# 1. Le module "NumPy", python scientifique

L'exécution d'un programme python est plus de 100 fois plus lent qu'un programme équivalent écrit en langage C. De plus, il ne supporte pas de manière native les tableaux "arrays". Les listes remplacent les "arrays" et sont plus générales, mais beaucoup plus lentes à traiter.
Python a la possibilité d'appeler des fonctions écrites en langage C, qui s'exécutent donc beaucoup plus rapidement que si elles étaient écrites en Python.
Pour des applications scientifiques, **scipy** a été créé. **scipy** est un ensemble de librairies écrites en langage C, destiné à un usage scientifique de Python et au traitement d'images. c.f. *www.scipy.org*.
La base de **scipy** est le module **numpy** qui a été écrit en langage C, spécifiquement pour le calcul scientifique sous Python. Il introduit le type "**ndarray**", qui gère des tableaux de plusieurs dimensions. "n Dimentional Arrays". Ne nombreuses fonctions ont été écrites en langage C pour manipuler ces "ndarray".

Ce cours se base en grande partie sur le livre de **Ashwin Pajankar** : "Python 3 Image Processing. Learn Image Processing with Python 3, NumPy, Matplotlib, and Scikit-image", édition bpb, 2019.
Les codes du livre sont disponibles sous : https://github.com/bpbpublications/Python-3-Image-Processing

## Installation

Pour utiliser les modules de **scipy** il faut installer diverses bibliothèques ("libraries" en anglais). Pour cela, l'utilitaire **pip3** est utile.
Depuis un terminal (cmd sou Windows), tapez les commandes suivantes :

```
sudo pip3 install ipykernel
sudo pip3 install jupyter
sudo pip3 install prompt-toolkit
sudo pip3 install numpy
sudo pip3 install matplotlib
```

Si un module a déjà été installé, cela sera indiqué et la commande tapée n'aura pas d'effet négatif.
Il faut que python et son installateur de module **pip3** aient été installés.

En tapant `pip3 list` on peut voir tous les modules installés.
Il est possible qu'il faille aussi installer "jupyter-notebook" en tapant :

```
sudo apt install jupyter-notebook
```

Divers **environnements de travail** peuvent être utilisé.
**IDLE** est un environnement minimaliste, simple.
**Thonny** est un environnement bien adapté pour Python, plus complet que IDLE.
**Spyder** est un autre environnement bien adapté pour Python encore plus complet.
**Jupyter** est une nouvelle manière de travailler, dans un navigateur, en mode très interactif.

## Premiers essais

Dans un Terminal, tapez : `jupyter notebook` ou `jupyter-notebook` ou `python3 -m notebook`
Normalement, votre navigateur par défaut devrait s'exécuter et afficher une fenêtre avec un menu, des boutons et "jupyter" écrit vers le haut de la fenêtre.
Si ce n'est pas le cas, et un mot de passe est demandé, il faut copier l'URL indiquée dans le Terminal et le copier dans la barre d'adresse du navigateur.
c.f. https://jupyter-notebook.readthedocs.io/en/stable/public_server.html pour des configurations.

Cliquez sur `adoc`, pour vous retrouver dans le dossier "adoc".
Cliquez sur le bouton : `New`
Puis sur : `Python3`

Tapez : `print("Bonjour!")`
puis cliquer sur le bouton "Run" ou pressez : Alt+Enter.
*Bonjour!* devrait s'écrire sous la cellule et une autre cellule devrait s'ouvrir.

Dans chaque cellule, plusieurs lignes de code peuvent être écrite.
"Enter" va simplement à la ligne.
"Alt+Enter" exécute le code, en ouvrant une nouvelle cellule en dessous.
"Shift+Enter" exécute le code, puis passe à la cellule suivante.

**Exercice 1.1 :**
Dans le jupyter notebook tapez :
*# À vous d'écrire un commentaire pertinent ici.*

```
import numpy as np
x = np.array([1, 2, 3], np.int16)
print(x)
print(type(x))
y = list(x)  # convertit le "ndarray" en "liste"
print(type(y))
print(type(x[0]))
```

Exécutez ce code (Alt+Enter), pour obtenir :

```
[1 2 3]
<class 'numpy.ndarray'>
<class 'list'>
<class 'numpy.int16'>
```

Renommez ce *notebook* en **np0110_debut**. Sauvegardez**.**

Cela crée un tableau "**array**" en anglais de 3 éléments de type entiers codés sur 16 bits, stocké dans la variable *x*. Un tel tableau correspond à une liste. Il peut facilement être converti en une liste.
Le tableau *x* peut être manipulé beaucoup plus efficacement et rapidement que la liste *y*.

