Utilisation de numpy et application à une simulation

I Tableaux/matrices numpy

```
1 import numpy as np
```

L'extension ***numpy*** *ne fait pas partie des connaissances exigibles du programme d'informatique. Néanmoins elle est souvent utilisée dans les sujets de concours et l'est systématiquement à l'épreuve orale « mathématiques 2 » du concours Centrale — avec fourniture d'une liste de rappels. La syntaxe n'est pas à retenir exactement, mais il est fortement recommandé d'y être familiarisée pour pouvoir l'utiliser immédiatement.*

L'utilisation des tableaux `numpy` présente certains avantages par rapport à l'utilisation des listes (ou des listes de listes) : ils sont plus faciles à définir ; on peut les manipuler "globalement" (calculs vectoriels). Par contre ils ont une taille fixée (non évolutive ; pas de méthode `append`), ce qui facilite leur utilisation et ce qui permet d'avoir d'excellentes performances.

I.1 Création de tableaux numpy

- `A = np.array([[1, 2, 3], [4, 5, 6]])`
  Cette instruction affecte à la variable `A` un tableau 2 lignes 3 colonnes dont la première ligne est le tableau à une dimension `[1, 2, 3]` et la deuxième ligne est le tableau à une dimension `[4, 5, 6]`.
- `np.zeros([n, p]) ou np.zeros((n, p)) # renvoie le tableau à n lignes et p colonnes rempli de zéros`
  Par exemple : `np.zeros([3, 2])` → `array([[0, 0], [0, 0], [0, 0]])`
- `np.ones([n, p]) ou np.ones((n, p)) # renvoie le tableau à n lignes et p colonnes rempli de uns`
  Par exemple : `np.ones([2, 2])` → `array([[1, 1], [1, 1]])`
- `np.eye(2) # renvoie la matrice identité d'ordre n`
  Par exemple : `np.eye(2)` → `array([[1, 0], [0, 1]])`
- `np.diag(L) # renvoie la matrice diagonale dont les coefficients diagonaux sont ceux de la liste L (ou du tuple L)`
  Par exemple : `np.diag([1, 2, 3])` → `array([[1, 0, 0], [0, 2, 0], [0 , 0, 3]])`
- Par défaut les coefficients d'un tableau `numpy` sont de type `float` (approximation de nombres réels, codés sur 64 bits = 8 octets). Pour économiser de l'espace en mémoire, le paramètre optionnel `dtype = ...` permet de spécifier un type de coefficients différent, par exemple :
  - `np.bool` : booléen (codé sur 1 bit) ;
  - `np.uint8` : entier naturel ("unsigned") de 0 à 255 (codé sur un 8 bits = 1 octet) ; $255 = 2^8 - 1$) ;
  - `np.int16` : entier relatif de $-32\,768$ à $32\,767$ (codé sur un 16 bits = 2 octet ; $32\,768 = 2^{16} - 1$) ;
  - `np.uint16`, `np.uint32`, `np.uint64`, `np.int8`, `np.int32`, `np.int64`, `np.float16`, `np.float32`...

  Par exemple : `A = np.array([[1, 2, 3], [4, 5, 6]], dtype = np.uint8)`

I.2 Modification d'un tableau déjà créé

- `A[i, j] = x # remplace l'élément i,j de A par x`
  Par exemple :
  `A[0, 1] = 3` ; `A` → `array([[1, 3, 3], [4, 5, 6]])`
- `A[i] = L # remplace la ligne i par L (liste numpy ou tableau 1xn numpy ou liste)`
  Par exemple :
  `L = np.array([1, 1, 1])` ; `A[0] = L` ; `A` → `array([[1, 1, 1], [4, 5, 6]])`
  `L = np.array([[1, 1, 1]])` ; `A[0] = L` ; `A` → `array([[1, 1, 1], [4, 5, 6]])`
  `A[0] = [1, 1, 1]` ; `A` → `array([[1, 1, 1], [4, 5, 6]])`
- `A[:, j] = L # remplace la colonne j par L (liste numpy ou tableau 1xn numpy ou liste)`
  Par exemple :
  `L = np.array([7, 7])` ; `A[:, 0] = L` ; `A` → `array([[7, 2, 3], [7, 5, 6]]`

