Chapitre 1

La bibliothèque NumPy

1.1 Introduction

NumPy est une bibliothèque (*library*) permettant la manipulation de **tableaux multidimensionnels** (vecteurs, matrices, etc.). Cette bibliothèque comprend une *vaste collection de fonctions mathématiques* pouvant opérer sur ces tableaux.
L'élément central de NumPy est le `ndarray`, un **tableau à n dimensions**. Ce tableau, à l'instar des listes de Python est un *container mutable*. En revanche, contrairement à une liste, cette structure est **homogène** (tous les éléments sont du même type) et sa **taille est fixée** à sa création. Ces contraintes permettent d'optimiser les calculs et un tableau NumPy autorise des **opérations "vectorisées"** où tout le "vecteur" (tableau) est traité comme une unité. L'exécution de ces opérations vectorisées est beaucoup plus efficace que le traitement séparé de chaque opération. Cette approche rend la bibliothèque NumPy **particulièrement performante**. Cette dernière a ainsi par exemple été l'un des éléments-clés ayant permis d'obtenir la première image d'un trou noir en 2019 (Trou noir M87, *Event Horizon Telescope Collaboration*) :

L'histoire de cette image et la description détaillée de la bibliothèque NumPy peuvent être consultées en ligne à l'adresse https://numpy.org/.

Une bonne manière d'importer la bibliothèque `numpy` est d'avoir en préambule d'un notebook Python la ligne :

```
import numpy as np
```

Une fonction `fct` de `numpy` doit alors être appelée par `np.fct`.
La touche `<TAB>` permet (dans presque tous les environnements) d'obtenir facilement les fonctions disponibles. Par exemple, `np.t <TAB>`, permet d'avoir accès à une trentaine de fonctions, parmi lesquelles on trouve la fonction `np.tan`. De plus amples informations (*docstrings*) sur la fonction `np.tan` (par exemple) peuvent ensuite être obtenues en utilisant le point d'interrogation :

```
np.tan?
```

1.2 Tableaux unidimensionnels

Les tableaux **en une dimension**, que l'on peut considérer comme étant des **"vecteurs"**, c'est-à-dire de **simples séquences de nombres**, sont des éléments essentiels de tout calcul scientifique.

1.2.1 Construction à partir d'une fonction de NumPy

La bibliothèque Numpy offre plusieurs fonctions permettant de construire de tels tableaux unidimensionnels :

1. **La fonction `np.linspace`** Des tableaux de réels (*floats*) **également espacés** peuvent être créés à l'aide de la fonction `np.linspace(start,stop,num)`. Ainsi, le code

```
import numpy as np
x = np.linspace(-2,3,10,endpoint=True,retstep=False)
print(x)
```

construit un **tableau de taille 10** (i.e. un tableau contenant 10 éléments (nombres) ; 50 étant la taille par défaut) :

```
[-2.         -1.44444444 -0.88888889 -0.33333333  0.22222222
 ↪  0.77777778
  1.33333333  1.88888889  2.44444444  3.        ]
```

→ le premier élément est `x[0]=-2`.
→ le dernier élément est `x[-1]=3` (`endpoint=True`, valeur par défaut)

→ le **pas** entre chaque valeur est alors donné par $\frac{3-(-2)}{10-1}$.

Si `retstep=True`, la fonction retourne un **tuple** renfermant le tableau et le pas :

```
import numpy as np
x_vecteur,dx=np.linspace(-2,3,10,endpoint=False,retstep=True)
print(x_vecteur,dx)
print(type(x_vecteur))
print(type(x_vecteur[1]))
print(type(dx))
```

```
[-2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2.   2.5] 0.5
<class 'numpy.ndarray'>
<class 'numpy.float64'>
<class 'numpy.float64'>
```

→ le **pas** entre chaque valeur est ici donné par $\frac{3-(-2)}{10}$.

2. **La fonction `np.logspace`**

   La fonction `np.logspace(start,stop,num)` produit quant à elle un tableau de N nombres **également espacés sur une échelle logarithmique**. Les arguments `start` et `stop` se refèrent cette fois à une puissance de 10 : le tableau commence à 10^{start} et se termine à 10^{stop}.

3. **La fonction `np.arange`**

   La fonction `np.arange(start,stop,step)` (*"ArrayRANGE"*) fournit une troisième manière de créer un tableau de nombres. Cette fonction est similaire à la fonction `range` (définie au semestre d'automne) permettant de créer des listes. Il est possible d'omettre le premier argument (qui prend alors la valeur 0) et/ou le troisième argument (qui prend alors la valeur 1).

