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 Python est fondamentalement lent et peu adapté à cela… sauf s'il manipule ces données *via* des bibliothèques conçues pour.
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 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])
>>> type(X)
>>> print(X)
>>> print(X[2])
```

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)
>>> print(Y[0,2]) # élément ligne 0, colonne 2
```

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**, fixée dès leur création. Oubliés les `append` et les concaténations !
2. Les éléments d'un tableau `numpy` sont tous du même type (ou bien entier, ou flottant, ou…) déterminé là aussi dès la création. 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 peut être obtenu par la variable `dtype` :

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

3. Un tableau peut avoir plusieurs dimensions (on dit **axes** dans le vocabulaire `numpy`) et une forme accessible par la variable `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

```
>>> Y = np.array([[1, 2, 3], [4, 5, 6]])
>>> Y.shape
(2, 3) # tableau à 2 lignes et 3 colonnes
```

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 même taille.

Ces contraintes reflètent en fait le fonctionnement profond de la mémoire de l'ordinateur : basiquement, celui-ci ne sait que réserver un gros bloc de cases mémoires consécutives pour y stocker des informations, et le bloc a une taille fixe et toutes les cases ont la même taille. Cela permet d'accéder très facilement au i-ème élément du tableau. Mais il n'est pas naturel de vouloir déplacer, concaténer, ou étendre ces blocs. Ainsi les tableaux `numpy` fonctionnent directement comme le fait la mémoire et sont nettement plus efficaces et plus rapides. On appellera fréquemment les tableaux `numpy` des **tableaux bas niveaux**, où l'expression « bas niveau » signifie proche du fonctionnement de l'ordinateur, par opposition à « haut niveau » qui signifie un langage ou un programme qui fournit beaucoup de facilités au programmeur mais en s'éloignant du fonctionnement profond de l'ordinateur. Pour en savoir plus, lire le cours 2 distribué !

I.2 Diverses manières de créer des tableaux

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

1. Conversion 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)
   ```

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

   ```
   >>> X = np.ones(10)
   >>> print(X)
   ```

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)
   >>> Y = np.arange(10, 21)
   >>> print(Y)
   ```

5. Tableau de n nombres aléatoires entre 0 et 1 : `np.random.random(n)` (oui, il y a deux fois `random`, car c'est la fonction `random()` du module `np.random` qui est un sous-module de `np`)

   ```
   >>> X = np.random.random(20)
   >>> print(X)
   ```

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

   ```
   >>> X = np.linspace(1, 5, 20)
   >>> print(X) # 20 nombres répartis entre 1 et 5
   ```

   Exercice 1. (mathématiques) Si `X = np.linspace(a, b, n)`, quelle formule donne `X[i]` ?

   Nous verrons que cela correspond à la notion mathématiques de *subdivision* (et même ici, subdivision régulière) d'un intervalle, 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.

7. Donner la forme (*shape*) voulue à un tableau : méthode `reshape`, prend en argument la nouvelle forme du tableau, testez par exemple :

   ```
   >>> X = np.arange(24)
   >>> print(X)
   >>> Y = X.reshape(4, 6) # converti en tableau de 4 lignes et 6 colonnes
   >>> print(Y)
   >>> Z = X.reshape(3, 2, 4) # superpositions de 3 tableaux de 2 lignes et 4 colonnes
   >>> print(Z)
   ```

   Remarquez que cela marche bien car $24 = 4 \times 6 = 3 \times 2 \times 4$. Sinon...

   ```
   >>> X.reshape(5, 6) # convertir 24 nombres dans un tableau 5×6, vous êtes sûr ???
   ```

   Souvent, la méthode `reshape` est appliquée directement après le création du tableau pour créer en une seule ligne un tableau de forme souhaitée, comme suit :

```
>>> X = np.zeros(15).reshape(3, 5)
>>> print(X)
>>> Y = np.random.random(14).reshape(2, 7)
>>> print(Y)
```

Remarque 1. En fait, lors d'une opération `reshape`, rien n'est modifié dans la mémoire, seule la façon d'accéder aux éléments change... par exemple les 6 nombres consécutifs $0, 1, 2, 3, 4, 5$ peuvent aussi bien être lus comme un tableau à une dimension de 6 nombres, ou bien le tableau à 2 lignes et 3 colonnes