Il est possible de changer la forme d'un tableau, mais le nombre total d'éléments reste inchangé :
Par exemple :

- `A = np.array([[1, 2, 3], [4, 5, 6]])` ; `A.reshape((3, 2))` ;
  `A` → `array([[1, 2], [3, 4], [5, 6]])`

I.3 Récupération d'éléments ou de parties d'un tableau

Les opérations de *slicing* sur les listes et les listes de listes s'étendent au tableaux `numpy`, avec davantage de possibilités.

Pour les exemples ci-dessus, on aura déjà créé : `A = np.array([[1,2,3],[4,5,6]])`

- `A[i][j] # renvoie l'élément de la ligne i, colonne j du tableau A`
  `A[i, j] # idem (syntaxe alternative)`
  Par exemple : `A[0][2]` → `3`
- `A[i] # renvoie le tableau numpy à 1 dimension qui est la ligne i de A`
  `A[i][:] # idem (syntaxe alternative)` (*car* ***`A[i][:]`*** *est* ***`(A[i])[:]`*** *càd.* ***`A[i]`***)
  `A[i, :] # idem (syntaxe alternative)`
  Par exemple : `A[0, :]` → `array([1, 2, 3])`
- `A[:, j] # renvoie le tableau numpy à 1 dimension qui est la colonne j de A`
  Par exemple : `A[:, 1]` → `array([2, 5])`
  ☞ `A[:][1]` → `array([4, 5, 6])` (*car* ***`A[:][1]`*** *est* ***`(A[:])[1]`*** *càd.* ***`A[1]`***)
- `A[i:ii, j:jj] # slicing qui renvoie le tableau numpy à deux dimensions formé des coefficients de lignes i à ii-1 (inclus) et des colonnes j à jj-1 (inclus)`
  Par exemple : `A[1:2, 1:3]` → `array([[5, 6]])`
- `x, y = A.shape # affecte à x et y les nombre de lignes et de colonnes de A`
  `(x, y) = A.shape # idem (syntaxe alternative avec un tuple)`
  `x, y = np.shape(A) # idem (syntaxe alternative)`
  Par exemple : `x, y = A.shape` ; `x` → `2` ; `y` → `3`

I.4 Opérations sur les tableaux numpy

Pour les exemples ci-dessous, on aura déjà créé :

`A = np.array([[1, 2], [3, 4]])` et `B = np.array([[5, 6], [7, 4]])`

- `np.concatenate((A, B)) # superposition du tableau A au dessus du tableau B`
  Par exemple : `np.concatenate(A, B))` → `array([[1, 2], [3, 4], [5, 6], [7, 4]])`
- `np.concatenate((A, B), axis = 1) # juxtaposition du tableau A à gauche du tableau B`
  Par exemple : `np.concatenate(A, B), axis = 1)` → `array([[1, 2, 5, 6], [3, 4, 7, 4]])`
- `A + B # somme vectorielle de deux tableaux de mêmes dimensions`
  Par exemple : `A + B` → `array([[6, 8], [10, 8]])`
- `x * A # multiplication par le scalaire x du vecteur/matrice A`
  Par exemple : `5 * A` → `array([[5, 10], [15, 20]])`
- `np.dot(A, B) # renvoie le produit de deux matrices de tailles compatibles`
  `A.dot(B) # idem (syntaxe alternative)`
  Le nombre de lignes de `A` doit être égal au nombre de colonnes de `B`. Si `B` un tableau à une dimension, le résultat est un tableau à une dimension.
  Par exemple : `np.dot(A, np.array([[0, -1], [-2, -3]])` → `array([[ -4, -7], [ -8, -15]])`
  `A.dot(np.array([1, -1]))` → `array([-1, -1])`
  `np.dot(np.array([1, 2, 3]), np.array([1, 2, 4]))` → `17 # produit scalaire`
- `np.transpose(A) #renvoie la matrice transposée de A`
  `A.T # idem (syntaxe alternative)`
  Par exemple : `np.transpose(A)` → `array([[1, 3], [2, 4]])`
  ☞ Le test de comparaison `A == B` est effectué coefficient par coefficient et renvoie un array avec le résultat de chacun des tests ;
  pour tester l'égalité globale, utiliser plutôt `np.array_equal(A,B)`.
  Par exemple : `A==B` → `array([[False,False],[False,True]], dtype=bool)`
  `np.array_equal(A,B)` → `False`

