# 1 - Le module Numpy et les vecteurs (tableaux à une dimension)

On a déjà eu l'occasion d'utiliser le module `numpy` pour les probabilités et pour tracer des courbes. En général, on l'importe avec l'alias `np` :

```
import numpy as np
```

Parmi ses nombreuses fonctionnalités, ce module apporte un type d'objet que l'on appelle *tableau* (*array* en anglais) et dont la syntaxe de base est la suivante :

```
np.array(liste,dtype = typ)
```

où `liste` est une liste de valeurs et `typ` le type de données vers lequel on veut que soient converties les valeurs dans la liste (par exemple `float` ou `complex`). En effet, contrairement aux listes, les éléments d'un tableau doivent être tous du *même* type.
Le paramètre `dtype` est optionnel : par défaut, en cas d'hétérogénéité de type, toutes les données de la liste seront automatiquement converties vers le type le plus fort (des flottants par exemple si la liste contient des flottants et des entiers).

```
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>> b = np.array([1,2.5,'a'])
>>> b
array(['1','2.5','a'])
```

L'implémentation en mémoire des tableaux est optimisée par rapport à celle des listes. Cela permet d'accéder et/ou de modifier plus rapidement les valeurs d'un élément, ce qui se révèle essentiel dans les méthodes numériques de calcul où les tableaux contiennent souvent des dizaines de milliers de valeurs.
Lé désignation des éléments d'un tableau est identique à celle des listes. Comme ces dernières, les tableaux sont des objets *mutables* : on peut modifier un élément dans un tableau sans toucher aux autres.

```
>>> a = np.array([1,2,3,4,5])
>>> a[0]
1
>>> a[1] = 7
>>> a
array([1,7,3,4,5])
>>> a = np.array([1,2,3,4,5])
>>> a[2] = 8.5
>>> a
np.array([1,2,8,4,5])
>>> a[3] = 'ecg'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-5c6c611e4369> in <module>()
----> 1 a[3] = 'ecg'
ValueError: invalid literal for int() with base 10: 'ecg'
```

Pour créer un tableau, on peut utiliser une syntaxe similaire à celle des listes.

```
>>> a = np.array([k for k in range(10)])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

On peut transformer une liste en tableau et inversement.

```
>>> a = np.array([k for k in range(10)])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = list(a)
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> c = np.array(b)
>>> c
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Certaines instructions permettent de créer des tableaux particuliers :

- `np.zeros(n,dtype=typ)` engendre un tableau de $n$ zéros dont le type est `typ` (le type par défaut est `float64`);
- `np.arange(start,stop,step)` est analogue à `range` mais avec la possibilité d'utiliser des arguments de type `float` (la valeur `stop` n'est pas comprise);
- `np.linspace(start,stop,num,endpoint=bool)` engendre un tableau de `num` valeurs espacées régulièrement (par défaut 50 valeurs) allant de `start` à `stop` (la valeur `stop` étant comprisee si `endpoint` est `True` ce qui est le cas par défaut).

```
>>> np.zeros(10,dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> np.arange(1,2,0.1)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> np.linspace(1,2,10,endpoint=False)
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> np.linspace(1 , 2 , 10)
array([ 1. , 1.11111111, 1.22222222, 1.33333333, 1.44444444, 1.55555556, 1.66666667,
    1.77777778, 1.88888889, 2.  ] )
>>> np.linspace(1,2,11)
array ([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

L'une des caractéristiques des tableaux – et cela constitue une différence essentielle par rapport aux listes – est leur comportement "classique" (*i.e.* case par case) par rapport aux opérateurs élémentaires (+,-,*,**,/), sous réserve de compatibilité au niveau des dimensions :