0	1	2
3	4	5

(2)

auquel cas le nombre sur la ligne i et la colonne j est $3 \times i + j$ ($i = 0, 1$ et $j = 0, 1, 2$) ou bien comme le tableau à 3 lignes et 2 colonnes

0	1
2	3
4	5

(3)

auquel car l'élément de la ligne i et de la colonne j est $2 \times i + j$ ($i = 0, 1, 2$ et $j = 0, 1$). Avec un peu d'arithmétique et les connaissances sur l'adressage dans les blocs de mémoire (cf. cours 2, § III.2), tous les tableaux de toutes les formes peuvent être écrits avec leurs éléments consécutifs, et faire varier la forme ne fait que changer la manière d'accéder à un élément ! Autrement dit lors d'une opération `reshape` rien n'est ré-organisé dans la mémoire : les données sont toujours écrites consécutivement, mais la façon de les regarder change. Cela fait partie de la grande flexibilité des tableaux `numpy` de pouvoir voir le tableau de différentes manières sans avoir à ré-organiser toute la mémoire.

Exercice 2. (mathématiques) On se donne un tableau à n lignes et p colonnes ($n, p \geqslant 1$). Les lignes sont numérotées de 0 à $n-1$, et les colonnes de 0 à $p-1$. On écrit toutes les lignes à la suite, sur une seule ligne, ce qui donne une liste de $n \times p$ coefficients. Quelle est alors l'application qui prend un couple d'indices (i, j) correspondant à une case du tableau et qui donne l'indice k où cette case apparaitra dans la liste de toutes les lignes ? Et l'application qui prend un indice k dans la liste de toutes les lignes et qui donne à quelles indices (i, j) cela correspond dans le tableau ? Ces applications sont-elles bijectives ?

I.3 Accès aux éléments

En dimension 1 cela se passe exactement comme pour les listes : on accède au `i`-ème élément du tableau `X` par `X[i]`, `len(X)` donne la longueur du tableau et on peut utiliser la boucle `for` dessus :

```
X = np.linspace(3, 4, 20)
for i in range(len(X)):
    print(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.

Remarque 2. 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).

On peut alors faire des boucles, de la même façon. Tout ceci se généralise à des tableaux à plus de deux axes.

```
>>> X = np.array([[3, 6, 8], [12, 25, 32]])
>>> X[1,0]
>>> X[0,1]
>>> X[1,2]
>>> X.shape
```

Comme pour les listes, on peut considérer des **tranches** `X[a:b]` (on rappelle que `b` est exclus), mais même dans plusieurs dimensions en même temps !

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
>>> Y = X[1:3, 1:4] # lignes 1, 2, colonnes 1, 2, 3, de X
>>> print(Y)
```

Ici toute modification de `Y` va affecter `X` : dans le vocabulaire `numpy` on dit que `Y` est une **vue** (en anglais **view**) de `X`. Si la variable `X` représente effectivement une suite de 24 nombres, considérée comme arrangée en un tableau de 4 lignes et 6 colonnes, alors la variable `Y` ne contient pas vraiment de données mais seulement un moyen d'accéder à ces même nombres, comme si c'était un tableau de 2 lignes et 3 colonnes. Tester par exemple :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
>>> Y = X[1:3, 1:4]
>>> print(Y)
>>> Z = np.arange(50, 56).reshape(2, 3)
>>> print(Z)
>>> X[1:3, 1:4] = Z
>>> print(X)
```

ce qu'on écrit en pratique plus rapidement

