TP 14
Numpy et Matplotlib

Nous introduisons deux bibliothèques qui sont d'utilité fondamentale dans toutes les sciences des données (traiter des grands tableaux, matrices, millions de données, faire des calculs et des statistiques dessus) et qui contribuent au succès croissant de Python dans ces domaines. En effet pour cela on a besoin de pouvoir travailler rapidement et efficacement sur des grandes quantités de données, utilisant beaucoup de mémoire, et le langage Python est fondamentalement lent et peu adapté à cela car il fonctionne avec une surcouche qui exécute les instructions de l'utilisateur au fur et à mesure… Sauf si on manipule les bibliothèques permettant d'envoyer d'un seul coup les données au processeur et de les faire traiter avec toute la puissance de calcul disponible.
Pour faire le lien immédiat avec les cours de mathématiques, cela nous sera utile autant pour traiter des matrices, que pour traiter de fonctions, dérivées et intégrales.
Comme dans les TP du début d'année, **ne pas hésiter à faire des tests** en mode interactif avec ses propres valeurs et ses propres initiatives ; dans la première partie il s'agit essentiellement de découvrir la bibliothèque `numpy` et jouer avec mais il n'y a pas vraiment de programme à écrire.

I Tableaux numpy

Le module `numpy` sera chargé une bonne fois pour toute au début avec la ligne

```
import numpy as np
```

Cela permet d'utiliser ses fonctions avec le préfixe `np`. Rien n'oblige à utiliser ce préfixe, et on pourrait aussi en choisir un autre ; mais celui-ci est tout à fait standard et est recommandé dans la documentation elle-même.

I.1 Le type `ndarray`

La première utilité de `numpy` est de fournir un type `ndarray` qu'on appellera tout simplement un **tableau** (en anglais **array**) même si en dimension 1 cela ressemble à une simple liste. Une première façon d'en créer est simplement de convertir une liste :

```
>>> X = np.array([1, 3, 5, 7])
>>> print(X)
[1 3 5 7]
>>> type(X)
<class 'numpy.ndarray'>
>>> print(X[2])
5
```

Mais on peut aussi créer un tableau à deux dimensions (ou plus !) en donnant la **liste** de ses **lignes** :

```
>>> Y = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(Y)
[[1 2 3]
 [4 5 6]]
>>> print(Y[0, 2]) # élément ligne 0, colonne 2
3
```

qui correspond au tableau

1	2	3
4	5	6

(1)

Il s'agit en quelque sorte d'une convention : on pourrait très bien vouloir travailler en donnant la liste des colonnes. Mais cela s'avèrera bien pratique ainsi…
Les différences suivantes avec les listes sont essentielles :

1. Les tableaux `numpy` ont une taille **fixe**, donnée dès leur création. Oubliés les `append` et les concaténations ! C'est parce que l'ordinateur doit réserver dès le départ un bloc de mémoire de la bonne taille.

2. Les éléments d'un tableau `numpy` sont tous du même type (ou bien entier, ou flottant, ou…), déterminé là aussi dès la création, ce qui est nécessaire pour bien les arranger en mémoire. Il n'y a pas d'équivalent d'une liste `[1, "oui", False]`, qui ne peut pas être convertie en tableau. Le type des éléments de `X` peut être obtenu par sa variable `X.dtype` :

```
>>> X = np.array([1, 3, 5])
>>> X.dtype
dtype('int64')
>>> Y = np.array([1.0, 2.0, 3.0])
>>> Y.dtype
dtype('float64')
```

   Le premier de ces tableaux contient des nombres entiers signés codés sur 64 bits, le second des nombres à virgule flottante sur 64 bits aussi (flottants double précision standards).