Une **liste** est une suite de pointeurs sur divers éléments, qui peuvent être de types différents.
Un **ndarray** est une suite de zones mémoires, chaque zone étant de même taille et stockant un nombre de même type.

Un **ndarray** peut être vu comme un **vecteur** ou comme une **matrice**. Nous verrons plus loin que des opérations vectorielles, matricielles et plus peuvent être effectuées sur des *ndarray*.

**Exercice 1.2 :**
Ajoutez à la suite du code précédent le code :

```
print(type(y[0]))
x[2] = x[2]**10
y[2] = y[2]**10
print(x[2], y[2])
print(type(x[2]))
print(type(y[2]))
```

Exécutez et observez que le type de x[2] ne change pas et reste un entier sur 16 bits.
Vu que "$3^{10} = 59'049 = 2^{16} – 6'487$" est trop grand pour être stocké comme un nombre entier signé, le résultat obtenu dans x[2] est faux. Il y a eu un **dépassement de capacité**.
Le type de y[2] a changé et le résultat est correcte.
° Le prix à payer pour l'efficacité des calculs avec des *ndarray* est le risque de dépassement de capacité.
° Le prix à payer pour le résultat exact du calcul avec une *liste* est la lenteur.
**Référence sur NumPy** : https://numpy.org/devdocs/

**Exercice 1.3 :**
Dans une nouvelle cellule, tapez :
```
a = np.array([[1, 2, 3], [4, 5, 6]], np.float64)
print(a)
print(type(a[0]))
print(type(a[0,0]))
```

Il existe au moins 35 types différents utilisables dans les *ndarray*.
Le type *np.float64* est le plus utilisé.

Commentez les lignes ci-dessus, pour expliquer ce qu'elles produisent.
Remarquez que `a` est une **matrice**.

**Exercice 1.4 :**
Que se passe-t-il si vous essayez d'accéder à `x[3]` ? Ou à `a[0, 3]` ?
Testez le code suivant et commentez-le pour indiquer à quoi il correspond.
```
print(a[:, 0])
print(a[:, 1])
print(a[:, 2])
print(a[0, :])
```

# Propriétés des ndarray

**Exercice 1.5 :**
Dans une nouvelle cellule, tapez :
```
print(a)
print(a.shape)
print(a.ndim)
print(a.dtype)
print(a.size)
print(a.nbytes)
```
Exécutez

Puis, dans une nouvelle cellule, tapez :
```
b = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]], np.float64)
print(b)
print(b.shape)
print(b.ndim)
print(b.dtype)
print(b.size)
print(b.nbytes)
```

Commentez les lignes pour expliquer la signification de chacune de ces propriétés.

**Exercice 1.6 :**
Dans une nouvelle cellule, tapez :
```
print(np.inf)
print(np.NAN)
print(np.NINF)
print(np.PINF)
print(np.NZERO)
print(np.PZERO)
print(np.e)
print(np.euler_gamma)
print(np.pi)
```

Exécutez et commentez.

# Création de ndarray, manipulations et opérations

**Exercice 1.7 :**
Dans une nouvelle cellule, tapez :

```
c = np.zeros([3,3], dtype=np.float64)
print(c)
I = np.eye(5, dtype=np.float64)
print(I)
d = np.eye(5, dtype=np.float64, k=-1)
print(d)
m1 = np.ones([5,5], dtype=np.float64)
print(m1)
m5 = np.full([2,3,4], dtype=np.float64, fill_value = 5)
print(m5)
r1 = np.random.randint(low=0, high=9, size=35)
print(r1)
r2 = np.sort(r1)
print(r2)
r3 = np.random.rand(2,6)
print(r3)
r4 = np.sort(r3)
print(r4)
```

Exécutez et commentez.
Les nombres aléatoires atteignent-ils les bornes *low* et *high* ?

**Exercice 1.8 :**
Dans une nouvelle cellule, tapez :

```
x1 = np.array([1, 2, 3, 4])
x2 = np.array([10, 20, 30, 40])
r1 = 3 * x1
print(r1)
r2 = x1 + x2
print(r2)
r3 = x1 * x2
print(r3)
x3 = np.random.rand(5)
print(x3)
print(np.mean(x3))    # c.f. https://numpy.org/devdocs/reference/routines.statistics.html
print(np.average(x3))
print(np.median(x3))
print(np.std(x3))
print(np.sort(x3))
print(np.sqrt(x3))
print(np.sin(x3))
```

Exécutez et commentez.

**Exercice 1.9 :**
Dans une nouvelle cellule, tapez :

```
z1 = np.array([1.0j, 1+1.0j, 2+1.0j, np.cos(np.pi/4)+ np.sin(np.pi/4)*1.j])
print(z1)
z2 = z1 * z1; print(z2)
z3 = np.exp(z1); print(z3); print(z3[0])
```

Exécutez et commentez.
Remarquez comme il est facile de traiter les nombres complexes.

