

TD 12: Numpy

1. Considérez les opérations élément par élément dans numpy :

Les opérations arithmétiques classiques (+, -, *, /, **) s'appliquent **élément par élément** entre tableaux de mêmes dimensions.

Exemples avec des scalaires :

- `np.array([1, 2, 3]) * 2` donne `[2, 4, 6]`
- `np.array([[1, 2], [3, 4]]) ** 2` donne $\begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$

Exemples entre tableaux de mêmes formes :

- `np.array([1, 2, 3]) + np.array([4, 5, 6])` donne `[5, 7, 9]`
- `np.array([[1, 2], [3, 4]]) - np.array([[1, 1], [1, 1]])` donne $\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$
- `np.array([1, 2, 3]) * np.array([4, 5, 6])` donne `[4, 10, 18]`

Écrivez une fonction `f` qui prend en entrée : une matrice `M` de forme (n, d) , un scalaire `s`, et une matrice `N` de même forme que `M`. Il retourne un tuple contenant :

- `M + s` (addition d'un scalaire)
- `M * s` (multiplication par un scalaire)
- `M ** 2` (carré élément par élément)
- `M + N` (addition de deux matrices)
- `M - N` (soustraction de deux matrices)
- `M * N` (multiplication élément par élément)

Contrainte : Utilisez uniquement les opérateurs arithmétiques (+, -, *, **). Aucune boucle.

Solution :

```
import numpy as np

def f(M, s, N):
    return (M + s, M * s, M ** 2, M + N, M - N, M * N)

M = np.array([[1, 2], [3, 4]])
N = np.array([[5, 6], [7, 8]])
s = 10

plus_scal, fois_scal, carre, plus_mat, moins_mat, produit = f(M, s, N)

print("M + s:\n", plus_scal)
print("M * s:\n", fois_scal)
print("M ** 2:\n", carre)
print("M + N:\n", plus_mat)
print("M - N:\n", moins_mat)
print("M * N:\n", produit)
```

2. Considérez les opérations matricielles suivantes avec numpy :

A.T retourne la transposée d'une matrice.
A @ B effectue la multiplication matricielle.
np.identity(n) crée la matrice identité $n \times n$.

Exemples :

• `A = np.array([[1,2],[3,4]])` → A.T donne $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

• `A = np.array([[1,2],[3,4]])` → `A @ A` donne $\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$

• `np.identity(3)` donne $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Soient deux matrices A et B (carrées de même taille). Calculez les expressions suivantes :

- A^T (transposée de A)
- $A + A$ (somme de A avec elle-même)
- AA^T (produit de A par sa transposée)
- $A^T A$ (produit de la transposée par A)
- AB (produit de A par B)
- Écrivez une fonction `f` qui prend A , B et un scalaire λ , et retourne $A(B - \lambda I)$ où I est la matrice identité de même taille que B .

Solution :

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("A^T:\n", A.T, "\n")
print("A+A:\n", A + A, "\n")
print("AA^T:\n", A @ A.T, "\n")
print("A^TA:\n", A.T @ A, "\n")
print("AB:\n", A @ B, "\n")

def f(A, B, mylambda):
    I = np.identity(B.shape[0])
    return A @ (B - mylambda * I)

print("A(B - lambda I) avec lambda =2:\n", f(A, B, 2))
```

3. Considérez les fonctions numpy suivantes :

`np.dot(a, b)` : produit scalaire (dot product).

`np.linalg.norm(x)` : norme euclidienne (longueur) du vecteur $\|x\| = \sqrt{\sum_i x_i^2}$.

Exemples :

- `np.dot([1, 2], [3, 4])` donne $1 \times 3 + 2 \times 4 = 11$
- `np.dot([1, 0, 0], [0, 1, 0])` donne 0 (vecteurs orthogonaux)
- `np.linalg.norm([3, 4])` donne $\sqrt{3^2 + 4^2} = 5.0$
- `np.linalg.norm([1, 1, 1])` donne $\sqrt{1^2 + 1^2 + 1^2} \approx 1.732$

La similarité cosinus entre deux vecteurs u et v mesure le cosinus de l'angle qui les sépare :

$$\text{cosine_sim}(u, v) = \frac{u \cdot v}{\|u\| \cdot \|v\|}$$

Elle retourne une valeur entre -1 (vecteurs opposés) et 1 (vecteurs identiques). Pour des vecteurs non nuls, 0 signifie orthogonalité.

Écrivez une fonction `cosinus_sim` qui prend en entrée deux vecteurs u et v (tableaux 1D de numpy de même longueur) et retourne leur similarité cosinus (un flottant). La fonction doit fonctionner même si l'un des vecteurs est nul (retournez 0 dans ce cas).

Contrainte : Utilisez `np.dot` et `np.linalg.norm`. Aucune boucle explicite.

Solution :

