

Python pour les statistiques

Fiche 2 : Numpy et Matplotlib

M2 Modélisation statistique, 2023–2024

Table des matières

1	Ce qu’il faut savoir	1
1.1	Créer des arrays, préciser leur type	2
1.2	Manipuler des arrays	3
1.2.1	Opérations vectorielles	3
1.2.2	Fonctions pratiques : somme, moyenne, etc.	4
1.2.3	Algèbre linéaire	5
1.3	Graphiques	5
1.3.1	Basiques	5
1.3.2	Avancés	6
1.4	Indexation	8
1.4.1	Slicing	8
1.4.2	Indexation par liste	9
1.4.3	Indexation par masques booléens	10
1.5	Manipuler l’aléatoire avec Numpy	10
1.6	Technique avancée : le broadcasting	11
2	Exercices	12

1 Ce qu’il faut savoir

Numpy est (selon la page d’accueil de sa [documentation](#)) « la librairie fondamentale du calcul scientifique en Python ». La raison en est que Numpy introduit les `ndarray`, des *structures homogènes*, c’est-à-dire des tableaux contenant des variables d’un même type numérique (pour faire simple : `bool`, `int`, `float`, ou `complex`). Ces `ndarray` peuvent être multi-dimensionnels (on se restreindra généralement aux dimensions 1 et 2, les objets seront donc à interpréter comme des *vecteurs* ou des *matrices*). Cette librairie ajoute à Python la possibilité de faire des opérations de manière vectorielle.

Exemple :

```
import numpy as np # Toujours importer Numpy de cette manière.
a = np.array([1, 2])
b = np.array([3, 4])
print(a + b) # [4 6]
print(a * b) # [3, 8]
print(a / b) # [0.33 0.5]
```

La librairie Matplotlib, elle, est la librairie fondamentale qui permet de tracer des graphiques en python. Elle bénéficie d’une syntaxe “assez” intuitive d’utilisation (au moins pour des graphiques simple), et d’une [documentation très complète](#) (même si pas forcément facile d’accès).

Elle est parfaitement compatible avec les objets Numpy (voir Figure 1 pour un exemple simple). À

noter : c'est en fait le sous-module `pyplot` que l'on utilise en général, d'où l'importation usuelle via la commande `import matplotlib.pyplot as plt` (et donc, son utilisation avec le préfixe `plt`).

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2.5, 200)
y = np.exp(x) * np.sin(6*x)
plt.plot(x, y)
plt.show()
```

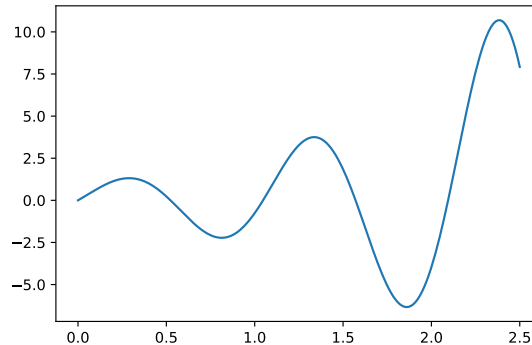


FIGURE 1 – Un code simple et le graphique produit.

Attention à la recherche dans la documentation : pour avoir les détails de la fonction `plt.plot`, il faut en fait (après avoir recherché le mot-clé “plot”) aller cliquer sur le lien correspondant à `matplotlib.pyplot.plot`.

1.1 Créer des arrays, préciser leur type

Comme dit ci-dessus, pour utiliser la librairie Numpy dans un script Python (ou dans une console Python) on l'importe avec la commande `import numpy as np`. Cela fait que tout ce qui est défini dans la librairie Numpy (ex : la fonction `dot`, la classe `ndarray`, le sous-module `random`) peut être utilisé dans votre code, simplement en **préfixant son nom par `np`** (ex : la fonction `np.dot`, la classe `np.ndarray`, le sous-module `np.random`). Dans tout ce qui suit, si l'on parle d'une fonction

Création d'un array avec la fonction `np.array`, à partir de n'importe quel objet indexable (`list`, `tuple`, etc.) ou doublement indexable (listes de listes – attention alors à ce que les longueurs des sous-listes soient les bonnes).

```
v = np.array([1, 2, 3]) # Crée un vecteur de longueur 3.
M = np.array([[1, 2], [0.5, -0.5]]) # Crée une matrice de dimensions (2, 2).
```

Importance des types. Attention : les types des données sont fixés à la création de l'objet. On peut récupérer le type d'un `ndarray` en regardant son attribut `dtype`. Concrètement ici on aurait :

```
print(v.dtype) # int64
print(M.dtype) # float64
```

Si rien n'est précisé, Numpy *devine* et associe un type selon sa règle interne. Pour les types qui nous intéressent, on a une hiérarchie `bool < int < float < complex`, au sens où les objets d'un type A peuvent être interprétés de manière canonique comme des objets d'un type B si `A < B`, mais pas inversement. Ainsi un `ndarray`, lors de sa création, prend automatiquement le “plus petit” type qui permet d'englober toutes ses entrées.

Pourquoi c'est important? Parce que l'on peut avoir de mauvaises surprises si l'on en est pas conscient. Car les `ndarray` sont mutables, au sens où leurs entrées peuvent changer, **mais leur type**

ne changera pas, et si l'on essaye de faire rentrer un “grand” type dans une “petite” case, voilà ce qui arrive :

