

INFORMATIQUE

TD n°3

Simulation informatique et utilisation de numpy

1 Tableaux/matrices numpy

```
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 familiarisé 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`).

1.1 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])` # 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])` # 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
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 64bits=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 -32768 à 32767 (codé sur un 16 bits=2 octet) ; $32768 = 2^{16-1}$;
 - `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)`*

1.2 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` (ligne 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]])`

Pour créer un tableau donné dont la taille est paramétrable, classiquement on commence par créer un tableau de la taille voulue avec des coefficients quelconques (par exemple des zéros avec `zeros` ou des uns avec `ones`), puis on modifie les coefficients un par un, ou par groupes, pour obtenir les coefficients voulus.

1.3 Récupération d'éléments ou de parties d'un tableau

Les opérations de *slicing* sur les listes et les listes de listes s'étendent aux 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 une dimension correspondant à la ligne `i` de `A`
`A[i,:]` # idem (syntaxe alternative)
`A[i,:]` # idem (syntaxe alternative)
Par exemple: `A[0,:] → array([1, 2, 3])`
- `A[:,j]` # renvoie le tableau numpy à une dimension qui est la colonne `j` de `A`
Par exemple: `A[:,1] → array([2,5])`
 `A[:,1]` `→ array([1,2,3])` (deux extractions successives où `A[:,1] → A`)
- `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`

1.4 Opérations sur les tableaux numpy

Pour les exemples ci-dessus, 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 ;
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`

2 Principes de la modélisation en temps discret

L'informatique permet d'étudier le comportement de systèmes complexes lors d'une succession d'instantanés numérotés par un entier à conditions de savoir décrire le passage d'un instant au suivant.

Le schéma général d'une telle modélisation est le suivant :

1. **Initialisation** : création de la structure de données qui va représenter la situation à un instant précis.
2. **Règle d'évolution** : calcul de la situation à l'instant $n+1$ en fonction de la situation à l'instant n .
Si la structure de donnée est complexe (liste ou liste de liste ou tableau numpy, potentiellement de grande taille), autant que possible, on effectue le travail **en place** c'est-à-dire à l'aide d'une procédure, c'est-à-dire une fonction qui ne renvoie rien mais modifie le paramètre.
3. **Simulation** : boucle qui permet d'effectuer la totalité de la simulation. Il y a plusieurs possibilités, notamment :
 - boucle `for` si le nombre d'instant à simuler est connu à l'avance ;
 - boucle `while` si l'on veut effectuer la simulation jusqu'à ce le système présente une caractéristique donnée, ou jusqu'à ce qu'il n'évolue plus, ou peu ;
 - boucle `for` interrompue ou boucle `while` si l'on est dans le cas précédent, mais l'on n'est pas sûr que la simulation va se terminer.

↪ lorsqu'il est complexe, le test d'arrêt de la boucle peut faire l'objet d'une fonction auxiliaire.
4. *Éventuellement* **mise en forme résultat**, par exemple d'une des manières suivantes :
 - renvoi (`return`) de tout ou d'une partie de la structure de données obtenue à la fin, ou du nombre d'itérations effectuées ;
 - affichage (`print` ou `plot`) de tout ou d'une partie de la structure de données obtenue à la fin.

3 Exercices 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 le 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 :

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

- 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`) renvoi 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 :

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

4 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 n, p 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 `L` 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(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 dans au-dessus lui et de celui de `T` situé à gauche de lui. *Par exemple :*
`suivant1(array([[ 1., 2.], [ 4., 0.], [ 5., 0.]]) → array([[ 1., 2.], [ 4., 1.5], [ 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 dans au-dessus lui et de celui de `TT` situé à gauche de lui.
Par exemple :
`suivant1(array([[ 1., 2.], [ 4., 0.], [ 5., 0.]]) → array([[ 1., 2.], [ 4., 1.5], [ 5., 3.25]])`

- 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)`). Lorsque c'est possible, 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.
- (b) Écrire cette fonction en utilisant la procédure `procSuivant1(T)`.
Quelle place en mémoire sera alors nécessaire pour stocker tous les tableaux utilisés, lorsque `L,C` sont de longueur 10 et `N` vaut 100.
- (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([[ 1., 2.][ 4., 0.]],[[ 1., 2.][ 4., .5]],0.2) → 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.
- 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?

5 Pour ceux qui ont déjà fini

EXERCICE. —

- 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]])`
- 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]])`

INFORMATIQUE

Corrigé du TD n°3

Q1