```
>>> X = np.arange(24).reshape(4, 6)
>>> X[1:3, 1:4] = np.arange(50, 56).reshape(2, 3)
```

Remarque 3. De même que dans la remarque 1, si `Y` est une vue de `X` obtenue par tranche alors `Y` ne contient pas de données mais c'est seulement un petit jeu d'arithmétique qui permet d'accéder aux éléments contenus dans `X` à travers `Y`. Par exemple, tout simplement en dimension 1, si on pose `Y = X[1:]` alors l'élément `Y[i]` est en fait `X[i+1]` (le premier élément de `Y`, celui d'indice 0, est l'élément d'indice 1 de `X`). On peut bien sûr combiner les tranches et les changement de forme, selon différents axes : les données ne changent jamais de place dans la mémoire, il s'agit seulement de faire quelques calculs pour changer la façon d'accéder aux éléments.

Comme pour les listes, on a aussi les possibilités suivantes sur les tranches, et ceci sur chaque axe :

1. Sélectionner tout à partir de l'indice `a` : `X[a:]`
2. Sélectionner tout jusqu'à l'indice `b` (exclus) : `X[:b]`
3. L'indice -1 correspond au dernier élément. On peut donc sélectionner les k derniers indices avec `X[-k:]`, ou tout sauf les k derniers avec `X[:-k]`
4. Un troisième argument dans la tranche permet de faire des *pas*, par exemple `X[a:b:2]` saute les indices de 2 en 2 en partant de `a`.
5. Avec un pas négatif, cela correspond à parcourir le tableau en sens inverse. Ainsi `X[::-1]` est le tableau `X` rangé en sens inverse. Remarquez que dans ce cas l'élément d'indice `i` de `X[::-1]` est donc `X[n-1-i]` (si `n` est la longueur de `X`) : comme on l'a dit, on ne retourne rien dans la mémoire, on change seulement la façon d'y accéder.
6. ... et on peut combiner tout cela selon plusieurs axes indépendamment, trancher dans un sens ou dans l'autres selon des lignes et selon des colonnes.
7. Éventuellement `X[:]` represénte `X` tout entier. Cela permet de trancher selon certains axes, mais ne rien faire selon d'autres.

Exercice 3. Produire les tableaux suivants, de la façon la plus directe possible à partir des opérations vues précédemment (il n'est donc pas question de donner la liste des éléments uns par uns...) :

```
>>> print(X1)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
>>> print(X2)
[[0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 0.]
 [0. 1. 1. 1. 0.]
 [0. 1. 1. 1. 0.]
 [0. 0. 0. 0. 0.]]
>>> print(X3)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [ 0  0  0  0  0]]
>>> print(X4)
[[ 9  8  7  6  5  4  3  2  1  0]
 [19 18 17 16 15 14 13 12 11 10]
 [29 28 27 26 25 24 23 22 21 20]
 [39 38 37 36 35 34 33 32 31 30]
 [49 48 47 46 45 44 43 42 41 40]]
```

Nous n'aurons pas besoin d'en savoir beaucoup plus pour l'instant ; l'idée est de donner un aperçu des nombreuses possibilités de manipulations de tableaux à plusieurs dimensions. Il n'est pas attendu d'un élève de savoir faire tout cela de façon autonome et quand nous en aurons besoin cela sera largement guidé.
Si on veut vraiment copier un tableau, sans en créer une nouvelle vue, on utilise la méthode `copy()` :

```
>>> Y = X.copy()
```

I.4 Opérations sur les tableaux

Les opérations arithmétiques habituelles s'effectuent élément par élément sur un tableau par exemple pour la multiplication par une constante :

```
>>> X = np.arange(10)
>>> print(X)
>>> Y = 2*X
>>> print(Y)
```

ou pour l'addition :

```
>>> X = np.array([1, 3, 10])
>>> Y = np.array([2, 12, 25])
>>> Z = X + Y
>>> print(Z)
```

Sur des tableaux à deux dimensions, cela correspond donc 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) :