3. Un tableau `X` peut avoir plusieurs dimensions (on dit **axes** dans le vocabulaire `numpy`) et une forme accessible par la variable `X.shape`. À une dimension, c'est presque comme une liste ; à deux dimensions, la forme correspond au nombre de lignes et de colonnes. Par exemple ci-dessus on doit avoir

```
>>> X = np.array([[1, 2, 3], [4, 5, 6]])
>>> X.shape
(2, 3)
```

   Dans le vocabulaire `numpy` il y a un axe de longueur 2 et un autre axe de longueur 3, et `Y.shape` est un tuple de longueur 2 (pour 2 axes). À trois axes, on peut y penser comme un empilement de tableaux de dimension 2 de la même taille.

On appellera fréquemment les tableaux `numpy` des **tableaux bas niveaux**, où l'expression « bas niveau » signifie proche du fonctionnement de l'ordinateur. Ils sont placés en mémoire de façon bien optimisée pour pouvoir accéder rapidement à n'importe quel élément et pouvoir effectuer des opérations sur tous les éléments en même temps, le plus rapidement possible.

I.2 Diverses manières de créer des tableaux

On pourra utiliser et tester les manières suivantes de créer des tableaux :

1. Convertir depuis une liste, ou une liste de listes, comme dans la section précédente.
2. Créer un tableau rempli de n zéros : `np.zeros(n)`

```
>>> X = np.zeros(10)
>>> print(X)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

3. Idem pour un tableau rempli de uns : `np.ones(n)`

```
>>> X = np.ones(10)
>>> print(X)
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

4. Créer un tableau des nombres entiers entre a et b : `np.arange(a, b)` (même syntaxe et convention que `range`, `b` est exclus) :

```
>>> X = np.arange(10)
>>> print(X)
[0 1 2 3 4 5 6 7 8 9]
```

5. Tableau de n nombres « linéairement espacés » entre a et b : `np.linspace(a, b, n)`, testez !

```
>>> X = np.linspace(1, 3, 11)
>>> print(X)
[1.  1.2 1.4 1.6 1.8 2.  2.2 2.4 2.6 2.8 3. ]
```

Nous verrons en intégration que cela correspond à la notion mathématique de **subdivision** d'un intervalle (et même ici, subdivision régulière), ne pas hésiter à l'utiliser même avec des grandes valeurs de `n`, voir la section suivante sur `matplotlib` ainsi que le cours 2 § II.3 sur les problèmes d'arrondis en nombres flottants.

Exercice 1 (Mathématiques). Si `X = np.linspace(a, b, n)`, quel est l'écart qui espace deux éléments consécutifs ? Quelle formule donne `X[i]` ?

Pour créer des tableaux à plusieurs axes, on peut :

1. Avec certaines fonctions `np.zeros`, `np.ones`, donner en argument non pas la longueur mais les dimensions voulues du tableau (attention, il y a deux paires de parenthèses : c'est un argument de type `tuple`) :

```
>>> X = np.zeros((3, 5))
>>> print(X)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
>>> Y = np.zeros((2, 3, 4))
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

  [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
```

   Ici `Y` est notre premier exemple de tableau tri-dimensionnel : c'est une superposition de 2 tableaux à 3 lignes et 4 colonnes, dont les éléments sont indicés `Y[i, j, k]` avec $0 \leqslant i < 2$, $0 \leqslant j < 3$ et $0 \leqslant k < 4$. Ainsi `Y[0, j, k]` correspond à la ligne j et la colonne k du premier tableau, et `Y[1, j, k]` au deuxième tableau.