```
import numpy as np

def cosinus_sim(u, v):
    produit = np.dot(u, v)
    norm_u = np.linalg.norm(u)
    norm_v = np.linalg.norm(v)

    if norm_u == 0 or norm_v == 0:
        return 0.0
    return produit / (norm_u * norm_v)

u1 = np.array([1, 2, 3])
v1 = np.array([4, 5, 6])
print(f"cos({u1}, {v1}) = {cosinus_sim(u1, v1):.4f}")

u4 = np.array([1, 1])
v4 = np.array([-1, -1])
print(f"cos({u4}, {v4}) = {cosinus_sim(u4, v4):.4f}")

u5 = np.array([0, 0, 0])
v5 = np.array([1, 2, 3])
print(f"cos(zero, v) = {cosinus_sim(u5, v5)}")
```

4. Considérez la fonction numpy suivante :

`np.cross(a, b)` calcule le produit vectoriel de deux vecteurs en 3D.

Pour deux vecteurs $\vec{a} = (a_x, a_y, a_z)$ et $\vec{b} = (b_x, b_y, b_z)$, le produit vectoriel est :

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

Le résultat est un vecteur orthogonal à \vec{a} et \vec{b} .

Exemples :

- `np.cross([1,0,0], [0,1,0])` donne `[0,0,1]`
- `np.cross([0,0,1], [0,1,0])` donne `[-1,0,0]`
- `np.cross([[1,0,0], [0,1,0]], [[0,1,0], [1,0,0]])` donne produit vectoriel par ligne

La **normale** à un triangle 3D défini par trois points p_1, p_2, p_3 est un vecteur orthogonal au plan du triangle. Elle se calcule par :

$$\vec{n} = (\overrightarrow{p_2 p_1}) \times (\overrightarrow{p_3 p_1})$$

puis on la normalise (division par sa norme).

Écrivez une fonction `normale_triangle` qui prend trois points p_1, p_2, p_3 (chacun est un tableau 1D de 3 coordonnées) et retourne la normale normalisée (vecteur unitaire orthogonal).

Contrainte : Un seul appel à `np.cross` et un seul appel à `np.linalg.norm`. Aucune boucle.

Solution :

```
import numpy as np

def normale_triangle(p1, p2, p3):
    u = p2 - p1
    v = p3 - p1
    n = np.cross(u, v)
    return n / np.linalg.norm(n)

p1 = np.array([0, 0, 0])
p2 = np.array([1, 0, 0])
p3 = np.array([0, 1, 0])
print("Normale :", normale_triangle(p1, p2, p3))

p1 = np.array([1, 0, 0])
p2 = np.array([0, 1, 0])
p3 = np.array([0, 0, 1])
print("Normale :", normale_triangle(p1, p2, p3))
```

5. Considérez la fonction numpy suivante :

`np.sum(a, axis=None)` calcule la somme des éléments. Le paramètre `axis` permet de choisir la dimension le long de laquelle on somme.

Exemples :

- `np.sum([[1,2],[3,4]])` donne 10 (somme de tous les éléments)
- `np.sum([[1,2],[3,4]], axis=0)` donne [4,6] (somme par colonne)
- `np.sum([[1,2],[3,4]], axis=1)` donne [3,7] (somme par ligne)

Soit une matrice A de forme (k, d) . Calculez un vecteur normes de forme $(k,)$ où chaque élément $normes[j] = \sum_{t=1}^d A[j, t]^2$.

Contrainte : Un seul appel à `np.sum` avec `axis`.

Solution :

```
import numpy as np

def normes_carrees(A):
    return np.sum(A**2, axis=1)

A = np.array([[1, 2], [3, 4], [5, 6]])
print(normes_carrees(A))
```

6. Considérez le broadcasting :

Le broadcasting permet d'effectuer des opérations entre tableaux de formes différentes. Quand on soustrait un vecteur d'une matrice, le vecteur est automatiquement répété pour correspondre à la forme de la matrice.

Exemples :

- `np.array([1,2,3]) - 1` donne `[0,1,2]` (scalaire répété)
- `np.array([[1,2],[3,4]]) - np.array([1,2])` donne $\begin{bmatrix} 0 & 0 \\ 2 & 2 \end{bmatrix}$
- `np.array([[1,2],[3,4]]) - np.array([1,1,1])` une erreur (formes incompatibles)

Soit une matrice A de forme (k, d) et un vecteur x de forme $(d,)$. Calculez la matrice `diff` de forme (k, d) où chaque ligne j contient $x - A[j]$.

Ensuite, calculez le vecteur `distances` de forme $(k,)$ où $\text{distances}[j] = \sum_{t=1}^d \text{diff}[j,t]^2$.

Contrainte : Utiliser le broadcasting et `np.sum`.

Solution :