```
>>> a = np.array([0, 1, 2, 2, 3, 5])
>>> b = np.array([1, 2, 0, 2, 5, 3])
>>> a+b
array([1, 3, 2, 4, 8, 8])
>>> a*b
array([0,  2,  0,  4, 15, 15])
>>> a**2
array([0, 1, 4, 4, 9, 25])
>>> 1/(a+1)
array([1. , 0.5, 0.33333333, 0.33333333, 0.25, 0.16666667])
>>> a = np.array([0, 1, 2, 2, 3, 5])
>>> np.floor(np.exp(a))
array([1. ,2. , 7. , 7. , 20. , 148. ])
```

## 2 - Tableaux bidimensionnels

Les tableaux bidimensionnels constituent la matérialisation des *matrices*. Pour créer un tableau bidimensionnel à $m$ lignes et $n$ colonnes, on peut utiliser la syntaxe suivante :

```
tab = np.array([[liste_1],...,[liste_m]])
```

où chacune des listes liste_k comporte $n$ éléments. On rappelle que ces éléments doivent être tous du *même type*.
Par exemple :

```
>>> tab1 = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Pour désigner un élément d'un tableau bidimensionnel, on utilise une double indexation reprenant la hiérarchie de la définition précédente. Par exemple :

```
>>> tab1[1,2]    # ou : tab1[1][2]
7
>>> tab2[1,0,1] # ou : tab2[1][0][1]
6
```

Ces tableaux possèdent leurs propres attributs et méthodes dont les plus classiques sont :

- ▷ `T.dtype` → type des données du tableau,
- ▷ `T.size` → taille du tableau, c'est-à-dire son nombre de cases (y compris dans le cas de tableaux multidimensionnels),
- ▷ `T.shape` → dimensions du tableau, c'est-à-dire tuple correspondant au nombre de lignes et au nombre de colonnes,
- ▷ `T.sum()` → somme de toutes les valeurs de `T`,
- ▷ `T.mean()` → moyenne de toutes les valeurs de `T`,
- ▷ `T.max()` → maximum de toutes les valeurs de `T` (idem avec `min()`),
- ▷ `T.argmax()` → plus petit indice `i` tel que `T[i]=T.max()` (idem avec `argmin()`),
- ▷ `T.round()` → arrondi de chaque valeur de `T` à la valeur entière la plus proche,
- ▷ `T.sort()` → tri des valeurs de `T` dans l'ordre croissant,
- ▷ `T.copy()` → copie totalement indépendante de `T`.

La lecture ou l'écriture d'éléments d'un tableau peut se faire par coupes (*slices* en anglais), suivant une ou éventuellement plusieurs des dimensions du tableau. Le format d'une coupe dans un tableau `T` est le suivant :

`T[debut:fin:pas]`

où `debut` désigne le premier indice de position (compris), `fin` le dernier indice de position (non compris) et `pas` la période de coupe.

```
>>> tab3 = np.array([k for k in range(10)])
>>> tab3
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> tab3[0:11:3]
array([0, 3, 6, 9])
>>> tab3[-1:0:-3]
 array([10,  7,  4,  1])
```

```
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1[1:3,0:2]
array([[ 5,  6],
       [ 9, 10]])
>>> tab1[1:3,:]
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1[:,1:3]
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

Comme pour les tableaux bimensionnels, les opérations usuelles sont appliquées élément par élément.

```
>>> tab1*2
array([[ 2,  4,  6,  8],
       [10, 12, 14, 16],
       [18, 20, 22, 24]])
>>> tab2*tab2
array([[[  1,   4],
        [  9,  16]],
       [[ 25,  36],
        [ 49,  64]],
       [[ 81, 100],
        [121, 144]]])
>>> np.exp(tab2)
array([[[  2.71828183e+00,   7.38905610e+00],
        [  2.00855369e+01,   5.45981500e+01]],
       [[  1.48413159e+02,   4.03428793e+02],
        [  1.09663316e+03,   2.98095799e+03]],
       [[  8.10308393e+03,   2.20264658e+04],
        [  5.98741417e+04,   1.62754791e+05]]])
```

La *transposition* consiste en une *inversion des indices* : au tableau dont l'élément courant est `tab[i,j,k,...]`, on associe le tableau dont l'élément courant est `tab[...,k,j,i]`. Cette opération est réalisée par la méthode `transpose()`. Par exemple :

```
>>> tab1
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> tab1.transpose()    # ou : tab1.T
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
```

La *concaténation* consiste en une agrégation de tableaux de *tailles compatibles* : deux tableaux accolés verticalement doivent avoir le même nombre de colonnes ; deux tableaux accolés horizontalement doivent avoir le même nombre de lignes.