**Raccourcis clavier :**

**Command Mode (press `Esc` to enable)**
`F` : find and replace
`Ctrl-Shift-F` : open the command palette
`Ctrl-Shift-P` : open the command palette
`Enter` : enter edit mode
`P` : open the command palette
**`Shift-Enter` : run cell, select below**
**`Ctrl-Enter` : run selected cells**
**`Alt-Enter` : run cell and insert below**
`Y` : change cell to code
`M` : change cell to markdown
`R` : change cell to raw
`1` : change cell to heading 1
`2` : change cell to heading 2
`3` : change cell to heading 3
`4` : change cell to heading 4
`5` : change cell to heading 5
`6` : change cell to heading 6
`K` : select cell above
`Up` : select cell above
`Down` : select cell below
`J` : select cell below
`Shift-K` : extend selected cells above
`Shift-Up` : extend selected cells above
`Shift-Down` : extend selected cells below
`Shift-J` : extend selected cells below

**Edit Mode (press `Enter` to enable)**
`Tab` : code completion or indent
`Shift-Tab` : tooltip
`Ctrl-]` : indent
`Ctrl-[` : dedent
`Ctrl-A` : select all
`Ctrl-Z` : undo
`Ctrl-/` : comment
`Ctrl-D` : delete whole line
`Ctrl-U` : undo selection
`Insert` : toggle overwrite flag
`Ctrl-Home` : go to cell start
`Ctrl-Up` : go to cell start
`Ctrl-End` : go to cell end
`Ctrl-Down` : go to cell end
`Ctrl-Left` : go one word left

`A` : insert cell above
`B` : insert cell below
`X` : cut selected cells
`C` : copy selected cells
`Shift-V : paste cells aboveV` : paste cells below
`Z` : undo cell deletion
`D,D` : delete selected cells
`Shift-M` : merge selected cells, or current cell with cell below if only one cell is selected
`Ctrl-S` : Save and Checkpoint
`S` : Save and Checkpoint
`L` : toggle line numbers
`O` : toggle output of selected cells
`Shift-O` : toggle output scrolling of selected cells
`H` : show keyboard shortcuts
`I,I` : interrupt the kernel
`0,0` : restart the kernel (with dialog)
`Ctrl-V` : Dialog for paste from system clipboard
`Esc` : close the pager
`Q` : close the pager
`Shift-L` : toggles line numbers in all cells, and persist the setting
`Shift-Space` : scroll notebook up
`Space` : scroll notebook down

`Ctrl-Right` : go one word right
`Ctrl-Backspace` : delete word before
`Ctrl-Delete` : delete word after
`Ctrl-Y` : redo
`Alt-U : redo selection`
`Ctrl-M` : enter command mode
`Ctrl-Shift-F` : open the command palette
`Ctrl-Shift-P` : open the command palette
`Esc` : enter command mode
`Shift-Enter` : run cell, select below
`Ctrl-Enter` : run selected cells
`Alt-Enter` : run cell and insert below
`Ctrl-Shift-Minus` : split cell at cursor
`Ctrl-S` : Save and Checkpoint
`Down` : move cursor down
`Up` : move cursor up

Notes personnelles

# 2. Le module "matplotlib" pour tracer des graphiques

Le module **MatPlotLib** fait partie de la base **scipy** du Python scientifique. Il permet de tracer des graphiques avec de nombreuses options. Il ne faut pas écrire ~~MathPlotLib~~ (avec un "h"), car "mat" provient de "matrice" et non de "mathématique".
**MatPlotLib** s'est basé sur le logiciel "MatLab" (Matrix Laboratory), qui permet de faire du calcul scientifique simplement et efficacement.
Une matrice est un "ndarray" de dimension deux.
Un vecteur est un "ndarray" de dimension un.

Créez un nouveau notebook et renommez-le : "*np0210_graphiques*". Sauvez-le.