II Échauffement sur les tableaux `numpy`

Q1 Écrire une **procédure** `combine_ligne(T, i, j, k)` (resp. une procédure `combine_colonne(T, i, j, k)`) qui modifie un tableau `T` en effectuant l'opération sur les lignes $L_i \leftarrow L_i + kL_j$ (resp. $C_i \leftarrow C_i + kC_j$).

Q2 Écrire une fonction `somme_pairs(T)` qui renvoie la somme des coefficients dont la valeur est paire du tableau `numpy` à deux dimensions `T`.

Q3 Écrire une fonction `tri_diag(l1, l2, l3)` où `l1`, `l2` et `l3` sont des listes de respectivement n, $n-1$ et $n-1$ nombres et qui renvoie la matrice `numpy` suivante :

$$\begin{pmatrix} \texttt{l1[0]} & \texttt{l3[0]} & 0 & \dots & 0 \\ \texttt{l2[0]} & \texttt{l1[1]} & \texttt{l3[1]} & \ddots & \vdots \\ 0 & \texttt{l2[1]} & & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \texttt{l3}[n-2] \\ 0 & \dots & 0 & \texttt{l2}[n-2] & \texttt{l1}[n-1] \end{pmatrix}$$

On pourra essayer écrire une version qui n'utilise pas de boucle (exploiter les possibilités de `numpy`)

Q4 Écrire une fonction `NB_hasard(n, p)` qui renvoie un tableau à `n` lignes et `p` colonnes d'entiers du type `np.uint8` choisis au hasard.
On rappelle que la fonction `randint(a, b)` (du module `random`) renvoit un entier tiré au hasard entre les entiers `a` et `b` (inclus).

Q5 Écrire une fonction `matrice(n)` qui renvoie sous forme de tableau `numpy` la matrice $M = (m_{i,j})_{1 \leq i,j \leq n}$ d'ordre n dont les coefficients sont donnés par :

$$m_{i,j} = \begin{cases} 1 & \text{si } i = n \text{ ou } j = n \text{ ou } i = n - j + 1 \\ 0 & \text{sinon} \end{cases}$$

III Un problème de simulation

Évolution d'une matrice dont deux bords sont donnés

Q6 Écrire une fonction `initialise(L, C)` qui prend en argument deux listes de nombres `L`, `C` de longueurs respectives `p`, `n` et qui renvoie un tableau `numpy` à `n+1` lignes et `p` colonnes dont tous les coefficients sont nuls sauf ceux de la première ligne qui sont ceux de `L`, dans l'ordre, et ceux de la première colonne (sauf le premier) qui sont ceux de `C`, dans l'ordre.
Par exemple :
`initialise([1, 2, 3], [4, 5])` → `array([[1., 2., 3.], [4., 0., 0.], [5., 0., 0.]])`

Q7 Écrire une fonction `extractionResultat(T)` qui prend en argument un tableau `numpy` à deux dimensions et qui renvoie la liste `[LL, CC]` où `LL` est la liste, dans l'ordre, des coefficients de la dernière ligne de `T`, et où `CC` est la liste, dans l'ordre, des coefficients de la dernière colonne de `T` sauf le dernier. *Par exemple :*
`extractionResultat(np.array([[1.,2.,3.],[4.,0.,0.],[5.,0.,0.]])`→ `[[5.,0.,0.],[3.,0.]]`

