38 Initiation à Python

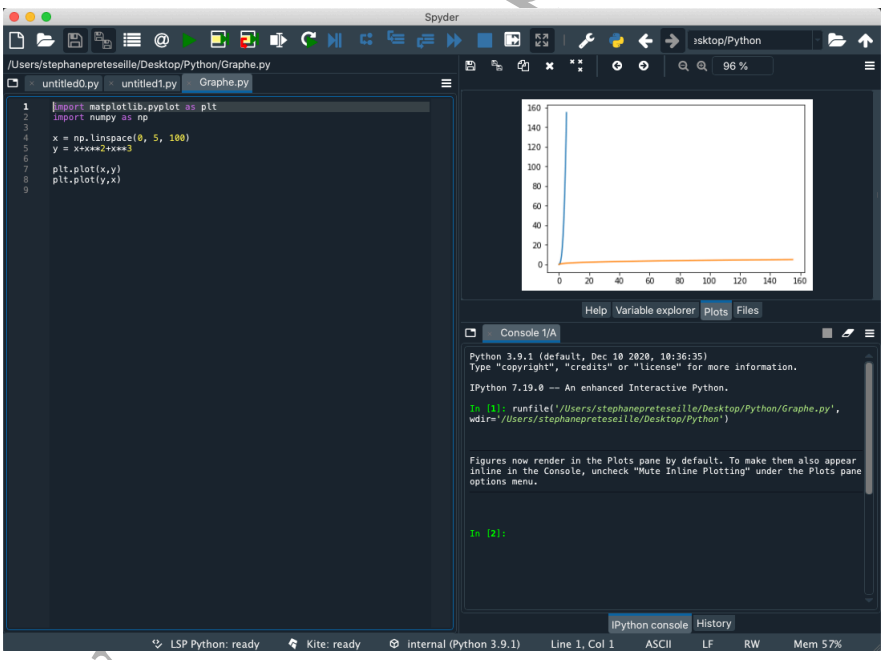
ECG Maths Approfondies
Semestres 1 à 4

A. Introduction

Python est un langage de programmation très employé par les informaticiens, particulièrement efficace en mathématiques, mais aussi dans l'analyse de données.
Il peut être téléchargé librement à la page `https://www.python.org/downloads/`, vous utiliserez alors l'environnement IDLE pour écrire et exécuter vos programmes. Vous pouvez également télécharger un environnement comme Spyder `https://www.spyder-ide.org`, que personnellement je trouve plus convivial.

A.1. Première prise de main de Python avec Spyder

À l'ouverture de Spyder, l'environnement par défaut contient l'éditeur de texte à gauche et l'interprète de commandes (shell ou kernel) à droite.
Les commandes peuvent être saisies dans l'interprète de commande, mais cette méthode n'est pas adaptée à la saisie de programmes et il est dans ce cas préférable de saisir les codes plus longs dans la console de gauche, car ils peuvent alors facilement être sauvegardés et modifiés.



Par exemple, si l'on saisit dans l'interprète de commandes `2+3*12` et `3**2-2**3` ou `print('2+3')`, en appuyant sur la touche « Entrée » ou « Retour », on obtient :

```
In [1]: 2+3*12
Out[1]: 38

In [2]: 3**2-2**3
Out[2]: 1

In [3]: print('2+3')
2+3
```

En revanche, si l'on saisit les commandes dans l'éditeur de texte et que l'on exécute le script, aucun résultat n'est affiché que si cela est demandé explicitement.

Par exemple, si l'on exécute le script suivant :

```
2+3*12
print(3**2-2**3)
5-6
```

Le seul résultat affiché est `1`. En effet, les instructions `2+3*12` et `5-6` ont été exécutées, mais les résultats n'ont pas été affichés, ni mémorisés, car aucune instruction n'a été faite en ce sens. En revanche, l'instruction `print(3**2-2**3)` a été exécutée et le résultat de l'opération `3**2-2**3)` a été affiché, car cela a été demandé à l'aide de la commande `print`.

Le shell peut évidemment servir de calculatrice, mais ce n'est pas sa fonction première et est avant tout un langage de programmation.

A.2. Un pas vers la programmation

L'utilisation du shell est très pratique lorsqu'il s'agit de faire un calcul simple et s'il n'y a pas de modification à faire. En revanche, on ne peut pas enregistrer et, si l'on souhaite enchaîner plusieurs instructions, la rédaction peut-être lourde, surtout si certaines se répètent (puisqu'on doit alors les écrire autant de fois qu'on veut les effectuer).
Lorsqu'on souhaite enchaîner plusieurs instructions, il sera donc préférable de rédiger un script (ou programme). Pour cela, on utilise l'éditeur de texte.
Dans un script, il est possible d'insérer des commentaires expliquant ce que font les différentes instructions ; ces commentaires commencent par # et permettent de faciliter la compréhension du script par un lecteur extérieur (le correcteur par exemple...).
Il est recommandé de créer un dossier dans lequel seront enregistrés tous les scripts rédigés, avec des noms de fichiers facilement repérables (sans espace, mais on peut utiliser le symbole underscore « _ » pour faciliter la lisibilité des noms de fichiers), terminés par l'extension « .py ». L'exécution d'un script peut se faire en cliquant sur l'icône « ▷ ».
Noter que l'aide de Python est assez complète et claire, donc en cas de question ou de doute, on peut faire une recherche dans l'aide.

B. L'environnement logiciel

B.1. Types et variables

Tout objet manipulé par Python possède un type, qui caractérise la manière dont celui-ci sera stocké en mémoire et la nature des opérations qui pourra être effectuée avec. Les types que nous utiliserons sont :

- le type `int` qui représente les nombres entiers,
- le type `float` qui représente les nombres réels (en fait des approximations décimales pour Python),
- le type `boolean` qui ne prend que deux valeurs : `True` (pour *vrai*) ou `False` (pour *faux*),
- le type `list` qui permet de créer des listes, et qui servira également à la création de matrices,
- le type `str` (ou *string*) pour les chaînes de caractères, qui seront saisies entre apostrophes ou des guillemets (indifféremment si la chaîne ne comporte ni apostrophe, ni guillemet, mais avec des apostrophes si la chaîne comporte des guillemets, et réciproquement).

B.2. Opérations élémentaires

De base, seul un nombre limité de fonctions sont disponibles sous Python. On dispose ainsi des opérations arithmétiques usuelles suivantes :

- l'addition : l'instruction `a+b` permet de calculer la somme $a+b$, tandis que l'instruction `a+=b` la valeur contenue dans `a` par $a+b$ (ainsi il est équivalent d'écrire `a=a+b` et `a+=b`),
- la soustraction : l'instruction `a-b` permet de calculer la différence $a-b$,
- la multiplication : l'instruction `a*b` permet de calculer le produit $a \times b$,
- la division : l'instruction `a/b` permet de calculer le quotient $\frac{a}{b}$ (sous réserve qu'il ait un sens),
- la puissance : l'instruction `a**b` permet de calculer a^b,
- la valeur absolue : l'instruction `abs(x)` renvoie la valeur de $|x|$.

Ces opérations respectent les règles de priorité usuelles.

B.3. Les bibliothèques additionnelles

Il existe un certain nombre de modules complémentaires que l'on peut importer sous Python et qui permettent d'utiliser la plupart des fonctions usuelles. Leur contenu sera décrit au fur et à mesure de ce chapitre. On dispose par exemple des bibliothèques suivantes :

- la librairie `numpy` contient les fonctions numériques usuelles (exp, ln, sin,...) et aussi bien d'autres commandes permettant de définir des matrices et de faire du calcul matriciel, d'utiliser des approximations de e ou de π par exemple,
- la librairie `numpy.linalg` apporte des fonctions utiles en algèbre (inversion de matrice, rang, résolution de systèmes linéaires, recherche de valeurs propres et de vecteurs propres),
- la librairie `numpy.random` apporte des fonctions utiles pour simuler des expériences aléatoires,
- la librairie `matplotlib.pyplot` apporte les fonctions utiles aux représentations graphiques.

Pour pouvoir utiliser ces modules dans un programme, on les appelle en ajoutant l'une des commandes `import module` (dans ce cas les fonctions seront appelées dans le programme par la commande `module.nom_fonction`) ou, ce qui sera plus simple à utiliser, `import module as xx` (dans ce cas les fonctions seront appelées dans le programme par la commande `xx.nom_fonction`).
Par exemple, si l'on insère la commande `import numpy as np` en début de programme, on disposera des fonctions usuelles :

- l'instruction `np.log(a)` permet de renvoyer la valeur de $\ln(a)$,
- l'instruction `np.exp(a)` permet de renvoyer la valeur de e^a,
- l'instruction `np.floor(a)` permet de renvoyer la valeur de $\lfloor a \rfloor$ (partie entière de a),
- l'instruction `np.abs(a)` permet de renvoyer la valeur de $|a|$,
- l'instruction `np.sqrt(a)` permet de renvoyer la valeur de $\sqrt{a}$,
- l'instruction `np.sin(a)` permet de renvoyer la valeur de $\sin(a)$,
- l'instruction `np.cos(a)` permet de renvoyer la valeur de $\cos(a)$,

ainsi que des constantes e et π :

- la commande `np.e` renvoie une valeur approchée de e,
- la commande `np.pi` renvoie une valeur approchée de π.