   Cette fonction permet ainsi par exemple de construire un vecteur renfermant les valeurs réelles comprises entre `start=-2` et `stop=3` (sans cette dernière valeur) séparées de la distance `step=0.2` :

```
import numpy as np
x = np.arange(-2,3,0.2)
print(x)
```

```
[-2.0000000e+00 -1.8000000e+00 -1.6000000e+00 -1.4000000e+00
 -1.2000000e+00 -1.0000000e+00 -8.0000000e-01 -6.0000000e-01
 -4.0000000e-01 -2.0000000e-01 -4.4408921e-16  2.0000000e-01
  4.0000000e-01  6.0000000e-01  8.0000000e-01  1.0000000e+00
```

```
  1.2000000e+00  1.4000000e+00  1.6000000e+00  1.8000000e+00
  2.0000000e+00  2.2000000e+00  2.4000000e+00  2.6000000e+00
  2.8000000e+00]
```

4. **Les fonctions `np.zeros`, `np.ones` et `np.empty`**

   Parfois, il peut être utile de construire un vecteur dont toutes les composantes valent 0, 1 ou ne sont pas initialisées (explicitement) :

```
import numpy as np
v_zero = np.zeros(3)
v_un = np.ones(5, dtype=int)
v_vide = np.empty(2)
print(v_zero, v_un, v_vide)
```

```
[0. 0. 0.] [1 1 1 1 1] [-5.73021895e-300 -2.68156159e+154]
```

   Les fonctions `np.zeros(shape)`, `np.ones(shape)` et `np.empty(shape)` ont chacune un argument obligatoire qui représente la forme du tableau (en fait, le plus souvent, il s'agit du nombre d'éléments dans le tableau), et un argument optionnel `dtype` qui spécifie le type des données du tableau (`bool`, `int`, `float` (défaut) ou `complex`).

5. **La fonction `np.array`**

   Finalement, un tableau peut être créé à l'aide de la fonction `np.array(object)` avec pour arguments un *container* (liste, tuple ou tableau) et éventuellement un paramètre `dtype` :

```
import numpy as np
print(np.array([-1,0,2.],dtype=complex))
```

```
[-1.+0.j  0.+0.j  2.+0.j]
```

   Si l'argument `dtype` est omis, la fonction `np.array` promeut automatiquement tous les éléments du tableau au type de l'entrée la plus générale du tableau (qui serait dans l'exemple ci-dessus le type `float`).

   Comme le montrent les quelques lignes de code suivantes, Numpy apporte à Python une grande diversité de **types numériques** :

```
import numpy as np
x=0.12345678912345678912345678 9
#
print('x =',x)
print('Python (default) type :',type(x))
print('\n')
#
y=np.array([0.12345678912345678912345678 9])
print('y =',y)
print('y[0] =',y[0])
print('y (Numpy default) type :',type(y))
print('y (Numpy default) data type :',y.dtype)
print('y[0] (Numpy default) type :',type(y[0]))
print('\n')
#
z16=np.array([0.12345678912345678912345678 9],dtype=np.float16)
print('NumPy, 16bits type :',z16.dtype)
print('z[0] =',z16[0])
print('\n')
#
z32=np.array([0.12345678912345678912345678 9],dtype=np.float32)
print('NumPy, 32bits type :',z32.dtype)
print('z[0] =',z32[0])
print('\n')
#
z64=np.array([0.12345678912345678912345678 9],dtype=np.float64)
print('NumPy, 64bits type :',z64.dtype)
print('z[0] =',z64[0])
```

Dans la sortie produite, on note que le type `numpy.float64` (ou `numpy.double`) fournit la même précision que le type `float` natif :

```
x = 0.12345678912345678
Python (default) type : <class 'float'>


y = [0.12345679]
y[0] = 0.12345678912345678
y (Numpy default) type : <class 'numpy.ndarray'>
y (Numpy default) data type : float64
y[0] (Numpy default) type : <class 'numpy.float64'>


NumPy, 16bits type : float16
z[0] = 0.1235

```

```
15
16 NumPy, 32bits type : float32
17 z[0] = 0.12345679
18
19
20 NumPy, 64bits type : float64
21 z[0] = 0.12345678912345678
```