Q8 Écrire une fonction `suivant1(T)` qui prend en argument un tableau numpy `T` et qui renvoie un tableau `TT` dont la première ligne et la première colonne sont celles de `T` et dont tout autre coefficient est la moyenne de celui de `T` situé au-dessus de lui et de celui de `T` situé à gauche de lui.
Par exemple :
`suivant1(np.array([[1.,2.],[4.,0.],[5.,0]]))` → `array([[1.,2.], [4.,3.0], [5.,2.5])`

Q9 Écrire une fonction `suivant1bis(T)` qui agit comme la précédente mais qui n'utilise pas de boucle. *On pensera à utiliser des opérations matricielles.*

Q10 Écrire une fonction `suivant2(T)` qui prend en argument un tableau numpy `T` et qui renvoie un tableau `TT` dont la première ligne et la première colonne sont celles de `T` et tel que tout autre coefficient est la moyenne de celui de `TT` situé au-dessus de lui et de celui de `TT` situé à gauche de lui.
Par exemple :
`suivant2(np.array([[ 1.,2.],[4.,0.],[5.,0]]))` → `array([[1.,2.],[4.,3],[5., 4.])`

Q11 Écrire une **procédure** `procSuivant1(T)` (resp. une procédure `procSuivant2(T)`) qui modifie `T` pour qu'il soit égal au résultat de la fonction `suivant1(T)` (resp. `suivant2(T)`). On donnera un code qui ne nécessite pas de créer une copie d'autre tableau numpy.

Q12 (a) Écrire une fonction `simulation1(L, C, N)` qui initialise un tableau numpy avec la fonction `initialise(L, C)` puis le fait évoluer `N` fois avec la fonction `suivant1`, et renvoie le tableau ainsi obtenu.
Quelle place en mémoire sera nécessaire pour stocker tous les tableaux utilisés, lorsque `L, C` sont de même longueur 10 et lorsque `N` vaut 100 ?

(b) Écrire cette fonction en utilisant la procédure `procSuivant1`.
Quelle place en mémoire sera à présent nécessaire pour stocker tous les tableaux utilisés avec `L, C` et `N` comme à la question précédente ?

(c) Même question qu'au (a) en remplaçant `suivant1` par `suivant2`.

(b) Même question qu'au (b) en remplaçant `procSuivant1` par `procSuivant2`.

Q13 On dit que deux tableaux numpy `T` et `TT` de même taille « diffèrent à `epsilon` près », où `epsilon` est un flottant strictement positif, lorsque l'un des couples de coefficients situés à la même place dans les deux tableaux diffèrent d'au moins de `epsilon`. Écrire une fonction `different(T, TT, epsilon)` qui renvoie `True` si `T` et `TT` diffèrent à `epsilon` près et qui renvoie `False` sinon. *Par exemple :*
`different(np.array([[1.,2.], [4.,0.]]), np.array([[1.,2.], [4.,0.1]]), 0.01)`→ `True`

Q14 Écrire une fonction `simulation1bis(L, C, epsilon)` qui initialise un tableau numpy avec la fonction `initialise(L, C)` puis le fait évoluer avec la procédure `procSuivant2` jusqu'à ce que deux valeurs successives du tableau soient égales à `epsilon` près. Cette fonction renvoie alors le dernier tableau obtenu. (On pourra utiliser la fonction `np.copy(T)` où `T` est un tableau `numpy`)

Q15 Écrire une fonction `suivant3(D)`, où `D` est une liste formée de deux listes `L,C`, qui initialise un tableau numpy avec la fonction `initialise(L, C)`, le fait évoluer avec la fonction `suivant2` puis renvoie la liste `[LL, CC]` obtenues avec la fonction `extractionResultat`.

