Finite difference method for a European option
M2 Modélisation aléatoire - Université Denis-Diderot
Cours EDP en Finance et Méthodes Numériques.

In this programming session, hints are given in `Python`, but you can use other programming languages. You should send back your program before Dec. 20 on moodle with a report in pdf or a pdf copy of the results. [1] The work to do is given in Section 6.

1 The Euler Forward (or Explicit Euler) scheme

We look for a numerical approximation of the European put function $v = v(t,s)$, $t \in [0,T]$, $s \in [0, S_{max}]$. It satisfies in first approximation the Black and Scholes backward PDE on the truncated domain $\Omega = [S_{min}, S_{max}]$:

$$\begin{cases} \frac{\partial}{\partial t}v - \frac{\sigma^2}{2}s^2 \frac{\partial^2}{\partial s^2}v - rs\,\frac{\partial}{\partial s}v + rv = 0, \quad t \in (0,T),\ s \in (S_{min}, S_{max}) \\ v(t, S_{min}) = v_\ell(t) \equiv Ke^{-rt} - S_{min}, \quad t \in (0,T) \\ v(t, S_{max}) = v_r(t) \equiv 0, \quad t \in (0,T) \\ v(0,s) = \varphi(s) := (K-s)_+, \quad s \in (S_{min}, S_{max}). \end{cases} \tag{1}$$

We will consider the following parameters:

$$K = 100, \quad S_{min} = 0,\ S_{max} = 200, \quad T = 1, \quad \sigma = 0.2, \quad r = 0.1$$

In particular, we aim at computing $v(t,s)$ at final time $t = T$.

We first introduce a discrete mesh as follows. Let $h := \frac{S_{max} - S_{min}}{I+1}$ and $\Delta t := \frac{T}{N}$ be the time step and spatial mesh step, and

$$\begin{aligned} s_j &:= S_{min} + jh, \quad j = 0, \dots, I+1 \text{ (mesh points)} \\ t_n &= n\Delta t, \quad n = 0, \dots, N \text{ (time mesh)} \end{aligned}$$

We are looking for U_j^n, an approximation of $v(t_n, s_j)$.

For any function $v \in C^2$ (or $v \in C^3$ for (4)), we recall the following approximations, as $h \to 0$,

$$v'(s_j) = \frac{v(s_j) - v(s_{j-1})}{h} + O(h) \tag{2}$$

$$v'(s_j) = \frac{v(s_{j+1}) - v(s_j)}{h} + O(h) \tag{3}$$

$$v'(s_j) = \frac{v(s_{j+1}) - v(s_{j-1})}{2h} + O(h^2). \tag{4}$$

[1] send back the program in the form `code.py` (or `code.ipynb`) and a short report in pdf format.

We therefore obtain several possible approximations by finite differences for the first order derivative:

$$\frac{\partial}{\partial s}v(t_n, s_j) \simeq \frac{U_j^n - U_{j-1}^n}{h} \quad \text{(backward difference approximation)} \tag{5}$$
$$\frac{\partial}{\partial s}v(t_n, s_j) \simeq \frac{U_{j+1}^n - U_j^n}{h} \quad \text{(forward difference approximation)} \tag{6}$$
$$\frac{\partial}{\partial s}v(t_n, s_j) \simeq \frac{U_{j+1}^n - U_{j-1}^n}{2h} \quad \text{(centered approximation)} \tag{7}$$

The first two approximations are said to be consistent of order 1 (in space), while the second one is consistent of order 2.

We also recall the approximation

$$-\frac{\partial^2}{\partial s^2}v(t_n, s_j) \quad \simeq \quad \frac{-U_{j-1}^n + 2U_j^n - U_{j+1}^n}{h^2}, \tag{8}$$

which is second-order consistent in space.