```
import numpy as np

def distances_point_centres(x, A):
    diff = x - A    #<-- broadcasting ici!
    return np.sum(diff**2, axis=1)

A = np.array([[0, 0], [1, 1], [2, 2]])
x = np.array([1, 1])
print( distances_point_centres(x, A) )
```

7. Considérez la fonction numpy suivante :

`np.where(condition)` retourne les indices où la condition est vraie.

Paramètres :

- `condition` : tableau booléen ou expression qui en produit un.
- La valeur de retour est un tuple contenant les indices des éléments non nuls.
- Pour un tableau 1D, `np.where(condition)[0]` donne un tableau 1D d'indices.

Exemples :

- `np.where([True, False, True])` donne `(array([0, 2]),)`
- `np.where([1,2,3,4] > 2)` donne `(array([2,3]),)`
- `np.where(np.abs([-2,-1,0,1,2]) >= 2)` donne `(array([0,4]),)`
- `indices = np.where(np.array([0,5,0,3,0]) != 0)[0]` donne `[1,3]`

Écrivez une fonction `g` qui prend en entrée `A`, un tableau 1D de numpy (type `np.array`) et `L`, un entier positif (le seuil). Il retourne :

- le **premier indice** `t` tel que $|A[t]| \geq L$ (c'est-à-dire la position où la valeur absolue atteint ou dépasse le seuil), ou
- `None` si aucun indice ne satisfait la condition.

Contrainte : Un seul appel à `np.where`. Aucune boucle explicite.

Solution :

```
import numpy as np

def g(A, L):
    indices = np.where(np.abs(A) >= L)[0]

    return indices[0] if indices.size > 0 else None

print("Exemple 1:", g(np.array([0, 1, 2, 3, 4, 5]), 3))
print("Exemple 2:", g(np.array([0, 1, 2, 1, 0, -1]), 3))
print("Exemple 3:", g(np.array([0, -1, -2, -3, -4]), 3))
print("Exemple 4:", g(np.array([5, 4, 3, 2, 1]), 3))
print("Exemple 5:", g(np.array([1, -3, 2, 4]), 3))
```

8. Considérez la fonction numpy suivante :

```
np.argmin(a, axis=None): retourne l'indice du minimum.
```

Exemples :

- `np.argmin([3,1,4,1,5])` donne 1 (première occurrence du minimum)
- `np.argmin([[2,1],[4,3]], axis=0)` donne [0,0] (colonne par colonne)
- `np.argmin([[2,1],[4,3]], axis=1)` donne [1,1] (ligne par ligne)

Écrivez une fonction `findClosest` qui prend une valeur z et un tableau A (1D) de type `np.array`, et retourne l'élément de A le plus proche de z .

Contrainte : Utilisez `np.argmin`. Aucune boucle explicite. Une seule ligne de code.

Solution :

```
import numpy as np

def findClosest(A, z):
    return A[np.argmin(np.abs(z - A))]

print( findClosest( np.array([2,1,5,1,22]), 1) )
```

9. Considérez les fonctions d'agrégation numpy suivantes :

```
np.max(a, axis=None) : valeur maximale.  
np.any(a, axis=None) : True si au moins un élément est vrai.
```

Exemples :

- `np.max([[1,5],[3,2]], axis=1)` donne `[5,3]` (ligne par ligne)
- `np.any([[False,True],[False,False]], axis=1)` donne `[True,False]` (ligne par ligne)

Soit une matrice A de forme (N, M) contenant des booléens (True ou False). Pour chaque ligne, calculez:

- La valeur maximale (convertir True \rightarrow 1, False \rightarrow 0)
- S'il existe au moins un True dans la ligne

Contrainte : Utilisez `np.max`, `np.any`.

Solution :

```
import numpy as np  
  
def analyse(A):  
    return np.max(A, axis=1), np.any(A, axis=1)  
  
A = np.array([[False, True, False],  
              [False, False, False],  
              [True, False, True]])  
  
print(analyse(A))
```

10. Considérez la fonction numpy suivante :