Dans la suite, quand ces bibliothèques seront utilisées, on les supposera importées comme suit :

```
import numpy as np
import numpy.linalg as al
import numpy.random as rd
import matplotlib.pyplot as plt
```

B.4. Affectations, entrée et sortie de valeurs

Affectations

Si une valeur (nombre réel, matrice ou autre) est susceptible d'être utilisée à plusieurs reprise, il est préférable de la stocker dans une variable, en utilisant le symbole =.

- L'instruction `a=2*13-2` a pour conséquence de stocker dans une variable `a` le résultat de l'opération $2 \times 13 - 2$ (donc 24), de sorte que, si l'on utilise la variable `a` dans un calcul ultérieur, ce sera cette valeur qui sera utilisée. Par exemple, si l'on saisit la suite d'instruction `a=2*13-2`, `b=a**2` puis `print(b)` en appuyant sur « Retour » après chaque instruction, la console affiche :

```
In [1]: a=2*13-2
In [2]: b=a**2
In [3]: print(b)
576
```

- La valeur stockée dans une variable peut être modifiée. Par exemple, si l'on saisit les instructions `a=1`, puis `a=a+2`, puis `a=a**3` et enfin `print(a)`, la dernière valeur stockée dans `a` est $(1+2)^3$, et la console affiche :

```
In [1]: a=1
In [2]: a=a+2
In [3]: a=a**3
In [4]: print(a)
27
```

Entrée de valeurs

Il peut arriver, notamment dans un script, que l'on souhaite utiliser une valeur saisie par l'utilisateur et la stocker dans une variable. Pour cela, on utilise l'instruction `input`.
Par exemple, si l'on saisit l'instruction `a=input('entrer un entier naturel : ')`, la console affiche le texte « entrer un entier naturel : » et attend la saisie par l'utilisateur d'une réponse ; lorsque l'utilisateur aura saisi une réponse et appuyé sur « Retour », cette valeur sera affectée à la variable `a`.

```
In [1]: a=input('entrer un entier naturel : ')
entrer un entier naturel : 2
In [2]: a
Out[2]: '2'
```

Noter la présence des apostrophes dans le résultat affiché à la ligne `Out[2]` : il indique que la valeur saisie (ici 2) n'est pas interprétée comme un nombre, mais comme une chaîne de caractère. Ainsi, si l'on cherche ensuite à effectuer l'opération `a+1`, on obtient le résultat suivant :

```
In [1]: a=input('entrer un entier naturel : ')
entrer un entier naturel : 2
In [2]: a
Out[2]: '2'
In [3]: a+1
Traceback (most recent call last):
  File "<ipython-input-20-98b939904c8e>", line 1, in <module>
    a+1
TypeError: can only concatenate str (not "int") to str
```

Si l'on saisit une valeur numérique, il s'agira donc d'en indiquer le type :

- la commande `n=int(input('entrer n'))` permet ainsi de stocker la valeur saisie dans la variable `n` de type `int` (entier),
- la commande `x=float(input('entrer x'))` permet ainsi de stocker la valeur saisie dans la variable `x` de type `float` (flottant, que l'on utilise pour les réels).

Lors de la saisie d'une valeur réelle à l'issue d'une instruction comme `x=float(input('entrer x'))`, celle-ci ne peut être le résultat d'une opération. Si l'on souhaite affecter à `x` la valeur $\frac{1}{3}$, il conviendra donc d'indiquer à Python qu'il doit effectuer le calcul, à l'aide de la fonction `eval()`, comme dans l'exemple suivant :

```
In [1]: a=float(eval(input('entrer un réel x : ')))
entrer un réel x : 1/3
In [2]: a
Out[2]: 0.3333333333333333
```

Sorties de texte ou de valeurs

Comme on l'a déjà vu précédemment, la commande `print` permet de demander d'afficher du texte ou le contenu d'une variable :

- l'instruction `print('texte')` permet d'afficher la chaîne de caractère `texte`,
- l'instruction `print(a)` permet d'afficher le contenu de la variable `a`.

Il est possible de demander l'affichage successifs de plusieurs informations (texte ou valeurs), en les séparant par des virgules, comme dans l'exemple suivant :

```
In [1]: a=2+3
In [2]: b=2**3
In [3]: print('a=',a,'b=',b)
a= 5 b= 8
```

C. Matrices numériques

On a vu que Python dispose d'un type `list`, qui nous sera utile pour les opérations matricielles. Une liste contient une série de valeurs, qui peuvent être de types différents (par exemple des entiers et des chaînes de caractères, ou même des listes). La déclaration d'une liste se fait en entrant série de valeurs séparées par des virgules, le tout encadré par des crochets.
Par exemple, la commande `x=[1,2,3]` crée une liste dont les trois coefficients sont $1, 2$ et 3, la commande `x=[[1,2,3],[4,5,6]]` crée une liste dont les deux coefficients sont les listes `[1,2,3]` et `[4,5,6]`.
Attention cependant, une liste n'est pas une matrice, et il n'est pas possible d'effectuer des opérations matricielles directement avec les listes. Par exemple, si l'on cherche à faire un produit, on obtient un message d'erreur :

```
In [1]: A=[[1,2],[2,1]]
In [2]: B=[[-1,1],[1,2]]
In [3]: A*B
Traceback (most recent call last):
  File "<ipython-input-48-47896efed660>", line 1, in <module>
    A*B
TypeError: can't multiply sequence by non-int of type 'list'
```

On peut en revanche concaténer des listes (ce qui revient à ajouter à la fin d'une première liste les éléments d'une seconde), ce qui s'avèrera utile pour la création de matrices. Ainsi, si `A` et `B` sont deux variables de type `list`, la commande `A+B` renvoie une liste, qui est la concaténation des listes `A` et `B` :

```
In [1]: A=[[1,2],[2,1]]
In [2]: B=[[-1,5]]
In [3]: A+B
Out[3]: [[1, 2], [2, 1], [-1, 5]]
```

C.1. Création de matrices

Le cas général

Pour créer des matrices à partir de listes, on peut importer la bibliothèque `numpy` et utiliser la fonction `np.array`. Ainsi, si une variable `A` contient une liste, la commande `np.array(A)` permet de convertir cette liste en matrice.
Par exemple, pour créer une variable `M` représentant la matrice

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

on pourra indifféremment utiliser les instructions suivantes :

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

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

On se souviendra que les lignes et les colonnes d'un tableau sont toujours numérotées à partir de 0.

Création de matrices particulières

S'il est évidemment souvent possible de saisir manuellement une matrice en entrant tous les coefficients, cela peut s'avérer fastidieux, voire impossible, dans le cas de vecteurs de grande taille.
Certaines fonctions de la bibliothèque `numpy` permettent de créer automatiquement certaines matrices :

- l'instruction `np.zeros(n)` renvoie la matrice (ligne) de $\mathcal{M}_{1,n}(\mathbb{R})$ dont tous les coefficients sont nuls,
- l'instruction `np.zeros([n,p])` renvoie la matrice de $\mathcal{M}_{n,p}(\mathbb{R})$ dont tous les coefficients sont nuls,
- l'instruction `np.ones(n)` renvoie la matrice (ligne) de $\mathcal{M}_{1,n}(\mathbb{R})$ dont tous les coefficients sont égaux à 1,
- l'instruction `np.ones([n,p])` renvoie la matrice de $\mathcal{M}_{n,p}(\mathbb{R})$ dont tous les coefficients sont égaux à 1,
- l'instruction `np.eye(n)` renvoie la matrice identité de $\mathcal{M}_n(\mathbb{R})$, c'est-à-dire la matrice dont tous les coefficients sont égaux à 0, sauf les coefficients diagonaux, tous égaux à 1,
- l'instruction `np.arange(a)` (où `a` est de type `float` et représente un réel strictement positif a) renvoie une matrice ligne dont les coefficients sont les entiers appartenant à $[0, a[$,

- l'instruction `np.arange(a,b)` (où `a` et `b`, de type `float`, représentent deux réels a et b tels que $a < b$) renvoie une matrice ligne dont les coefficients sont les termes appartenant à $[a, b[$ de la suite arithmétique de raison 1 et de premier terme a,
- l'instruction `np.arange(a,b,r)` (où `a`, `b` et `r`, de type `float`, représentent trois réels a, b, r tels que $a < b$ et $r > 0$) renvoie une matrice ligne dont les coefficients sont les termes appartenant à $[a, b[$ de la suite arithmétique de raison r et de premier terme a,
- l'instruction `np.arange(a,b,r)` (où `a`, `b` et `r`, de type `float`, représentent trois réels a, b, r tels que $a > b$ et $r < 0$) renvoie une matrice ligne dont les coefficients sont les termes appartenant à $]b, a]$ de la suite arithmétique de raison r et de premier terme a,
- l'instruction `np.linspace(a,b,n)` (où `a` et `b` sont de type `int` ou `float` et `n` est de type `int`) envoie une matrice (ligne) de $\mathcal{M}_{1,n}(\mathbb{R})$ dont les coefficients sont n premiers termes d'une suite arithmétique x telle que $x_0 = a$ et $x_{n-1} = b$, c'est-à-dire n réels $x_0, \dots, x_{n-1}$ tels que $x_1 - x_0 = x_2 - x_1 = \cdots = x_{n-1} - x_{n-2}$) et tels que $x_0 = a$ et $x_{n-1} = b$, autrement dit le vecteur `x` dont chaque coefficient `x(k)` vaut $a + k\,\frac{b-a}{n-1}$.