```
x = np.array([1, 2])
x[0] = 0.5
print(x)  # [0 2] et non [0.5 2]
```

Pour résoudre ce souci, si l'on sait que nos données seront d'un certain type, par exemple des `float`, on le précise lors de la création : `x = np.array([1, 2], dtype=float)`.

Créer des arrays de tailles / formes fixées.

La **taille** (nombre d'entrées) d'un array se récupère par l'attribut `size`, sa **forme** (taille de chaque dimension) par l'attribut `shape`. En pratique, avec les arrays `v = np.array([1, 2, 3])` et `M = np.array([[1, 2], [0.5, -0.5]])` :

```
print(v.size)  # 3
print(v.shape) # (3,)
print(M.size)  # 4
print(M.shape) # (2, 2)
```

On peut créer de manière automatique plusieurs arrays : voici les fonctions principales. De manière générale, on peut préciser **une taille ou une forme**, le type des données, et parfois d'autres paramètres précisés dans les cas particuliers suivants.

- Des arrays de zéros :
`np.zeros(20)` # array de forme (20,), de type float.
`np.zeros((5,2,2), dtype=bool)` # array de forme (5,2,2), de type bool.
Notons qu'un zéro de type `bool` est bien sûr un **False**.
- Des arrays de uns :
`np.ones(20)` # array de forme (20,), de type float.
`np.ones((3,3), dtype=bool)` # array de forme (3,3), de type bool.
Notons qu'un “un” de type `bool` est bien sûr un **True**.
- Des arrays de valeurs de `a` (inclus, 0 par défaut) à `b` (exclus) :
`np.arange(4)` # [0 1 2 3]
`np.arange(4, step=.5)` # [0. 0.5 1. 1.5 2. 2.5 3. 3.5]
`np.arange(1, 10, step=2, dtype=float)` # [1. 3. 5. 7. 9.]
- Des arrays de valeurs de `a` (inclus) à `b` (inclus par défaut), de **taille fixée** :
`np.linspace(0, 1, 5)` # [0. 0.25 0.5 0.75 1.]
`np.linspace(0, 10, 4, endpoint=False)` # [0. 2.5 5. 7.5]

1.2 Manipuler des arrays

1.2.1 Opérations vectorielles

Maintenant que l'on sait les créer, on peut les manipuler et faire des opérations. Par défaut, elles sont effectuées coordonnée par coordonnée, ainsi si l'on a `x = np.array([1,2,3])` et `y = np.array([1,0,-1])`, on aura :

```
print(x + y)  # [2 2 2]
print(x * y)  # [1 0 -3]
print(y / x)  # [1. 0. -0.333]
```

Notons que le troisième résultat est un array de `float`, alors que l'on avait des entiers. En effet, quand on écrit une telle opération, il y a création d'un nouveau vecteur qui contient le résultat, et comme auparavant Numpy choisit automatiquement le type de celui-ci.

Fonctions usuelles. Les fonctions mathématiques usuelles sont disponibles dans Numpy et fonctionnent également de manière vectorielle (calculées coordonnée par coordonnée) :

```
print(np.exp(x)) # [ 2.71828183  7.3890561  20.08553692]
print(np.tan(x)) # [ 1.55740772 -2.18503986 -0.14254654]
print(np.pi)    # 3.141592653589793
```

NB : quand on utilise Numpy, **on n'importe pas le module** `math`. Les fonctions analogues de celui-ci ne seront pas capable de gérer les vecteurs.

Opérations booléennes. De la même manière que pour les opérateurs algébriques, les opérateurs booléens sont aussi interprétés coordonnée par coordonnée.

Exemple :

```
M = np.array([[1, 2], [0.5, -0.5]])
matrice_test = (M > 0)
print(matrice_test) # [[True  True]
                    # [True False]]
```

NB : les parenthèses ci-dessus sont optionnelles, elles sont là pour souligner que l'expression `M > 0` définit un nouvel objet : un ndarray de mêmes dimensions que `M` et dont les coordonnées sont de type `bool`.

1.2.2 Fonctions pratiques : somme, moyenne, etc.

Supposons que l'on dispose de deux vecteurs `x` et `y`.

- Somme de toutes les entrées : `np.sum(x)` ou `x.sum()`
- Moyenne : `np.mean(x)` ou `x.mean()`
- Variance : `np.var(x)` ou `x.var()`
Attention, contrairement à R, Numpy calcule par défaut la *variance non corrigée*, c'est-à-dire que `x.var()` renverra la même chose que `((x-x.mean())**2).mean()`.
- Écart-type : `np.std(x)` ou `x.std()`
- Covariance : `np.cov(x, y, bias=True)[0,1]` ou `((x-x.mean())*(y-y.mean())).mean()`
Attention, la fonction `np.cov` n'est pas cohérente avec `np.var` : elle renvoie une *matrice* de covariance *corrigée* par défaut. Mieux vaut se rappeler de la formule et utiliser la seconde expression donnée.

Si `x` est une matrice, toutes ces opérations (somme, moyenne, variance, écart-type) sont par défaut effectuées sur l'array *aplati* correspondant – c'est-à-dire le vecteur ligne composé de toutes les entrées successives de `x`. Si, au contraire, l'on veut effectuer les sommes de **toutes les lignes** (1^{ère} dimension, numérotée 0) ou de **toutes les colonnes** (2^{ème} dimension, numérotée 1), on utilisera un argument optionnel `axis`. Exemple :