2. Il existe aussi la méthode `X.reshape` qui permet de modifier la forme d'un tableau pour lui donner une forme voulue. Souvent on l'applique directement après la création du tableau :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> Y = np.arange(24).reshape(3, 2, 4)
[[[ 0  1  2  3]
 [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]
```

   Remarquez que cela marche bien car $24 = 4 \times 6 = 3 \times 2 \times 4$. Sinon, une erreur se produit...

Il est important de comprendre que les différentes façons d'arranger un tableau à plusieurs dimensions n'ont rien à voir avec la disposition des éléments en mémoire. Observons par exemples les deux tableaux suivants :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

(2)

Dans les deux cas, il s'agit de 15 cases de mémoire rangées consécutivement et numérotées de 0 à 14. Mais c'est *parce que* le premier est de dimensions $(3,5)$ que l'élément numéroté 8 correspond à la ligne 1 colonne 3, alors que dans le deuxième de dimensions $(5,3)$ il est en ligne 2 colonne 2. De même, la case $(2,1)$ correspond à l'élément 11 dans le premier tableau mais à 7 dans le deuxième. Bref, les éléments ne bougent pas, mais la forme change la façon dont on calcule les coordonnées lignes et colonnes des éléments !

Exercice 2 (Mathématiques)**.** Le cas général est le suivant : dans un tableau à n lignes et p colonnes $(n,p \geqslant 1)$, tout numéroté à partir de 0, on peut accéder à une case soit par ses indices (i,j) de lignes et colonnes, soit en les numérotant par un entier k « en ligne droite » comme ci-dessus. Comment calcule-t-on k à partir de (i,j) ? Et réciproquement ?

Ce procédé correspond en fait à deux applications bijectives, réciproques l'une de l'autre, entre $[\![0,n-1]\!]\times[\![0,p-1]\!]$ et $[\![0,n\times p-1]\!]\ldots$

I.3 Accès aux éléments

En dimension 1, la syntaxe pour accéder aux éléments d'un tableau `numpy` est la même que celle pour les listes. Le `i`-ème élément de `X` est `X[i]`, `len(X)` donne la longueur du tableau, et on peut utiliser la boucle `for` dessus :

```
for i in range(len(X)):
    ... X[i] ...
```

En dimension plus grande, on parle de l'élément de la ligne `i` et de la colonne `j` auquel on accède par `X[i, j]`. Il n'y a pas de longueur `len(X)` mais ici c'est `X.shape` qui est un tuple donnant la taille du tableau selon chaque axe. On récupère ses deux coordonnées avec la syntaxe `(n, p) = X.shape`.

Remarque 1. Voilà pourquoi on évite d'abuser du mot *dimension* : on confondrait le nombre d'axes (1 pour une liste, 2 pour un tableau au sens usuel, 3 pour un empilement de tableaux), avec la taille d'un axe (longueur d'une liste, nombre de lignes *ou* nombre de colonnes d'un tableau).

Comme pour les listes, on peut considérer des **tranches** dont on rappelle brièvement la syntaxe :

1. `X[a:b]` : sélectionne de l'indice `a` à l'indice `b` exclus.
2. `X[a:]` : sélectionne tout à partir de l'indice `a`.
3. `X[:b]` : sélectionne tout jusqu'à l'indice `b`.
4. On peut aussi utiliser des indices négatifs, par exemple `X[:-1]` correspond à tout sauf le dernier élément.
5. Un troisième argument dans la tranche permet de faire des *pas*, par exemple `X[::2]` prend un terme sur deux en partant de 0 (donc les indices pairs) alors que `X[1::2]` prend en terme sur deux mais en partant de 1 (ce sont donc les indices impairs).
6. Avec un pas négatif, cela correspond à parcourir le tableau en sens inverse. Ainsi `X[::-1]` est le tableau `X` rangé en sens inverse.
7. Éventuellement `X[:]` représente `X` tout entier.

Toutes ces opérations peuvent s'utiliser *indépendamment* sur chaque axe ! Observez :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> print(X[:3, 1:4]) # lignes 0, 1, 2, colonnes 1, 2, 3
[[ 1  2  3]
 [ 7  8  9]
 [13 14 15]]
>>> print(X[:, ::-1]) # même lignes, renverse l'indice colonnes
[[ 5  4  3  2  1  0]
 [11 10  9  8  7  6]
 [17 16 15 14 13 12]
 [23 22 21 20 19 18]]
```

Cela permet aussi de copier un bout de tableau dans un autre :

```
>>> X[-2:, -3:] = np.zeros((2, 3))
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14  0  0  0]
 [18 19 20  0  0  0]]
```