Des exemples valant toujours mieux qu'une longue explication, on relira les lignes précédentes pour comprendre les instructions et résultats suivant :

```
In [1]: np.zeros(3)
Out[1]: array([0., 0., 0.])
In [2]: np.zeros([2,3])
Out[2]:
array([[0., 0., 0.],
       [0., 0., 0.]])
In [3]: np.ones(2)
Out[3]: array([1., 1.])
In [4]: np.ones([3,2])
Out[4]:
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
In [5]: np.eye(3)
Out[5]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
In [6]: np.arange(5)
Out[6]: array([0, 1, 2, 3, 4])
In [7]: np.arange(2,6)
Out[7]: array([2, 3, 4, 5])
In [8]: np.arange(1,11,2)
Out[8]: array([1, 3, 5, 7, 9])
In [9]: np.arange(11,1,-2)
Out[9]: array([11, 9, 7, 5, 3])
In [10]: np.linspace(1,4,5)
Out[10]: array([1. ,1.75, 2.5, 3.25, 4. ])
```

En particulier, on peut remarquer que les instructions `np.arange(0,n+1)` et `np.linspace(0,n,n+1)` renvoient la même matrice, à ceci-près que les coefficients de `np.arange(0,n+1)` sont de type `int`, tandis que ceux de `np.linspace(0,n,n+1)` sont de type `float`.

C.2. Opérations matricielles

Extraction ou modification d'éléments d'une matrice

Il est possible de modifier ou d'utiliser certains coefficients d'une matrice, en n'oubliant pas que les lignes et les colonnes sont toujours numérotées à partir de 0.
Si `x` est une matrice ligne et si `i`, `j` et `r` sont des variables contenant des entiers i, j, r, alors :

- `x[i]` renvoie le coefficient *numéro* i de la matrice, c'est-à-dire le $(i+1)^{ème}$ coefficient de `x`,
- `x[i:j]` renvoie une matrice lignes dont les $j - i$ coefficients sont `x[i]`, `x[i+1]`,..., `x[j-1]`,
- `x[i:j:r]` renvoie une matrice lignes dont les $j - i$ coefficients sont `x[i]`, `x[i+r]`,..., `x[i+kr]` où k est le plus grand entier tel que $i + kr < j$.

On pourra relire les lignes précédentes pour comprendre l'exemple suivant :

```
In [1]: x=np.arange(-3,15,2)
In [2]: x[0]
Out[2]: -3
In [3]: x[5]
Out[3]: 7
In [4]: x[1:5]
Out[4]: array([-1, 1, 3, 5])
In [5]: x[2:9:3]
Out[5]: array([ 1, 7, 13])
In [6]: x
Out[6]: array([-3, -1, 1, 3, 5, 7, 9, 11, 13])
```

Si `A` est une variable contenant une matrice $A = (a_{i,j})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant p}}$ de $\mathcal{M}_{n,p}(\mathbb{R})$ et si `i`, `j`, `i1`, `i2`, `j1` et `j2` sont des variables contenant des entiers i, j, i_1, i_2, j_1, j_2, alors :

- l'instruction `A[i,j]` renvoie le coefficient de la $i^{ème}$ ligne et de la $j^{ème}$ colonne **de la variable** `A`, c'est-à-dire $a_{i+1,j+1}$,
- l'instruction `A[i,:]` et `A[i]` renvoient la $i^{ème}$ ligne de la variable `A`, c'est-à-dire la $(i+1)^{ème}$ ligne de la matrice A,
- l'instruction `A[:,j]` renvoie **une matrice ligne** dont les coefficients sont ceux de la $j^{ème}$ colonne de la variable `A`,
- l'instruction `A[i1:i2,j1:j2]` renvoie **une matrice** de $i_2 - i_1$ lignes et $j_2 - j_1$ colonnes dont les coefficients sont ceux situés sur les lignes i_1 à $i_2 - 1$ et les colonnes j_1 à $j_2 - 1$ de la variable `A`,
- si `x=[x[0],...,x[m]]` et `y=[y[0],...,y[m]]` sont deux vecteurs dont les éléments appartiennent respectivement à $[\![1, n]\!]$ et $[\![1, p]\!]$, l'instruction `A[x,y]` renvoie une **matrice ligne** dont les m coefficients sont `A[x[0],y[0]]`,..., `A[x[m],y[m]]`.

On pourra relire les lignes précédentes pour comprendre l'exemple suivant :

```
In [1]: A=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In [2]: A[1,2]
Out[2]: 7
In [3]: A[2]
Out[3]: array([ 9, 10, 11, 12])
In [4]: A[0,:]
Out[4]: array([1, 2, 3, 4])
In [5]: A[:,1]
Out[5]: array([ 2, 6, 10])
In [6]: A[2,1]=0
In [7]: A
Out[7]:
array([[ 1, 2, 3, 4],
       [ 5, 6, 7, 8],
       [ 9, 0, 11, 12]])
In [8]: A[[0,2],[1,2]]
Out[8]: array([ 2, 11])
In [9]: A[0,:]=np.zeros(4)
In [10]: A
Out[10]:
array([[ 0, 0, 0, 0],
       [ 5, 6, 7, 8],
       [ 9, 0, 11, 12]])
In [11]: A[0:2,1:4]
Out[11]:
array([[0, 0, 0],
       [6, 7, 8]])
```

Les opérations usuelles

Si `A` et `B` sont deux variables contenant des matrices A et B définies avec la bibilothèque `numpy` et si `a` est une variable contenant un réel x, Python autorise les opérations suivantes, utilisant les bibliothèques `numpy` et `numpy.linalg` :

- l'addition : l'instruction `A+B` permet de calculer la somme $A+B$,
- la soustraction : l'instruction `A-B` permet de calculer la différence $A-B$,
- la multiplication par un réel : l'instruction `xA` permet de calculer la différence xA,
- la multiplication : l'instruction `np.dot(A,B)` permet de calculer le produit AB,
- la transposition : l'instruction `np.transpose(A)` permet de calculer ${}^tA$,
- l'inversion : l'instruction `al.inv(A)` permet de calculer A^{-1},
- l'élévation à une puissance entière : si `n` contient un entier, l'instruction `al.matrix_power(A,n)` permet de calculer A^n,
- calcul du rang : l'instruction `al.matrix_rank(A)` permet de déterminer le rang de A,
- format d'une matrice : l'instruction `np.shape(A)` permet de renvoyer le format de la matrice A (dans l'ordre, nombre de lignes et nombre de colonnes),
- nombre de coefficients d'une matrice : si `A` est une matrice de format $n \times p$, l'instruction `np.size(A)` permet de renvoyer la valeur np.