```
np.roll(a, shift, axis=None): décale circulairement.
```

Exemples :

- `np.roll([1,2,3,4], 2)` donne `[3,4,1,2]`
- `np.roll([1,2,3,4], -1)` donne `[2,3,4,1]`
- `np.roll([[1,2],[3,4]], 1, axis=0)` donne $\begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix}$ (colonne par colonne)
- `np.roll([[1,2],[3,4]], 1, axis=1)` donne $\begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$ (ligne par ligne)

Écrivez une fonction `auto_corr_circ` qui prend un vecteur `a` de type `np.array` avec `n = len(a)`, et retourne l'autocorrélation circulaire $c[j] = \sum_i a[i] \cdot a[(i+j) \bmod n]$.

Contrainte : Utilisez n appels à `np.roll`.

Solution :

```
import numpy as np

def auto_corr_circ(a):
    n = len(a)
    c = np.zeros(n)
    for j in range(n):
        c[j] = np.dot(a, np.roll(a, j))
    return c

print( auto_corr_circ(np.array([1,2,3])) )
```

11. Considérez la fonction numpy suivante :

`np.mean(a, axis=None)` calcule la moyenne. Le paramètre `axis` permet de choisir la dimension le long de laquelle la moyenne est calculée.

Exemples :

- `np.mean([[1,2],[3,4]])` donne 2.5 (moyenne de tous les éléments)
- `np.mean([[1,2],[3,4]], axis=0)` donne [2,3] (moyenne par colonne)
- `np.mean([[1,2],[3,4]], axis=1)` donne [1.5, 3.5] (moyenne par ligne)

Soit une matrice `points` de forme (m, d) contenant m points dans \mathbb{R}^d . Calculez le centre (moyenne) de ces points. Le résultat doit être un vecteur de forme $(d,)$.

Contrainte : Un seul appel à `np.mean` avec le paramètre `axis` approprié.

Solution :

```
import numpy as np

def centre_points(points):
    return np.mean(points, axis=0)

points = np.array([[1, 2], [3, 4], [5, 6]])
print(centre_points(points))
```

12. Considérez la fonction numpy suivante :

`np.allclose(a, b, rtol=1e-05, atol=1e-08)` vérifie si deux tableaux sont égaux à une tolérance près.

Paramètres :

- Deux valeurs a et b sont considérées proches si $|a - b| \leq \text{atol} + \text{rtol} \cdot |b|$.
- Avec $\text{rtol}=0$, la condition devient simplement $|a - b| \leq \text{atol}$.

Exemples :

- `np.allclose([1.0, 2.0], [1.0001, 2.0001])` donne True (car $10^{-4} \leq \text{tolérance}$)
- `np.allclose([1.0, 2.0], [1.1, 2.0])` donne False
- `np.allclose([0, 0], [1e-9, 0], atol=1e-8)` donne True
- `np.allclose([1000,2000], [1001,2000], rtol=1e-3)` donne True ($1 \leq 1000 \times 10^{-3} = 1$)

Écrivez une fonction `proche` qui prend deux tableaux a et b de même forme, ainsi qu'une tolérance absolue eps , et retourne True si tous les éléments de a et b diffèrent en valeur absolue de moins de eps , c'est-à-dire $|a_i - b_i| < \text{eps}$ pour tout i . Utilisez `np.allclose` avec $\text{rtol}=0$.

Contrainte : Un seul appel à `np.allclose`.

Solution :

```
import numpy as np

def proche(a, b, eps=1e-4):
    return np.allclose(a, b, atol=eps, rtol=0)

a = np.array([1.00001, 2.00001, 3.00001])
b = np.array([1.0, 2.0, 3.0])
print( proche(a, b, eps=1e-4) )
print( proche(a, b, eps=1e-6) )
```

13. Considérez la fonction numpy suivante :

`np.polyval(p, x)` évalue un polynôme (coefficients par degré décroissant).

Exemples :

- `np.polyval([1,0,0], [2])` donne 4 (x^2 en $x = 2$)
- `np.polyval([1,-2,1], [0,1,2])` donne `[1,0,1]` pour $(x - 1)^2$

Évaluez $P(x) = 3x^3 - 2x^2 + 5$ aux points `[0, 1, 2, 3]`.

Contrainte : Un seul appel à `np.polyval`.

Solution :

```
import numpy as np

def evalue_polynome(x):
    coeffs = [3, -2, 0, 5]
    return np.polyval(coeffs, x)

print(evalue_polynome(np.array([1,2,3])))
```

14. Considérez la fonction numpy suivante :

`np.roots(p)` calcule les racines d'un polynôme.

Exemples :

- `np.roots([1,-3,2])` donne `[2.,1.]`
- `np.roots([1,0,-1])` donne `[-1.,1.]`

Trouvez les racines de $x^4 - 5x^2 + 4 = 0$.

Contrainte : Un seul appel à `np.roots`.

Solution :