```
>>> tab3 = np.array([[13, 14, 15, 16]])
>>> tab3
array([[13, 14, 15, 16]])
```

```
>>> tab4 = np.concatenate((tab1,tab3),axis=0)  # noter la présence des parenthèses
>>> tab4                                       # axis=0 -> concaténation verticale
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
>>> tab5 = np.concatenate((tab1,tab3),axis=1)  # axis=1 pour concaténation
    horizontale
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-95-66e734811591> in <module>()
----> 1 tab5 = np.concatenate((tab1,tab3),axis=1)
ValueError: all the input array dimensions except for the concatenation axis
must match exactly
```

L'erreur est due à une incompatibilité entre le nombre de lignes de `tab1` (3 lignes) et `tab3` (1 ligne).

```
>>> tab5 = np.concatenate((tab1.T,tab3.T),axis=1)
>>> tab5
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

Le *produit matriciel* de deux matrices A et B de termes génériques respectifs $a_{i,j}$ et $b_{i,j}$ est la matrice C dont le terme générique $c_{i,j}$ est donné par la relation :

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

où $n$ représente le nombre de colonnes de A et le nombre de lignes de B.
Avec `numpy`, le produit matriciel entre tableaux aux dimensions compatibles peut être réalisé par l'intermédiaire de la méthode dot(). Par exemple :

```
>>> A = np.array([[1,2,3],[1,2,3]])
>>> A
array([[1, 2, 3],
       [1, 2, 3]])
>>> B = np.array([[1,3],[2,4],[1,3],[2,4]])
>>> B
array([[1, 3],
       [2, 4],
       [1, 3],
       [2, 4]])

>>> A.dot(B)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-107-7fbaa337fd94> in <module>()
----> 1 A.dot(B)
ValueError: objects are not aligned
>>> B.dot(A)
array([[ 4,  8, 12],
       [ 6, 12, 18],
       [ 4,  8, 12],
       [ 6, 12, 18]])
```

***Exercice 1***

Soit la matrice carrée :

$$\mathrm{M}(n) = \begin{bmatrix} 0 & 1 & 2 & \vdots & n \\ 1 & 1 & 2 & \vdots & \vdots \\ 2 & 2 & 2 & \vdots & n \\ \cdots & \cdots & \cdots & \ddots & \vdots \\ n & \cdots & n & \cdots & n \end{bmatrix}$$

Écrire la fonction `mat(n)` retournant le tableau semblable à la matrice ci-dessus. On proposera deux solutions fondées sur :

— une boucle ajoutant une nouvelle “couche” à chaque itération,

— une fonction récursive permettant de définir la matrice $\mathrm{M}(n)$ à partir de la matrice $\mathrm{M}(n-1)$ .

## 3 - Calculs d’algèbre linéaire

On va utiliser les deux modules `numpy` et `numpy.linalg`.

```
import numpy as np
import numpy.linalg as al
```

Considérons l’exemple de la matrice $\mathrm{M} = \begin{pmatrix} 0 & 3 & -1 \\ -1 & 2 & 1 \\ -1 & 3 & 0 \end{pmatrix}$.

Le calcul du rang montre que cette matrice est inversible :

```
>>> M = np.array([[0, 3, -1], [-1, 2, 1], [-1, 3, 0]])
>>> al.matrix_rank(M)
3
```

L’instruction `al.eig` fournit le tableau des valeurs propres et une matrice avec des vecteurs propres en colonnes :

```
>>> al.eig(M)
(array([ 1.,  2., -1.]), array([[-8.16496581e-01,  5.77350269e-01,  7.07106781e-01],
       [-4.08248290e-01,  5.77350269e-01, -5.55111512e-17],
       [-4.08248290e-01,  5.77350269e-01,  7.07106781e-01]]))
```

ce qui correspond à :

$$\ker(\mathrm{M}-\mathrm{I}_3) = \mathrm{Vect}\big((2,1,1)\big),\ \ker(\mathrm{M}-2\,\mathrm{I}_3) = \mathrm{Vect}\big((1,1,1)\big) \text{ et } \ker(\mathrm{M}+\mathrm{I}_3) = \mathrm{Vect}\big((1,0,1)\big).$$