```
x = np.arange(6).reshape((2,3))
print(x.sum()) # 15
```

```
# Somme les lignes.
print(x.sum(axis=0)) # [3 5 7]
# Somme les colonnes.
print(x.sum(axis=1)) # [ 3 12]
```

1.2.3 Algèbre linéaire

Les arrays bidimensionnels peuvent être interprétés comme des matrices, et les opérations matricielles usuelles (produit matriciel, transposé, inverse, décomposition en valeurs propres, etc) sont codées Numpy. De plus, les vecteurs (unidimensionnels) sont interprétés en priorité comme des vecteurs lignes **sauf** dans le cas où ils se trouvent à gauche d'un produit matriciel, auquel cas Numpy les interprètent directement comme le vecteur colonne correspondant (la transposé du vecteur ligne).

À part le produit matriciel, c'est le sous-module `np.linalg` qui gère les fonctions avancées d'algèbre linéaire.

Exemples avec `A = np.arange(6).reshape((2,3))`, `B = np.array([[0.5, -1], [1, 0.5]])`, `x = np.array([1, -1])`, et `y = np.arange(1,4)` :

- **produit matriciel** : `np.dot(A, y)` ou `A.dot(y)` ou `A @ y`
Avec nos exemples, on peut aussi avoir : `x @ A` `x @ B` `B @ x`
Si `x` et `y` avaient la même taille, `x @ y` renverrait le produit scalaire.
- **transposition** : `np.transpose(A)` ou `A.transpose()` ou `A.T`
- **inverse** de matrice : `np.linalg.inv(B)`
- **puissances de matrices** : `np.linalg.matrix_power(B, 5)`
- **valeurs et vecteurs propres** : `np.linalg.eig(B)`

1.3 Graphiques

1.3.1 Basiques

Maintenant que l'on sait calculer des choses, on peut les afficher, c'est là qu'intervient Matplotlib. La syntaxe basique de la fonction `plt.plot` est simple (cf. Figure 1) : on écrit `plt.plot(x,y)`, où `x` et `y` sont deux vecteurs **de même taille**. La fonction crée une figure qui relie successivement les points `(x[0],y[0]) -- (x[1], y[1]) -- ... (x[-1], y[-1])`. La fonction `plt.show()` (appelée sans argument) permet d'afficher le graphique.

Ainsi, pour afficher le graphe d'une fonction mathématique f sur un intervalle $[a, b]$, il suffit en général d'écrire :

```
x = np.linspace(a, b, 200)
plt.plot(x, calcul_de_f(x))
plt.show()
```

À savoir :

- Les échelles sont automatiquement ajustées, la taille de la figure est fixée par défaut (pour changer ces choses-là, chercher dans la doc!)
- On peut tracer **plusieurs courbes sur le même graphique**, il suffit d'appeler plusieurs commandes `plt.plot(x,y)` à la suite avant d'afficher le graphe. Les couleurs sont choisies automatiquement, selon une série de couleurs par défaut.

- Pour rajouter des **légendes** aux différentes courbes, on rajoute un *argument par mot-clé* `label` à la fonction `plt.plot`, puis on active l'affichage avec la fonction `plt.legend`. Par exemple, testez :

```
x = np.linspace(0, 10, 200)
plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend()
plt.show()
```

- Pour compléter un graphique, il faut un **titre et des légendes d'axes** : à faire avec `plt.title`, `plt.xlabel` et `plt.ylabel`.
- Pour tracer un **scatterplot** (si vos `x` ne sont pas ordonnés par exemple), utiliser la commande `plt.scatter(x, y)`.
- Pour tracer un **histogramme** (quand vous disposerez d'un échantillon `x`), on utilise la syntaxe `plt.hist(x, density=True)`. L'histogramme par défaut n'est pas très lisible mais ça fera l'affaire ! Note : on peut intercaler un entier en second argument de `plt.hist` pour choisir le nombre de barres à la main. Question : pourquoi préciser `density=True` ?
- Pour afficher une matrice de valeurs avec une échelle de couleur :

```
x = np.arange(100).reshape((10,10))
plt.imshow(x)
plt.show()
```

1.3.2 Avancés

Figures multiples. Une figure peut contenir des sous-figures : ces différents éléments sont gérés par Matplotlib par des classes `Figure` et `Axes`. Avec la fonction `plt.subplots`, on peut récupérer un objet figure et plusieurs objets axes que l'on peut ensuite utiliser séparément pour gérer les sous-figures.

Le plus simple est d'adapter l'exemple :

```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(10,5))
x1 = np.linspace(0, 2, 200)
x2 = np.linspace(-5, 5, 200)
for i in range(1,4):
    ax1.plot(x1, np.exp(x1/i)-1, label=f'exp(x/{i})-1')
    ax2.plot(x2, np.sin(i*x2), label=f'sin({i}x)')
ax1.legend()
ax2.legend()
fig.show()
```

Graphiques 3D. Pour tracer dans une figure en trois dimensions, il faut créer un objet `Axes` adapté, avec `fig = plt.figure()` puis `ax = fig.add_subplot(projection='3d')`. Cette dernière commande, pour les versions de Matplotlib inférieure à 3.2.0, ne fonctionnera qu'après avoir importé le sous module `mpl_toolkits.mplot3d`. Ainsi on commencera par les trois lignes (la première pour être sûr, suivant votre version) :

```
import mpl_toolkits.mplot3d
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
```

Puis l'on peut :

- Tracer une courbe 3D : utiliser `ax.plot` avec trois vecteurs au lieu de deux. Exemple dans la Figure 2.