1.2.2 Construction à partir d'un objet préexistant

Supposons que l'on définisse un vecteur `x` dont les composantes sont, par exemple, régulièrement espacées. Il peut être utile de définir un vecteur de la même taille et du même type que `x`. Trois constructeurs différents permettent d'obtenir un vecteur non initialisé ("vide"), un vecteur rempli de 0 et un vecteur ne contenant que des 1 : `np.empty_like()`, `np.zeros_like()` et `np.ones_like()`.
Ainsi, le code

```
1 import numpy as np
2 x = np.linspace(0,9,10, dtype=int)
3 y_vide = np.empty_like(x)
4 y_zero = np.zeros_like(x)
5 y_un = np.ones_like(x)
6 print(y_vide, y_zero, y_un)
```

produit la sortie suivante :

```
1 [                  0 4607182418800017408 4611686018427387904
2  4613937818241073152 4616189618054758400 4617315517961601024
3  4618441417868443648 4619567317775286272 4620693217682128896
4  4621256167635550208] [0 0 0 0 0 0 0 0 0 0] [1 1 1 1 1 1 1 1 1 1]
```

1.2.3 Opérations arithmétiques sur les tableaux

En Python, il est possible d'additionner, de soustraire, de multiplier ou de diviser **deux vecteurs de même taille**. Durant ces opérations, la $i$ème composante du vecteur résultant est obtenue par l'addition, la soustraction, la multiplication ou la division des $i$ème composantes des deux vecteurs :

```
1 import numpy as np
2 v_1 = np.array([-1,2,4])
3 v_2 = np.linspace(1,2,3)
4 print(v_1, v_2)
```

```
print(v_1+v_2, v_1-v_2)
print(v_1*v_2, v_1/v_2)
```

```
[-1  2  4] [1.  1.5 2. ]
[0.  3.5 6. ] [-2.   0.5  2. ]
[-1.  3.  8.] [-1.         1.33333333  2.        ]
```

Les opérations d'addition et de soustraction de deux vecteurs sont identiques à celles définies en géométrie euclidienne. En revanche, avec des vecteurs habituels, le produit (scalaire) entre deux vecteurs, par exemple $\vec{v}_1 = (a, b)$ et $\vec{v}_2 = (c, d)$, fournit un scalaire :

$$\vec{v}_1 \cdot \vec{v}_2 = ac + bd\,,$$

alors que dans NumPy le produit de deux `ndarray` est un tableau de même taille. "Vectoriellement", cela aurait la signification suivante :

$$v\_1 * v\_2 = [ac, bd]\,.$$

Remarquons également qu'en géométrie, la division d'un vecteur par un autre n'a pas de sens, alors que c'est une opération bien définie en Python :

$$v\_1/v\_2 = [a/c, b/d]\,.$$

En géométrie, on est fréquemment amené à multiplier un vecteur par un scalaire. Comme le montre le code suivant, NumPy autorise le même type de manipulations :

```
import numpy as np
v = np.array([-1,2,4])
print(5*v, v*5)
print(v/5, 5/v)
print(v**4)
print(v+5)
```

```
[-5 10 20] [-5 10 20]
[-0.2  0.4  0.8] [-5.    2.5   1.25]
[  1  16 256]
[4 7 9]
```

De plus, NumPy fournit des fonctions appelées **universal functions** (ou simplement **ufuncs**) qui peuvent être appliquées à un scalaire, de manière à produire un scalaire, ou à un tableau de façon à générer un tableau de même taille (en s'appliquant alors à chaque composante). Parmi ces *ufuncs*, on trouve notamment les **fonctions**...

- ... **trigonométriques** (`sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`)

- ... **hyperboliques** (`sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`)
- ... **exponentielle** (`exp`)
- ... **logarithmes** (`log`, `log10`)
- ... **racine carrée** (`sqrt`)
- ... **permettant de manipuler des nombres complexes** (`angle`, `real`, `complex`, `conj`)
- ... **signe** (`sign`)
- ... **valeur absolue** (`abs`)

Ainsi, il est par exemple possible de construire très facilement un vecteur `y` dont les composantes sont les racines carrées des composantes correspondantes d'un vecteur `x` :

```
import numpy as np
x = np.linspace(0,9,10)
y = np.sqrt(x)
print(x)
print(y)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
[0.         1.         1.41421356 1.73205081 2.         2.23606798
 2.44948974 2.64575131 2.82842712 3.        ]
```

Notons que Python permet le calcul de la racine carrée d'un nombre négatif pour autant que l'on manipule des nombres complexes.
Rappel : La racine carrée de -1 est le nombre complexe i : $i = \sqrt{-1}$. En Python, ce nombre se note `j`.
Dans le code suivant, la troisième ligne produit un "warning" (et retourne une composante `nan` (*not a number*)) parce que les éléments de `v` sont des nombres réels. La cinquième ligne en revanche s'exécute sans problème, les éléments de `a*v` étant complexes.

```
import numpy as np
v = np.array([-1,2,4])
print(np.sqrt(v))
a = complex(1,0)
print(np.sqrt(a*v))
```

```
[       nan 1.41421356 2.        ]
[0.        +1.j 1.41421356+0.j 2.        +0.j]
```