Hence we obtain the so-called "Euler Forward scheme" (or Explicit Euler scheme), hereafter denoted "EE", using the centered approximation, as follows:

$$\begin{aligned}
&\frac{U_j^{n+1} - U_j^n}{\Delta t} + \frac{\sigma^2}{2}s_j^2\,\frac{-U_{j-1}^n + 2U_j^n - U_{j+1}^n}{h^2} - rs_j\frac{U_{j+1}^n - U_{j-1}^n}{2h} + rU_j^n = 0 \qquad \begin{array}{l} n = 0, \ldots, N-1, \\ j = 1, \ldots, I \end{array}\\
&U_0^n = v_\ell(t_n) \equiv Ke^{-rt_n} - S_{min}, \quad n = 0, \ldots, N\\
&U_{I+1}^n = v_r(t_n) \equiv 0, \quad n = 0, \ldots, N\\
&U_j^0 = \varphi(s_j) \equiv (K - s_j)_+, \quad j = 1, \ldots, I
\end{aligned} \tag{9}$$

Let us remark that we have taken $j = 1$ and $j = I$ as extremal index in j. For $j = 1$, the scheme uses the value $U_0^n := v_\ell(t_n)$ (left boundary value). For $j = I$, the scheme uses the value $U_{I+1}^n := v_r(t_n)$ (right boundary value).

2 Programming Explicit Euler (EE)

2.1 Preliminaries.

We choose to work with the unkown vector corresponding to $(v(t_n, s_j))_{j=1,\ldots,I}$:

$$U^n = \begin{pmatrix} U_1^n \\ \vdots \\ U_I^n \end{pmatrix}.$$

We would like to write (9) under the vector form as follows:

$$\frac{U^{n+1} - U^n}{\Delta t} + AU^n + q(t_n) = 0, \quad n = 0, \ldots, N-1 \tag{10}$$
$$U^0 = (\varphi(s_i))_{1 \le i \le I} \tag{11}$$

where A is a square matrix of dimension I and $q(t)$ is a column vector of size I. Then the EE scheme can be programmed with initialization $U^0 = (\varphi(s_i))$ and the following recursion

$$U^{n+1} = U^n - \Delta t(AU^n + q(t_n)), \quad n = 0, \dots, N-1.$$

Let us denote

$$\alpha_j := \frac{\sigma^2}{2}\frac{s_j^2}{h^2}, \quad \beta_j := r\frac{s_j}{2h}.$$

In view of (9), we look for A and $q(t)$ such that

$$\begin{aligned}
&\alpha_i(-U_{i-1}^n + 2U_i^n - U_{i+1}^n) - \beta_i(U_{i+1}^n - U_{i-1}^n) + rU_i^n \\
&= \; (-\alpha_i + \beta_i)U_{i-1}^n + (2\alpha_i + r)U_i^n + (-\alpha_i - \beta_i)U_{i+1}^n \\
&\equiv \; (AU + q(t_n))_i.
\end{aligned}$$

By identification we see that A is a tridiagonal matrix

$$A \;:=\; \begin{bmatrix}
2\alpha_1 + r & -\alpha_1 - \beta_1 & & & & 0 \\
-\alpha_2 + \beta_2 & 2\alpha_2 + r & -\alpha_2 - \beta_2 & & & \\
\ddots & \ddots & \ddots & & & \\
& -\alpha_i + \beta_i & 2\alpha_i + r & -\alpha_i - \beta_i & & \\
& \ddots & \ddots & \ddots & & \\
0 & & & -\alpha_I + \beta_I & 2\alpha_I + r &
\end{bmatrix}$$

and $q(t_n)$ contains the boundary values $U_0^n = v_\ell(t_n)$ and $U_{I+1}^n = v_r(t_n)$ as follows:

$$q(t_n) \;:=\; \begin{pmatrix} (-\alpha_1 + \beta_1)U_0^n \\ 0 \\ \vdots \\ 0 \\ (-\alpha_I - \beta_I)U_{I+1}^n \end{pmatrix} \equiv \begin{pmatrix} (-\alpha_1 + \beta_1)v_\ell(t_n) \\ 0 \\ \vdots \\ 0 \\ (-\alpha_I - \beta_I)v_r(t_n) \end{pmatrix}.$$

2.2 Programming indications.

Financial parameters should be denoted `r,sigma,K,T` and should be defined at the begining of the program.