```
t = np.linspace(0, 50, 1000)
ax.plot(t * np.sin(t),
        t * np.cos(t),
        np.log(1 + t))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Une courbe 3D')
fig.show()
```

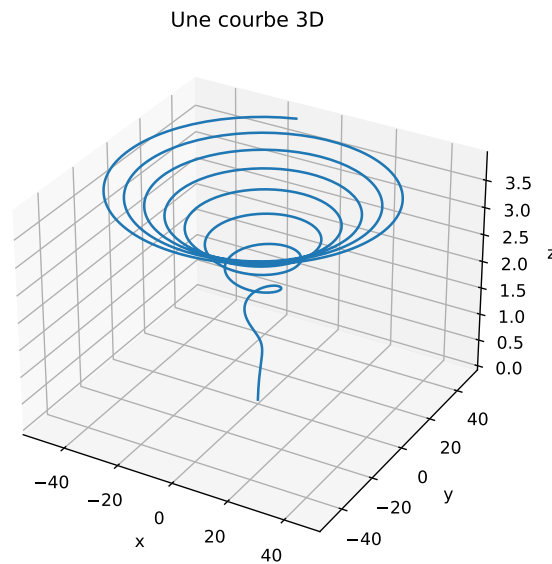


FIGURE 2 – Une courbe en 3D avec `ax.plot`.

- Tracer le graphe d'une fonction bidimensionnelle (une "nappe"). Pour cela on utilisera la syntaxe `ax.plot_surface(X, Y, Z)`, où les trois arguments sont des matrices de même dimensions. Les entrées correspondantes dans chaque matrice donneront l'abscisse, l'ordonnée et la cote de chaque point de la nappe. Pour obtenir des X et Y correspondant à une discrétisation régulière d'une zone $[a, b] \times [c, d]$, on utilisera la fonction `np.meshgrid`. Exemple dans la Figure 3.
- Faire un *scatterplot* 3D. Pour cela on utilisera la syntaxe `ax.scatter` avec trois vecteurs au lieu de deux. Les entrées correspondantes dans chaque matrice donneront l'abscisse, l'ordonnée et la cote de chaque point du *scatterplot*.

Animation. C'est un peu complexe, il y a trop de choses à voir pour que l'on puisse détailler ça ici (et je suis loin de maîtriser cet aspect de Matplotlib). Je réfère donc aux [exemples officiels](#).

Pour faire simple et obtenir un fichier `.gif`, on adaptera cet exemple :

```
from matplotlib.animation import PillowWriter
fig, ax = plt.subplots()

duree = 5 # Durée de l'animation en secondes.
fps = 25 # Nombre d'images par seconde.

writer = PillowWriter(fps)
with writer.saving(fig, "animation.gif", dpi=100):
    t = np.linspace(0, 10, 400)
    x = 0 # Décalage de la courbe.
```

```

a, b, c, d = 0, 5, 0, 5
x = np.linspace(a, b, 100)
y = np.linspace(c, d, 100)
X, Y = np.meshgrid(x, y)
def f(u, v):
    return (np.sin(u*v)
            / (0.1 + u + v)
            + (u + v)/10)
ax.plot_surface(X, Y, f(X, Y))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Une nappe')
fig.show()

```

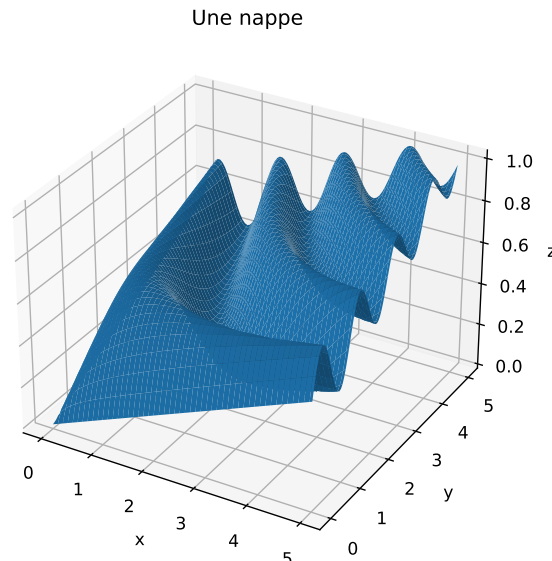


FIGURE 3 – Un graphique 3D avec `ax.plot_surface`.