Évidemment, ces opérations ne donnent un résultat que si elles ont un sens (formats compatibles, matrice inversible...).
On pourra ainsi comprendre les exemples suivants :

```
In [1]: M=np.array([[1,2,3],[1,0,0],[0,1,1]])
In [2]: al.inv(M)
Out[2]:
array([[ 0., 1., 0.],
       [-1., 1., 3.],
       [ 1., -1., -2.]])
In [3]: M=np.ones([3,3])
In [4]: al.inv(M)
Out[4]: LinAlgError: Singular matrix
In [5]: A=np.array([[1,2,0],[1,1,0]])
In [6]: B=np.array([[1,1,2],[1,1,2]])
In [7]: A+B
Out[7]:
array([[2, 3, 2],
       [2, 2, 2]])
In [8]: A=np.array([[1,2],[1,1]])
In [9]: B=np.array([[1,1,2],[1,1,2]])
In [10]: A+B
Out[10]: ValueError: operands could not be
broadcast together with shapes (2,2) (2,3)
```

Opérations coefficient par coefficient

Si `A` et `B` sont deux variables contenant des matrices A et B de même format $n \times p$, le coefficient de la ligne i et de la colonne j de A (respectivement B) est notée $a_{i,j}$ (respectivement $b_{i,j}$), `numpy` autorise les opérations suivantes :

- la multiplication terme à terme : l'instruction `A*B` renvoie la matrice $C = (a_{i,j}\, b_{i,j})$,
- la division terme à terme : l'instruction `A/B` renvoie la matrice $C = (a_{i,j} \,/\, b_{i,j})$ (sous réserve que les coefficients de B soient tous non nuls évidemment),
- l'élévation à la puissance de chaque terme : si `k` est un entier, l'instruction `A**k` renvoie la matrice $C = (a_{i,j}^k)$,
- l'addition d'un même réel `x` à tous les coefficients d'une matrice `A` : l'instruction `B=A+x` ou `B=x+A` renvoie la matrice $B = (a_{i,j} + x)$,
- la multiplication de tous les coefficients d'une matrice `A` par un même réel `x` : l'instruction `B=A*x` ou `B=x*A` renvoie la matrice $B = (x\, a_{i,j})$.

Plus généralement, si `f` est une fonction numérique définie sous Python et si `A` est une matrice dont les coefficients sont `A(i,j)`, alors l'instruction `f(A)` renvoie une matrice dont les coefficients sont `f(A(i,j))`.

Minimum, maximum, moyenne

La bibliothèque `numpy` permet de déterminer le maximum et le minimum d'une matrice.
Si `A` est une variable représentant une matrice A de $\mathcal{M}_{n,p}(\mathbb{R})$:

- l'instruction `np.min(A)` renvoie le plus petit coefficient de `A`,
- l'instruction `np.min(A,0)` renvoie une matrice ligne dont le $j^{ème}$ coefficient est le plus petit des coefficients de la colonne numéro j de la variable `A`,
- l'instruction `np.min(A,1)` renvoie une matrice ligne dont le $i^{ème}$ coefficient est le plus petit des coefficients de la ligne numéro i de la variable `A`,
- l'instruction `np.max(x)` renvoie le plus grand élément de `xA`,
- l'instruction `np.max(A,0)` renvoie une matrice ligne dont le $j^{ème}$ coefficient est le plus grand des coefficients de la colonne numéro j de la variable `A`,
- l'instruction `np.max(A,1)` renvoie une matrice ligne dont le $i^{ème}$ coefficient est le plus grand des coefficients de la ligne numéro i de la variable `A`,
- l'instruction `np.mean(x)` renvoie la moyenne des coefficients de `A`,
- l'instruction `np.mean(A,0)` renvoie une matrice ligne dont le $j^{ème}$ coefficient est la moyenne des coefficients de la colonne numéro j de la variable `A`,
- l'instruction `np.mean(A,1)` renvoie une matrice ligne dont le $i^{ème}$ coefficient est la moyenne des coefficients de la ligne numéro i de la variable `A`.

On peut ainsi comprendre l'exemple suivant :

```
In [1]: A=np.array([[-1,1,0,3],[5,4,-2,6],[0,2,4,8]])
In [2]: np.max(A)
Out[2]: 8
In [3]: np.min(A)
Out[3]: -2
In [4]: np.max(A,0)
Out[4]: array([5, 4, 4, 8])
In [5]: np.max(A,1)
Out[5]: array([3, 6, 8])
In [6]: np.min(A,0)
Out[6]: array([-1, 1, -2, 3])
In [7]: np.min(A,1)
Out[7]: array([-1, -2, 0])
In [8]: np.mean(A)
Out[8]: 2.5
In [9]: np.mean(A,0)
Out[9]: array([1.33333333, 2.33333333, 0.66666667, 5.66666667])
In [10]: np.mean(A,1)
Out[10]: array([0.75, 3.25, 3.5 ])
```

C.3. Calculs de sommes

Lorsqu'on dispose d'une variable `A` représentant une matrice A de taille $n \times p$, il est possible de calculer certaines sommes de coefficients automatiquement :

- la commande `np.sum(A)` renvoie la somme de tous les coefficients de A,
- la commande `np.sum(A,0)` renvoie une matrice ligne dont le coefficient de la colonne j est égal à la somme des coefficients de la colonne j de la variable `A`,
- la commande `np.sum(A,1)` renvoie une matrice ligne dont le coefficient de la colonne i est égal à la somme des coefficients de la ligne i de la variable `A`,
- la fonction `np.cumsum(A)` renvoie une **matrice ligne**, dont $i^{ème}$ coefficient est égal à la somme des i premiers coefficients de A (les coefficient étant numérotées de 1 à np, ligne par ligne),
- la fonction `np.cumsum(A,0)` renvoie une matrice de même format que `A` dont le coefficient situé sur la $i^{ème}$ ligne et la $j^{ème}$ colonne est égal à la somme des i premiers coefficients de la colonne j de `A`,
- la fonction `np.cumsum(A,1)` renvoie une matrice de même format que `A` dont le coefficient situé sur la $i^{ème}$ ligne et la $j^{ème}$ colonne est égal à la somme des j premiers coefficients de la ligne j de `A`.

On peut désormais comprendre l'exemple suivant :

```
In [1]: A=np.array([[-1,1,0,3],[5,4,-2,6],[0,2,4,8]])
In [2]: np.sum(A)
Out[2]: 30
In [3]: np.sum(A,0)
Out[3]: array([ 4, 7, 2, 17])
In [4]: np.sum(A,1)
Out[4]: array([ 3, 13, 14])
In [5]: np.cumsum(A)
Out[5]: array([-1, 0, 0, 3, 8, 12, 10, 16, 16, 18, 22, 30])
In [6]: np.cumsum(A,0)
Out[6]:
array([[-1, 1, 0, 3],
       [ 4, 5, -2, 9],
       [ 4, 7, 2, 17]])
In [7]: np.cumsum(A,1)
Out[7]:
array([[-1, 0, 0, 3],
       [ 5, 9, 7, 13],
       [ 0, 2, 6, 14]])
```

Noter que des fonctions analogues existent pour effectuer les produits ou les produits cumulés des coefficients d'une matrice (fonctions `np.prod` et `np.cumprod`) mais qui ne figurent pas au programme.

D. Structures de contrôle

Contrairement à beaucoup de langages informatiques, les blocs d'instructions liées entre elles ne sont pas séparées par mots-clefs (du type `begin... end` ou par des accolades. Avec le langage de Python, c'est l'indentation (l'espace en début de ligne) qui permet d'indiquer les groupes d'instructions. Il faudra donc veiller à bien respecter l'indentation pour éviter les bugs.

D.1. Les structures conditionnelles

Les variables booléennes

Le résultat d'une proposition logique peut être stocké dans une variable. Une telle variable est appelée variable booléenne et ne prend que deux valeurs : `True` (pour vrai) ou `False` (pour faux). Il sera intéressant de se souvenir que, d'une certaine façon, Python gère ce type de variable comme si elle contenant les valeurs 1 (pour `True`) et 0 (pour `False`), permettant par exemple l'addition et la multiplication (ainsi `True+True` retourne 2 par exemple).

Les opérateurs de comparaison

Deux types d'opérateurs renvoient des résultats de type booléen :

- les opérateurs de comparaison ou de test :
  - l'opérateur `==` : l'instruction `a==b` renvoie `True` si les valeurs stockées dans `a` et `b` sont égales et renvoie `False` sinon,
  - l'opérateur `!=` : l'instruction `a!=b` renvoie `Trure` si les valeurs stockées dans `a` et `b` sont différentes et renvoie `False` sinon,
  - l'opérateur `>` : l'instruction `a>b` renvoie `True` si la valeur stockée dans `a` est strictement supérieure à celle stockée dans `b` et renvoie `False` sinon,
  - l'opérateur `>=` : l'instruction `a>=b` renvoie `True` si la valeur stockée dans `a` est supérieure ou égale à celle stockée dans `b` et renvoie `False` sinon,
  - l'opérateur `<` : l'instruction `a<b` renvoie `True` si la valeur stockée dans `a` est strictement inférieure à celle stockée dans `b` et renvoie `False` sinon,
  - l'opérateur `<=` : l'instruction `a<=b` renvoie `Truesi` la valeur stockée dans `a` est inférieure ou égale à celle stockée dans `b` et renvoie `False` sinon,
- les opérateurs relationnels :
  - l'opérateur `and` : l'instruction `A and B` renvoie `True` si les propositions `A` et `B` sont toutes les deux vraies et renvoie `False` sinon,

— l'opérateur `or` : l'instruction `A or B` renvoie `True` si l'une au moins des propositions `A` et `B` est vraie et renvoie `False` sinon,

— l'opérateur `not` : l'instruction `not A` renvoie `True` si la proposition `A` est fausse et renvoie `False` sinon.

