Le module `numpy` : opérations

Il est possible d'effectuer des opérations sur les tableaux `numpy` soit termes à termes, soit des opérations vectorielles ou matricielles. En général, ces opérations sont à privilégier car beaucoup plus efficace (elles sont programmées dans un langage bas niveau, ce qui améliore les performances).

In [1]:
```
import numpy as np
x_4 = np.random.rand(4)
x_5 = np.random.rand(5)
A = np.random.rand(5, 4)
Aint = np.random.randint(10, size=(5, 4))
xint = np.random.randint(10, size=(4,))
```

Opérations termes à termes

Toutes les opérations classiques sur les nombres sont possibles en utilisant les mêmes opérateurs (`+` , `-` , `*` , `/` , `**` , `<` , `<=` , `>` , `>=` , `//` , `%` ...)

In [4]:
```
print(A)
print(2.1*A)
print(A**2)
print(A[0, 0]**2, (A**2)[0, 0])
print(A+1)
```

```
[[0.45839443 0.21391285 0.65310194 0.50678854]
 [0.76879939 0.12859906 0.96495489 0.91983475]
 [0.69938218 0.64103234 0.6353177  0.740514  ]
 [0.72167421 0.34237241 0.5777261  0.5088241 ]
 [0.8319637  0.7991838  0.62544682 0.59158447]]
[[0.96262829 0.44921698 1.37151407 1.06425593]
 [1.61447872 0.27005802 2.02640527 1.93165298]
 [1.46870257 1.34616792 1.33416718 1.55507939]
 [1.51551584 0.71898207 1.21322481 1.06853062]
 [1.74712377 1.67828597 1.31343831 1.2423274 ]]
[[0.21012545 0.04575871 0.42654214 0.25683462]
 [0.5910525  0.01653772 0.93113795 0.84609597]
 [0.48913543 0.41092247 0.40362858 0.54836098]
 [0.52081366 0.11721887 0.33376745 0.25890197]
 [0.6921636  0.63869474 0.39118372 0.34997219]]
0.21012544954509835 0.21012544954509835
[[1.45839443 1.21391285 1.65310194 1.50678854]
 [1.76879939 1.12859906 1.96495489 1.91983475]
 [1.69938218 1.64103234 1.6353177  1.740514  ]
 [1.72167421 1.34237241 1.5777261  1.5088241 ]
 [1.8319637  1.7991838  1.62544682 1.59158447]]
```

In [9]:
```
print(A)
print(A+A - 2*A)
print(x_4)
print(A+x_4)  # Attention, dans ce cas, `x_4` est augmenté pour avoir 2 dimensions
print(x_4[None, :].repeat(5, axis=0))
print(A + x_4)
print(A+x_5[:, None].repeat(4, axis=1))  # ne fonctionne pas : problème de dimension
```

```
[[0.45839443 0.21391285 0.65310194 0.50678854]
 [0.76879939 0.12859906 0.96495489 0.91983475]
 [0.69938218 0.64103234 0.6353177  0.740514  ]
 [0.72167421 0.34237241 0.5777261  0.5088241 ]
 [0.8319637  0.7991838  0.62544682 0.59158447]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[0.09134069 0.14594231 0.9367839  0.85810111]
[[0.54973511 0.35985516 1.58988584 1.36488964]
 [0.86014007 0.27454137 1.9017388  1.77793586]
 [0.79072286 0.78697465 1.57210161 1.5986151 ]
 [0.81301489 0.48831472 1.51451001 1.36692521]
 [0.92330438 0.94512611 1.56223072 1.44968558]]
[[0.09134069 0.14594231 0.9367839  0.85810111]
 [0.09134069 0.14594231 0.9367839  0.85810111]
 [0.09134069 0.14594231 0.9367839  0.85810111]
 [0.09134069 0.14594231 0.9367839  0.85810111]
 [0.09134069 0.14594231 0.9367839  0.85810111]]
[[0.54973511 0.35985516 1.58988584 1.36488964]
 [0.86014007 0.27454137 1.9017388  1.77793586]
 [0.79072286 0.78697465 1.57210161 1.5986151 ]
 [0.81301489 0.48831472 1.51451001 1.36692521]
 [0.92330438 0.94512611 1.56223072 1.44968558]]
[[1.14000433 0.89552275 1.33471185 1.18839844]
 [0.89591784 0.25571751 1.09207335 1.04695321]
 [1.11063148 1.05228165 1.046567   1.1517633 ]
 [0.74264698 0.36334519 0.59869887 0.52979688]
 [1.54288093 1.51010102 1.33636404 1.3025017 ]]
```

