# Tableaux et calcul matriciel avec NumPy

Dans cette page, nous utilisons une importation de **NumPy** et l'attribution d'un alias `np`. Il existe une autre façon de procéder en exploitant un module nommé « pylab ». Celui-ci autorise un style de programmation qui se rapproche plus du logiciel Matlab. Toutefois, cette approche n'est pas recommandée car l'origine des fonctions issues de **NumPy** n'apparaît pas de manière explicite. Vous pouvez néanmoins trouver une présentation de cette démarche dans la page Tableaux et calcul matriciel avec PyLab

Nous allons voir comment créer des tableaux avec la fonction `numpy.array()` de **NumPy**. Ces tableaux pourront être utilisés comme des vecteurs ou des matrices grâce à des fonctions de **NumPy** ( `numpy.dot()` , `numpy.linalg.det()` , `numpy.linalg.inv()` , `numpy.linalg.eig()` , etc.) qui permettent de réaliser des calculs matriciels utilisés en algèbre.

Premièrement, nous allons importer le module `numpy`. Pour cela, il suffit de faire :

```
import numpy as np
```

**Note**

on importe la totalité du module `numpy` et on lui donne un alias pour alléger ensuite l'écriture de l'appel des fonctions. L'alias qui est le plus couramment utilisé est `np`. Pour en savoir plus sur l'importation et la création d'un alias, vous pouvez consulter la page Modules et importations.

## Tableaux - `numpy.array()`

Pour créer des tableaux, nous allons utiliser `numpy.array()` .

### Tableaux monodimensionnels (1D)

Pour créer un tableau 1D, il suffit de passer une liste de nombres en argument de `numpy.array()` . Un liste est constituée de nombres séparés par des virgules et entourés de crochets ( `[` et `]` ).

```
>>> a = np.array([4,7,9])
>>> a
array([4, 7, 9])
```

Pour connaître le type du résultat de `numpy.array()` , on peut utiliser la fonction `type()` .

```
>>> type(a)
numpy.ndarray
```

On constate que ce type est issu du package `numpy`. Ce type est différent de celui d'une liste.

```
>>> type([4, 7, 9])
list
```

## Tableaux bidimensionnels (2D)

Pour créer un tableau 2D, il faut transmettre à `numpy.array()` une liste de listes grâce à des crochets imbriqués.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

## La fonction `numpy.size()`

La fonction `numpy.size()` renvoie le nombre d'éléments du tableau.

```
>>> a = np.array([2,5,6,8])
>>> np.size(a)
4
>>> b = np.array([[1, 2, 3],
                  [4, 5, 6]])
>>> np.size(b)
6
```

## La fonction `numpy.shape()`

La fonction `numpy.shape()` (*forme*, en anglais) renvoie la taille du tableau.

```
>>> a = np.array([2,5,6,8])
>>> np.shape(a)
(4,)
>>> b = np.array([[1, 2, 3],
                  [4, 5, 6]])
>>> np.shape(b)
(2, 3)
```

On distingue bien ici que `a` et `b` correspondent à des tableaux 1D et 2D, respectivement.

## Produit terme à terme

Il est possible de réaliser un produit terme à terme grâce à l'opérateur `*`. Il faut dans ce cas que les deux tableaux aient la même taille.

```
>>> a = np.array([[1, 2, 3],
                  [4, 5, 6]])
>>> b = np.array([[2, 1, 3],
                  [3, 2, 1]])
>>> a*b
array([[ 2,  2,  9],
       [12, 10,  6]])
```

## Produit matriciel - `numpy.dot()`

Un tableau peut jouer le rôle d'une matrice si on lui applique une opération de calcul matriciel. Par exemple, la fonction `numpy.dot()` permet de réaliser le produit matriciel.

```
>>> a = np.array([[1, 2, 3],
                  [4, 5, 6]])
>>> b = np.array([[4],
                  [2],
                  [1]])
>>> np.dot(a,b)
array([[11],
       [32]])
```

Le produit d'une matrice de taille `n x m` par une matrice `m x p` donne une matrice `n x p`.

A partir de la version 3.5 de Python, il est également possible d'effectuer le produit matriciel en utilisant `@`.

```
>>> a @ b
array([[11],
       [32]])
```

## Transposé

```
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

## Complexe conjugué - `numpy.conj()`

```
>>> u = np.array([[  2j, 4+3j],
                  [2+5j, 5   ],
                  [   3, 6+2j]])
>>> np.conj(u)
array([[ 0.-2.j,  4.-3.j],
       [ 2.-5.j,  5.+0.j],
       [ 3.+0.j,  6.-2.j]])
```

## Transposé complexe conjugué

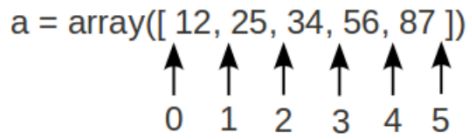
```
>>> np.conj(u).T
array([[ 0.-2.j,  2.-5.j,  3.+0.j],
       [ 4.-3.j,  5.+0.j,  6.-2.j]])
```

# Tableaux et slicing

Lors de la manipulation des tableaux, on a souvent besoin de récupérer une partie d'un tableau. Pour cela, Python permet d'extraire des *tranches* d'un tableau grâce une technique appelée **slicing** (tranchage, en français). Elle consiste à indiquer entre crochets des indices pour définir le début et la fin de la *tranche* et à les séparer par deux-points `:`.