Les opérateurs de comparaison peuvent être utilisés avec des matrices `A` et `B`, du moment qu'elles sont de même format ; dans ce cas, chaque case `A(i,j)` est comparée avec la case `B(i,j)`, comme on peut le voir dans l'exemple suivant :

```
In [1]: A=np.array([[-1,1,0,3],[5,4,-2,6],[0,2,4,8]])
In [2]: B=np.array([[1,-1,1,3],[4,4,-2,6],[0,2,2,8]])
In [3]: A==B
Out[3]:
array([[False, False, False, True],
       [False, True, True, True],
       [ True, True, False, True]])
```

Les structures conditionnelles

Il peut arriver que l'on veuille effectuer des opérations différentes selon que l'on est dans une situation ou une autre.
Dans ce cas, on rédige l'instruction

```
if condition1:
    instructions1
    ...
else:
    instructions3
    ....
```

On se souviendra que :

- s'il y a plusieurs instructions à effectuer lorsque la condition est vraie (respectivement fausse), elles doivent toutes avoir la même indentation (l'espace horizontal à gauche du texte),
- l'alternative `else` n'est pas obligatoire (si l'on ne souhaite rien faire lorsque la condition n'est pas vraie), auquel cas la commande s'arrête après `instructions1` (qui comporte éventuellement plusieurs lignes),
- si l'instruction `else` est suivie d'un `if`, on peut contracter les deux en écrivant `elif`.

Par exemple, si l'on exécute les commandes

```
if x>0:
    y=np.log(x)
    print('ln(x)=',y)
else:
    print("ln(x) n'existe pas")
```

la console affichera le message « $\ln(x) =$ » suivi de la valeur de $\ln(x)$ si $x > 0$ et affichera le message « $\ln(x)$ n'existe pas » si $x \leqslant 0$.
De même, si l'on exécute les commandes

```
if x>0:
    print(1)
elif x==0:
    print(0)
else:
    print(-1)
```

la console affichera la valeur 1 si $x > 0$, affichera la valeur 0 si $x = 0$ et la valeur -1 si $x < 0$.

D.2. Les structures répétitives

Les boucles `for`

Si l'on veut répéter une suite d'instructions un nombre fixé de fois, on peut utiliser l'instruction `for`, que l'on rédigera ainsi :

```
for k in range(a,b+1):
    Instructions
    .....
```

On se souviendra que :

- l'instruction `for k in range(a,b+1)` permet de répéter une ou plusieurs instructions lorsque `k` prend successivement les valeurs `a`, `a+1`, ..., `b`, donc $b-a+1$ fois en tout (attention à la dernière valeur prise par `k` !),
- la variable `k` est une variable muette et peut évidemment être remplacée par n'importe quelle nom de variable,
- `range(a,b+1)` indique
- l'instruction `for k=p:i:n instruction(s), end` permet de répéter une ou plusieurs instructions (séparées, le cas échéant, par des virgules ou des points-virgules) lorsque `k` prend successivement les valeurs `p`, `p+i`, `p+2i`, ..., `p+mi` (où m est tel que $p+mi \leqslant n < p+(m+1)i$).

Par exemple, si l'on veut calculer et afficher la somme $\sum_{k=0}^{100} k^3$, on peut écrire le script suivant :

```
S=0
for k in range(1,101):
    S=S+k**3
print(S)
```

Cependant, l'utilisateur averti aura aussi remarqué que l'on obtient le même résultat en saisissant les instructions

```
S=0
for k in range
    (1,101):
    S=S+k**3
print(S)
```

Les boucles `while`

Si l'on veut répéter une suite d'instructions tant qu'une certaine condition est vraie (ou ne l'est pas), on peut utiliser l'instruction `while`, que l'on présentera ainsi :

```
while condition:
    instructions
    .....
```

Par exemple, si l'on sait que la suite $u = \left((n+1)^2\,\mathrm{e}^{-n}\right)_{n\in\mathbb{N}}$ converge vers 0 et que l'on cherche le premier entier naturel n tel que $|u_n| < 10^{-9}$, on peut écrire le programme suivant :

```
n=0
while (n+1)**2*np.exp(-n)>=10**(-9):
    n=n+1
print(n)
```

E. Définition de fonctions

Si Python connaît déjà un certain nombre de fonctions, il est possible de définir des fonctions supplémentaires, ce qui permet de n'écrire qu'une seule fois une série d'instructions amenées à être effectuées plusieurs fois. La création d'une fonction est annoncée par une ligne de commande du type `def nom_fonction(paramètres d'appel):` où `nom_fonction` sera évidemment remplacé par le nom que l'on souhaite donner à la fonction (et qui servira à l'appeler dans le programme) et `paramètres d'appel` sera remplacé par la liste des valeurs qui sont utilisées dans les calculs. Le plus souvent, une fonction servira à effectuer une série de calculs et à renvoyer un résultat, auquel cas elle se terminera par une ligne de commande du type `return résultat`, mais cette ligne peut ne pas être utile si l'on ne souhaite pas renvoyer de résultat mais juste effectuer une série d'instructions (visant à afficher un ou plusieurs résultats par exemple).
Par exemple, si l'on souhaite définir la fonction f définie sur $\mathbb{R}$ par :

$$\forall x \in \mathbb{R},\ f(x) = \mathrm{e}^x + \frac{1}{x^2+1}$$

on pourra rédiger la fonction suivante :

```
import numpy as np
def f(x):
    return np.exp(x)+1/(x**2+1)
```

De même, si l'on souhaite définir la fonction g définie sur $\mathbb{R}$ par :

$$\forall x \in \mathbb{R},\ g(x) = \begin{cases} \frac{\mathrm{e}^x-1}{x} & \text{si } x \neq 0 \\ 1 & \text{si } x = 0 \end{cases}$$

on pourra rédiger la fonction suivante :

```
import numpy as np
def g(x):
    if x==0:
        return 1
    else:
        return (np.exp(x)-1)/x
```

L'appel de la fonction se fera dans le programme naturellement. Par exemple, si l'on veut écrire un programme demandant l'entrée d'un réel x puis calculant et affichant la valeur de $f(x)$ (la fonction f étant la première fonction donnée en exemple), le programme pourra être rédigé ainsi :

```
import numpy as np
def f(x):
    if x==0:
        return 1
    else:
        return (np.exp(x)-1)/x
x=float(input('x='))
print(f(x))
```

On se souviendra que :

- le plus souvent, une fonction réelle `f` d'une variable réelle x peut être appliquée à une matrice `A`, auquel cas le résultat renvoyé sera une matrice de même format que `A` et dont les coefficients sont les images `f(A[i,j]` des coefficients de `A` ; c'est le cas par exemple de la fonction `f` définie ci-dessus, et l'exécution de la commande `f(np.arange(-1,2))` renvoie ainsi le résultat suivant :

```
array([0.86787944, 2. , 3.21828183])
```

- il arrive cependant que cela ne soit pas possible directement, du fait de la nature des opérations effectuées (par exemple s'il y a des comparaisons) ; dans ce cas, on pourra utiliser la fonction `np.vectorize`, qui permet de créer une nouvelle fonction, compatible avec les matrices ; c'est le cas par exemple de la fonction `g` introduite précédemment, et à l'exécution de la commande `g(np.arange(-1,2))`, on obtient le message suivant :

```
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

On réécrira dans ce cas le programme ainsi :

```
import numpy as np
def g(x):
    if x==0:
        return 1
    else:
        return (np.exp(x)-1)/x
G=np.vectorize(g)
```

L'exécution de la commande `G(np.arange(-1,2))` renvoie alors le résultat suivant :