```
import numpy as np

def racines():
    return np.roots([1, 0, -5, 0, 4])

print(racines())
```

15. Considérez la fonction numpy suivante :

`np.searchsorted(a, v, side='left')` trouve l'index d'insertion pour maintenir l'ordre dans un tableau trié `a`.

Paramètres :

- `a` : tableau 1D trié (dans l'ordre croissant)
- `v` : valeur(s) à insérer (scalaire ou tableau)
- `side='left'` : retourne l'indice de la première position où `v` peut être inséré sans violer l'ordre. Si la valeur existe déjà, l'insertion se fait **avant** l'élément existant.
- `side='right'` : retourne l'indice de la dernière position où `v` peut être inséré. Si la valeur existe déjà, l'insertion se fait **après** l'élément existant.

Exemples :

- `np.searchsorted([1,3,5], 4)` donne 2 (car $3 < 4 < 5$)
- `np.searchsorted([1,3,5], [0,3,6])` donne `[0,1,3]`
- `np.searchsorted([1,3,3,5], 3, side='left')` donne 1 (insertion avant le premier 3)
- `np.searchsorted([1,3,3,5], 3, side='right')` donne 3 (insertion après le dernier 3)

Complexité : $O(\log n)$ par élément (recherche binaire).

Écrivez une fonction `insertion_triee` qui prend en entrée un tableau trié `arr` (1D) et une valeur `x`, et retourne un nouveau tableau où `x` est inséré à la position correcte pour maintenir l'ordre croissant.

Si `x` existe déjà dans `arr`, insérez-le **après** les éléments existants.

Contrainte : Utilisez `np.searchsorted` et `np.insert`.

Solution :

```
import numpy as np

def insertion_triee(arr, x):
    idx = np.searchsorted(arr, x, side='right')
    return np.insert(arr, idx, x)

# Exemples
print(insertion_triee(np.array([1, 3, 5]), 4))      # [1, 3, 4, 5]
print(insertion_triee(np.array([1, 3, 3, 5]), 3))  # [1, 3, 3, 3, 5]
print(insertion_triee(np.array([1, 2, 3]), 0))    # [0, 1, 2, 3]
print(insertion_triee(np.array([1, 2, 3]), 4))    # [1, 2, 3, 4]
```

16. Considérez la fonction numpy suivante :

```
np.unique(ar, return_counts=False) retourne les éléments uniques d'un tableau.
```

Paramètres :

- ar : tableau d'entrée
- return_counts : si True, retourne aussi les fréquences de chaque valeur unique

Exemples :

- np.unique([1, 2, 1, 3, 2, 1]) donne [1, 2, 3]
- np.unique([1, 2, 1, 3], return_counts=True) donne (array([1,2,3]), array([2,1,1]))
- valeurs, counts = np.unique(['a', 'b', 'a', 'c', 'b', 'a'], return_counts=True) donne ['a','b','c'] et [3,2,1]

Le **mode** d'un tableau est la valeur qui apparaît le plus fréquemment. Par exemple, dans le tableau [1, 2, 1, 3, 1, 2], le mode est 1 (qui apparaît 3 fois).

Écrivez une fonction **mode** qui prend en entrée un tableau 1D arr (type np.array) et retourne la valeur la plus fréquente.

Contrainte : Aucune boucle.

Solution :

```
import numpy as np

def mode(arr):
    valeurs, counts = np.unique(arr, return_counts=True)
    return valeurs[np.argmax(counts)]

print(mode(np.array([1, 2, 1, 3, 1, 2])))
print(mode(np.array([1, 1, 2, 2, 3])))
print(mode(np.array([5, 5, 5, 2, 2, 2, 3])))
print(mode(np.array(['a', 'b', 'a', 'c', 'a', 'b'])))
```

17. Considérez la fonction numpy suivante :