Dans l'exemple suivant, NumPy calcule le logarithme où il le peut, et retourne `nan` si l'opération est illégale (calcul du logarithme d'un nombre négatif) et `-inf` pour le logarithme de zéro. Les autres valeurs dans le tableau sont retournées correctement :

```
import numpy as np
b = np.arange(-2.,4,1)
print(np.log(b))
```

```
[       nan        nan       -inf 0.         0.69314718
 ↪  1.09861229]
```

Nous savons qu'une expression telle que $\vec{v} > 0$, où $\vec{v}$ est un vecteur n'a pas de sens. Dans le cas d'un tableau unidimensionnel de NumPy, une telle opération de comparaison est autorisée, **élément par élément**. On obtient un vecteur contenant les valeurs booléennes (valeurs "de vérité") `True` ou `False` :

```
import numpy as np
v = np.linspace(-2,2,9)
y = v > 0
print(v)
print(y)
```

```
[-2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2. ]
[False False False False False  True  True  True  True]
```

Un vecteur "logique" tel que le vecteur `y` ci-dessus permet par exemple de calculer facilement la valeur absolue de `x` :

```
import numpy as np
v = np.linspace(-2,2,9)
w = v
w[w<0]=-w[w<0]
print(w)
print(v)
```

```
[2.  1.5 1.  0.5 0.  0.5 1.  1.5 2. ]
[2.  1.5 1.  0.5 0.  0.5 1.  1.5 2. ]
```

Remarquons que pour conserver le vecteur `v` original, il faudrait remplacer la troisième ligne par `w = v.copy()`.

1.2.4 Indexation et découpage (*slicing*) des tableaux

Les tableaux peuvent être **découpés** de la même manière que les listes et on **accède à un élément** d'un tableau unidimensionnel grâce à l'indice de l'élément placé entre des crochets qui suivent l'identificateur du tableau.
Pour comprendre comment fonctionne ce découpage, nous allons imaginer une expérience de chute libre, durant laquelle la distance verticale `y` (en mètres) parcourue par un objet est mesurée en fonction du temps `t` (en secondes). Les mesures permettent alors de remplir les deux tableaux suivants :

```
import numpy as np
y = np.array([0,1.31,4.99,10.9,19.7,29.8,43.9])
t = np.array([0,0.51,1.01,1.49,2.,2.5,2.99])
```

Il est possible de calculer la vitesse moyenne de l'objet en s'intéressant au taux de variation de la position par rapport au temps :

$$v_{\text{moy},i} = \frac{y_i - y_{i-1}}{t_i - t_{i-1}}\,, \quad i \geq 1\,,$$

où les y_i et les t_i sont les éléments des tableaux `y` et `t`.
Pour ce faire, nous envisageons le découpage des tableaux `y` et `t`. Par exemple, pour `y`, nous allons considérer

```
print(y) # tableau complet des mesures de distance
print(y[1:]) # tableau amputé de la première mesure de distance
print(y[:-1]) # tableau amputé de la dernière mesure de distance
```

```
[ 0.    1.31  4.99 10.9  19.7  29.8  43.9 ]
[ 1.31  4.99 10.9  19.7  29.8  43.9 ]
[ 0.    1.31  4.99 10.9  19.7  29.8 ]
```

et faire la différence **élément par élément** (idem pour le tableau `t`) :

```
v_moyenne = (y[1:]-y[:-1])/(t[1:]-t[:-1])
print(v_moyenne)
```

```
[ 2.56862745  7.36       12.3125     17.25490196 20.2
28.7755102 ]
```