```

vitesse = 1 # Vitesse d'augmentation de x, en unités par seconde.
for i in range(duree * fps):
    plt.clf()
    plt.plot(t, np.cos((x + t)**2) / (1 + x + t))
    plt.ylim((-1,1))
    writer.grab_frame() # Ajoute une image à l'animation.
    x += vitesse / fps

```

Note : on peut aussi obtenir un fichier `.mp4` si l'on remplace `PillowWriter` par `FFMpegWriter` et que l'on change le nom du fichier de manière appropriée.

1.4 Indexation

1.4.1 Slicing

Comme les listes de bases, les arrays Numpy peuvent être “slicés”. Le *slicing* est le fait d'utiliser une syntaxe du type `x[a:b]` pour désigner une partie d'un vecteur ou d'un tableau. Comme les `ndarray` sont multidimensionnel, avec Numpy le slicing permet de prendre ceci en compte. Pour une matrice `M`, on aura :

- `M[a:b]`, ou `M[a:b, :]`, désigne la sous-matrice composée des *lignes* indexées de `a` à `b-1`.
- `M[:, a:b]` désigne la sous-matrice composée des *colonnes* indexées de `a` à `b-1`.
- `M[a:b, c:d]` désigne la sous-matrice composée des éléments sur les lignes indexées de `a` à `b-1` et les colonnes indexées de `c` à `d-1`.

En complément, on rappelle que

- la syntaxe `a:b` de slicing peut s'écrire plus généralement :
 - `a:` pour désigner tous les éléments indexés à partir de `a`.
 - `:b` pour désigner tous les éléments indexés jusqu'à `b-1`.

- : pour désigner tous les éléments.
- a et/ou b peuvent être des entiers négatifs, auquel cas ils désignent les éléments en partant de la fin de la liste (-1 pour le dernier élément, -2 pour l'avant-dernier, etc).
- s'il y a un deuxième :, alors le chiffre qui suit désigne le "pas de sélection" des éléments : a:b:p désigne les éléments dont les indices s'obtiennent en commençant par l'indice a, puis en incrémentant l'indice par la valeur p successivement, tant que l'indice n'atteint pas la valeur b. Ce pas peut être négatif, auquel cas a doit être supérieur à b, sinon une liste vide (ou un array vide) sera renvoyée.

Exemples :

```
M = np.arange(20).reshape((4,5))
print(M)           # [[ 0  1  2  3  4]
                   #  [ 5  6  7  8  9]
                   #  [10 11 12 13 14]
                   #  [15 16 17 18 19]]

print(M[1:3,2:])    # [[ 7  8  9]
                   #  [12 13 14]]

print(M[:, :3, -2::-2]) # [[ 3  1]
                   #  [18 16]]
```

Modification des slices. Une slice peut être utilisée en tant qu'array à part entière, la plupart du temps elle sera copiée en mémoire et l'objet original non modifié :

```
x = M[1,:] - 7
print(x) # [-2 -1  0  1  2]
print(M) # M n'est pas modifié.
```

Mais l'on peut utiliser la slice d'un array à gauche d'un opérateur d'assignation = ou d'un +=, -=, *=, /=. Alors, c'est l'array original qui est modifié.

```
M[1,:] -= 7
M[:,0] = 3*M[:,0] - M[:,3]
print(M) # [[-3  1  2  3  4]
          #  [-7 -1  0  1  2]
          #  [17 11 12 13 14]
          #  [27 16 17 18 19]]
```

1.4.2 Indexation par liste

En plus du slicing, il existe de nombreuses façons de récupérer une partie d'un array. Une première manière est de passer une **liste d'indices** entre les crochets :

```
x = np.arange(-10,11,2) # [-10 -8 -6 -4 -2  0  2  4  6  8 10]
print(x[[0, 2, 2, 1, -4, -5, -4]]) # [-10 -6 -6 -8  4  2  4]
```

Remarquons que l'on peut passer plusieurs fois le même indice.

On peut remplacer la liste par un array d'entiers, et celui-ci peut même être multi-dimensionnel, cela créera un nouvel array de mêmes dimensions :

```
y = np.array([[0,1],[1,2]])
print(x[y]) # [[-10 -8]
            #  [-8 -6]]
```

Remarque : comme pour le slicing, on peut modifier les éléments d'un array indexés par une liste :

```
x[[0, 1, 3, 6, -1]] +=1
print(x) # [-9 -7 -6 -3 -2  0  3  4  6  8 11]
```

1.4.3 Indexation par masques booléens

On reprend ici notre matrice M :

```
M = np.arange(20).reshape((4,5))
M[1,:] -= 7
M[:,0] = 3*M[:,0] - M[:,3]
print(M) # [[-3  1  2  3  4]
          #  [-7 -1  0  1  2]
          #  [17 11 12 13 14]
          #  [27 16 17 18 19]]
```

Une autre façon de faire du slicing est de mettre entre crochets une matrice de même taille que M dont le dtype est `bool` :

```
print(M[M <= 1]) # [-3  1 -7 -1  0  1]
print(M[(M % 2) == 0]) # [ 2  4  0  2 12 14 16 18]
```

Une particularité à noter : si l'on "utilise" le résultat d'une telle expression, on obtient un array unidimensionnel. Par contre, si l'on met cette expression à gauche d'un opérateur d'assignation (=, etc.), on **modifie les cases concernées** dans la matrice.

Exemple :