**Exercice 2.1 :**
Dans la première cellule, tapez :
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(5)
print(x)
y=x
plt.plot(x, y, 'o--')
plt.plot(x, -y, 'o-')
plt.title('y=x et y=-x')
plt.show()
```

La première ligne sert à ce que les graphiques suivent dans la page. Est-elle utile ?
Exécutez et commentez.
Remarquez comme il est facile de tracer des graphiques.

**Exercice 2.2 :**
Dans une nouvelle cellule, tapez :
```
N = 11
x = np.linspace(0, 10, N)
print(x)
y = x**2
print(y)
plt.plot(x, y, 'o--')
plt.show()
```

`np.linspace(0, 10, N)` crée un vecteur de $N$ éléments, espacés régulièrement entre 0 et 10, bornes comprises.
Exécutez et commentez.

**Exercice 2.3 :**
Dans une nouvelle cellule, tapez :
```
N = 10
x = np.linspace(0.1, 1, N)
y = np.logspace(0.1, 1, N)
print(x)
print(y)
plt.plot(x, y, 'o--')
plt.show()
print(10**0.1, 10**0.2, 10**0.3, 10**0.4)
print(np.log10(y))
```

`np.logspace(0.1, 1, N)` crée un vecteur de $N$ éléments, espacés de manière logarithmique entre $10^{0,1}$ et $10^{1}$, bornes comprises. $10^{0,1}$ $10^{0,2}$ ; $10^{0,3}$ ... $10^{1,0}$ sont les 10 nombres du vecteur.
Exécutez et commentez.

**Exercice 2.4 :**
Dans une nouvelle cellule, tapez :

```
N = 11
x = np.linspace(0, 10, N)
plt.plot(x, x**2)
plt.plot(x, x**3)
plt.plot(x, 2**x)
plt.plot(x, 2*x)
plt.show()
```

Exécutez et commentez.

**Exercice 2.5 :**
Dans une nouvelle cellule, tapez :

```
plt.figure(figsize=(11, 8))
N = 11
x = np.linspace(0, 10, N)
plt.plot(x, x**2, x, x**3, x, 2**x, x, 2*x)
plt.grid(True)
plt.axis([0, 10, 0, 1000]) # idem : plt.xlim([0, 10]); plt.ylim([0, 1000])
plt.xticks(np.linspace(0,10,11,endpoint=True))
plt.yticks(np.linspace(0,1000,11,endpoint=True))
fig = plt.gcf(); fig.set_size_inches(8, 6) # autre manière de faire
#fig.savefig('test_25.png', dpi=100)
plt.show()
```

Exécutez et commentez.
Enlevez le commentaire devant "fig.savefig...", pour créer un fichier.
Mettez la ligne "fig = plt.gcf()" en commentaire, pour voir la différence.
C.f. https://matplotlib.org/api/pyplot_api.html
C.f. https://python.developpez.com/tutoriels/graphique-2d/matplotlib/
C.f. https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html

**Exercice 2.6 :**
Dans une nouvelle cellule, tapez :

```
N = 11
x = np.linspace(0, 10, N)
plt.plot(x, x**2, label='x**2')
plt.plot(x, x**3, label='x**3')
plt.plot(x, 2**x, label='2**x')
plt.plot(x, 2*x, label='2*x')
plt.legend(loc='upper left')
plt.grid(True)
plt.axis([0, 10, 0, 1000]) # idem : plt.xlim([0, 10]); plt.ylim([0, 1000])
plt.xticks(np.linspace(0,10,11,endpoint=True))
plt.yticks(np.linspace(0,1000,11,endpoint=True))
plt.xlabel('x = linspace(...)')
plt.ylabel('y = f(x)')
plt.title('Exemple de 4 courbes')
plt.show()
```

Exécutez et commentez.

Téléchargez les codes du livre : https://github.com/bpbpublications/Python-3-Image-Processing
Regardez les codes du "Chapter 07" pour plus d'exemples.
Nous verrons plus de possibilités lorsque nous en aurons besoin.

# 3. Algèbre linéaire avec NumPy

**Contexte :**
Une matrice est un tableau de nombres.
Un vecteur s'écrit généralement comme une colonne de nombres.
L'objectif de ce qui suit est d'apprendre à traiter les matrices et vecteurs avec NumPi et de voir leur lien avec les systèmes linéaires d'équations.

**Théorie : Matrice, vecteur colonne et produit matriciel**
Une matrice est un tableau de nombres.

Voici une matrice de 3 lignes et 3 colonnes : $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$

Voici un vecteur colonne qui est une matrice de 3 lignes et une colonne : $v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$

Par définition, le produit de la matrice $A$ par le vecteur $v$ est :

$$A \cdot v = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} a_{11} \cdot v_1 + a_{12} \cdot v_2 + a_{13} \cdot v_3 \\ a_{21} \cdot v_1 + a_{22} \cdot v_2 + a_{23} \cdot v_3 \\ a_{31} \cdot v_1 + a_{32} \cdot v_2 + a_{33} \cdot v_3 \end{pmatrix}$$

Attention, $A$ est à gauche et $v$ à droite, l'ordre compte !
C'est le "**produit ligne - colonne**".
Remarquez que le terme à droite de l'égalité est un vecteur colonne.
Nous allons voir ci-dessous que cette définition du produit matriciel est très pratique.

**Exercice 3.1 :**
Sur une feuille de papier :
a) Écrivez une matrice $A$ de 2 lignes et de 2 colonnes.
b) Écrivez une matrice $B$ de 3 lignes et de 2 colonnes.
c) Combien de lignes doit avoir un vecteur $v$ pour que la multiplication $A \cdot v$ soit définie ?
d) Combien de lignes doit avoir un vecteur $v$ pour que la multiplication $B \cdot v$ soit définie ?

Ce qui suit se base en grande partie sur le livre de **Leo (Liang-Huan) Chin et Tanmay Dutta** : "NumPy Essentials", édition Packt publishing, April 2016.
Les codes du livre sont disponibles sous : https://github.com/bpbpublications/Python-3-Image-Processing

Au lieu d'utiliser un "notebook de jupyter", il est possible d'utiliser l'IDLE ou **spyder3**.

Exécutez le programme *spyder3* disponible sous "Développement".

**Exercice 3.2 :**
Dans la partie "Éditeur", tapez le code suivant, pour tester le logiciel.