Notons d’ailleurs que l’on a bien :

```
>>> al.matrix_rank(M - np.eye(3,3))
2
>>> al.matrix_rank(M - 2*np.eye(3,3))
2
>>> al.matrix_rank(M + np.eye(3,3))
2
```

La relation $M = PDP^{-1}$ est aisément vérifiable :

```
>>> P = al.eig(M)[1]
>>> D = np.diag(al.eig(M)[0])
>>> P
array([[-8.16496581e-01,  5.77350269e-01,  7.07106781e-01],
       [-4.08248290e-01,  5.77350269e-01, -5.55111512e-17],
       [-4.08248290e-01,  5.77350269e-01,  7.07106781e-01]])
>>> D
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0., -1.]])
>>> al.inv(P)
array([[-2.44948974e+00,  4.89506384e-15,  2.44948974e+00],
       [-1.73205081e+00,  1.73205081e+00,  1.73205081e+00],
       [-3.14018492e-16, -1.41421356e+00,  1.41421356e+00]])
>>> np.dot(np.dot(P, D), al.inv(P))
array([[ 6.66133815e-16,  3.00000000e+00, -1.00000000e+00],
       [-1.00000000e+00,  2.00000000e+00,  1.00000000e+00],
       [-1.00000000e+00,  3.00000000e+00, -8.88178420e-16]])
```

On peut aussi calculer des puissances de matrices :

```
>>> np.dot(np.dot(P, al.matrix_power(D, 10)), al.inv(P))
array([[-1022.,  1023.,  1023.],
       [-1023.,  1024.,  1023.],
       [-1023.,  1023.,  1024.]])
>>> al.matrix_power(M, 10)
array([[-1022,  1023,  1023],
       [-1023,  1024,  1023],
       [-1023,  1023,  1024]])
```

***Exercice 2***

Faire des manipulations analogues pour les matrices suivantes :

$$A = \begin{pmatrix} 2 & 1 & 1 \\ -3 & -4 & -3 \\ 3 & 5 & 4 \end{pmatrix}, B = \begin{pmatrix} -4 & 1 & 5 \\ -3 & 2 & 3 \\ -3 & 1 & 4 \end{pmatrix}, C = \begin{pmatrix} 2 & 3 & 3 \\ -1 & 0 & -1 \\ 1 & -1 & 0 \end{pmatrix}, D = \begin{pmatrix} -1 & 8 & -2 \\ -3 & 5 & 3 \\ -3 & 8 & 0 \end{pmatrix},$$

$$E = \begin{pmatrix} 5 & 3 & 3 \\ -8 & -11 & -8 \\ 8 & 13 & 10 \end{pmatrix}, F = \begin{pmatrix} 7 & 5 & -9 \\ 2 & 3 & -2 \\ 2 & 5 & -4 \end{pmatrix}, G = \begin{pmatrix} 3 & -2 & -2 \\ -5 & -7 & -5 \\ 5 & 12 & 10 \end{pmatrix}.$$

***Exercice 3***

Déterminer le noyau des matrices :

$$A = \begin{pmatrix} 1 & -1 & -1 & 5 \\ -2 & 2 & 2 & 2 \\ -2 & 2 & 2 & 2 \\ 1 & -1 & -1 & 5 \end{pmatrix} \text{ et } B = \begin{pmatrix} 9 & -5 & -9 & 1 \\ 3 & -1 & -3 & -3 \\ 3 & -1 & -3 & -3 \\ 7 & -7 & -7 & 3 \end{pmatrix}.$$

Signalons également l'instruction `al.solve(A, b)` donnant la solution du système $Ax = b$ dans le cas où A est inversible :

```
>>> A
array([[ 0,  3, -1],
       [-1,  2,  1],
       [-1,  3,  0]])
>>> b = np.array([1, 2, 1])
>>> b
array([1, 2, 1])
>>> al.solve(A, b)
array([2., 1., 2.])
```