```
np.random.choice(a, size=None, replace=True, p=None) tire aléatoirement des échantillons.
```

Paramètres :

- `a` : tableau 1D ou entier. Si entier, il est interprété comme `np.arange(a)`.
- `size` : forme du tableau de sortie (entier ou tuple). Si `None`, retourne un scalaire.
- `replace` : `True` pour tirage avec remise, `False` pour tirage sans remise.
- `p` : probabilités associées à chaque élément (optionnel, par défaut uniforme).

Exemples :

- `np.random.choice([-1, 1], size=5)` donne un vecteur 1D de 5 valeurs -1 ou $+1$
- `np.random.choice([-1, 1], size=(3,4))` donne une matrice 3×4 de valeurs -1 ou $+1$
- `np.random.choice(10, size=(2,2))` donne une matrice 2×2 d'entiers entre 0 et 9
- `np.random.choice(100, 10, replace=False)` donne 10 entiers distincts entre 0 et 99

Écrivez une fonction `random_operations` qui :

- Génère une matrice A de taille (N, M) (avec $N = 100$, $M = 50$) contenant des valeurs -1 ou $+1$ avec probabilité $1/2$.
- Calcule la somme des éléments de chaque ligne de A (vecteur de taille N).
- Choisit aléatoirement $k = 5$ indices distincts parmi les N lignes (sans remise).
- Retourne ces trois éléments : la matrice A , le vecteur des sommes, et le tableau des indices sélectionnés.

Contrainte : Utilisez `np.random.choice` deux fois (une fois avec une liste, une fois avec un entier), `np.sum` avec `axis=1`, et `replace=False` pour la sélection sans remise. Aucune boucle.

Solution :

```
import numpy as np

def random_operations(N=100, M=50, k=5):
    A = np.random.choice([-1, 1], size=(N, M))

    sommes = np.sum(A, axis=1)

    indices = np.random.choice(N, k, replace=False)

    return A, sommes, indices

A, sommes, indices = random_operations(10, 5, 3)
print("Matrice A:\n", A)
print("Sommes par ligne:", sommes)
print("Indices selectionnes:", indices)
```

18. Considérez la fonction numpy suivante :

`np.cumsum(a, axis=None)` calcule la somme cumulée (prefix sum) des éléments.

Paramètres :

- `a` : tableau d'entrée
- `axis` : axe le long duquel on cumule. Si `None`, le tableau est aplati.

Exemples :

- `np.cumsum([1, 2, 3, 4])` donne `[1, 3, 6, 10]`
- `np.cumsum([[1,2],[3,4]], axis=0)` donne $\begin{bmatrix} 1 & 2 \\ 4 & 6 \end{bmatrix}$ (cumul par colonne)
- `np.cumsum([[1,2],[3,4]], axis=1)` donne $\begin{bmatrix} 1 & 3 \\ 3 & 7 \end{bmatrix}$ (cumul par ligne)

Soit une matrice A de taille (N, M) où chaque ligne contient des valeurs -1 ou $+1$. Écrivez une fonction `f` qui prend A et retourne un tuple contenant :

- Un vecteur `positions_finales` de taille N , où chaque élément est la position finale de la ligne correspondante (dernière colonne).
- Un vecteur `max_abs` de taille N , où chaque élément est la valeur absolue maximale atteinte par la ligne correspondante (le pic maximum de la marche).

Contrainte : Utilisez `np.cumsum` et `np.max` avec `axis=1` pour le maximum. Aucune boucle.

Solution :

```
import numpy as np

def f(A):
    positions = np.cumsum(A, axis=1)

    positions_finales = positions[:, -1]

    max_abs = np.max(np.abs(positions), axis=1)

    return positions_finales, max_abs

A1 = np.array([[ 1, 1, 1, 1], [-1, 1, -1, 1]])
finales, max_abs = f(A1)
print("A1:\n", A1)
print("Positions finales:", finales)
print("Max valeur absolue:", max_abs)

np.random.seed(42)
A2 = np.random.choice([-1, 1], size=(5, 10))
finales, max_abs = f(A2)
print("A2:\n", A2)
print("Positions finales:", finales)
print("Max valeur absolue:", max_abs)
```

19. Considérez la multiplication de matrices par blocs (optimisation cache) :

La multiplication matricielle naïve $C = A \times B$ (où A et B sont des matrices de taille $n \times n$) avec trois boucles imbriquées accède à la mémoire de manière non localisée. En découpant les matrices en petits blocs, on améliore la localité des données.

Nouvel algorithme :

- On choisit une taille de bloc B_s .
- On réécrit la multiplication comme une triple boucle sur les blocs :

$$C[i:i+B_s, j:j+B_s] += A[i:i+B_s, k:k+B_s] @ B[k:k+B_s, j:j+B_s],$$

où les indices i, j, k parcourent les positions de début de bloc $(0, B_s, 2B_s, \dots)$.

- À l'intérieur, on utilise la multiplication matricielle standard ($@$) sur des blocs de taille au plus $B_s \times B_s$.

Notez que la taille n n'est pas forcément un multiple de B_s ; donc on utilise $\min(i+B_s, n)$ pour ne pas dépasser.

Complexité : $O(n^3)$ (toujours), mais le facteur constant est bien meilleur.

Implémentez la multiplication de matrices $C = A \times B$ par blocs de taille $B_s = 64$ pour n quelconque. Utilisez des vues par indexation et l'opérateur $@$ pour les blocs. La fonction doit retourner la matrice produit.

Contrainte : Utilisez $A[i:i+bs, k:k+bs]$, l'opérateur $@$ pour la multiplication des blocs, et $np.zeros$ pour initialiser C . La fonction doit fonctionner pour toute taille n (pas nécessairement multiple de B_s).

Solution :