```
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(5)
print(x)
y=x
plt.plot(x, y, 'o--')
plt.plot(x, -y, 'o-')
plt.title('y=x et y=-x')
plt.show()
```

Exécutez pour voir dans la console IPython le résultat.
Sauvegardez le code précédent dans le dossier **adoc** sous le nom **spy0310_debut.py**

**Exercice 3.3 :**
Testons quelques opérations matricielles. Dans un nouveau fichier, tapez le code suivant :

```
print('-------- 3.3 -------------')
import numpy as np
A = np.array([[2, 3], [4, 5]], np.float64)
print(A)
B = np.array([[3, 5], [7, 9]], np.float64)
print(B)
C = A * B
print(C)
D = np.dot(A, B)
print(D)
print(type(D[0,0]))
vec = np.array([4, 6], np.float64)
print(np.dot(A, vec))
```

Exécutez pour voir dans la console IPython le résultat.
Sauvegardez le code précédent dans le dossier **adoc** sous le nom **spy0320_ioeratuibs_matricielles.py**
À quoi correspond les deux opérations A*B et np.dot(A, B) ?

**Exercice 3.4 :**
Ajoutez à la suite du code précédent :

```
print('-------- 3.4 -------------')
Ainv = np.linalg.inv(A)
print(Ainv)
print(np.dot(A, Ainv))
```

Exécutez pour voir dans la console IPython le résultat.
Sauvegardez le code précédent dans le dossier **adoc** sous le nom **spy0320_ioeratuibs_matricielles.py**
À quoi correspond la matrice *Ainv* ?

**Exercice 3.5 :**

° Écrivez sur papier sous forme matricielle le système linéaire de 2 équations à 2 inconnues suivant :

$$\begin{cases} 4v_1 - 2v_2 = 11 \\ 9v_1 - 5v_2 = 23 \end{cases}$$

° Écrivez sur papier la manière de résoudre ce système de manière matricielle.

° À l'aide du produit matriciel et de l'inversion de matrice vu précédemment, résolvez ce système linéaire de 2 équations à 2 inconnues.

° À l'aide du produit matriciel, vérifiez votre solution.

Sauvegardez votre code à la suite du précédent, ou dans un nouveau fichier.

**Exercice 3.6 :**

En supposant que vous avez nommé $A$ la matrice précédente et $b$ le vecteur <11; 23>, ajoutez à la suite du code précédent :

```
print('-------- 3.6 -------------')
v_sol = np.linalg.solve(A, b)
print(v_sol)
print(np.dot(A, Ainv))
```

Commentez ce code.

**Exercice 3.7 :**

° Écrivez sur papier sous forme matricielle le système linéaire de 3 équations à 3 inconnues suivant :

$$\begin{cases} x + y + z = -3 \\ x + 2y + 3z = -4 \\ 3x + 5y - 6z = 15 \end{cases}$$

° Écrivez sur papier la manière de résoudre ce système de manière matricielle.

° À l'aide du produit matriciel et de l'inversion de matrice vu précédemment, résolvez ce système linéaire de 3 équations à 3 inconnues.

° À l'aide du produit matriciel, vérifiez votre solution.

Sauvegardez votre code à la suite du précédent, ou dans un nouveau fichier.
Quel code avez-vous utilisé pour résoudre ce problème ?

**Exercice 3.8 :**

° Écrivez sur papier sous forme matricielle le système linéaire de 4 équations à 4 inconnues suivant :

$$\begin{cases} 4v_1 - 2v_2 + 4v_3 + 4v_4 = 12 \\ 2v_1 - 5v_2 + 7v_3 - 9v_4 = 13 \\ 3v_1 - 2v_2 + 3v_3 + 4v_4 = 8 \\ 7v_1 + 3v_2 - 3v_3 + 8v_4 = 6 \end{cases}$$

° À l'aide du produit matriciel et de l'inversion de matrice vu précédemment, résolvez ce système linéaire de 4 équations à 4 inconnues.

° À l'aide du produit matriciel, vérifiez votre solution.

Sauvegardez votre code à la suite du précédent, ou dans un nouveau fichier.

**Exercice 3.9 : Que se passe-t-il si le système linéaire d'équations n'a pas de solution ?**

° Soit le système d'équations :

$$\begin{cases} 4v_1 - 6v_2 = 17 \\ 2v_1 - 3v_2 = 8 \end{cases}$$

° Remarquez que le système d'équations ci-dessus n'a pas de solution.

° Tentez de résoudre ce système avec une méthode vue précédemment.
Que se passe-t-il ?

Pour évitez une plantée de Python, voici la manière standard de faire :

