L2-S3 : UE Méthodes numériques

SEANCE 8

Calcul matriciel, Systèmes d'équations linéaires

28 novembre 2024

Calcul matriciel : création de matrices

In [1]:
```
1 import numpy as np
2 a = np.zeros(4)
3 print(a)
```

Out[1]:
```
array([ 0., 0., 0., 0.])
```

In [2]:
```
1 nx, ny = 2, 2
2 a=np.zeros((nx,ny))
3 print(a)
```

Out[2]:
```
array([ [ 0.,  0. ]
        [ 0.,  0.] ]
```

Calcul matriciel : création de matrices

Il existe également les fonctions `np.ones`, `np.eye`, `np.identity`, `np.empty`, ... Par exemple :

- `np.identity(3)` crée la matrice identité d'ordre 3,
- `np.empty` crée un tableau vide,
- `np.ones(5)` crée le vecteur `[1 1 1 1 1]`,
- `np.eye(3,2)` crée la matrice à 3 lignes et 2 colonnes contenant des 1 sur la diagonale et des zéro partout ailleurs.

Par défaut les éléments d'un tableaux sont des `float` (un réel en double précision) ; mais on peut donner un deuxième argument qui précise le type (`int`, `complex`, `bool`, ...).

Calcul matriciel : création de matrices

In [3]:
```
1 np.eye(2, dtype=int)
```

Out[3]:
```
array([[1, 0],
       [0, 1]])
```

In [4]:
```
1 b = np.arange(10)
2 print(b)
```

Out[4]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [5]:
```
1 L1, L2 = [1, 2, 3], [4, 5, 6]
2 a = np.array([L1, L2])
3 print(a)
```

Out[5]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Calcul matriciel : opérations

- `np.argmax(a, axis=i)` : renvoie un tableau d'indices des valeurs maximales selon l'axe i
- `np.max(a)` : renvoie le maximum
- `np.conj(a)` : renvoie le conjugué de a
- `np.sum(a)` : renvoie la somme des éléments de a
- `np.prod(a)` : renvoie le produit des éléments de a
- `np.transpose(a)` ou `a.T` : renvoie le transposé de a
- `a.astype(type)` : renvoie un tableau de a converti sous un certain type

Calcul matriciel : opérations

La fonction `np.where(condition)` renvoie les indices pour lesquels les valeurs d'un tableau remplissent une condition logique.

In [6]:

```
a = np.array([[7,8,9],[4,5,6],[1,2,3]])
print ("a\n",a)
maxcol = np.argmax(a,axis=0)
print("numero de la ligne avec le max de la
    colonne", maxcol)
maxline = np.argmax(a, axis = 1)
print("numero de la colonne avec le max de la
    ligne", maxline)
ind = np.where(a > 5)
print("ind\n", ind)
```

Out[6]:

```
a
 [[7 8 9]
 [4 5 6]
 [1 2 3]]
numero de la ligne avec le max de la colonne [0
     0 0]
numero de la colonne avec le max de la ligne [2
     2 2]
ind
 (array([0, 0, 0, 1]), array([0, 1, 2, 2]))
```

Calcul matriciel : opérations

Les indices ainsi obtenus peuvent servir directement pour traiter le sous-ensemble des éléments d'un tableau vérifiant la condition.

In [7]:
```
a[ind] = 0
print(a)
```

Out[7]:
```
[[0 0 0]
 [4 5 0]
 [1 2 3]]
```

C'est une fonction très pratique pour traiter les données ou les images.