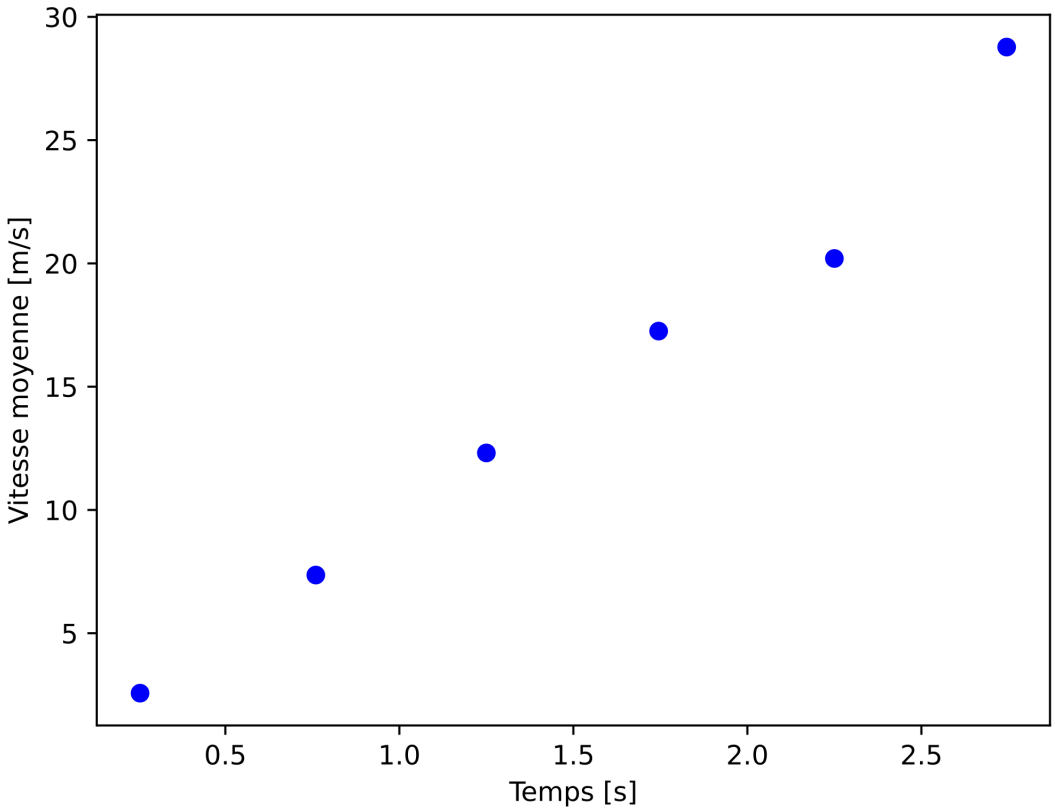
La division se fait également élément par élément et le tableau `v_moyenne` obtenu contient un élément de moins que les tableaux `y` et `t` de départ. Il est alors naturel d'associer

les vitesses obtenues à un nouveau tableau temporel dont les entrées correspondent à une moyenne entre deux mesures de temps successives :

```
t_moyenne = (t[1:]+t[:-1])/2
print(t_moyenne)
```

```
[0.255 0.76  1.25  1.745 2.25  2.745]
```

Les tableaux `v_moyenne` et `t_moyenne` renferment ainsi le même nombre d'éléments. Nous verrons au chapitre suivant comment représenter de telles données expérimentales :



Une telle représentation permet d'avoir rapidement une "vérification" visuelle du comportement (linéaire) de la vitesse en fonction du temps dans le cas de la chute libre :

$$v(t) = A\,t\,,\text{ où } A \text{ est une constante}\,.$$

1.3 Attributs caractérisant les tableaux

Trois attributs permettent de caractériser un tableau NumPy à une, deux ou n dimensions :

- `ndim` : le **nombre de dimensions** (ou le **nombre d'axes**) du tableau ;
- `shape` : un tuple de taille `ndim` qui donne l'**extension** (la longueur) du tableau le long de chaque axe ;
- `dtype` : le **type** des éléments du tableau.

Ainsi, pour le cas particulier d'un **tableau unidimensionnel** (**vecteur**),

```
import numpy as np
x = np.linspace(-2,3,10)
```

nous avons affaire à un objet **à une dimension**, **de forme** $(10,)$ et dont les éléments sont des **nombres en virgule flottante** :

```
print(x.ndim)
print(x.shape)
print(x.dtype)
```

```
1
(10,)
float64
```

On note que l'attribut `shape` est bien un tuple.