```
array([0.63212056, 1. , 1.71828183])
```

F. Générateurs de nombres aléatoires et simulation

F.1. Générateur de nombres aléatoires

Le langage Python permet de simuler des expériences aléatoires grâce à la bibliothèque `numpy.random`. Ainsi, si `n,p,a,b` sont des variables contenant des entiers naturels n, p et relatifs a, b, alors :

- l'instruction `rd.randint(b+1)` renvoie un nombre réel choisi au hasard dans l'intervalle $[\![0, b]\!]$; autrement dit, `rd.randint(b+1)` permet de simuler une expérience aléatoire en renvoyant la valeur prise par une variable aléatoire suivant la loi uniforme sur $[\![0, b]\!]$,
- l'instruction `rd.randint(a,b+1)` renvoie un nombre réel choisi au hasard dans l'intervalle $[\![a, b]\!]$; autrement dit, `rd.randint(a,b+1)` permet de simuler une expérience aléatoire en renvoyant la valeur prise par une variable aléatoire suivant la loi uniforme sur $[\![a, b]\!]$,
- l'instruction `rd.randint(a,b+1,n)` renvoie une simulation de n variables aléatoires $X_1, \dots, X_n$ indépendantes et suivant la loi uniforme sur $[\![a, b]\!]$,
- l'instruction `rd.randint(a,b+1,[n,p])` renvoie une matrice de $\mathcal{M}_{n,p}(\mathbb{R})$ dont les np coefficients sont des simulations de np variables aléatoires $X_{1,1}, X_{1,2}, \dots, X_{n,p}$ indépendantes et suivant la loi uniforme sur $[\![a, b]\!]$,
- l'instruction `rd.random()` renvoie un nombre réel choisi au hasard dans l'intervalle $[0, 1[$; autrement dit, `rand` permet de simuler une expérience aléatoire en renvoyant la valeur prise par une variable aléatoire suivant la loi uniforme sur $[0, 1[$,
- l'instruction `rd.random(n)` renvoie une simulation de n variables aléatoires $X_1, \dots, X_n$ indépendantes et suivant la loi uniforme sur $[0, 1[$,
- l'instruction `rd.random([n,p])` renvoie une matrice de $\mathcal{M}_{n,p}(\mathbb{R})$ dont les np coefficients sont des simulations de np variables aléatoires $X_{1,1}, X_{1,2}, \dots, X_{n,p}$ indépendantes et suivant la loi uniforme sur $[0, 1[$.

Par exemple, si l'on souhaite simuler le tirage de 5 boules au hasard et avec remise dans une urne contenant 10 boules numérotées de 1 à 10, on peut proposer le programme suivant :

```
import numpy.random as rd
Tirage=rd.randint(1,11,5)
print(Tirage)
```

De même, si l'on veut simuler le tirage de 5 boules au hasard et avec remise dans une urne contenant 3 boules blanches, 5 boules noires et 2 boules rouges, on peut modéliser le tirage d'une boule par le choix d'un numéro dans $[\![1, 10]\!]$ puis décider que les numéros 1 à 3 correspondent à une boule blanche, que les numéros 4 à 8 correspondent à une boule noire et que les numéros 9 et 10 correspondent à une boule rouge, ce qui nous permet de proposer le programme suivant :

```
import numpy.random as rd
Valeurs=rd.randint(1,11,5)
Tirage=[]
for k in range(0,5):
    if Valeurs[k]<4:
        Tirage=Tirage+['Blanche']
    elif Valeurs[k]<9:
        Tirage=Tirage+['Noire']
    else:
        Tirage=Tirage+['Rouge']
print(Tirage)
```

Enfin, si l'on souhaite simuler le tirage de 5 boules au hasard et avec remise dans une urne contenant une proportion p (avec $p \in]0,1[$) de boules blanches et une proportion $1-p$ de boules noires et afficher le nombre de boules blanches obtenues, on peut proposer le programme suivant :

```
import numpy.random as rd
p=float(input('p='))
Tirage=rd.random(5)
B=0
for k in range(0,5):
    if Tirage[k]<p:
        B=B+1
print(B)
```

F.2. Simulation de variables aléatoires discrètes

La bibliothèque `numpy.random` propose également des simulateurs de la plupart des lois usuelles. Si l'on cherche à simuler une loi usuelle :

- `rd.randint(a,b+1)` permet de simuler une variable aléatoire suivant la loi uniforme sur $[\![a,b]\!]$ (a et b étant deux entiers tels que $a \leqslant b$),
- `rd.binomial(n,p)` permet de simuler une variable aléatoire suivant la loi binomiale $\mathcal{B}(n,p)$ ($n \in \mathbb{N}^*$ et $p \in]0,1[$),
- `rd.geometric(p)` permet de simuler une variable aléatoire suivant la loi géométrique $\mathcal{G}(p)$ ($p \in]0,1[$),
- `rd.poisson(lambda)` permet de simuler une variable aléatoire suivant la loi de Poisson $\mathcal{P}(\lambda)$ ($\lambda \in \mathbb{R}_+^*$).

Plus généralement, si l'on souhaite simuler plusieurs variables aléatoires indépendantes et toutes de même loi, on peut utiliser :

- `rd.loi(paramètres,n)` (où `loi` est à remplacer par l'une des lois mentionnées ci-dessus, `paramètres` étant à remplacer par les paramètres adéquats comme vu précédemment) permet de simuler n variables aléatoires indépendantes et toutes de même loi `loi(paramètres)`,
- `rd.loi(paramètres,[n,p])` permet de renvoyer une matrice de $\mathcal{M}_{n,p}(\mathbb{R})$ dont les coefficients sont une simulations de np variables aléatoires indépendantes et toutes de même loi `loi(paramètres)`.

Par exemple, si l'on souhaite effectuer une simulation de 1000 variables aléatoires $X_1, \dots, X_{1000}$ indépendantes et toutes de même loi géométrique de paramètre 0.4, puis renvoyer la valeur du minimum de $(X_1, \dots, X_{1000})$, on peut procéder ainsi :

```
import numpy.random as rd
import numpy as np
X=rd.geometric(0.4,1000)
print(np.min(X))
```

F.3. Simulation de variables aléatoires à densité

La bibliothèque `numpy.random` propose également des simulateurs de la plupart des lois usuelles. Si l'on cherche à simuler une loi usuelle (ou plusieurs, de même que dans le paragraphe précédent) :

- `rd.random()` permet de simuler une variable aléatoire suivant la loi uniforme sur $[0,1[$,

- `rd.exponential(m)` permet de simuler une variable aléatoire suivant la loi exponentielle d'espérance m, donc de paramètre $\lambda = \frac{1}{m}$ ($m \in \mathbb{R}_+^*$),
- `rd.normal(m,sigma)` permet de simuler une variable aléatoire suivant la loi normale d'espérance m et d'écart-type σ, c'est-à-dire la loi normale $\mathcal{N}(m, \sigma^2)$ ($m \in \mathbb{R}$, $\sigma \in \mathbb{R}_+^*$),
- `rd.gamma(nu,1)` permet de simuler une variable aléatoire suivant la loi gamma $\gamma(\nu)$ ($\nu > 0$).

G. Graphisme en deux dimensions

G.1. La fonction `plt.plot`

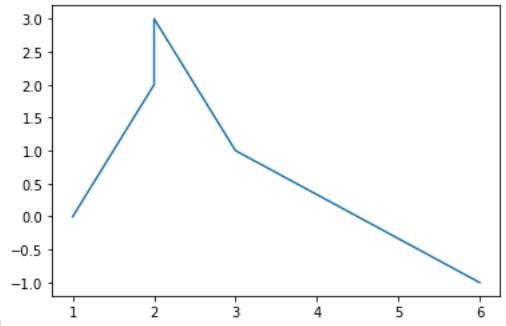
La bibliothèque `matplotlib.pyplot` permet la représentation graphique de suites ou de fonctions. Les tracés de courbes dans le plan peuvent se faire avec la commande `plt.plot` :

- si `x` et `y` sont deux variables contenant des matrices lignes à n colonnes, l'instruction `plt.plot(x,y)` permet de tracer la ligne brisée obtenue en reliant les points de coordonnées `(x[0],y[0])`,..., `(x[n],y[n])` par des segments de droite,
- si `A` et `B` sont deux variables contenant des matrices de format $n \times p$, l'instruction `plt.plot(A,B)` permet de tracer dans un même graphique les n lignes brisées obtenues, pour tout $j \in [\![1, n]\!]$, en reliant les points de coordonnées `(A[0,j],B[0,j])`,..., `(A[n,j],B[n,j])` par des segments de droite.

On peut noter qu'avec les notations précédentes, il est équivalent d'exécuter la commande `plt.plot(A,B)` et d'exécuter les commandes `plt.plot(A[:,j],B[:j])` pour tout j appartenant à $[\![0, n-1]\!]$.
Par exemple, si l'on exécute le programme suivante :

```
import numpy as np
import matplotlib.pyplot as plt
x=np.array([1,2,2,3,6])
y=np.array([0,2,3,1,-1])
plt.plot(x,y)
plt.show()
```

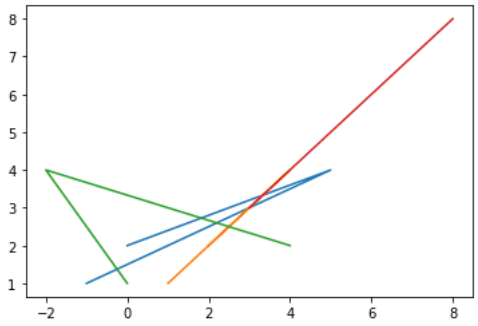
on obtient le graphique suivant :



De même, si l'on exécute le programme suivante :