The following functions should be also programmed at the begining of the code:

- payoff function `u0` (for φ) as a function of s.
- functions `uleft` (for v_ℓ) and `uright` (for v_r), as functions of the time
- parameters `Smin`, `Smax` for the domain boundary
- parameter `Sval` (a specific value `Sval`$= \bar{s}$ where we want to evaluate $v(T, \bar{s})$).

Then the numerical parameters should be defined, like this:

```
# NUMERICAL PARAMETERS (EXAMPLE)
N=10
I= 9
SCHEME='EE'
Smin=0; Smax=200;
```

We advice to print some of the data like this:

```
print('N=%3i' % N,  'I=%3i' % I,  'SCHEME=%s' % SCHEME)
...
```

The initialization and recursion steps should take the form

```
# init
U=phi(s)

# main loop
for n in range(0:N):
  t=n*dt
  U = U - dt * (A@U + q(t))
  # ... plots, prints, etc.
```

d) Program the matrix `A` of size $I \times I$; Program the function `q(t)`. We STRONGLY advice the following types, using module `numpy`:

```
import numpy as np
A=np.zeros((I,I))
# fill A correctly
# ...


def q(t):
  y=np.zeros((I,1))
  # fill y correctly
  # ...
  return y
```

Also one can use vectors of $\mathbb{R}^I$, or column vectors of $\mathbb{R}^I$ (more precisely, matrices of size $I \times 1$) as elements of type `np.array`. The mesh can be programmed as follows:

```
s=Smin + h*np.arange(1:I+1);  # => s[0],...,s[I-1] encodes s_1,...,s_I
```

The payoff function `phi` can be programmed also as follows:

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

Then for `I=9` the values of `s` and `phi(s)` should look like:

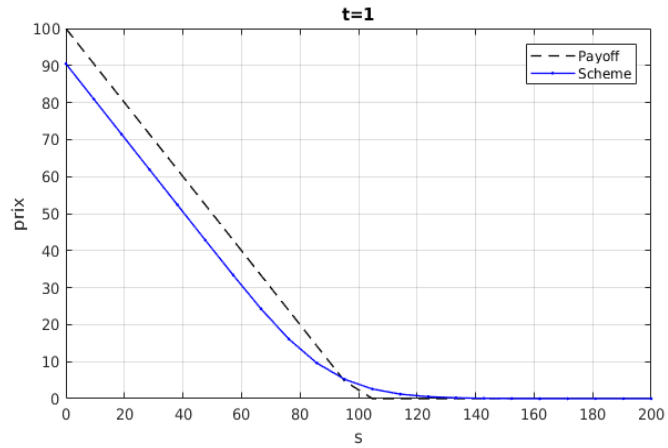

Figure 1: European put option. $T = 1$, $N = I = 20$.

```
array([ 20.,  40.,  60.,  80., 100., 120., 140., 160., 180.])

array([[80.],
       [60.],
       [40.],
       [20.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.]])
```

Because of the reshape operation `.reshape(I,1)`, the matrix $\times$ vector operation is possible, such as `U=phi(s); V=A@U+q(t)` Once the scheme is well programmed, you should be able to obtain the typical plot of Figure 1.

3 First numerical tests

a) Test the Euler forward scheme (EE). First fix $N = 10$ and take $I = 10, 20, 50, \ldots$. Then take the following $N = I$ values : $10, 20, 50, 100$. Observe that:

- the scheme is not always numerically stable (the norm of U^n may explode after a finite number of iterations)

- it does not always give a positive solution (i.e. we do not always have $U_j^n \geq 0\ \forall n, j$)

We would prefer to avoid these effects.

b) In order to understand the origin of the oscillations or explosion when they occur, for instance, fix $N = 10$ and $I = 50$, and look at the "amplification" matrix defined as

$$B := I_d - \Delta t A.$$

Check that the coefficients of B are not all positive [2] and that they may have a modulus greater than 1 (look in particular at the diagonal elements of B). Compute also the norms $\|B\|_\infty$ and $\|B\|_2$ (see at the end of the document for python commands) and check that they are larger than one.

