], 'r--', lab
        plt.xlabel('Stock Price')
```

```
        plt.ylabel('Option Value')
        plt.title(f'Call Option - {self.scheme} (T = {self.T})')
        plt.legend()
        plt.grid()
        plt.show()
```

In [23]:

```
I = N = 640
call = CallOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I, N=
call.run()
```



In [26]:

```
option = CallOption(r=0.1, sigma=0.2, K=100, T=1, Smin=0, Smax=200, I=I,
option.compute_error_table(80)
```

Out[26]:

	I	N	U(s)	error	alpha	errex	tcpu
0	10.0	1.0	4.209675	0.000000	0.000000	0.000000	0.000215
1	20.0	2.0	3.433124	0.776551	0.000000	0.643203	0.002359
2	40.0	4.0	3.072803	0.360322	1.147695	0.282881	0.002280
3	80.0	8.0	2.916853	0.155950	1.229979	0.126932	0.003799
4	160.0	16.0	2.849080	0.067773	1.213137	0.059159	0.014741
5	320.0	32.0	2.818345	0.030735	1.145964	0.028424	0.091108