Comme dans la section précédente, il est important de comprendre qu'une tranche ne change rien du tout à la disposition des éléments dans la mémoire, mais seulement à la façon d'y accéder. Par exemple l'élément d'indice i de `X[1:]` est tout simplement l'élément d'indice $i+1$ de `X` ; ou bien l'élément d'indice i de `X[::-1]` est l'élément d'indice $n-1-i$ de `X`. Rien n'est tranché ni renversé dans la mémoire, mais c'est seulement un petit jeu de calcul sur les indices qui varie, et cela se combine très bien avec les tableaux à plusieurs axes.
Enfin notons que toutes ces opérations (`reshape` et tranches) ne copient pas le tableau mais créent des références, par exemple si on déclare `Y = X[a:b]` alors toute modification de `Y` va affecter `X`, et réciproquement. Si on veut vraiment copier un tableau pour en avoir un nouveau sans lien avec le précédent, on utilise la méthode `Y = X.copy()`.

I.4 Opérations sur les tableaux

Toutes les opérations arithmétiques habituelles ont été reprogrammées pour s'effectuer élément par élément sur des tableaux de mêmes dimensions. On parle d'**opérations vectorielles**.

```
>>> X = np.array([1, 3, 5])
>>> Y = np.array([1, 7, 2])
>>> Z = X + 2 * Y
>>> print(Z)
[ 3 17  9]
```

Sur des tableaux à deux dimensions, cela correspond exactement à l'addition de matrices et au produit par une constante !
La bibliothèque `numpy` contient aussi de nombreuses fonctions mathématiques usuelles : `np.exp()`, `np.sin()`, `np.cos()`, `np.arctan()`, `np.log()`, `np.sqrt()`, `np.pi`... qui s'appliquent sur chaque terme du tableau (y compris à plusieurs axes). Typiquement pour obtenir les valeurs de la fonction exponentielle sur $[0,1]$ en divisant cet intervalle en 100 points, on écrit simplement

```
>>> X = np.linspace(0, 1, 100)
>>> Y = np.exp(X)
```

Cette deuxième ligne est grossièrement équivalente à

```
for i in range(100):
    Y[i] = exp(X[i])
```

mais s'exécute beaucoup, beaucoup plus vite, un ordre de grandeur du millier de fois plus rapide ! C'est parce que les opérations vectorielles ont été programmées pour s'exécuter d'un seul coup sur tout le tableau en utilisant toutes les optimisations possibles de l'ordinateur, effectuant plusieurs opérations en parallèle, occupant la place mémoire et la vitesse du processeur disponible, là où Python seul serait obligé d'effectuer des opérations unes par unes à chaque passage dans la boucle.
La clé du bon programmeur Python/Numpy, c'est de savoir utiliser au maximum les opérations vectorielles combinées à toutes les manipulations ci-dessus, changer la forme, manipuler les axes, tranches, sans écrire de boucles !

II Représentations graphiques avec `matplotlib`

On charge maintenant, une bonne fois pour toute, le sous-module `pyplot` de la librairie de représentations graphiques `matplotlib` avec :

```
import matplotlib.pyplot as plt
```

Là encore cet alias est assez standard, d'autant plus qu'il évite d'écrire `matplotlib.pyplot` ce qui est un peu long...
La seule fonction que nous utiliserons vraiment est `plt.plot(X, Y)` qui prend au moins deux arguments : `X` et `Y` sont tous les deux, soit des listes, soit des tableaux `numpy`, de même taille, avec `X` une liste d'abscisses et `Y` une liste d'ordonnées. Les points vont être automatiquement reliés. Ensuite la fonction `plt.show()` permet d'afficher le graphique. Éventuellement, plusieurs appels successifs à `plt.plot` vont superposer les graphiques (attention alors à configurer correctement le style de points, de lignes, et la couleur).