Q16 Écrire une fonction `simulation3(L, C, x)` qui fait évoluer le couple `[L, C]` avec la fonction `suivant3` jusqu'à ce que le coefficient `L[0]` dépasse `x` et qui renvoie le nombre d'itérations effectuées.

Q17 La fonction précédente se termine-t-elle forcément ? Si ce n'est pas le cas comment peut-on la modifier ?

IV Pour ceux qui ont déjà fini

Exercice 1.

Q18 *Écrire une fonction* `cible(L)` *qui renvoie une cible carrée qui est un tableau* `numpy` *dont les zones concentriques en partant de l'extérieur sont remplies avec les coefficients successifs de la liste* `L`.

Par exemple : `cible([4, 2, 1])` →
```
array([[4, 4, 4, 4, 4],
       [4, 2, 2, 2, 4],
       [4, 2, 1, 2, 4],
       [4, 2, 2, 2, 4],
       [4, 4, 4, 4, 4]])
```

On donnera une version qui contient une seule boucle et une version dans laquelle chaque coefficient n'est écrit qu'un seule fois.

Q19 *En déduire une fonction* `coin(L)` *qui renvoie un tableau* `numpy` *rempli à partir du coin en bas à gauche en zones concentriques par les coefficients de la liste* `L`.

Par exemple : `coin([4, 2, 1])` →
```
array([[1, 1, 1],
       [2, 2, 1],
       [4, 2, 1]])
```

Corrigé

Q1

```
def combine_ligne(T, i, j, k):
    '''on suppose 0 <= i <= n-1 et 0 <= i <= p-1 avec n, p = np.shape(T)
    '''
    T[i] += k * T[j]
    return T

def combine_colonne(T, i, j, k):
    '''on suppose 0 <= i <= n-1 et 0 <= i <= p-1 avec n, p = np.shape(T)
    '''
    T[:, i] += k * T[:, j]
    return T

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
assert (combine_ligne(A, 0, 2, -1) == np.array([[-6, -6, -6],
                                                [ 4,  5,  6],
                                                [ 7,  8, 9]] )).all()
```

Q2

```
def somme_pairs(T):
    n, p = np.shape(T) # donne la taille de T
    somme = 0
    for i in range(n):
        for j in range(p):
            if T[i, j] % 2 == 0:
                somme += T[i, j]
    return somme

A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
assert somme_pairs(A) == 20
```

Q3

```
def tri_diag(l1, l2, l3):
    n = len(l1)
    T1 = np.diag(l1)
    T2 = np.zeros((n, n))
    T2[1:, :-1] = np.diag(l2)
    T3 = np.zeros((n, n))
    T3[:-1, 1:] = np.diag(l3)
    return T1 + T2 + T3

L1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
L2 = [1, 1, 1, 1, 1, 1, 1, 1]
L3 = [1, 1, 1, 1, 1, 1, 1, 1]
assert (tri_diag(L1, L2, L3) == [[1., 1., 0., 0., 0., 0., 0., 0., 0.],
                                 [1., 2., 1., 0., 0., 0., 0., 0., 0.],
                                 [0., 1., 3., 1., 0., 0., 0., 0., 0.],
                                 [0., 0., 1., 4., 1., 0., 0., 0., 0.],
                                 [0., 0., 0., 1., 5., 1., 0., 0., 0.],
                                 [0., 0., 0., 0., 1., 6., 1., 0., 0.],
                                 [0., 0., 0., 0., 0., 1., 7., 1., 0.],
                                 [0., 0., 0., 0., 0., 0., 1., 8., 1.],
                                 [0., 0., 0., 0., 0., 0., 0., 1., 9.]]).
   all()
```

Q4

```
from random import randint

def NB_hasard(n, p):
    T = np.zeros((n, p), dtype = np.uint8)
    for i in range(n):
        for j in range(p):
            T[i, j] = randint(0, 255)
    return T
```