```
M[M <= 1] = -1
M[(M % 2) == 0] = 50 + 2*np.arange((M % 2) == 0).sum())
print(M) # [[-1 -1 50  3 52]
          #  [-1 -1 -1 -1 54]
          #  [17 11 56 13 58]
          #  [27 60 17 62 19]]
```

Remarque : le code `((M % 2) == 0).sum()` compte le nombre d'éléments pairs de la matrice M. En effet, la fonction `sum()`, appelée sur un array de booléens, fait la somme des 1 pour les éléments égaux à `True` (et des 0 pour les éléments égaux à `False`).

1.5 Manipuler l'aléatoire avec Numpy

Le module Numpy fournit son générateur de nombres (pseudo-)aléatoires avec le sous-module `np.random`.

Attention : la bibliothèque standard Python offre un module `random` aussi, mais celui-ci ne permet pas de générer des objets Numpy. Donc dans un projet utilisant Numpy (et dans tout ce cours), on n'utilisera **jamais le module random standard**, mais bien le sous-module `np.random`, dont les fonctionnements diffèrent. Soyez donc particulièrement vigilant, sur ce sujet, à **chercher dans la bonne documentation** : celle de Numpy.

Grâce à `np.random`, il est possible de générer rapidement des text de nombres aléatoires i.i.d. d'une certaine loi. La façon recommandée d'opérer est la suivante :

1. On définit au début du script un **générateur de nombres aléatoires** (*random number generator*) `rng` avec `rng = np.random.default_rng()`.
2. On l'utilise pour générer ce que l'on veut :
 - `rng.uniform(...)` pour des variables uniformes un intervalle réel.
 - `rng.integers(...)` pour des variables uniformes sur un intervalle d'entiers.
 - `rng.binomial(...)` pour des variables binomiales.
 - `rng.poisson(...)` pour des variables de Poisson.
 - `rng.normal(...)` pour des variables gaussiennes.
 - et bien d'autres lois ! Cf. [la doc](#) pour la liste complète.

Entre les parenthèses des fonctions précédentes, on met les paramètres des lois si besoin, et une taille / forme (argument optionnel `size`). La taille peut être un entier seul, pour générer un échantillon de longueur donnée, où un tuple (n_1, \dots, n_k) , qui donne les dimensions du ndarray que l'on obtiendra.

Exemple : pour obtenir une matrice A de dimensions 3×4 dont les coefficients sont uniformes dans $[0, 1]$, on tape `A = rng.uniform(size=(3,4))`.

Important : il est possible de *seeder* l'aléa de vos simulations en donnant un argument à la fonction `rng = np.random.default_rng(seed=0)`. Cet argument est appelé la *seed* (graine, ici donnée comme valant 0), et permet de reproduire les simulations à l'identique (c'est le principe du pseudo-aléatoire). Sans argument, le générateur est "auto-seedé", et vos simulations ne seront pas reproductibles à l'identique.

1.6 Technique avancée : le broadcasting

Prenons la matrice `M = np.array([[1, 2], [-0.5, 0.5]])`. Remarquons que le code `M > 0` effectue la comparaison de chaque entrée de M avec 0. Ceci a le même effet que si l'on crée une matrice Z , de mêmes dimensions que M et remplie de zéros, puis que l'on effectue `M > Z`.

```
Z = np.zeros_like(M)
matrice_test = (M > Z)
print(matrice_test)  # [[True  True]
                    #  [True False]]
```

On appelle ce qui se passe ici le **broadcasting** : un array de dimensions inférieures (ici le vecteur de taille 1 constitué d'un seul zéro) est interprété comme une matrice entière en se répliquant dans toutes les dimensions manquantes. C'est aussi ce qui se passe quand on effectue les opérations suivantes :

```
print(M + 1)          # [[2.  3. ]
                    #  [1.5 0.5]]

print(3 * M)          # [[ 3.   6. ]
                    #  [ 1.5 -1.5]]

print(M / np.array([2, 10])) # [[ 0.5  0.2 ]
                    #  [ 0.25 -0.05]]
```

La dernière ligne est plus subtile que les autres : ici l'array qui est *broadcast* est un vecteur de longueur deux. Numpy, dans une opération avec une matrice de dimensions $(n, 2)$ l'interprétera comme une matrice de même dimensions dont toutes les lignes sont égales au vecteur $[2 \ 10]$. Plus de détails sur le [broadcasting dans la doc officielle](#).

2 Exercices

Exercice 1 : indexation, masques booléens

a) Copier le code

```
rng = np.random.default_rng(0)
A = rng.integers(6, size=(3,4))
mask = A <= 1
```

Afficher la matrice A. Quel genre d'objet Numpy est stocké dans la variable mask ?

b) Utiliser la fonction `np.logical_or()` afin de modifier mask, qui devra contenir le masque booléen correspondant aux éléments de A qui sont égaux à 0, 1, ou 5.

c) En une commande, modifier la matrice A en remplaçant chaque élément x mentionné ci-dessus par son opposé $-x$.

Exercice 2 : graphes de fonctions

a) Importer numpy et le sous-module pyplot de matplotlib avec leurs alias standards.

b) Que font les commandes `x = np.arange(0, 10, 0.05)` et `x = np.linspace(0, 10, 200)` ? Quels sont les avantages de chacune ? À quelles commandes R correspondent-elles ?

c) Tracer le graphe de la fonction $x \mapsto x \sin(x)^2$, sur l'intervalle $[0, 10]$.

d) Tracer le graphe de la fonction $t \mapsto \log(1 + t^2)$ pour $t \in [-3, 3]$.

e) Utiliser une boucle `for` pour tracer les graphes des fonctions $t \mapsto n \log(1 + t^2/n)$ pour $n \in \{1, 2, 5, 10, 20\}$, et celui de $t \mapsto t^2$, assortis de légendes, sur le même graphique.

f) Rajouter des légendes sur les axes des abscisses et ordonnées, puis un titre, avec `plt.xlabel`, `plt.ylabel` et `plt.title`.

Exercice 3 :

a) Créer une première matrice avec la syntaxe