Calcul matriciel : opérations

Si le fait d'obtenir les indices est peu important à garder en mémoire pour les transformations à apporter à la matrice, on peut aussi directement écrire les conditions comme ceci :

In [8]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a > 5) # renvoie un masque booleen
```

Out[8]:
```
[[False False False]
 [False False  True]
 [ True  True  True]]
```

In [9]:
```
a[a>5] = 0
print(a)
```

Out[9]:
```
[[1 2 3]
 [4 5 0]
 [0 0 0]]
```

Calcul matriciel : opérations

In [10]:
```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
a = np.where(a>5, 0, a)
print(a)
```

Out[10]:
```
[[1 2 3]
 [4 5 0]
 [0 0 0]]
```

Calcul matriciel : opérations

Le produit matriciel se réalise avec la fonction `dot()` ou l'opérateur `@`.
Attention :

- si le deuxième argument est un vecteur ligne, il sera transposé si besoin en vecteur colonne, à cause du fait qu'il est plus simple de définir un vecteur ligne, par exemple `v=np.array([0,1,2])`, qu'un vecteur colonne, ici `v=np.array([[0],[1],[2]])`
- le résultat est toujours présenté comme un vecteur ligne (même quand cela doit être un vecteur colonne).

In [11]:
```
A=np.array([[1,1,1],[0,1,1],[0,0,1]])
v=np.array([0,1,2])
print("v.A", np.dot(v,A))
```

Out[11]:
```
v.A: [0 1 3]
```

In [12]:
```
print("v.A", v @ A)
```

Out[12]:
```
v.A: [0 1 3]
```

In [13]:
```
print("A.v", A @ v)
```

Out[13]:
```
A.v: [3 3 2]
```

Calcul matriciel : opérations

Pour élever une matrice au carré il faut écrire `np.dot(A, A)` ou `A @ A`.

Pour élever une matrice carrée à une puissance n il faut faire une boucle :

In [14]:
```
B = A
n = 5
for i in range(1,n):
    B = A @ B
```

A la sortie de cette boucle B contient A^5.

Attention : l'opération `A**2` correspond à une élévation au carré terme à terme. De même `A*B` correspond au produit terme à terme des éléments des tableaux `A` et `B`.

Inversion d'une matrice n x n, calcul du déterminant

Une matrice carrée $n \times n$ est inversible si et seulement si son déterminant est non nul.
Pour calculer le déterminant d'une matrice carrée, il faut utiliser la fonction `det` du sous module de numpy appelé `np.linalg`. Exemple :

In [15]:
```
1 a = np.array([[1,0,0], [0,2,0], [0,0,3]])
2 print(" Matrice a:\n", a)
```

Out[15]:
```
Matrice a:
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
```

In [16]:
```
1 print("Determinant de a: ", np.linalg.det(a))
```

Out[16]:
```
Determinant de a:  6.0
```

Inversion d'une matrice n x n, calcul du déterminant

Exemple 2 :

In [17]:
```
b = 10*np.array([[1,2,3], [4,5,6], [7,8,9]])
print(" Matrice a:\n", a)
```

Out[17]:
```
Matrice b:
 [[10, 20, 30],
 [40, 50, 60],
 [70, 80, 90]]
```

In [18]:
```
np.linalg.det(b)
```

Out[18]:
```
-3.1974423109204e-12
```

Attention : Une matrice non inversible possède un déterminant nul, mais le calcul numérique renvoie rarement exactement 0.

Inversion d'une matrice n x n, calcul du déterminant

L'inversion d'une matrice carrée se réalise avec la fonction `np.linalg.inv()`, si la matrice est inversible.
Exemple :

In [19]:
```
c=np.array([[1,1,1],[0,1,1],[0,0,1]])
inv_c = np.linalg.inv(c)
print("Matrice c:\n", c)
print("Inverse de c:\n", inv_c)
print("produit de c par son inverse:\n",
    c@inv_c)
```

Out[19]:
```
Matrice c:
 [[1 1 1]
 [0 1 1]
 [0 0 1]]
Inverse de c:
 [[ 1. -1.  0.]
 [ 0.  1. -1.]
 [ 0.  0.  1.]]
produit de c par son inverse:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Inversion d'une matrice n x n, calcul du déterminant

Exemple 2 : Matrice b précédente :

In [20]:

```
print("Matrice b:\n", b)
print("Determinant de b: ", np.linalg.det(b))
inv_b = np.linalg.inv(b)
print("Inverse de b\n: ", inv_b)
print("produit de b par son inverse:\n",
    b@inv_b)
```