```
import numpy as np

def mult_blocs(A, B, bs=64):
    n = A.shape[0]
    C = np.zeros((n, n))
    for i in range(0, n, bs):
        for j in range(0, n, bs):
            for k in range(0, n, bs):
                i_end = min(i+bs, n)
                j_end = min(j+bs, n)
                k_end = min(k+bs, n)
                C[i:i_end, j:j_end] += A[i:i_end, k:k_end] @ B[k:k_end, j:j_end]
    return C

A = np.random.random( (3,3) )
B = np.random.random( (3,3) )
print(A@B)
print(mult_blocs(A,B))
```

20. Considérez la simulation d'une marche aléatoire avec barrières absorbantes :

On simule un grand nombre de marches aléatoires pour estimer le temps moyen avant qu'elles ne touchent une barrière. Une marche aléatoire 1D part de 0. À chaque pas, on ajoute -1 ou $+1$ avec probabilité $1/2$. Les barrières sont en $-L$ et $+L$: la marche s'arrête dès qu'elle atteint ou dépasse ces bornes.

Simulation :

- Générer une matrice A de taille $(N, \text{pas_max})$ avec des valeurs -1 ou $+1$ aléatoires.
- Calculer les positions cumulées avec `np.cumsum`.
- Pour chaque simulation i , trouver le premier indice t où $\text{pos}[i][t] \leq -L$ ou $\text{pos}[i][t] \geq L$.
- Le temps d'absorption est $t + 1$ (car t est 0-indexé). Si aucune absorption n'est détectée, on assigne `pas_max`.
- Retourner la moyenne des temps.

Écrivez une fonction `randomwalk` qui prend en entrée :

- `L` (int) : la distance des barrières (par défaut $L = 10$)
- `n_sim` (int) : le nombre de simulations (par défaut $n_sim = 10^4$)
- `max_steps` (int) : le nombre maximal de pas par simulation (par défaut $max_steps = 10^5$)

et retourne un float qui est le temps moyen d'absorption estimé.

Contrainte : Utilisez `np.random.choice` pour générer A , `np.cumsum` pour les positions cumulées.

Solution :

```
import numpy as np

def randomwalk(L=10, n_sim=10000, max_steps=100000):
    A = np.random.choice([-1, 1], size=(n_sim, max_steps))
    positions = np.cumsum(A, axis=1)

    temps = []
    for i in range(n_sim):
        indices = np.where(np.abs(positions[i]) == L)[0]

        if len(indices) > 0:
            temps.append(indices[0] + 1)
        else:
            temps.append(max_steps)

    return sum(temps) / len(temps)

print( randomwalk (3, 10, 12) )
```

21. Considérez l'algorithme k-means (clustering) :

Le k-means partitionne n points dans \mathbb{R}^d en k clusters. On alterne entre deux étapes : assigner chaque point au centre le plus proche, puis déplacer chaque centre à la moyenne de ses points assignés.

Soit X une matrice $n \times d$ (chaque ligne représente un point). On initialise les centres C (taille $k \times d$) en choisissant k points aléatoires parmi X sans remise.

Étape 1 : Assignment. Pour chaque point i et chaque centre j , on calcule la distance euclidienne au carré :

$$\text{distances}[i, j] = \sum_{t=1}^d (X[i, t] - C[j, t])^2$$

Puis on assigne chaque point au centre avec la distance minimale :

$$\text{labels}[i] = \arg \min_j \text{distances}[i, j]$$

Étape 2 : Mise à jour des centres. Pour chaque cluster j , on recalcule le centre comme la moyenne des points $X[i]$ tels que $\text{labels}[i] = j$. Si un cluster est vide, on garde l'ancien centre.

On s'arrête quand tous les centres changent de moins de $\varepsilon = 10^{-4}$ (en utilisant `np.allclose`) ou quand on atteint `max_iter`.

Écrivez une fonction