```
import numpy as np
import matplotlib.pyplot as plt
A=np.array([[-1,1,0,3],[5,4,-2,6],[0,2,4,8]])
B=np.array([[1,1,1,3],[4,4,4,6],[2,2,2,8]])
plt.plot(A,B)
plt.show()
```

on obtient le graphique suivant :



La commande `plt.show()` a pour vocation, à l'exécution du programme, d'ouvrir une fenêtre pour afficher le graphique (une sorte d'équivalent de la commande `print` pour les graphiques). On notera cependant que, dans certains éditeurs, la commande `plt.show()` n'a aucun effet. C'est le cas notamment de Spyder, car c'est un éditeur interactif ; par conséquent nous nous dispenserons de mettre cette commande dans les programmes qui suivent.

G.2. Courbes représentatives de fonctions

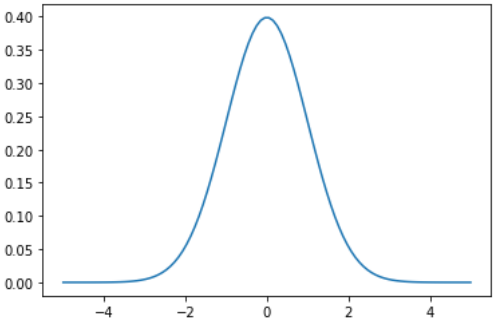
En utilisant la commande `plt.plot`, on peut tracer des allures de courbes représentatives de fonctions numériques réelles. L'idée de base est de placer suffisamment de points, masquant ainsi le côté anguleux pour que la ligne brisée soit suffisamment proche de la courbe.

Si l'on veut représenter l'allure de la courbe représentative d'une fonction $f : [a, b] \to \mathbb{R}$ (a et b étant deux réels tels que $a < b$), on peut donc définir deux variables `x=[x[0],...,x[n]]` et `y=[y[0],...,y[n]]` où `[x[0],...,x[n]]` est une suite strictement croissante telle que `x[0]=a`, `x[n]=b)` et `y[i]=f(x[i])` pour tout i compris entre 0 et n.

Par exemple, si l'on veut représenter graphiquement la fonction $x \mapsto \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{x^2}{2}}$ sur $[-5, 5]$, on peut exécuter le programme suivant :

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(-5,5,100)
y=np.exp(-x**2/2)/np.sqrt(2*np.pi)
plt.plot(x,y)
```

qui retourne le graphique suivant :

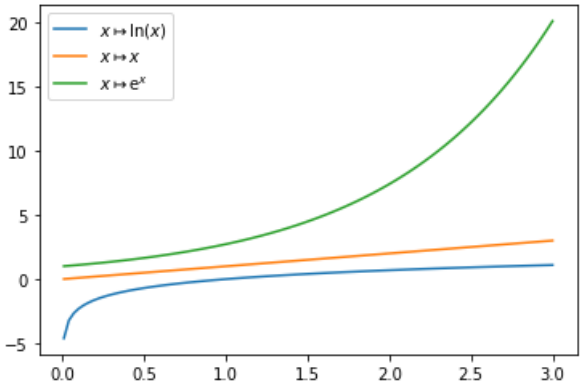


Pour créer le vecteur `x` dont les composantes sont les abscisses des points souhaités, on utilise en général les fonctions `np.linspace` ou `np.arange`.

On peut également représenter graphiquement plusieurs courbes simultanément. Dans ce cas, les vecteurs devront nécessairement être des vecteurs colonnes. Par exemple, si l'on veut illustrer les croissances comparées, on peut saisir les instructions suivantes :

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0.01,3,100)
y=np.zeros([100,3])
y[:,0]=np.log(x)
y[:,1]=x
y[:,2]=np.exp(x)
graph=plt.plot(x,y)
plt.legend(['$x \mapsto \ln(x)$','$x \mapsto x$','$x \mapsto \mathrm{e}^x$'])
```

on obtient la représentation graphique suivante :



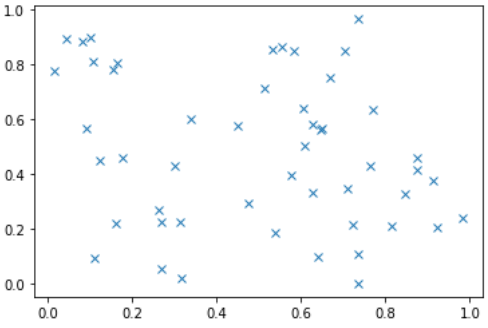
G.3. Nuages de points, représentation graphique de suites

Représenter un nuage de points

Par défaut, si l'on donne deux matrices lignes `x` et `y` de même format, la fonction `plot` relie les points de coordonnées `(x[i],y[i])` et `(x[i+1,y[i+1])`. Il est toutefois possible d'éviter cela si l'on veut représenter un nuage de points.
Il suffit pour cela de saisir les abscisses et les ordonnées des points à représenter dans deux matrices `x` et `y` de même format et d'utiliser la commande `plt.plot(x,y,'x')`, l'option `'x'` suivant les variables permettant de choisir le format d'affichage (dans ce cas, des croix, on peut aussi mettre 'o' pour des cercles par exemple). Par exemple, si l'on saisit les instructions

```
import numpy.random as rd
import matplotlib.pyplot as plt
x=rd.random(50)
y=rd.random(50)
plt.plot(x,y,'x')
```

on obtient la représentation graphique suivante :



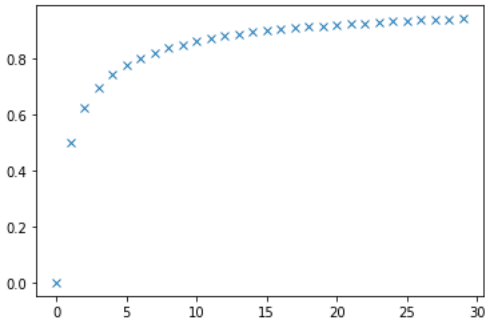
où l'on a représenté 50 points dont les abscisses (respectivement les ordonnées) sont les réalisations de 50 variables aléatoires indépendantes et de même loi uniforme sur $[0,1]$.

Représenter les premiers termes d'une suite

On peut représenter graphiquement les termes d'une suite u en représentant un nuage de points de coordonnées (n, u_n). Par exemple, si l'on souhaite représenter graphiquement les 30 premiers termes de la suite u définie par $u_0 = 0$ et par la relation $u_{n+1} = \frac{1+u_n^2}{2}$, on peut écrire le programme suivant :

```
import matplotlib.pyplot as plt
import numpy as np
n=np.arange(0,30)
u=np.zeros(30)
for k in range(1,30):
    u[k]=(1+u[k-1]**2)/2
plt.plot(n,u,'x')
```

On obtient alors la représentation graphique suivante :



G.4. Fonction de répartition de la loi normale

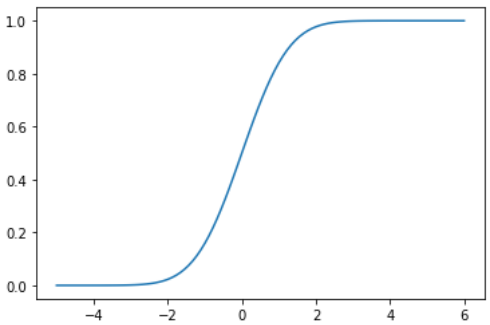
La fonction `ndtr` de la bibliothèque `scipy.special` représente la fonction de répartition Φ de la loi normale centrée réduite :

$$\forall x \in \mathbb{R},\ \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \mathrm{e}^{-\frac{t^2}{2}}\,\mathrm{d}t$$

Par exemple, si l'on saisit les instructions

```
import matplotlib.pyplot as plt
import scipy.special as sp
import numpy as np
x=np.linspace(-5,6,100)
y=sp.ndtr(x)
plt.plot(x,y)
```

on obtient la représentation graphique suivante :



G.5. Graphes 3D de fonctions de deux variables

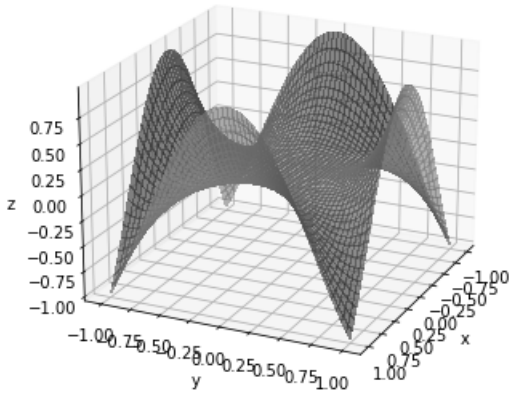
Python offre la possibilité d'obtenir des représentations graphiques de surfaces, à l'aide de la fonction `Axe3D` et `fplot3d`. La connaissance de ses arguments n'est pas exigible.
On retiendra néanmoins que, pour représenter graphiquement une fonction f de deux variables :

- il faut définir la fonction `f`,
- il faut créer deux variables `x=[x[0],...,[x[n-1]]` et `y=[y[0,...,y[p-1]]` désignant les valeurs de x et de y pour lesquelles on sera amené à calculer $f(x,y)$ (en prenant suffisamment de points pour obtenir un graphe réaliste et éviter les graphes anguleux),
- on crée ensuite une grille de points (x_i, y_j) pour $(i,j) \in [\![0, n-1]\!] \times [\![0, p-1]\!]$:
  la commande `X,Y=np.meshgrid(x,y)` renvoie deux matrices `X` et `Y` appartenant à $\mathcal{M}_{p,n}(\mathbb{R})$ tels que :
  — les p lignes de `X` sont toutes égales à `x`,
  — les n colonnes de `Y` sont toutes égales à `y` (à la transposition près),
- on crée le vecteur `Z=f(X,Y)`,
- on représente graphiquement la surface avec la commande `plot_surface(X,Y,Z)`.

Par exemple, si l'on exécute le script suivant :

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
ax=Axes3D(plt.figure())
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
def f(x, y):
    return(x**4-3*x**2*y**2+y**4)
x=np.linspace(-1, 1, 200)
y=np.linspace(-1, 1, 200)
X,Y=np.meshgrid(x,y)
Z=f(X,Y)
ax.plot_surface(X,Y,Z)
```