Out[20]:

```
Matrice b:
 [[10 20 30]
 [40 50 60]
 [70 80 90]]
Determinant de b:  -4.263256414560599e-12
Inverse de b:
 [[ 7.03687e+13 -1.40737e+14  7.03687e+13]
 [-1.40737e+14  2.81474e+14 -1.40737e+14]
 [ 7.03687e+13 -1.40737e+14  7.03687e+13]]
produit de b par son inverse:
 [[ 0.84375  0.      -0.21875]
 [ 0.1875   0.       0.0625 ]
 [-0.96875  0.       0.59375]]
```

Inversion d'une matrice n x n, calcul du déterminant

La précédente matrice `b` n'est pas inversible (déterminant nul) mais comme le déterminant calculé n'était pas exactement nul, un résultat (absurde) est tout de même renvoyé par `inv()`.
Moralité, il faut garder un œil critique sur les résultats.

Résolution de systèmes de n équations linéaires à n inconnues

Représentons un système de n équations linéaires à n inconnues sous la forme :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n &= b_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + \ldots + a_{4n}x_n &= b_4 \\ \qquad\qquad\qquad\quad \ldots & \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n \end{cases}$$

où les inconnues sont les x_i, tandis que les a_{ij} et b_i sont des coefficients constants (qui peuvent être nuls). Le problème se reformule sous la **forme matricielle** :

$$A \cdot X = B$$

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \ldots & a_{3n} \\ a_{41} & a_{42} & a_{43} & \ldots & a_{4n} \\ & & & \ldots & \\ a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \ldots \\ x_n \end{pmatrix}, \quad B = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \ldots \\ b_n \end{pmatrix}$$

Résolution par inversion de la matrice A

Si la matrice carrée A est inversible, alors ce système admet une solution unique. On peut obtenir cette solution en calculant l'inverse de la matrice A, puis le produit :

$$X = A^{-1} \cdot B$$

avec la méthode `np.linalg.inv(A)` qui renvoie un tableau contenant l'inverse de A.

Syntaxe : `X = np.dot(np.linalg.inv(A), B)`

Mais il se trouve que les algorithmes de résolution directe d'un système d'équations sont **beaucoup plus efficaces** en général que ceux de calcul d'inverse de matrice. Cette méthode, la plus directe et logique au sens mathématique, n'est donc pas la plus efficace. C'est un argument très fort lorsque le nombre n d'équations devient important.

Résolution par la méthode LU

Imaginons que l'on sache décomposer la matrice carrée inversible A en un produit de deux matrices :

$$A = L \cdot U$$

où L est une matrice triangulaire inférieure ($L =$ "lower") qui n'a des éléments que sur ou sous la diagonale, et U est une matrice triangulaire supérieure ($U =$ "upper") qui n'a des éléments que sur ou au-dessus de la diagonale :

$$L = \begin{pmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ l_{41} & l_{42} & l_{43} & \dots & 0 \\ & & & \dots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{pmatrix}, \qquad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ 0 & 0 & 0 & \dots & u_{4n} \\ & & & \dots & \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix}$$

Résolution par la méthode LU

Le système initial s'écrit alors $L \cdot (U \cdot X) = B$
et il reste à chercher un vecteur intermédiaire Y tel que $L \cdot Y = B$
puis le vecteur solution X tel que $U \cdot X = Y$

Résolution par la méthode LU

L'avantage de procéder ainsi est que maintenant le **système est trivial** à cause de la forme **triangulaire** des deux matrices :

$$\begin{cases} y_1 = b_1/\ell_{11} \\ y_2 = (b_2 - \ell_{21}y_1)/\ell_{22} \\ y_3 = [b_3 - (\ell_{31}y_1 + \ell_{32}y_2)]/\ell_{33} \\ \dots \end{cases}$$

permet de déterminer successivement tous les y_i en partant de y_1

$$\begin{cases} x_n = \dfrac{y_n}{u_{nn}} \\ x_{n-1} = \dfrac{1}{u_{n-1,n-1}}(y_{n-1} - u_{n-1,n}x_n) \\ x_{n-2} = \dfrac{1}{u_{n-2,n-2}}[y_{n-2} - (u_{n-2,n}x_n + u_{n-2,n-1}x_{n-1})] \\ \dots \end{cases}$$