```
print('--------- 3.9 ------------')
A = np.array([[4, -6], [2, -3]], np.float64)
b = np.array([17, 8], np.float64)
try:
    v_sol = np.linalg.solve(A, b) # Résolution du système d'équations.
    print(v_sol)
except :
    print('Erreur, pas de solution.')
```

**Exercice 3.10 : Comment obtenir une solution approchée du système précédent ?**

Tapez le code suivant :

```
print('--------- 3.10 ------------')
A = np.array([[4, -6], [2, -3]], np.float64)
b = np.array([17, 8], np.float64)
v_sol = np.linalg.lstsq(A, b, rcond=None)[0] # Solution approchée.
print(v_sol)
print(np.dot(A, v_sol))
```

Remarquez que la solution obtenue ne satisfait pas les égalités désirées, mais presque.
"**lstsg**" signifie "Least Square Solution".

Le problème est changé en : $\|A \cdot v - b\|^2 = min$ .

Parmi les solutions $v$, on prend celle de norme minimale.
Ce problème a toujours une solution !

**Exercice 3.11 : Un autre exemple sans solution.**

° Soit le système d'équations :

$$\begin{cases} 4v_1 - 2v_2 + 4v_3 = 12 \\ 2v_1 - 5v_2 + 7v_3 = 13 \\ 6v_1 - 7v_2 + 11v_3 = 26 \end{cases}$$

° Remarquez que le système d'équations ci-dessus n'a pas de solution.

° Trouvez "la meilleure" solution à ce système.

**Exercice 3.12 : Systèmes sur dimensionnés.**

° Soit le système d'équations :

$$\begin{cases} 4v_1 - 2v_2 = 14 \\ 2v_1 - 5v_2 = -5 \\ 3v_1 - 2v_2 = 8 \end{cases}$$

Remarquez qu'habituellement, lorsqu'il y a plus d'équations que d'inconnues, le système d'équations n'a pas de solution.
Malgré cela, *np.linalg.lstsq* fournit une solution, qui n'est pas trop éloigné de ce qui est désiré, ce qui peut être très utile pour déterminer les paramètres d'un modèle, ayant plus de mesures que de paramètres.
Comme précédemment, trouvez la solution fournie par *np.linalg.lstsq*.
Vérifiez que la solution fournie est assez proche de ce que l'on peut espérer !

**Contexte :**

Ce qui précède montre comment résoudre des systèmes d'équations linéaires.
Ce qui suit utilise cette possibilité pour interpoler, approximer et déterminer les paramètres optimaux de divers modèles

**Exercice 3.13 : Droite passant par 2 points donnés.**

° Soient les deux points : P1 = ( –2 ; 1) et P2 = (3 ; 5).
On désire la droite passant par ces deux points.

a) Déterminez le système à résoudre.
b) Écrivez-le sous forme matricielle.
c) Déterminez l'équation de la droite passant par ces deux points.
d) Affichez les coefficients de l'équation de la droite. (a et b de y = a*x + b).
e) Affichez ces deux points dans un graphique.
f) Dans le même graphique, tracez la droite.
g) Afficher un titre et des légendes au graphique. Voici des instructions utiles :
c.f. https://matplotlib.org/tutorials/text/text_intro.html

```
plt.text(-3, 5.5, 'a = ' + '{:10.6f}'.format(a), color='black', fontsize=12)
plt.text(-3, 5.0, 'b = ' + '{:10.6f}'.format(b), color='black', fontsize=12)
```

**Exercice 3.14 : Droite passant au mieux 4 points donnés.**

° Soient les quatre points : P1 = ( –2 ; 1) ; P2 = ( 3 ; 5) ; P3 = ( 8 ; 9) ; et P4 = (13 ; 12)
On désire la droite passant "au mieux" par ces points.

a) Déterminez le système à résoudre.
b) Écrivez-le sous forme matricielle.
c) Déterminez l'équation de la droite passant "au mieux" par ces points.
d) Affichez les coefficients de l'équation de la droite. (a et b de y = a*x + b).
e) Affichez ces points dans un graphique.
f) Dans le même graphique, tracez la droite.
g) Afficher un titre et des légendes au graphique.

Voici un code assez général pour résoudre l'exercice 14 et des exercices similaires.