Q5

```
def matrice(n):
    T = np.zeros((n, n))
    U = np.ones((1, n))
    T[-1], T[:, -1] = U, U
    for i in range(1, n-1):
        T[i, n-i-1] = 1
    return T
# attention au décalage entre l'indexation des matrices (1 à n) et les
   indices python (0 à n-1)

# question posée à l'oral de Centrale PC, 2017

assert (matrice(4) == np.array([[0., 0., 0., 1.],
                                [0., 0., 1., 1.],
                                [0., 1., 0., 1.],
                                [1., 1., 1., 1.]])).all()
```

Q6

```
def initialise(L, C):
    n, p = len(C), len(L)
    T = np.zeros((n+1, p))
    T[0] = L
    T[1:, 0] = C # Attention : T[1:, :1] est une erreur car de dimension
    2
    return T

assert (initialise([1, 2, 3], [4, 5]) == np.array([[1., 2., 3.],
                                                   [4., 0., 0.],
                                                   [5., 0., 0.]])).all()
```

Q7

```
def extractionResultat(T):
    LL = T[-1, :]
    CC = T[:-1, -1]
    return [LL, CC]

assert (extractionResultat(np.array([[1., 2., 3.], [4., 0., 0.], [5.,
   0., 0.]]))[0] == np.array([5., 0., 0.])).all()
assert (extractionResultat(np.array([[1., 2., 3.], [4., 0., 0.], [5.,
   0., 0.]]))[1] == np.array([3., 0.])).all()
```

Q8

```
106 def suivant1(T):
107     n, p = np.shape(T)
108     TT = np.zeros((n, p))
109     TT[0] = T[0]
110     TT[:, 0] = T[:, 0] # ou TT[1:, 0] = T[1:, 0]
111     for i in range(1, n):
112         for j in range(1, p):
113             TT[i, j] = (T[i-1, j] + T[i, j-1]) / 2
114     return TT
115 
116 assert (suivant1(np.array([[1., 2.], [4., 0.], [5., 0]])) == np.array
    ([[1. , 2. ], [4. , 3. ], [5. , 2.5]])).all()
117 
118 assert (suivant1(np.array([[1., 2., 3.], [4., 1., 7.], [5., 6., 8.]]))
    == np.array([[1. , 2., 3. ], [4. , 3., 2. ], [5. , 3., 6.5]])).all()
```

Q9

```
121 def suivant1bis(T):
122     n, p = np.shape(T)
123     T1, T2, T3 = np.zeros((n, p)), np.zeros((n, p)), np.zeros((n, p))
124     T1[0] = T[0]
125     T1[:, 0] = T[:, 0] # ou T1[1:, 0] = T[1:, 0]
126     T2[1:, 1:] = T[1:, :-1]
127     T3[1:, 1:] = T[:-1, 1:]
128     return T1 + (T2 + T3) / 2
129 
130 assert (suivant1bis(np.array([[1., 2.], [4., 0.], [5., 0]])) == np.array
    ([[1. , 2. ], [4. , 3. ], [5. , 2.5]])).all()
```

Q10

```
133 def suivant2(T):
134     n, p = np.shape(T)
135     TT = np.zeros((n, p))
136     TT[0, :] = T[0, :]
137     TT[1:, 0] = T[1:, 0]
138     for i in range(1, n):
139         for j in range(1, p):
140             TT[i, j] = (TT[i-1, j] + TT[i, j-1]) / 2
141     return TT
142 
143 assert (suivant2(np.array([[ 1., 2.],[ 4.,0.],[5.,0]])) == np.array
    ([[1., 2.], [4., 3.], [5., 4.]])).all()
```