permet ensuite de déterminer tous les x_i en partant de x_n

Résolution par la méthode LU

Évidemment tout ceci repose sur le fait qu'on sache décomposer la matrice A en deux matrices triangulaires. Il se trouve que c'est tout à fait faisable simplement si on impose que $\forall i, \ell_{ii} = 1$ avec :

$$\forall i \leqslant j, u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj}, \quad \forall i > j, \ell_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} u_{kj} \right)$$

Tous les termes dans les sommes précédentes sont calculables à condition de calculer alternativement les lignes de U et les colonnes de L :

- première ligne de U : $u_{1j} = a_{1j}$, première colonne de L : $\ell_{i1} = \frac{a_{i1}}{a_{11}}$
- seconde ligne de U : $u_{2j} = a_{2j} - \ell_{21} u_{1j} = a_{2j} - \frac{a_{21}}{a_{11}} a_{1j}$, seconde colonne de L : $\ell_{i2} = \frac{a_{i2} - \ell_{i1} u_{12}}{u_{22}}$, ...

La décomposition LU est utilisée par la méthode `np.linalg.solve(A,B)`, qui retourne le vecteur solution. Elle est plus rapide que la méthode par inversion, mais ne s'applique de même qu'aux matrices A carrées et inversibles.
Syntaxe : **X = np.linalg.solve(A,B)**

Méthode des moindres carrés

La méthode des moindres carrés consiste à trouver le vecteur solution X qui **minimise la norme euclidienne**

$$\|B - A \cdot X\|^2$$

Cet algorithme est utilisé par `np.linalg.lstsq(A,B)`, et fonctionne **quelles que soient les propriétés de la matrice A.**

En effet, avec `np.linalg.lstsq(A,B)`, si la matrice est carrée et inversible ($\det(A) \neq 0$), la solution est exacte (on a bien $\|B - A \cdot X\| = 0$). Sinon, la norme $\|B - A \cdot X\|$ est minimale mais non nulle. Dans le cas où le nombre de solutions est infini, la solution avec la norme la plus faible est retournée.
Quel intérêt ? la rapidité pour certains systèmes à grand nombre d'équations
Syntaxe : **X = np.linalg.lstsq(A,B)**

Cas d'une matrice non inversible, rang d'une matrice

Le rang d'une matrice A est égal au rang de sa matrice transposée. Ce rang est égal au nombre de vecteurs colonnes de A qui sont indépendants. Il est aussi égal au nombre de vecteurs lignes de A qui sont indépendants. Si la matrice $(n \times n)$ A n'est pas inversible, alors l'une au moins des lignes de cette matrice peut s'écrire comme la combinaison linéaire des autres lignes de la matrice. Le rang de la matrice est donc strictement inférieur à n.

Cas d'une matrice non inversible, rang d'une matrice

Supposons que la ligne i de A s'écrive come combinaison linéaire des autres lignes de A.

- Si le coefficient b_i est égal à la même combinaison linéaire des autres b, alors l'équation i s'écrit comme combinaison linéaire des autres équations, et est automatiquement vérifiée si les autres équations le sont. On a donc en réalité un système à $n-1$ équations et n inconnues, qui admet une infinité de solutions.
- dans le cas contraire, alors l'équation i n'est pas vérifiée lorsque toute les autres le sont et le système n'admet aucune solution.

Pour distinguer ces deux cas, on recherche le rang de la matrice à n lignes et $n+1$ colonnes, formée de la matrice A à laquelle on a accolé le vecteur B comme $n+1$ème colonne .

- Si le rang de cette matrice est égal au rang de A (et donc strictement inférieur à n, alors le système admet une infinité de solutions.
- Si le rang de cette matrice est supérieur au rang de A, alors le système n'admet aucune solution.

Pour déterminer le rang d'une matrice : `r=np.linalg.matrix_rank(A)`