```
print('------ 3.14 ----------------')
points = np.array([[-2, 1], [3, 5], [8, 9], [13, 12]], np.float64) # Données
matA = np.array([np.ones(points.shape[0]), points[:,0]]).T
vecb = points[:,1]
v_sol = np.linalg.lstsq(matA, vecb, rcond=None)[0] # Solution approchée
print("Solution = ", v_sol)
b = v_sol[0]
a = v_sol[1]

xx = np.array([-3, 14]) # Abscisses des extrémités de la droite
yy = a*xx + b # Ordonnées des extrémités de la droite
plt.plot(points[:,0], points[:,1], 'o') # Affiche les points
plt.plot(xx, yy, '-') # Trace la droite
plt.title('droite passant "au mieux" par les points.')
plt.xlabel('x')
plt.ylabel('y = a*x + b')
plt.text(-2.5, 12, 'a = ' + '{:10.6f}'.format(a), color='black', fontsize=12)
plt.text(-2.5, 11, 'b = ' + '{:10.6f}'.format(b), color='black', fontsize=12)
plt.axis([-3, 14, 0, 14]) # Limites des axes
plt.show()
```

**Exercice 3.15 : Droite passant au mieux 5 points donnés.**

° Soient les points : P1=( –2 ; 13) ; P2=( 0 ; 10) ; P3=( 2 ; 8) ; P4=( 4 ; 7) ; et P5=(6 ; 5)
On désire la droite passant "au mieux" par ces points.

a) Déterminez le système à résoudre.
b) Écrivez-le sous forme matricielle.
c) Déterminez l'équation de la droite passant "au mieux" par ces points.
d) Affichez les coefficients de l'équation de la droite. (a et b de y = a*x + b).
e) Affichez ces points dans un graphique.
f) Dans le même graphique, tracez la droite.
g) Afficher un titre et des légendes au graphique.

**Exercice 3.16 : Détermination d'un polynôme d'interpolation.**

° Soient les 5 points définis par : ax = [-2; -1; 0; 1; 2]; ay=sin(ax);
On cherche les coefficients $c_i$, i allant de 0 à 4, qui satisfont :
$c_0 + c_1 \cdot ax[k] + c_2 \cdot ax[k]^2 + c_3 \cdot ax[k]^3 + c_4 \cdot ax[k]^4 = ay[k]$, pour k allant de 0 à 4.
Cela forme donc un système de 5 équations aux 5 inconnues $c_i$.

a) Écrivez sur une feuille le système sous forme matricielle !!!
b) Entrez les données : `ax = np.array([-2, -1, 0, 1, 2], , np.float64)`
`ay = np.sin(ax)`
c) Entrez la matrice `matA`
d) Résolvez le système pour obtenir les coefficients $c_0$ à $c_4$.
e) Affichez ces points dans un graphique.
f) Dans le même graphique, tracez la courbe du sinus, pour $x$ allant de – 2,5 à 2,5.
g) Dans le même graphique, tracez la courbe du polynôme, pour $x$ allant de – 2,5 à 2,5.
h) Afficher un titre et des légendes au graphique.

Voici un code assez général pour résoudre l'exercice 16 et des exercices similaires.

```
print('------ 3.16 ---------------')
ax = np.array([-2, -1, 0, 1, 2], np.float64) # Donnée
ay = np.sin(ax) # Donnée

# Création de la matrice
matA = np.zeros((ax.shape[0], ax.shape[0]), np.float64)
matA[:, 0] = np.ones(ax.shape[0]) # Première colonne de la matrice

# Remplissage des autres colonnes de la matrice
for jj in range(1, ax.shape[0]):
    matA[:, jj] = ax ** jj

coefs = np.linalg.lstsq(matA, ay, rcond=None)[0] # Coefficients du polynôme
print("Solution = ", coefs)

# Graphique, les points, le polynôme et le sinus.
xx = np.linspace(-2.5, 2.5, 81,endpoint=True) # Abscisses des points pour tracer le polynôme
yy = coefs[0] + coefs[1]*xx + coefs[2]*xx**2 + coefs[3]*xx**3 + coefs[4]*xx**4
plt.plot(ax, ay, 'o') # Affiche les points
plt.plot(xx, yy, '-') # Trace le polynôme
plt.plot(xx, np.sin(xx), '-') # Trace le sinus
plt.title('Polynôme passant par les points.')
plt.xlabel('x')
plt.ylabel('y = sin(x)')
plt.legend(['points', 'y=poly(x)', 'y=sin(x)'], loc='upper left')
plt.axis([-2.5, 2.5, -1.5, 1.5]) # Limites des axes
plt.show()
```

Python offre une méthode plus simple pour déterminer un polynôme passant au mieux par une série de points donnés. Voici le code :