Q11

```
146 def procSuivant1(T):
147     n, p = np.shape(T)
148     # on parcourt T à partir de la fin
149     for i in range(n-1, 0, -1):
150         for j in range(p-1, 0, -1):
151             T[i, j] = (T[i-1, j] + T[i, j-1]) / 2
152     # pas de return (procédure)
153 
154 T = np.array([[1., 2.], [4., 0.], [5., 0]]); procSuivant1(T)
155 (T == np.array([[1. , 2. ], [4. , 3. ], [5. , 2.5]])).all()
```

```
158 def procSuivant2(T):
159     n, p = np.shape(T)
160     # on parcourt T à partir du début
161     for i in range(1, n):
162         for j in range(1, p):
163             T[i, j] = (T[i-1, j] + T[i, j-1])/2
164     #pas de return (procédure)
165 
166 T = np.array([[1., 2.], [4., 0.], [5., 0]]); procSuivant2(T)
167 (T == np.array([[1., 2.], [4., 3.], [5., 4.]])).all()
```

Q12

```
170 ##(a)
171 def simulation1(L, C, N):
172     T = initialise(L, C)
173     for i in range(N):
174         T = suivant1(T)
175     return T
176 
177 L = [1, 2, 3]; C = [4, 5]; N = 4
178 print(initialise(L, C))
179 assert (simulation1(L, C, N) == [[1., 2., 3.], [4., 3., 3.], [5., 4.,
    3.5]]).all()
180 
181 # on initialise un tableau de flottants de taille n x (p+1) à chaque ité
    ration, d'où N x n x (p+1) flottants (8 octets chacun) stockés. Avec
    N = 100, n = 10 et p = 10, on obtient 100 x 10 x 11 x 8 soit 88 000
    octets.
182 
183 
184 ##(b)
185 def simulation1Proc(L, C, N):
186     T = initialise(L, C)
187     for i in range(N):
188         procSuivant1(T)
189     return T # sans cela T n'est pas accessible
190 
191 L = [1, 2, 3]; C = [4, 5]; N = 4
192 assert (simulation1Proc(L, C, N) == [[1., 2., 3.], [4., 3., 3.], [5.,
    4., 3.5]]).all()
193 
194 # on initialise un tableau de flottants (8 octets chacun) de taille n x
    (p+1), d'où n x (p+1) x 8 octets stockés, donc 10 x 11 x 8 soit 880
    octets. Ils sont écrasés à chacune des N itérations de la procédure.
    Donc, ces 880 octets sont suffisants.
195 
196 
197 ##(c)
198 def simulation2(L, C, N):
199     T = initialise(L, C)
200     for i in range(N):
201         T = suivant2(T)
202     return T
203 
204 L = [1, 2, 3]; C = [4, 5]; N = 4
205 print(initialise(L, C))
206 assert (simulation2(L, C, N) == [[1., 2., 3.], [4., 3., 3.], [5., 4.,
    3.5]]).all()
207 
```

```
# on initialise 4 tableaux (TT,T1,T2,T3) de flottants de taille n x (p+1) à chaque itération, d'où 4 x N x n x (p+1) flottants (8 octets chacun) stockés. Avec N = 100, n = 10 et p = 10, on obtient 4 * 100 x 10 x 11 x 8 soit 352 000 octets.


##(d)
def simulation2Proc(L, C, N):
    T = initialise(L, C)
    for i in range(N):
        procSuivant2(T)
    return T # sans cela T n'est pas accessible

L = [1, 2, 3]; C = [4, 5]; N = 4
assert (simulation1Proc(L, C, N) == [[1., 2., 3.], [4., 3., 3.], [5., 4., 3.5]]).all()

# on initialise un tableau de flottants (8 octets chacun) de taille n x (p+1), d'où n x (p+1) x 8 octets stockés, donc 10 x 11 x 8 soit 880 octets. Ils sont écrasés à chacune des N itérations de la procédure. Donc, ces 880 octets sont suffisants.
```

Q13

```
def different(T, TT, epsilon):
    n, p = np.shape(T)
    for i in range(n):
        for j in range(p):
            if abs(TT[i, j] - T[i, j]) > epsilon:
                return True
    return False

assert different(np.array([[1.,2.], [4.,0.]]), np.array([[1.,2.], [4.,0.1]]), 0.01) == True
assert different(np.array([[1.,2.], [4.,0.]]), np.array([[1.,2.], [4.,0.1]]), 0.2) == False
```