II.1 Graphes de fonctions

Pour tracer le graphe d'une fonction f, il suffit de tracer un nombre suffisamment grand de points sur le graphe de f. Mais nous avons largement introduit tous les pré-requis pour cela. Par exemple supposons qu'on veuille tracer le graphe de la fonction carré sur l'intervalle $[-1, 3]$. On génère les points d'abscisse avec la fonction `np.linspace`, on calcule leur ordonnée directement avec les opérations vectorielles, et on fournit cela à `plt.plot()` ce qui donne le code tout simple :

```
# abscisses
X = np.linspace(-1, 3, 10)
# ordonnées
Y = X**2
# tracer
plt.plot(X, Y)
plt.show()
```

Si on fait varier le nombre de points dans `linspace`, on observe des courbes tracées plus ou moins finement.

Exercice 3. Tracer les graphes des fonctions suivantes, en faisant varier le nombre de points pour `linspace` :

1. $x \mapsto x^3 - 5x$ sur $[-4, 4]$
2. $x \mapsto \sin(x)$ sur $[0, 2\pi]$,
3. $x \mapsto e^x - 3x + 1$ sur $[-3, 3]$,

La bibliothèque `matplotlib` contient de nombreuses options pour tracer, des paramètres de couleurs et de points à relier, et des paramètres de la fenêtre d'affichage. Mais ce n'est pas vraiment notre priorité... Renvoyons vers l'aide officielle :
https://matplotlib.org/stable/tutorials/introductory/pyplot.html
ou vers le Mémo Agro-véto et citons seulement :

- `plt.title("titre")` : donne un titre à la fenêtre.
- `plt.xlabel("titre")` : donne un titre à l'axe des x.
- `plt.ylabel("titre")` : de même pour l'axe des y.
- `plt.xlim(a, b)` : fixe les bornes sur l'axes des x entre a et b (sans, les bornes sont ajustées automatiquement pour faire rentrer le graphique).
- `plt.ylim(a, b)` : de même pour l'axe des y.
- Un troisième argument passé à `plt.plot()` sous la forme d'une chaine de caractère permet à la fois de changer la couleur et le type de point, ainsi que le style du tracé (y compris pour savoir s'il faut ou non relier les points). Voir la documentation.

II.2 Courbes paramétrées

Une *courbe paramétrée* est le tracé de points x et y qui sont *tous les deux* des fonctions d'une même variable t. On note cela $t \mapsto (x(t), y(t))$. Il s'agit donc de calculer, pour des valeurs de t régulièrement espacées avec `linspace`, les deux tableaux `X` et `Y` puis afficher tous ces points calculés.

Exercice 4. Les *courbes de Lissajous* sont les courbes paramétrées définies la donnée de deux entiers non-nuls (p, q) et la paramétrisation

$$\begin{aligned} [0, 2\pi] &\longrightarrow \mathbb{R}^2 \\ t &\longmapsto \begin{cases} x(t) = \sin(pt) \\ y(t) = \cos(qt) \end{cases} \end{aligned} \tag{3}$$

Écrire une fonction `lissajous(p, q)` qui trace et affiche la courbe de Lissajous ci-dessus, et tester pour différentes valeurs de p et de q (le résultat dépend surtout du rapport p/q).

II.3 Dériver des fonctions

On rappelle que la dérivée d'une fonction f en un point x est définie comme :

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{4}$$

Pour obtenir un tableau `numpy` qui représente la dérivée d'une fonction, et représenter graphiquement la dérivée, on fait la chose suivante :

1. On part d'un tableau `X` obtenu avec `linspace`, représentant une subdivision d'un intervalle avec un pas suffisamment petit,
2. On construit le tableau `Y` des valeurs de la fonction f en chacun des points de `X`, comme précédemment, à l'aide des opérations vectorielles,
3. On construit un tableau `Z` dont les valeurs sont les `(Y[i+1]-Y[i])/(X[i+1]-X[i])`... attention, la longueur du tableau `Z` sera un de moins que celle de `X` et `Y`.

Exercice 5. Écrire une fonction `dérive(X, Y)` renvoyant le tableau `Z` ci-dessus.

Exercice 6. Représenter graphiquement les dérivées pour les fonctions de l'exercice 3, en testant avec différentes valeurs du nombre de subdivisions dans `linspace`.