```
>>> X = np.arange(9, 26)
>>> print(X)
>>> Y = np.sqrt(X) # racine carrée de tous les nombres entre 9 et 25
>>> print(Y)
```

Bref, on peut faire les calculs tout comme si `X`, `Y` étaient de simples variables représentant des nombres, utiliser toutes les opérations habituelles et les fonctions `numpy` ci-dessus, et les calculs se font automatiquement sur tous les éléments du tableau.

Remarque 4. Bien sûr, on saurait très bien écrire les fonctions ci-dessus avec une simple boucle `for` qui calcule sur chaque case du tableau, une par une. Cependant les opérations fournies par `numpy` qui agissent d'un seul coup sur tout un tableau sont en général beaucoup, beaucoup plus rapides. En effet, au lieu de fournir uns par uns les éléments du tableau à une fonction à calculer, on fournit d'un seul coup tout le tableau et les fonctions `numpy` sont optimisée pour traiter de nombreux calculs en même temps en parallèle, ce que ne permet pas la boucle Python. On parle d'opérations **vectorielles** (ou : vectorialisées) pour signifier qu'on opère pas sur un seul élément à la fois mais sur tout un tableau d'un seul coup. Les opérations vectorielles peuvent être des milliers de fois plus rapides si elles sont bien utilisées. 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) ce qui exploite les capacités bas niveau et ultra optimisées des bibliothèques sans attendre que ce soit Python qui traite tout lentement et séquentiellement ! Idéalement même, on peut tout écrire *sans* utiliser de boucle `for`. Mais pour la pédagogie, ce n'est pas encore l'objectif...

Nous aurons d'autres applications dans la section suivante.

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 rien n'oblige à utiliser l'alias `plt` mais ceci est absolument standard et recommandé dans la documentation. Et puis sinon, toutes les fonctions que nous allons utiliser devraient être précédées de `matplotlib.pyplot` ce qui est un peu long à écrire...

II.1 Graphiques simples

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 ; `X` est une liste d'abscisses et `Y` une liste d'ordonnées, pour des points, qui vont ensuite être automatiquement reliés. Ensuite la fonction `plt.show()` permet d'afficher le graphique. Essayez par exemple :

```
plt.plot([1, 2, 3], [1, 5, 2])
plt.show()
```

et vérifiez qu'il s'agit bien des points de coordonnées $(1,1)$, $(2,5)$, $(3,2)$ reliés, dans l'ordre dans lequel ils sont donnés. La fenêtre est automatiquement ajustée pour faire rentrer toute la figure.

Exercice 4. Tracer un carré de côté 2 dont le coin bas gauche a pour coordonnée $(1,0)$.

II.2 Graphe 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, 5)
# ordonnées
Y = X**2
# tracer
plt.plot(X, Y)
plt.show()
```

Normalement, le dessin n'est pas très joli : on voit bien les 5 points... mais il suffit de reprendre en augmentant peu à peu cette valeur (par exemple 10, puis 30).

Exercice 5. Tracer les graphes des fonctions suivantes, en ajustant soi-même la subdivision pour que le graphe soit assez fluide :

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
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. De même il y a `plt.ylabel("titre")`
- `plt.xlim(a, b)` : fixe les bornes sur l'axes des x entre a et b. De même il y a `plt.ylim(a, b)`
- 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. Voir la documentation.

III Dériver et intégrer

III.1 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 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 6. Écrire les tableaux représentant les `X`, `Y`, `Z` ci-dessus, puis représenter graphiquement les dérivées, pour les fonctions de l'exercice 5. 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.

Exercice 7. Au lieu d'une boucle... pouvez-vous *vectorialiser* l'opération qui consiste à calculer les `(Y[i+1]-Y[i])/(X[i+1]-X[i])`

Remarque 5. 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(X[:-1], Z)

plt.show()
```

Éventuellement il faut forcer à la main les limites des axes des abscisses (les tableaux X et Z n'ont pas exactement la même longueur) pour que les deux figures soient bien alignées.

III.2 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.

FIGURE 1 – Dessin de la méthode des rectangles : au tableau

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. Donner des approximations des intégrales suivantes (plus le pas est petit, plus l'approximation est précise).

1. $\int_0^1 \frac{\mathrm{d}x}{1+x^2}$, puis multiplier le résultat par 4,
2. $\int_1^2 \frac{\mathrm{d}x}{x}$
3. $\lim_{A\to+\infty} \int_0^A e^{-x^2}\,\mathrm{d}x$ (on prend A le plus grand possible)

Exercice 9. Idem, pouvez-vous vectorialiser ces opération ? La bibliothèque `numpy` contient la fonction `np.sum()` qui calcule la somme de tous les éléments d'un tableau. Comparer son utilisation avec la somme calculée avec une boucle `for` pour un million, puis pour dix millions, de subdivisions.