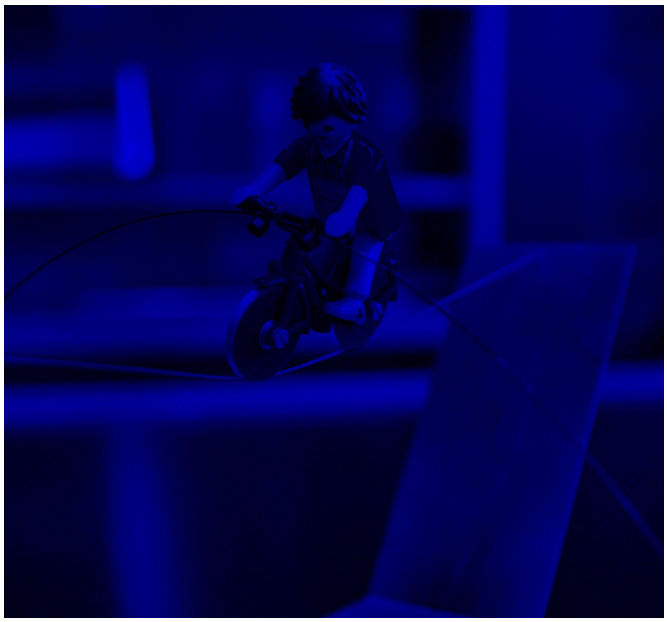
1.4 Tableaux multidimensionnels et matrices

Si les tableaux à une dimension sont omniprésents dans le calcul scientifique, les tableaux multidimensionnels revêtent également une grande importance.

Ainsi, par exemple, une image en niveaux de gris (ou de bleus) peut être représentée par un tableau **bidimensionnel** (que l'on appelle également parfois une **"matrice"**), alors qu'une image en couleurs correspond à un tableau de **dimension trois**.

Une image en couleurs peut être représentée par un tableau de dimension trois. Les deux premières dimensions fournissent la localisation (les coordonnées) des pixels colorés de l'image, alors que la troisième dimension donne l'information de couleur. Dans cette troisième dimension, on trouve trois nombres (entre 0 et 255) correspondant à la proportion (luminosité) de rouge, de vert et de bleu (canaux **RGB** en anglais) permettant de reconstruire la couleur du pixel concerné. Cette façon de faire permet de produire $256 \cdot 256 \cdot 256 = 16777216$ couleurs différentes ! Parfois, un quatrième nombre est ajouté dans cette troisième dimension pour avoir une information de transparence. On parle alors des **quatre canaux RGBA** de l'image, où A désigne le quatrième canal appelé également *canal alpha*.

Une image en niveaux de gris (ou en niveaux de bleu) peut être représentée par un tableau de dimension deux. En effet, l'information de couleur se résume alors à un seul nombre et une simple matrice permet de reconstuire l'image.



1.4.1 Construction à partir d'une fonction de NumPy

Numpy offre plusieurs fonctions permettant de construire des tableaux multidimensionnels.

1. **La fonction `np.array`**
   Il est possible de créer un tableau NumPy en convertissant une liste à l'aide de la fonction `np.array(object)` :

```
import numpy as np
x = np.array([[1,2,3], [4,5,6]])
print(x)
print(x.ndim,x.shape,x.dtype)
```

```
[[1 2 3]
 [4 5 6]]
2 (2, 3) int64
```

A la deuxième ligne du code ci-dessus, deux listes à une dimension, $[1, 2, 3]$ et $[4, 5, 6]$, sont placées dans des crochets pour créer une liste à deux dimensions. Lors de la conversion de cette liste par la fonction `np.array`, les éléments prennent le type le plus général rencontré en parcourant la liste.

2. **Les fonctions `np.zeros`, `np.ones` et `np.empty`**

   Les fonctions `np.zeros(shape)`, `np.ones(shape)` et `np.empty(shape)` que nous avons déjà rencontrées s'appliquent également dans le contexte plus général des tableaux multidimensionnels. Ainsi, le code

```
import numpy as np
print(np.zeros((3,20), dtype=int))
```

produit un tableau possédant 3 lignes et 20 colonnes dont toutes les entrées sont posées égales à 0, 0 étant ici un entier :

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

L'argument obligatoire de ces fonctions est un tuple dont le nombre d'éléments fournit la dimension du tableau.

En observant le résultat produit par les trois lignes suivantes

```
import numpy as np
y = np.ones((2,3,1,8))
print(y,y.ndim,y.shape, sep='      ')
```

```
[[[[1. 1. 1. 1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1. 1. 1. 1.]]]
```

```
4
5    [[1. 1. 1. 1. 1. 1. 1. 1.]]]
6
7
8   [[[1. 1. 1. 1. 1. 1. 1. 1.]]
9
10   [[1. 1. 1. 1. 1. 1. 1. 1.]]
11
12   [[1. 1. 1. 1. 1. 1. 1. 1.]]]]      4      (2, 3, 1, 8)
```

on note que l'axe correspondant au dernier élément du tuple est **affiché horizontalement**.