On the contrary, check that for $N = I = 10$, coefficients of B are (almost) all positive, and smaller that 1.

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

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

and print it. Check that there is no stability problem when μ is sufficiently small.

d) Compute the P_1-interpolated value at $\bar{s} = 90$ (because $\bar{s}$ may not be on the mesh!). If $\bar{s} \in [s_i, s_{i+1}]$, then this interpolated value can be obtained by using the affine (P1) approximation

$$U(\bar{s}) \simeq \frac{s_{i+1} - \bar{s}}{h} U_i + \frac{\bar{s} - s_i}{h} U_{i+1}.$$

e) Numerical order of the scheme.

First consider the following values of I and of N: $N = I = 10, 20, 40, 80, 160, 320$, and print the corresponding table (as in Fig. 2). You should observe a stability issue.

Then use $I = 10, 20, 40, 80, \ldots$, and $N = I^2/10$ (that is, $N = 10, 40, \ldots$). This is in order to have $\Delta t \simeq h^2$. Fill in the corresponding error table

In order to estimate the error, several methods are possible. Here will simply estimate the difference between to successive computations, say $U^{(k-1)}$ and $U^{(k)}$ (obtained with mesh (I_{k-1}, N_{k-1}) and (I_k, N_k), resp.), so that

$$e_k := \left| U^{(k-1)}(\bar{s}) - U^{(k)}(\bar{s}) \right|$$

(and the first error term e_1 is not defined). It is also possible to compare the value $U^{(k)}$ with the value given by the (exact) Black and Scholes formula (see complement 1), and then to take $e_{ex,k} := \left| U^{(k)}(\bar{s}) - v(T, \bar{s}) \right|$. [3]

Typical values for N of the order of I^2 (here with $\bar{s} = 80$):

	I	N	U(s)	error	alpha	errex	tcpu
0	10	10	14.448406	0.000000	0.000000	0.000000	0.001655
1	20	40	13.611061	0.837345	0.274530	0.337398	0.005495
2	40	160	13.361562	0.249499	1.809707	0.087899	0.062304
3	80	640	13.296062	0.065500	1.964225	0.022399	0.320479
4	160	2560	13.279318	0.016744	1.985612	0.005655	2.444337

Figure 2: Convergence table, at `Sval=80`, using module `penda`

[2] For a scheme that would be simply written $U^{n+1} = BU^n$ (ie with zero boundary conditions), positivity of the matrix B matrix ($B \geq 0$ compentwise) ensures that if $U^0 \geq 0$ (componentwise) then $U^n \geq 0$ for all n.

[3] Other measurement for the errors are possible, for instance in L^∞ norm $e_k = \|U^{(k-1)} - U^{(k)}\|_\infty$ and $e_{ex,k} = \|U^{(k)} - (v(T, s_i))_i\|_\infty$ are possible, etc.

More precisely, if the error is e_k for a given parameters $I = I_k$ and $N = N_k$ we can estimate the order (at step k) by the formula

$$\alpha_k := \frac{\log(e_{k-1}/e_k)}{\log(h_{k-1}/h_k)} \simeq \alpha$$

where h_k is the spatial mesh step at step k (corresponding to I_k). The idea is to try to detect a behavior of the form $e_k = C\, h_k^\alpha$, where C est a constant

If $h_k = h_{k-1}/2$ as in the left table, then we get $e_k/e_{k-1} \simeq 1/2^\alpha$ and the previous formula gives $\alpha_k = \frac{\log(e_{k-1}/e_k)}{\log(2)}$ for an estimate of α. (the order in space should be 2)

Notice that we can also look at the order in time, trying to look at a behavior in the form of $e_k \simeq C\Delta t_k^\beta$, we are lead to try the estimate

$$\beta_k := \frac{\log(e_{k-1}/e_k)}{\log(\Delta t_{k-1}/\Delta t_k)} \simeq \beta.$$

Here using $N_k = C\ I_k^2$ leads to $\Delta t_k = Const.h_k^2$ and an order $\beta_k = \frac{\alpha_k}{2}$ (therefore the order in time should be 1).