```

4  def combine_ligne(T,i,j,k):
5      T[i,:]=T[i,:]+k*T[j,:]
6
7  def combine_colonne(T,i,j,k):
8      T[:,i]=T[:,i]+k*T[:,j]

```

Q2

```

11 def somme_paires(T):
12     n,p=np.shape(T) #donne les dimensions de T
13     S=0
14     for i in range(n):
15         for j in range(p):
16             if T[i,j]%2==0:
17                 S=S+T[i,j]
18     return S

```

Q3

```

21 def tri_diag(l1,l2,l3):
22     n=len(l1)
23     M=np.zeros([n,n])
24     for i in range(n):
25         M[i,i]=l1[i]
26     for i in range(0,n-1):
27         M[i,i+1]=l3[i]
28         M[i+1,i]=l2[i]
29     return M
30
31 #ou avec np.diag
32 def tri_diag2(l1,l2,l3):
33     n=len(l1)
34     M=np.diag(l1)
35     for i in range(0,n-1):
36         M[i,i+1]=l3[i]
37         M[i+1,i]=l2[i]
38     return M
39
40 #voire, avec encore plus de np.diag
41 def tri_diag3(l1,l2,l3):
42     n=len(l1)
43     N,Q=np.zeros([n,n]),np.zeros([n,n])
44     N[l:n,:n-1]=np.diag(l2)
45     Q[:n-1,l:n]=np.diag(l3)
46     return np.diag(l1)+N+Q

```

Q4

```
49 from random import randint
50
51 def NB_hasard(n,p):
52     M=np.zeros([n,p],dtype=np.uint8)
53     for i in range(n):
54         for j in range(p):
55             M[i,j]=randint(0,255)
56     return M
```

Q5

```
59 def matrice(n):
60     M=np.zeros([n,n])
61     for i in range(n):
62         M[i,n-1]=1
63         M[n-1,i]=1
64         M[i,n-1-i]=1
65     return M
66 #attention au décalage entre l'indexation des matrices (1 à n) et les indices
    python 0 à n-1
67
68 #question posée à l'oral de Centrale PC, 2017
```

Q6

```
71 def initialise(L,C):
72     n=len(L)
73     p=len(C)
74     T=np.zeros((n,p+1))
75     T[0,:]=L
76     T[1:,0]=C
77     return T
```

Q7

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

Q8

```
86 def suivant1(T):
87     n,p=np.shape(T)
88     TT=np.zeros((n,p))
89     for i in range(1,n):
90         for j in range(1,p):
91             TT[i,j]=(T[i-1,j]+T[i-1,j-1])/2
92     return TT
```

Q9

```
86 def suivant1(T):
87     n,p=np.shape(T)
88     TT=np.zeros((n,p))
```

```

89     for i in range(1,n):
90         for j in range(1,p):
91             TT[i,j]=(T[i-1,j]+T[i-1,j-1])/2
92     return TT
93
94     ###Q9###
95     def suivant1bis(T):
96         n,p=np.shape(T)
97         T1,T2,T3=np.zeros((n,p)),np.zeros((n,p)),np.zeros((n,p))
98         T1[0,:]=T[0,:]
99         T1[1:,0]=T[1:,0] #T1 matrice nulle sauf les bords qui sont ceux de T
100        T2[1:,1:]=T[0:n-1,1:n] #T2 matrice des coefficients au dessus (et deux
bords nuls)
101        T3[1:,1:]=T[1:n,0:n-1] #T2 matrice des coefficients au de gauche (et deux
bords nuls)
102    return T1+0.5*(T2+T3)

```

Q10

```

105     #ici il faut impérativement remplir le tableau de haut en bas et de gauche à
droite
106     def suivant2(T):
107         n,p=np.shape(T)
108         TT=np.zeros((n,p))
109         for i in range(1,n):
110             for j in range(1,p):
111                 TT[i,j]=(TT[i-1,j]+TT[i,j-1])/2
112     return TT

```

Q11