```
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

b) Vérifier que l'on obtient bien la même matrice avec l'expression `A = np.arange(1, 10).reshape(3, 3)`.

c) Utiliser une syntaxe de *slicing* pour extraire de A les sous-matrices $B = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$ et $C = \begin{pmatrix} 1 & 2 \\ 7 & 8 \end{pmatrix}$.

d) Écrire une fonction `col_paires(A)` qui renvoie la sous-matrice constituée de toutes les colonnes d'une matrice A dont les indices sont pairs (les colonnes 0, 2, etc.).

e) Que fait la ligne `A[2, :] += A[0, :]` ? La variable C (définie en question (c)) est-elle affectée ?

f) Constater l'effet du *broadcasting* en évaluant les expressions `A + 1`, `A + np.arange(3)`, et `A + A[:, 1]`.

- g) L'assignation est également *broadcastable* : que fait la commande `A[:] = np.arange(3)`, et pourquoi ne peut-on pas omettre le “[:]” ?

Exercice 4 : algèbre linéaire avec Numpy.

- Écrire une fonction `matrice_carree(n)` qui renvoie une matrice de dimensions $n \times n$ dont les entrées sont, dans l'ordre, les entiers de 0 à $n^2 - 1$. On posera `A = matrice_carree(3)`.
- Générer une matrice aléatoire `B` de taille 3×3 , dont les coefficients sont indépendants et de loi uniforme entre -1 et 1 .
- On veut perturber `A` en lui ajoutant `B`. La ligne `A += B` renvoie-t-elle une erreur ? Que faire pour régler le problème ?
- Définir un vecteur `v = np.array([1, 2, -1])`.
- Vérifier que ce vecteur peut être interprété comme vecteur ligne ou colonne en fonction du contexte : `A @ v` ou `v @ A` renvoient bien les multiplications matricielles qui ont du sens, en prenant `v` comme vecteur colonne dans le premier cas, ligne dans le second.
- Définir `Ainv` comme l'inverse de la matrice `A`. A-t-on bien la matrice identité quand on effectue `A @ Ainv` ?
- Refaire la dernière question avec la matrice `A` originale perturbée par des valeurs uniformes entre -10^{-15} et 10^{-15} . Expliquer ce qui se passe.

Exercice 5 : régression linéaire. Considérons les données

```
x1 = [5.18, 3.35, 2.2 , 2.08, 6.07, 6.56, 5.03, 5.65, 4.72, 6.68]
x2 = [5.63, 4.01, 5.71, 4.07, 5.46, 4.35, 5.73, 5.08, 4.6 , 4.85]
y = [3.38, 2.15, -5.77, -4.23, 8.47, 12.2, 4.44, 11.78, 8.08, 13.26]
```

- Utiliser Numpy et ses méthodes d'algèbre linéaire pour estimer par la méthode des moindres carrés ordinaires les coefficients du modèle linéaire

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon.$$

- Calculer la somme des carrés des résidus, les comparer avec ceux obtenus dans le modèle sans constante.
- Sur un graphique 3D, tracer le scatterplot des données et le plan correspondant aux coefficients trouvés. Rajouter un titre et les étiquettes x_1 , x_2 et y sur les axes correspondants.

Exercice 6 : méthode de Newton n -dimensionnelle. En dimension deux, la récurrence de la méthode de Newton devient

$$x_{n+1} = x_n - df(x_n)^{-1} \cdot f(x_n),$$

où f est une fonction $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ et df est sa jacobienne :

$$df : (x, y) \mapsto \left(\frac{\partial}{\partial x} f(x, y), \frac{\partial}{\partial y} f(x, y) \right) \in \mathcal{M}_2(\mathbb{R}).$$

- Réécrire la fonction `newton2(f, df, eps=1e-10, N=1000)` en supposant que les arguments donnés sont une fonction $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ et sa jacobienne.

b) À l'aide de la question précédente, résoudre numériquement le système d'équation

$$\begin{cases} x = 1 + \sin(y) \\ y^2 = \cosh(x). \end{cases}$$

Exercice 7 :

- Choisir un réel $\lambda > 0$ et générer un échantillon x de $n=1000$ variables aléatoires uniformes i.i.d. de loi exponentielle de paramètre λ .
- Tracer l'histogramme *normalisé* de x .
- Tracer la densité théorique de l'échantillon sur le même graphique. Ajouter toutes les légendes.

Exercice 8 :

- Générer un échantillon x de taille 10000 de votre loi préférée.
- Illustrer la loi des grands nombres en traçant

$$\bar{x}_k = \frac{1}{k} \sum_{i=0}^{k-1} x_i$$

en fonction de k .

- Tracer sur le même graphique le graphe de la fonction constante égale à l'espérance. Ajouter toutes les légendes.

Exercice 9 : courbes paramétrées.

- Que représente la courbe paramétrée par

$$\begin{aligned} x(t) &= \frac{\sin(t)}{1 + \cos(t)^2} \\ y(t) &= \frac{\sin(t) \cos(t)}{1 + \cos(t)^2}, \end{aligned}$$

pour $t \in [0, 2\pi]$?