Q14

```
def simulation1bis(L, C, epsilon):
    T1, T2 = initialise(L, C), initialise(L, C)
    procSuivant2(T2)
    while different(T1, T2, epsilon):
        T1 = np.copy(T2)
        procSuivant2(T2)
    return T1, T2

L = [1, 2, 3]; C = [4, 5]; epsilon = 0.01
assert (simulation1bis(L, C, epsilon)[0] == np.array([[1. , 2. , 3. ], [4. , 3. , 3. ], [5. , 4. , 3.5]])).all()
assert (simulation1bis(L, C, epsilon)[1] == np.array([[1. , 2. , 3. ], [4. , 3. , 3. ], [5. , 4. , 3.5]])).all()
```

Q15

```
def suivant3(D):
    L, C = D
    return extractionResultat(suivant2(initialise(L, C)))

D = [[1, 2, 3], [4, 5]]
assert (suivant3(D)[0] == np.array([5. , 4. , 3.5])).all()
assert (suivant3(D)[1] == np.array([3., 3.])).all()
```

Q16

```
def simulation3(L, C, x):
    D = [L, C]
    iteration = 0
    while D[0][0] < x:
        D = suivant3(D)
        iteration += 1
    return iteration

L = [1, 2, 3]; C = [4, 5]; x = 5
assert simulation3(L, C, x) == 1
```

Q17

```
# La fonction précédente ne se termine pas toujours, par exemple si tous les coefficients du bord sont négatifs et x positif.

def simulation3N(L, C, x, N):  # N est le nombre maximum d'itérations
    for i in range(N):
        if L[0] >= x:
            return i
        [L, C] = suivant3([L, C])
    return "la simulation ne se termine pas"

L = [1, 2, 3]; C = [4, 5]; x = 5; N = 10
assert simulation3N(L, C, x, N) == 1
L = [1, 2, 3]; C = [4, 5]; x = 6; N = 10
assert simulation3N(L, C, x, N) == "la simulation ne se termine pas"
```

Q18

```
def cible(L):
    n = 2*len(L)-1
    rayon = n
    T = np.zeros((n, n))
    T[0, :],T[-1, :],T[:, 0],T[:, -1] = L[0], L[0], L[0], L[0]
    for i in range(1,len(L)):
        T[i, i:-i],T[i:-i, i],T[i:-i, -1-i],T[-1-i, i:-i] = L[i], L[i], L[i], L[i]
    return T

assert (cible([4, 2, 1]) == [[4., 4., 4., 4., 4.],
                             [4., 2., 2., 2., 4.],
                             [4., 2., 1., 2., 4.],
                             [4., 2., 2., 2., 4.],
                             [4., 4., 4., 4., 4.]]).all()

## code multiplicatif mais récritures multiples des mêmes coefficients
def cibleB(L):
    n = len(L)
    M = np.zeros([2*n - 1, 2*n - 1])
    for i in range(n):
        M[i:2*n - 1-i, i:2*n - 1-i] = L[i] * np.ones([2*n - 1- 2*i, 2*n - 1- 2*i])
    return M

assert (cible([4, 2, 1]) == [[4., 4., 4., 4., 4.],
                             [4., 2., 2., 2., 4.],
                             [4., 2., 1., 2., 4.],
                             [4., 2., 2., 2., 4.],
                             [4., 4., 4., 4., 4.]]).all()
```

Q19

```
315 def coin(L):
316     n = len(L)
317     L = [L[n-1-i] for i in range(len(L))] #on inverse L
318     T = cible(L)
319     return T[:n, n-1:]
320 
321 assert (coin([4, 2, 1]) == [[1., 1., 1.],
322                             [2., 2., 1.],
323                             [4., 2., 1.]]).all()
```