```

115     #il est nécessaire de créer un tableau pour le résultat car, chaque valeur
modifiée dépend de valeurs de T qui ont été modifiées auparavant
116     def procSuivant1(T):
117         T=suivant(T)
118
119     #il n'est pas nécessaire de créer de tableau supplémentaire
120     def procSuivant2(T):
121         n,p=np.shape(T)
122         for i in range(1,n):
123             for j in range(1,p):
124                 T[i,j]=(T[i-1,j]+T[i,j-1])/2
125     #pas de return (procédure)

```

Q12

```

128     #(a)
129     def simulation1(L,C,N):
130         T=initialise(L,C)
131         for i in range(N):
132             T=suivant1(T)
133         return T
134
135     #(b)
136     def simulation1(L,C,N):
137         T=initialise(L,C)

```



```

138     for i in range(N):
139         procSuivant1(T)
140     return T
141 #on crée un tableau de flottant de taille n x (p+1) à chaque itération, d'où
    N x n x (p+1) flottants (8 octets chacun) stockés, soit 880000 octets
142
143 #(c)
144 def simulation2(L,C,N):
145     T=initialise(L,C)
146     for i in range(N):
147         T=procSuivant2(T)
148     return T
149
150 #(d)
151 def simulation2(L,C,N):
152     T=initialise(L,C)
153     for i in range(N):
154         procSuivant2(T)
155     return T
156 #on crée un seul tableau de flottants de taille n x (p+1) d'où n x (p+1)
    flottants (8 octets chacun) stockés, soit 8800 octets

```

Q13

```

159 def different(T,TT,epsilon):
160     n,p=shape(T)
161     for i in range(n):
162         for j in range(p):
163             if abs(T[i,j]-TT[i,j])>epsilon:
164                 return True
165     return False

```

Q14

```

168 #le démarrage de la boucle necessite d'avoir deux tableaux sucessifs
169 def simulation1bis(L,C,epsilon):
170     T=initialise(L,C)
171     test=True
172     while test:
173         TT=T[:,:] #VRAIE copie de T
174         procSuivant2(T)
175         test=different(T,TT,epsilon)
176     return T
177
178 #ou bien
179 def simulation1bis(L,C,epsilon):
180     T=initialise(L,C)
181     TT[:,:] #VRAIE copie de T
182     procedure(T)
183     while different(T,TT,epsilon):
184         TT=T[:,:] #VRAIE copie de T
185         procSuivant2(T)
186     return T

```

Q15

```

189 def suivant3(D):
190     [L,C]=D
191     return extractionResultat(suivant2(Tinitialise(L,C)))

```

Q16

```

195 def simulation3(L,C,x):
196     i=0
197     while L[0]<x:
198         [L,C]=suivant3([L,C])
199         i=i+1
200     return i

```

Q17

```

202 #la fonction précédente ne se termine pas toujours, par exemple si tous les
    coefficients du bord sont négatifs et x positif.
203
204 def simulation3(L,C,x,N): #N nombre maximum d'itérations
205     for i in range(N):
206         if L[0]>=x:
207             return i
208         [L,C]=suivant3([L,C])
209     return "la simulation ne se termine pas"

```

Q18

```

213 def cible(L):
214     n=len(L)
215     M=np.zeros([2*n-1,2*n-1])
216     for i in range(n):
217         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])
218     return M
219 #code court, mais récritures multiples des mêmes coefficients
220
221 def cible2(L): #autre version
222     n=len(L)
223     M=np.zeros([2*n-1,2*n-1])
224     for i in range(n-1):
225         print(i)
226         l=L[i]*np.ones(2*n-1-2*i)
227         c=L[i]*np.ones(2*n-3-2*i)
228         M[i,i:2*n-1-i]=l
229         M[2*n-2-i,i:2*n-1-i]=l
230         M[i+1:2*n-2-i,i]=c
231         M[i+1:2*n-2-i,2*n-2-i]=c
232     M[n-1,n-1]=L[n-1]
233     return M
234 #code plus pénible à écrire mais exécution plus efficace

```

Q19

```

237 def coin(L):
238     n=len(L)
239     LL=[L[n-1-i] for i in range(n)] #on renverse l'ordre des coefficients
240     M=cible(LL)

```

241

```
return M[:n,n-1:] #quart supérieur droit de M
```