Écrire les tableaux représentant les `X`, `Y`, `Z` ci-dessus, puis représenter graphiquement les dérivées, pour les fonctions de l'exercice 3. On mettra dans une variable `N` le nombre de subdivisions de l'intervalle (le paramètre passé à `linspace`, que l'on pourra faire varier depuis des toutes petites valeurs pour tester l'effet.

Remarque 2. On pourrait tracer l'un en dessous de l'autre la fonction et sa dérivée, dans une même fenêtre. Pour cela il faut créer des *subplots*, la syntaxe est la suivante :

```
plt.subplot(2, 1, 1) # deux lignes, une colonne, première figure
plt.plot(X, Y)
plt.subplot(2, 1, 2) # deux lignes, une colonne, deuxième figure
plt.plot(...)
plt.show()
```

Éventuellement il faut forcer à la main les limites des axes des abscisses avec `plt.xlim` et `plt.ylim` car les tableaux X et Z n'ont pas exactement la même longueur, pour que les deux figures soient bien alignées.

Exercice 7. Pouvez-vous écrire la fonction `dérive` *sans* boucle `for` mais en utilisant seulement les opérations `numpy` vues jusque là ?

II.4 Intégrer

Pour calculer l'intégrale d'une fonction continue f sur une intervalle $[a, b]$, on subdivise l'intervalle $[a, b]$. Sur un petit intervalle $[x, x+h]$, on trace un rectangle de hauteur $f(x)$. L'aire de ce rectangle approche bien l'intégrale de f sur $[x, x+h]$; et l'intégrale de f est la somme des aires de ces petits rectangles.
En résumé on fait la chose suivante pour calculer l'intégrale de f :

1. On part d'un tableau `X` représentant une subdivision d'un intervalle avec un pas suffisamment petit,

2. On construit le tableau `Y` des valeurs de la fonction f en chacun des points de `X`, comme précédemment,
3. On somme tous les `(X[i+1]-X[i]) * Y[i]`... là encore attention aux bornes.

Exercice 8.
1. Écrire une fonction `intègre(X, Y)` qui prend en argument, comme précédemment, deux tableaux `X`, `Y` représentant une fonction f sur un intervalle subdivisé, et qui renvoie l'intégrale de la fonction f obtenue par cette méthode.
2. Pouvez-vous écrire cette fonction *sans* boucle `for` ? Sachant que la fonction `np.sum(X)` calcule la somme de toutes les valeurs d'un tableau `X`.
3. (Mathématiques) Calculer $4 \times \displaystyle\int_0^1 \frac{1}{1+x^2}\,\mathrm{d}x$.
4. Tester la fonction `intègre` avec cette fonction-là, et une subdivision de plus en plus fine de l'intervalle $[0, 1]$.

II.5 Représenter une suite

Pour représenter les termes d'une suite récurrente $u_{n+1} = f(u_n)$ (graphique en escalier, ou en toile d'araignée), on peut faire les choses suivantes :

1. D'une part, tracer la fonction f et la droite d'équation $y = x$, avec `numpy` comme ci-dessus,
2. D'autre part, calculer les termes de la suite (dans une liste comme d'habitude).

On peut ensuite vouloir représenter les termes de la suite sur le graphique, en utilisant `plt.plot` mais sous forme de points non reliés cette fois ! On peut soit placer les termes de la suite seulement sur l'axe des abscisses, soit sur la courbe de f. Et tout cela avec des couleurs et des styles de points différents.

Exercice 9. Représenter graphiquement les suites suivantes définies par $u_{n+1} = f(u_n)$:

1. $u_{n+1} = \sqrt{2u_n + 3}$, $u_0 = 1$, sur $[-3/2, 4]$.
2. $u_{n+1} = \frac{1}{1+u_n}$, $u_0 = 1$, sur $[0, 2]$.

Pour tracer le graphique en escalier, il faut alors relier tous ces points, ce qui peut nécessiter de créer à part une liste des abscisses et une liste des ordonnées qu'on passera d'un seul coup à `plt.plot`. Attention à bien calculer les coordonnées successives des points, il y a une alternance une fois sur deux !