3. **La fonction `np.eye`**

   NumPy propose également une fonction `np.eye(N)` qui permet de créer un tableau bidimensionnel carré $N \times N$ avec des 1 sur la diagonale principale et des 0 ailleurs, c'est-à-dire une **"matrice identité"** :

```
1  import numpy as np
2  print(np.eye(3))
```

```
1  [[1. 0. 0.]
2   [0. 1. 0.]
3   [0. 0. 1.]]
```

4. **La fonction `np.reshape`**

   Notons encore qu'un tableau multidimensionnel peut être obtenu à partir d'un vecteur (i.e. d'un tableau unidimensionnel) grâce à la fonction `np.reshape(a,shape)` .

   Le code suivant fournit par exemple un tableau bidimensionnel 3×6 :

```
1  import numpy as np
2  oned = np.arange(18)
3  multid = np.reshape(oned, (3,6))
4  print(oned, multid, sep = '\n')
```

```
1  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]
2  [[ 0  1  2  3  4  5]
3   [ 6 7 8  9 10 11]
4   [ 12 13 14 15 16 17]]
```

1.4.2 Accès aux éléments d'un tableau

L'accès aux éléments individuels d'un tableau peut se faire en utilisant la syntaxe employée avec les listes (quatrième ligne du code ci-dessous), ou en utilisant la syntaxe proposée à la cinquième ligne du code ci-dessous :

```
import numpy as np
oned = np.arange(24)
multid = np.reshape(oned, (3,8))
print(multid[1][3])
print(multid[1,3])
```

```
11
11
```

1.4.3 Opérations "matricielles"

Les opérations arithmétiques décrites plus haut s'appliquent toujours dans le cas plus général des tableaux multidimensionnels :

```
import numpy as np
x = np.array([[1,2,3], [4,5,6]])
print(1-x)
print(x*5)
print(x**3)
```

```
[[ 0 -1 -2]
 [-3 -4 -5]]
[[ 5 10 15]
 [20 25 30]]
[[  1   8  27]
 [ 64 125 216]]
```

Ces opérations sont effectuées **élément par élément**.
Une multiplication de deux tableaux (de **forme (*shape*) identique**) va donc produire un résultat différent de celui attendu en effectuant le produit matriciel des deux objets (matrices) :

```
import numpy as np
A = np.array([[1,2,3], [4,5,6]])
B = 2*np.ones((2,3))
print(A)
```

```
print(B)
print(A*B)
```

```
[[1 2 3]
 [4 5 6]]
[[2. 2. 2.]
 [2. 2. 2.]]
[[ 2.  4.  6.]
 [ 8. 10. 12.]]
```

La multiplication matricielle habituelle en algèbre linéaire peut être obtenue en exploitant la fonction `np.dot(a,b)` :

```
import numpy as np
A = np.array([[1,2,3], [4,5,6]])
B = 2*np.ones((2,3))
#print(np.dot(A,B))      #produit une erreur
print(np.dot(A,B.T))
```

```
[[12. 12.]
 [30. 30.]]
```

Ici, la transposition de la matrice `B` (à l'aide de la méthode `.transpose()` ou `.T`) est nécessaire.
Le sous-module `linalg` de NumPy permet d'effectuer un grand nombre d'opérations matricielles plus spécialisées, par exemple sur les matrices carrées :

```
import numpy as np
C = np.array([[0,2,1], [6,5,4], [2,0,0]])
print(np.linalg.det(C))
print(np.linalg.inv(C))
valeurs_propres, vecteurs_propres = np.linalg.eig(C)
print(np.linalg.eig(C))
```

Les troisième et quatrième lignes du code ci-dessus calculent **le déterminant** et **l'inverse** de la matrice `C` :

```
6.0
[[ 0.          0.          0.5       ]
 [ 1.33333333 -0.33333333  1.        ]
 [-1.66666667  0.66666667 -2.        ]]
```

```
(array([ 7.09302745, -1.54580305, -0.54722439]),
array([[ 0.28085964,  0.60972679, -0.24417939],
       [ 0.95647598, -0.07681862, -0.37940392],
       [ 0.07919316, -0.78888031,  0.89242876]]))
```

Nous reviendrons sur quelques-unes de ces opérations en exercices et de plus amples informations peuvent être trouvées en parcourant la documentation (par exemple grâce à `np.linalg?`).

1.5 Importation et exportation de données textuelles

Dans le calcul scientifique, il est essentiel de pouvoir fournir des informations à l'ordinateur (*input*) et de pouvoir récupérer le résultat du traitement des données (*output*). Cette communication avec la machine au travers de Python passe souvent par la lecture et l'écriture de fichiers textes.
La fonction `np.loadtxt(fname)` de NumPy permet **de lire et d'exploiter** un fichier texte contenant des données placées dans une ou plusieurs colonnes séparées par un délimiteur (en général un (ou plusieurs) espace(s)). Le paramètre optionnel `delimiter` permet de choisir un délimiteur particulier.
La fonction `np.savetxt(fname)` de NumPy permet quant à elle la **sauvegarde** de données dans un fichier texte.