```
ax = np.array([-2, -1, 0, 1, 2], np.float64) # Donnée
ay = np.sin(ax) # Donnée

coefs = np.polyfit(ax, ay, 4) # Coefficients du polynôme de degré 4
coefs = np.flip(coefs)
print("Solution = ", coefs)
```

La suite est comme précédemment.

**Exercice 3.17 : Polynôme passant au mieux par des points donnés.**

° Soient les points définis par : `ax = np.linspace(-2, 2, 21, np.float64)`
`ay = sin(ax)`
`nDeg = 3`
On cherche les coefficients $c_i$, *i allant de 0 à nDeg*, qui satisfont :
$c_0 + c_1 \cdot ax(k) + c_2 \cdot ax(k)^2 + ... + c_{nDeg} \cdot ax(k)^{nDeg} = ay(k)$, pour *k allant de 0 à 20*
Cela forme donc un système surdimensionné.

° Indiquez le nombre d'équations et le nombre d'inconnues.

° Grace à la fonction `np.polyfit` il n'y a pas besoin de déterminer la matrice du système.

a) Entrez les données : `ax=np.linspace(-2, 2, 21, np.float64); ay=sin(ax); nDeg=3`

b) Déterminez les coefficients $c_0$ à $c_{nDeg}$.

c) Affichez ces points dans un graphique.

d) Dans le même graphique, tracez la courbe du sinus, pour $x$ allant de – 2,5 à 2,5.

e) Dans le même graphique, tracez la courbe du polynôme, pour $x$ allant de – 2,5 à 2,5.

f) Afficher un titre et des légendes au graphique.

g) Dans un autre graphique, tracez la différence entre le polynôme et la fonction sinus.

En changeant le nombre de points (21), a-t-on une meilleure approximation ?
En changeant le degré du polynôme, a-t-on une meilleure approximation ?

**Exercice 3.18 : Approximation d'une fonction paire par une somme de cosinus.**
Dans certaines situations, on doit revenir au système matricielle pour déterminer les coefficients inconnus.

Le but est d'approximer la fonction $f(x)=e^{-5\cdot x^2}$ sur l'intervalle $[-\pi\,;\pi]$ par la fonction :

$$Fourier(x)=\sum_{j=0}^{n-1} a_j\cdot\cos(j\cdot x)\,.$$

Le nombre $n$ est un paramètre que l'on fixera à 7 pour la suite, les nombres $a_k$ sont à déterminer pour avoir une approximation optimale.
Une telle approximation s'appelle une approximation de Fourier.
On veut tracer dans un graphique les points ayant servis à calculer l'approximation, la courbe de la fonction *Fourier(x)* et dans un autre graphique la différence entre la fonction et son approximation.

a) Définissez un vecteur `ax` de 15 valeurs réparties régulièrement dans l'intervalle $[-\pi\ ;\ \pi]$.

b) Définissez le vecteur `aY` des 15 images des valeurs de `ax` par la fonction $f(x)=e^{-5\cdot x^2}$.

c) Sur papier, écrivez la matrice de ce système d'équations surdimensionné.

d) Déterminez les nombres $a_0$ jusqu'à $a_6$ de telle sorte à satisfaire au mieux les 15 équations :
$a_0+a_1\cdot\cos(x_j)+a_2\cdot\cos(2\cdot x_j)+a_3\cdot\cos(3\cdot x_j)+\ldots+a_6\cdot\cos(6\cdot x_j)=e^{-5\cdot x_j^2}$ pour
$x_j=-\pi+j\cdot\pi/7 \quad j=0$ *à* $14$ étant les valeurs définies en a).

e) Tracez les points $\left(x_j\ ;\ e^{-5\cdot x_j^2}\right)$ dans un graphique.

f) Tracez dans le même graphique la fonction $\sum_{j=0}^{n-1} a_j\cdot\cos(j\cdot x)$ pour $x$ allant de $-\pi$ à $\pi$.
coefs est le vecteur des valeurs $a_j$ déterminés au point d).

g) Tracez dans un autre graphique la différence entre la fonction $f(x)=e^{-5\cdot x^2}$ et son approximation de Fourier.

h) Que faut-il changer au programme si au lieu de 15 équations, on en voulait 75 ?
Cela améliore-t-il l'approximation ?