Since $N \simeq I^2$ (coming from the CFL condition) is costly in terms of number of time iterations, it motivates the use of implicit schemes in order to avoid mesh step restrictions.

4 Implicit Euler (IE) scheme

The Implicit Euler scheme ("IE" scheme), with centered difference approximation for the first spatial derivative, is:

$$\begin{aligned}
&\frac{U_j^{n+1} - U_j^n}{\Delta t} + \frac{\sigma^2}{2} s_j^2\, \frac{-U_{j-1}^{n+1} + 2U_j^{n+1} - U_{j+1}^{n+1}}{h^2} - r s_j \frac{U_{j+1}^{n+1} - U_{j-1}^{n+1}}{2h} + rU_j^{n+1} = 0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad n = 0, \dots, N-1, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad j = 1, \dots, I \\
&U_0^{n+1} = v_\ell(t_{n+1}) \equiv K e^{-rt_{n+1}} - S_{min}, \quad n = 0, \dots, N-1 \\
&U_{I+1}^{n+1} = v_r(t_{n+1}) \equiv 0, \quad n = 0, \dots, N-1 \\
&U_j^0 = (K - s_j)_+, \quad j = 1, \dots, I
\end{aligned} \tag{12}$$

Check that the scheme, in vector form, can be written

$$\frac{U^{n+1} - U^n}{\Delta t} + AU^{n+1} + q(t_{n+1}) = 0, \quad n = 0, \dots, N-1$$

with

$$U^0 = (\varphi(s_i))_{1 \le i \le I}.$$

Program IE. By setting the parameter `SCHEMA='IE'` at the begining of the **same** program, the code should switch to the IE method.

In order to solve a linear system of the form $Ax = b$ one may use the linear solver as follows

```
x=nlp.solve(A,b)
```

(We may have to recast into an $I \times 1$ array, in particular when using a sparse solver. In that case use for instance `x=x.reshape(-1,1)`.)

a) Check that with the IE scheme there is no more stability problems (with for instance $N = 10$ and $I = 50$).

b) Draw the corresponding table with $N = I$ and with $N = I/10$, as before (in particular, there is no need anymore to use N or the order of I^2 !).

5 Crank-Nicolson scheme

The Crank Nicholson scheme ("CN" scheme), with centered difference approximation for the first spatial derivative, is:

$$\begin{aligned} &\frac{U_j^{n+1} - U_j^n}{\Delta t} + \frac{1}{2}\Big(-\frac{\sigma^2}{2} s_j^2 \frac{U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}}{h^2} - r s_j \frac{U_{j+1}^{n+1} - U_{j-1}^{n+1}}{2h} + rU_j^{n+1} \Big) \\ &\qquad + \frac{1}{2}\Big(-\frac{\sigma^2}{2} s_j^2 \frac{U_{j-1}^{n} - 2U_j^{n} + U_{j+1}^{n}}{h^2} - r s_j \frac{U_{j+1}^{n} - U_{j-1}^{n}}{2h} + rU_j^{n} \Big) = 0 \qquad (13) \\ &\qquad\qquad n = 0, \dots, N-1, \\ &\qquad\qquad j = 1, \dots, I \end{aligned}$$

with adequate boundary conditions and initial conditions, such as:

$$\begin{aligned} &U_0^{n+1} = v_\ell(t_{n+1}) \equiv K e^{-r t_{n+1}} - S_{min}, \quad n = 0, \dots, N-1 \qquad (14) \\ &U_{I+1}^{n+1} = v_r(t_{n+1}) \equiv 0, \quad n = 0, \dots, N-1 \\ &U_j^0 = (K - s_j)_+, \quad j = 1, \dots, I \end{aligned}$$

a) Program the Crank-Nicolson scheme (CN) (`SCHEME='CN'`). First, write the CN scheme in vector form. Then program the Crank-Nicolson scheme (θ-scheme with $\theta = \frac{1}{2}$), `SCHEME='CN'`

b) Draw the corresponding table with $N = I$ and $N = I/10$. (The numerical order should be clear for $N = I$.)