In [7]:
```
print(A < 0.5)
B = A * (A < 0.5) - A * (A >= 0.5)
print(B)
```

```
[[ True  True  True False]
 [False False  True False]
 [False  True  True  True]
 [False  True  True  True]
 [False  True  True False]]
[[ 0.4643984   0.43683496  0.08359677 -0.94394679]
 [-0.64338395 -0.65930351  0.03223884 -0.55564108]
 [-0.59362203  0.13417642  0.29878549  0.26113081]
 [-0.94191874  0.24188861  0.10074206  0.33878247]
 [-0.85994567  0.3888307   0.09734452 -0.74184998]]
```

In [13]:
```
print(Aint)
print((2*Aint+1) % Aint)
```

```
[[6 2 3 2]
 [9 6 5 2]
 [2 8 6 6]
 [7 8 3 9]
 [1 4 8 6]]
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [0 1 1 1]]
```

Test de vitesse

In [14]:
```
%timeit B = 2*A
```

```
888 ns ± 29.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loo
ps each)
```

In [15]:
```
%%timeit
B = np.zeros(A.shape)
for i in range(A.shape[0]):
    for j in range(B.shape[1]):
        B[i, j] = 2*A[i, j]
```

```
14.5 µs ± 1.08 µs per loop (mean ± std. dev. of 7 runs, 10000 loop
s each)
```

Opérations vectorielles

Il est assez facile d'identifier les tableaux `numpy` à des objets algébriques comme les vecteurs ou les matrices. Attention toutefois aux vecteurs qui sont des éléments d'espaces vectoriels de dimension 1. Il est également possible d'identifier ces vecteurs à des matrices lignes ou colonnes.

Petit rappel d'algèbre linéaire

Si l'on considère un espace vectoriel E muni d'une base $(e_1, \dots, e_n)$ et un espace vectoriel F muni d'une base $(f_1, \dots, f_n)$ (donc E est de dimension n et F de dimension m), une application linéaire $\phi : E \to F$ est caractérisée par l'image de la base (e_i) qui s'écrit dans la base des (f_j) :

$$\phi(e_i) = \sum_{j=1}^{m} A_{i,j} f_j, \qquad 1 \leq i \leq n.$$

On appelle matrice de ϕ le tableau des nombres $A_{i,j}$.

Un vecteur x de E (*resp.* y de F) s'écrit dans la base des (e_i) (*resp.* (f_j))

$$x = \sum_{i=1}^{n} x_i e_i, \qquad y = \sum_{j=1}^{m} y_j f_j.$$

On assimile donc le vecteur x (*resp.* y) au tableau monodimensionnel $(x_1, \dots, x_n)$ (*resp.* $(y_1, \dots, y_m)$).

Cependant, étant donné un vecteur x de E, il est possible de considérer la forme linéaire $\phi_x : E \to \mathbb{R}$ définie par

$$\phi_x(a) = \sum_{i=1}^{n} x_i a_i, \qquad a \in E.$$

Le vecteur x est alors assimilé à une matrice ligne avec $x_{1,i} = x_i$, $1 \leq i \leq n$. Il est également possible de plonger $\mathbb{R}$ dans E en utilisant l'application linéaire $\psi_x : \mathbb{R} \to E$ définie par

$$\psi_x(a) = (ax_1, ax_2, \dots ax_n), \qquad a \in \mathbb{R}.$$

Le vecteur x est alors assimilé à une matrice colonne avec $x_{i,1} = x_i$, $1 \leq i \leq n$.