```
>>> a = np.array([12, 25, 34, 56, 87])
>>> a[1:3]
array([25, 34])
```

Dans la tranche [n:m], l'élément d'indice `n` est inclus, mais pas celui d'indice `m`. Un moyen pour mémoriser ce mécanisme consiste à considérer que les limites de la tranche sont définies par les numéros des positions situées entre les éléments, comme dans le schéma ci-dessous :




Il est aussi possible de ne pas mettre de début ou de fin.

```
>>> a[1:]
array([25, 34, 56, 87])
>>> a[:3]
array([12, 25, 34])
>>> a[:]
array([12, 25, 34, 56, 87])
```

## Slicing des tableaux 2D

```
>>> a = np.array([[1, 2, 3],
                  [4, 5, 6]])
>>> a[0,1]
2
>>> a[:,1:3]
array([[2, 3],
       [5, 6]])
>>> a[:,1]
array([2, 5])
>>> a[0,:]
array([1, 2, 3])
```

**Avertissement**

a[:,n] donne un tableau 1D correspondant à la colonne d'indice n de a. Si on veut obtenir un tableau 2D correspondant à la colonne d'indice n, il faut faire du slicing en utilisant a[:,n:n+1].

```
>>> a[:,1:2]
array([[2],
       [5]])
```

## Tableaux de 0 - `numpy.zeros()`

zeros(n) renvoie un tableau 1D de n zéros.

```
>>> np.zeros(3)
array([ 0.,  0.,  0.])
```

zeros((m,n)) renvoie tableau 2D de taille `m x n`, c'est-à-dire de *shape* (m,n).

```
>>> np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

## Tableaux de 1 - `numpy.ones()`

```
>>> np.ones(3)
array([ 1.,  1.,  1.])
>>> np.ones((2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

## Matrice identité - `numpy.eye()`

eye(n) renvoie tableau 2D carré de taille `n x n`, avec des *uns* sur la diagonale et des *zéros* partout ailleurs.

```
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

**Exercice**

Effectuer le produit suivant :

$$\begin{pmatrix} 2 & 3 & 4 \\ 1 & 5 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Produire un tableau de taille `7 x 8` ne contenant que des 3.

# Algèbre linéaire

## Déterminant - `numpy.linalg.det()`

```
>>> from numpy.linalg import det
>>> a = np.array([[1, 2],
                  [3, 4]])
>>> det(a)
-2.0
```

## Inverse - `numpy.linalg.inv()`

```
>>> from numpy.linalg import inv
>>> a = np.array([[1, 3, 3],
                  [1, 4, 3],
                  [1, 3, 4]])
>>> inv(a)
array([[ 7., -3., -3.],
       [-1.,  1.,  0.],
       [-1.,  0.,  1.]])
```

## Résolution d'un système d'équations linéaires - `numpy.linalg.solve()`

Pour résoudre le système d'équations linéaires 3 * x0 + x1 = 9 et x0 + 2 * x1 = 8 :

```
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

Pour vérifier que la solution est correcte :

```
>>> np.allclose(np.dot(a, x), b)
True
```

**Voir aussi**

https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html

## Valeurs propres et vecteurs propres - `numpy.linalg.eig()`

```
>>> from numpy.linalg import eig
>>> A = np.array([[ 1, 1, -2 ], [-1, 2, 1], [0, 1, -1]])
>>> A
array([[ 1,  1, -2],
       [-1,  2,  1],
       [ 0,  1, -1]])
>>> D, V = eig(A)
>>> D
array([ 2.,  1., -1.])
>>> V
array([[  3.01511345e-01,  -8.01783726e-01,   7.07106781e-01],
       [  9.04534034e-01,  -5.34522484e-01,  -3.52543159e-16],
       [  3.01511345e-01,  -2.67261242e-01,   7.07106781e-01]])
```

Les colonnes de V sont les vecteurs propres de A associés aux valeurs propres qui apparaissent dans D.

**Exercice :** Vérifier que les colonnes de V sont bien des vecteurs propres de A

# Changement de la taille d'un tableau

Il est possible de changer la taille d'un tableau en utilisant l'attribut **shape** de ce tableau.

```
>>> u = np.arange(1, 16)
>>> u
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> np.shape(u)
(15,)
>>> u.shape = (3, 5)
>>> u
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
>>> np.shape(u)
(3, 5)
```

## Obtention d'un tableau 2D ligne ou colonne

```
>>> a = np.arange(1, 6)
>>> a
array([1, 2, 3, 4, 5])
>>> a.shape = (1, np.size(a))
>>> a
array([[1, 2, 3, 4, 5]])
>>> a.shape = (np.size(a), 1)
>>> a
array([[1],
       [2],
       [3],
       [4],
       [5]])
```