6 Work to do

1. Program EE, EI, CN. Draw some figures such as in Figure 1.
2. Fill the tables for EE, EI, CN (two tables for EE: $N = I$ and $N = I^2/10$), two tables for IE (with $N = I$ and $N = I/10$), two tables for CN (with $N = I$ and $N = I/10$).
3. Write a short conclusion on your numerical tests. Discuss on the orders numerically observed and if they are coherent with the theoretical analysis.
4. Complement 1: program the Black and Scholes formula (see last section), comment if you observe a different order analysis when using the exact formula for computing the errors.

5. Complement 2: program the sparse matrices (see last section). Comment if you observe cputime gain with sparse matrices.

6. Complement 3: program the call option (see last section).

7 Complements

Complement. 1 ("Black and Scholes formula") *Program the Black and Scholes formula for the put option. A formula corresponding to $v(t, S)$ is*

$$v(t, S) := Ke^{-rt}N(-d_-) - SN(-d_+)$$

with

$$d_\pm := \frac{log(S/K) + (r \pm \frac{1}{2}\sigma^2)t}{\sqrt{\sigma^2 t}} \quad \text{and} \quad N(y) := \int_{-\infty}^{y} e^{-u^2/2} \frac{du}{\sqrt{2\pi}} \tag{15}$$

Draw the previous tables (in particular for EI and CN) with now the exact error ***errex***$:= U(\bar{s}) - v(t, \bar{s})$.

Complement. 2 Improved efficiency using sparse matrices. *The matrix A has only a few non zero elements (about $3I$ non zero elements). It is possible to code only the non-zero elements of A by using a sparse type matrix. Typical modules:*

```
from scipy.sparse import csr_matrix as sparse
from scipy.sparse.linalg import spsolve
```

Warning: after the use of a command such as `x=spsolve(B, b)` *for solving* `B*x=b`*, it is possible than you need to reshape the result (*`x=x.reshape(I,1)`*).*

$\Rightarrow$ *Modify slighlty your code in order to program EE, IE and CN with sparse matrices. Compare the speed of the new code with respect to the full matrix approach. (Execution time is in general improved for large N, I).*

Complement. 3 ("Call option") *a) Propose adequate left and right boundary conditions (that is, at $s = S_{min}$ and $s = S_{max}$) for the call option, in the form ($v(t, S_{min}) = v_\ell(t)$ and $v(t, S_{max}) = v_r(t)$, where v_ℓ, v_r are function to determine.*[4]

b) Propose a PDE for the call option, using these boundary conditions.

c) Program and test the Implicit Euler scheme for the call option.

8 Some useful modules and commands

```
import numpy as np               # array
import numpy.linalg as lng       # linear algebra
import matplotlib.pyplot as plt  # plot functions
import time
import sys                       # command sys.exit()
```

[4] $v_\ell(t, s) \equiv 0$, $v_r(t, s) \equiv s - Ke^{-rt}$.

To print with `panda` :

```
import pandas as pd
tab=np.array(tab)
df=pd.DataFrame(tab,columns=['I', 'N', 'U(s)','error','alpha','errex','tcpu'])
df  # with jupiter notebook, otherwise "print(df)" will work also
```

To obtain the cputime for a sequence of instructions, use:

```
t0=time.time()
# instructions ...
t1=time.time(); print('tcpu=%5.2f' % (t1-t0))
```

Miscellaneous : for a matrix A of type `np.array`:

`print(A)`	Nice print of an nd.array `A`
`print(np.round(A,decimals=3))`	
`lng.norm(A,np.inf)`	$\|A\|_\infty$ $(= \max_{x\neq 0} \|Ax\|_\infty/\|x\|_\infty)$
`lng.norm(A,2)`	$\|A\|_2$ (largest singular value)

For computing the cumulative normal distribution function of a scalar or np.array `x`:

```
import scipy.stats as stats
stats.norm.cdf(x, 0.0, 1.0)
```