Produit matrice vecteur

Soit $A = (A_{i,j})_{1 \leq i,j \leq N}$ et $x = (x_i)_{1 \leq i \leq N}$, le produit Ax est le vecteur défini par

$$(Ax)_i = \sum_{j=1}^{N} A_{i,j} x_j, \qquad 1 \leq i \leq N.$$

ATTENTION

> En python, des tableaux de forme `(n, )`, `(n, 1)` et `(1, n)` peuvent être utilisés pour représenter des vecteurs mathématiques (et l'on peut facilement les transformer pour passer d'une version à l'autre). Mais il fait faire attention à ce que l'on fait car les opérations dessus ne sont pas les mêmes...

Sommes et produits

La somme de deux tableaux (lorsque les tailles sont compatibles) a déjà été vue : c'est la même opération que termes à termes. On utilise le `+`.

Pour le produit, il faut utiliser la fonction `np.matul()` ou de manière équivalente le `@`. Tant que l'on n'utilise pas de tableaux avec plus de 3 dimensions, on peut aussi utiliser de manière équivalente la fonction `np.dot`.

Pour les vecteurs, on a aussi `np.inner` pour le produit scalaire, `np.outer` pour le produit extérieur :

$$x \cdot y = \sum_{i=1}^{N} x_i y_i, \qquad (x \otimes y)_{i,j} = x_i y_j, \quad 1 \leq i, j \leq N.$$

In [17]:
```
print(Aint)
print(xint)
print(Aint @ xint)          # produit matrice-vecteur
print(np.dot(Aint, xint))   # produit matrice-vecteur
print(xint @ xint)          # produit scalaire de deux vecteurs
print(np.dot(xint, xint))   # produit scalaire de deux vecteurs
print(np.inner(xint, xint))
print(np.outer(xint, xint))
```

```
[[6 2 3 2]
 [9 6 5 2]
 [2 8 6 6]
 [7 8 3 9]
 [1 4 8 6]]
[8 4 2 1]
[ 64 108  66 103  46]
[ 64 108  66 103  46]
85
85
85
[[64 32 16  8]
 [32 16  8  4]
 [16  8  4  2]
 [ 8  4  2  1]]
```

Transposition

On calcule la transposée d'une matrice à l'aide de la fonction `np.transpose`. Lorsque c'est possible (lorsque `python` estime que c'est possible), une vue est renvoyée sans faire de copie...

In [23]:
```
print(A)
print(A.T)
print(A.shape)
print(A.transpose().shape)    # transposition d'une matrice
print(x_4, x_4.T)
x_4bis = x_4[None, :]
print(x_4bis)
print(x_4bis.T)
print(x_4bis.shape)
print(x_4bis.T.shape)
print(x_4.shape)
print(x_4.transpose().shape)  # ne fait rien sur les tableaux avec
une seule dimension
```

```
[[0.45839443 0.21391285 0.65310194 0.50678854]
 [0.76879939 0.12859906 0.96495489 0.91983475]
 [0.69938218 0.64103234 0.6353177  0.740514  ]
 [0.72167421 0.34237241 0.5777261  0.5088241 ]
 [0.8319637  0.7991838  0.62544682 0.59158447]]
[[0.45839443 0.76879939 0.69938218 0.72167421 0.8319637 ]
 [0.21391285 0.12859906 0.64103234 0.34237241 0.7991838 ]
 [0.65310194 0.96495489 0.6353177  0.5777261  0.62544682]
 [0.50678854 0.91983475 0.740514   0.5088241  0.59158447]]
(5, 4)
(4, 5)
[0.09134069 0.14594231 0.9367839  0.85810111] [0.09134069 0.145942
31 0.9367839  0.85810111]
[[0.09134069 0.14594231 0.9367839  0.85810111]]
[[0.09134069]
 [0.14594231]
 [0.9367839 ]
 [0.85810111]]
(1, 4)
(4, 1)
(4,)
(4,)
```

Le module `linalg`

Le module `linalg` est un sous-module de `numpy` qui apporte un grand nombre de méthodes numériques adaptées à l'algèbre linéaire. Citons par exemple :

- `np.linalg.det` pour calculer le déterminant d'une matrice ;
- `np.linalg.solve` pour résoudre un système linéaire ;
- `np.linalg.inv` pour calculer l'inverse d'une matrice...

In [ ]:
```
```