- Tracer une [spirale](#) avec Matplotlib.

Non, la courbe de la question précédente n'a rien à voir avec une spirale, c'est simplement pour vous montrer que l'on peut tracer des courbes paramétrées.

Exercice 10 : équation de la chaleur. L'équation de la chaleur sur le cercle, en dimension 1, est l'équation

$$\partial_t f = \partial_x^2 f,$$

où la solution $f = f(t, x)$ est définie sur $\mathbb{R}_+ \times \mathbb{S}$, à partir d'une condition initiale $f_0 = f(0, \cdot)$, et où \mathbb{S} désigne le cercle : le segment $[0, 1]$ où l'on a identifié les deux extrémités 0 et 1. Le but de cet exercice est de proposer une résolution numérique de ce problème.

- a) Étant donné un entier n , on pose $\delta = 1/n$ et $x_i = i\delta$ pour $i \in \{0, \dots, n-1\}$. Les points $(x_i)_{0 \leq i < n}$ forment une discrétisation de l'intervalle $[0, 1]$ (donc ici, du cercle \mathbb{S}). On représentera une fonction $u : \mathbb{S} \rightarrow \mathbb{R}$ par un vecteur de taille $n : (u(x_i))_{0 \leq i < n}$. La fonction $v = \partial_x^2 u$ est alors représentée par

$$v_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{\delta^2},$$

où les indices sont à comprendre modulo n .

Écrire une fonction `laplacien(u)` qui renvoie $\partial_x^2 u$, sans utiliser de boucle `for`. *Attention aux indices aux extrémités du vecteur : on rappelle que l'on est sur le cercle.*

Dans la suite, pour tester nos fonctions et faire tourner le code, on se fixera $n = 100$.

- b) Écrire une fonction `chaleur_etape(u, dt)` qui renvoie, étant donnée u qui représente $f(t, \cdot)$, renvoie $f(t + dt, \cdot)$.
- c) Écrire une fonction `chaleur_resol(f0, t_final=0.02, dt=5e-5)` qui renvoie la solution de l'équation de la chaleur partie de f_0 sous forme de matrice : une ligne par pas de temps.
- d) Visualiser sous forme de “nappe 3D” une solution partie de plusieurs conditions initiales choisies par vous. Suggestions : la fonction $f_0 : x \mapsto \sin(4\pi x)$, la fonction indicatrice de $[1/3, 2/3]$.

Exercice 11 : analyse en composantes principales. On considère une matrice $X \in \mathcal{M}_{np}(\mathbb{R})$, qui représente un échantillon de taille n de p variables. On notera $\bar{X}_j = \frac{1}{n} \sum_i X_{ij}$, et $\sigma_j^2 = \frac{1}{n} \sum_i (X_{ij} - \bar{X}_j)^2$.

- a) Écrire une fonction `centrage(X, reduction=False)` qui prend une matrice $X = (X_{ij})_{i,j}$ en entrée et renvoie la matrice $\tilde{X} = (X_{ij} - \bar{X}_j)_{i,j}$ (resp. $\tilde{X} = ((X_{ij} - \bar{X}_j)/\sigma_j)_{i,j}$) dans le cas `reduction=False` (resp. dans le cas `reduction=True`).

La décomposition en valeurs singulières de la matrice \tilde{X} est le fait de l'écrire sous la forme $\tilde{X} = UDV^T$, où $(U, V) \in \mathcal{M}_n \times \mathcal{M}_p$ sont des matrices unitaires (les colonnes forment une base orthonormée), et $D \in \mathcal{M}_{n,p}$ est une matrice dont les coefficients diagonaux sont des réels positifs décroissants, les autres étant nuls.

On note v_j la j -ème colonne de V , u_j la j -ème colonne de U , d_j le j -ème coefficient diagonal de D , et on pose $Y_j = \tilde{X}v_j = d_j u_j$. Le vecteur v_j représente une combinaison linéaire des variables \tilde{X}_j telle que si l'on “effectue le changement de variable” $(\tilde{X}_j) \leftarrow (Y_j)$, l'on obtient des variables qui sont non corrélées, et de variance respectives d_j^2/n .

- b) En utilisant la fonction `np.svd(A)` qui effectue la décomposition en valeurs singulières de la matrice A , écrire une fonction `acp(X, reduction=False)` qui, après centrage de X , effectue l'ACP de nos données. La fonction doit renvoyer :
- La matrice $Y = (Y_j)_{1 \leq j \leq p}$ qui représente les données dans la nouvelle base.
 - La liste $(d_j^2/n)_{1 \leq j \leq p}$ des variances empiriques des Y_j .
 - La matrice V , qui détermine le changement de variables effectué.
- c) Écrire une fonction `plot_acp(Y, var)`, qui trace les données dans le repère engendré par les deux premières variables (Y_1, Y_2) du changement effectué par l'ACP. En légende sur chaque axe, préciser la proportion de la variance des données expliquée le long de cet axe.
- d) Tester les codes ci-dessus avec les données fournies dans le fichier `data_td2_acp.txt` (d'abord importer le vecteur X avec `X = np.loadtxt('data_td2_acp.txt')`). Quelle est la dimension du jeu de données original ? Sur combien de dimensions peut-on le ramener grâce à l'ACP, tout en gardant au moins 95% de la variance ? 99% de la variance ?