1.5.1 Lecture de données

Pour comprendre comment utiliser les deux fonctions évoquées ci-dessus, nous allons considérer le fichier texte suivant, obtenu lors d'une expérience de chute libre.
Fichier texte `'ChuteLibreData.txt'` :

```
Données collectées (masse m en chute libre)
Date : 20 février 2025

No de la mesure  Temps[s]  Distance parcourue[m]  Incertitude[m]
0 0 0 0
1 0.51 1.41 1.5
2 1.01 4.39 2.3
3 1.49 11.9 2.5
4 2. 20.7 2.9
5 2.5 28.8 2.7
6 2.99 45.9 2.9
```

Le code suivant permet par exemple de récupérer les trois premières colonnes (paramètre `usecols` de `np.loadtxt`) du fichier `ChuteLibreData.txt` sans tenir compte des quatre premières lignes (paramètre `skiprows` de `np.loadtxt`) :

```
import numpy as np
Data_array = np.loadtxt('ChuteLibreData.txt', usecols = (0,1,2),
 ↪  skiprows = 4, unpack = False)
print(np.shape(Data_array))
identification, xdata, ydata = np.loadtxt('ChuteLibreData.txt',
 ↪  usecols = (0,1,2), skiprows = 4, unpack = True)
print(xdata)
```

Lorsque le paramètre `unpack` de `np.loadtxt` prend la valeur `True` (la valeur `False` est la valeur par défaut), le tableau produit est transposé ce qui permet de récupérer les données (colonnes) dans des tableaux unidimensionnels (quatrième ligne du code ci-dessus). Sortie (résultat) du code Python ci-dessus :

```
(7, 3)
[0.   0.51 1.01 1.49 2.   2.5  2.99]
```

1.5.2 Ecriture de données

La structure générale de la fonction `np.savetxt` est la suivante :

```
np.savetxt(nom_du_fichier, tableau, fmt="%0.18e", delimiter=" ",
 ↪  newline="\n", header="", footer="", comments="# ")
```

- Premier argument : nom du fichier texte à créer.
- Deuxième argument : tableau devant être écrit dans le fichier.
  Si les données que l'on cherche à sauvegarder se trouvent dans plusieurs tableaux unidimensionnels, il faut les rassembler en un seul tableau.
- Les autres arguments sont optionnels et peuvent parfois être compliqués à appréhender. Ainsi, l'option de formatage, décrite dans ce lien, permet de décrire précisément comment les valeurs doivent être représentées. L'argument `fmt="\%0.18e"` permet d'obtenir un affichage à 18 décimales en notation scientifique (exponentielle). Dans l'exemple ci-dessous, nous nous contentons de deux chiffres après la virgule et d'une largeur de colonne minimale de 5.

Le code suivant fournit un exemple concret de sauvegarde de données. Il utilise les données que nous avons déjà rencontrées plus haut :

```
import numpy as np
identification, xdata, ydata, erreur =
 ↪  np.loadtxt("ChuteLibreData.txt", skiprows=4, unpack=True)
informations = 'Données de la chute libre après avoir été
 ↪  retravaillées'
informations += '\nDate de creation : 22 février 2025'
informations += '\nVersion : 2.02'
```

```
informations += '\n\n\nMesure-'
informations += 'temps (en s)-'
informations += 'distance (en m)-'
informations += 'erreur estimée (en m)'
np.savetxt('ChuteLibreDataOut.txt', np.transpose([identification,
 ↪  xdata, ydata, erreur]), header=informations, fmt="%5.2f")
```

Le mot clé `header` de `np.savetxt` permet de placer une en-tête au-dessus des données, alors que le mot clé `footer` est utilisé pour clore le tableau par une série de commentaires. Sortie (résultat) du code Python :

```
# Données de la chute libre après avoir été retravaillées
# Date de creation : 22 février 2025
# Version : 2.02
#
#
# Mesure-temps (en s)-distance (en m)-erreur estimée (en m)
 0.00  0.00  0.00  0.03
 1.00  0.51  1.31  0.02
 2.00  1.01  4.99  0.03
 3.00  1.49 10.90  0.03
 4.00  2.00 19.70  0.03
 5.00  2.50 29.80  0.03
 6.00  2.99 43.90  0.02
```