```
def kmeans(X: np.ndarray, k: int, max_iter: int = 100)
```

Entrées :

- X : matrice numpy de forme (n, d) contenant n points dans \mathbb{R}^d
- k : nombre de clusters (entier, $1 \leq k \leq n$)
- `max_iter` : nombre maximal d'itérations (entier, par défaut 100)

Sorties :

- `labels` : vecteur numpy de forme $(n,)$ contenant pour chaque point l'indice du cluster assigné (entiers de 0 à $k - 1$)
- `centres` : matrice numpy de forme (k, d) contenant les centres finaux des clusters

Contrainte : Utilisez `np.random.choice` pour l'initialisation, une double boucle `for` (sur i et j) pour calculer les distances, `np.argmin` pour l'assignation, `np.where` pour trouver les indices par cluster, et `np.mean` pour calculer les centres.

Solution : J'ajoute une visualisation, afin que vous puissiez exécuter le code et voir le résultat.

```
import numpy as np

def kmeans(X, k, max_iter=100):
    n, d = X.shape

    idx = np.random.choice(n, k, replace=False)
    centres = X[idx].copy()

    for iteration in range(max_iter):
        labels = np.zeros(n)
        for i in range(n):
            distances = np.zeros(k)
            for j in range(k):
                distances[j] = np.sum((X[i] - centres[j]) ** 2)
            labels[i] = np.argmin(distances)

        nouveaux_centres = np.zeros((k, d))
        for j in range(k):
            indices_j = np.where(labels == j)[0]
            if len(indices_j) > 0:
                nouveaux_centres[j] = np.mean(X[indices_j], axis=0)
            else:
                nouveaux_centres[j] = centres[j]

        if np.allclose(centres, nouveaux_centres, atol=1e-4):
            print(f"Convergence apres {iteration + 1} iterations")
            break
        centres = nouveaux_centres

    return labels, centres

# =====
# EXEMPLE 1 : (4 points, 2 clusters)
# =====
print("=" * 50)
print("EXEMPLE 1 : 4 points, 2 clusters")
print("=" * 50)

X1 = np.array([
    [1, 1], # point 0
    [1, 2], # point 1
    [8, 8], # point 2
    [9, 8] # point 3
])

print("Points:")
print(X1)
print()

labels1, centres1 = kmeans(X1, k=2, max_iter=10)
print(f"\nLabels: {labels1}")
print(f"Centre cluster 0: {centres1[0]}")
print(f"Centre cluster 1: {centres1[1]}")

# =====
# EXEMPLE 2 : (2 clusters naturels)
# =====
print("\n" + "=" * 50)
print("EXEMPLE 3 : Points sur un cercle (2 cotes)")
print("=" * 50)

np.random.seed(42)
theta1 = np.random.uniform(0, np.pi, 30)
theta2 = np.random.uniform(np.pi, 2*np.pi, 30)
r = 5

X3_haut = np.array([r * np.cos(theta1), r * np.sin(theta1)]).T
X3_bas = np.array([r * np.cos(theta2), r * np.sin(theta2)]).T
X3 = np.vstack([X3_haut, X3_bas])

labels3, centres3 = kmeans(X3, k=2, max_iter=20)
print(f"\nRepartition:")
print(f" Cluster 0: {np.sum(labels3 == 0)} points")
print(f" Cluster 1: {np.sum(labels3 == 1)} points")
print(f"\nCentres:")
print(f" Centre 0: ({centres3[0][0]:.2f}, {centres3[0][1]:.2f})")
print(f" Centre 1: ({centres3[1][0]:.2f}, {centres3[1][1]:.2f})")

# =====
# Fonction pour visualiser
# =====
def plot_clusters(X, labels, centres, title="k-means Clustering"):
    import matplotlib.pyplot as plt

    colors = ['red', 'blue', 'green', 'orange', 'purple', 'brown', 'pink', 'gray']

    plt.figure(figsize=(8, 6))

    for j in range(centres.shape[0]):
        mask = labels == j
        plt.scatter(X[mask, 0], X[mask, 1],
                   c=colors[j % len(colors)],
                   label=f'Cluster {j}',
                   alpha=0.6, s=50)

    plt.scatter(centres[:, 0], centres[:, 1],
               c='black', marker='X', s=200,
               edgecolors='white', linewidth=2,
               label='Centres')

    plt.title(title)
    plt.legend()
    plt.axis('equal')
    plt.show()

plot_clusters(X3, labels3, centres3, "k-means sur deux demi-cercles")
```