On obtient alors la représentation graphique suivante (ici de la fonction $(x,y) \mapsto x^4 - 3x^2y^2 + y^4$ sur $[-1,1]^2$).



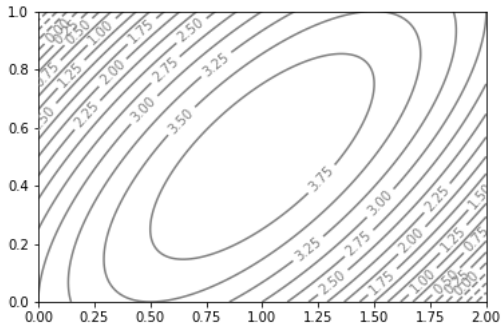
Noter que, comme il s'agit d'une représentation en deux dimensions d'un graphique qui en a trois, il peut être intéressant de tourner autour des axes pour mieux visualiser la nappe. Pour pouvoir bénéficier de la partie interactive des graphiques de Spyder, il faut modifier les préférences par défaut (menu Préférences->Python Console->Graphics->Graphics backend : Qt) : cela permet ensuite de déplacer la figure à l'aide de la souris.

G.6. Lignes de niveau

La bibliothèque `matplotlib.pyplot` permet également de représenter graphiquement des lignes de niveau d'une fonction de deux variables, à l'aide de la fonction `contour`. Les différentes options n'étant pas exigibles, nous ne les détaillerons pas ici, mais on pourra s'aider des exemples suivants pour en comprendre l'usage :
Exemple 1. L'exécution du programme suivant permet de définir la fonction $f : (x, y) \mapsto 3-2x^2-4x^2+4xy+2x$ et de représenter, à l'aide de la commande `plt.contour(X,Y,Z),20)`, 20 lignes de niveaux (les valeurs étant choisies par Python pour rendre un graphique harmonieux)

```
import matplotlib.pyplot as plt
import numpy as np
def f(x, y):
    return(3-2*x**2-4*y**2+4*x*y+2*x)
x=np.linspace(0,2,200)
y=np.linspace(0,1,200)
X,Y=np.meshgrid(x,y)
Z=f(X,Y)
graphe=plt.contour(X,Y,Z,20,colors='grey')
plt.clabel(graphe,inline=1,fontsize=10,fmt='%3.2f')
```

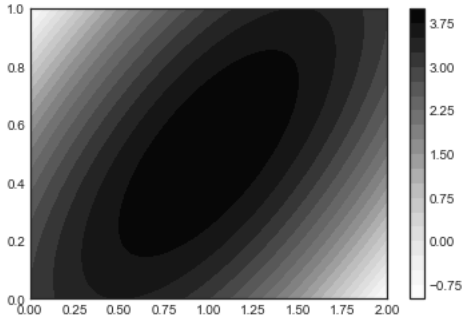
et renvoie la représentation graphique suivante :



Exemple 2. L'exécution du programme suivant fait la même chose, avec des options différentes (couleurs entre les lignes de niveau, plutôt que d'afficher les valeurs des lignes de niveau)

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import numpy as np
def f(x,y):
    return(3-2*x**2-4*y**2+4*x*y+2*x)
x=np.linspace(0,2,200)
y=np.linspace(0,1,100)
X,Y=np.meshgrid(x,y)
Z=f(X,Y)
plt.contourf(X,Y,Z,20,cmap='Greys')
plt.colorbar()
```

et renvoie la représentation graphique suivante :



G.7. Gradients

La bibliothèque `matplotlib.pyplot` permet enfin de représenter graphiquement des gradients d'une fonction de deux variables, à l'aide de la fonction `quiver` :

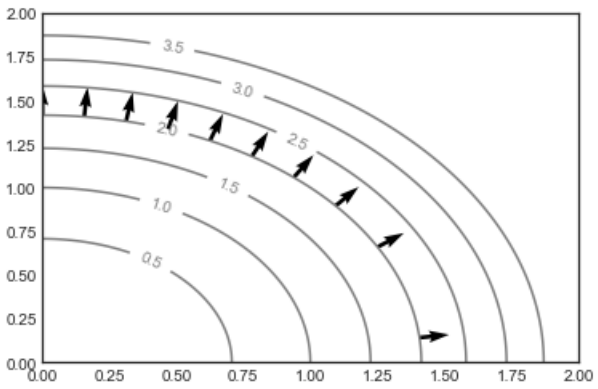
- si `x,y,u,v` désignent des réels, l'exécution de `plt.quiver(x,y,u,v)` représente graphiquement le représentant du vecteur de coordonnées `[u,v]` et d'origine `[x,y]`,
- si `x,y,u,v` désignent des tableaux de réels tous de même format, l'exécution de `plt.quiver(x,y,u,v)` représente graphiquement les représentants des vecteurs de coordonnées `[u(i,j),v(i,j)]` et d'origine `[x(i,j),y(i,j)]`.

Encore une fois, les différentes options ne sont pas exigibles et on s'aidera des exemples suivants pour en comprendre l'usage :

Exemple 1. L'exécution du programme suivant

```
import matplotlib.pyplot as plt
import numpy as np
def f(x,y):
    return x**2+y**2
x=np.linspace(0,2,100)
y=np.linspace(0,2,100)
X,Y=np.meshgrid(x,y)
graphe=plt.contour(X,Y,f(X,Y),np.arange(0,4,0.5),colors='gray')
xx=np.linspace(0,np.sqrt(1.98),10)
yy=np.sqrt(2-xx**2)
u=2*xx
v=2*yy
plt.quiver(xx,yy,u,v)
plt.clabel(graphe,inline=1,fontsize=10,fmt='%1.1f')
```

renvoie la représentation graphique suivante :

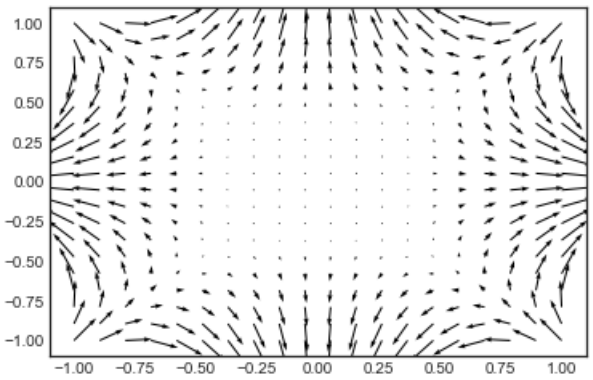


On y a représenté les lignes de niveau $k \in \{0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5\}$ de la fonction $f : (x, y) \mapsto x^2 + y^2$ sur $[0, 2]^2$ (à l'aide de la commande `plt.contour(X,Y,f(X,Y),np.arange(0,4,0.5))` puis, en remarquant que $\nabla f(x, y) = (2x, 2y)$ pour tout $(x, y) \in \mathbb{R}^2$, on a représenté les gradients de f en certains points de la ligne de niveau 2 (d'équation $x^2 + y^2 = 2$) ; on peut y voir que les gradients en un point (x, y) sont orthogonaux à la ligne de niveau contenant (x, y).

Exemple 2. L'exécution du programme

```
import matplotlib.pyplot as plt
import numpy as np
def f(x,y):
    return x**4-3*x**2*y**2+y**4
x=np.linspace(-1,1,20)
y=np.linspace(-1,1,20)
X,Y=np.meshgrid(x,y)
Z=f(X,Y)
u=4*X**3-6*X*Y**2
v=-6*X**2*Y+4*Y**3
plt.quiver(X,Y,u,v)
plt.show()
```

renvoie la représentation graphique suivante, semblant indiquer la présence d'un point selle en $(0,0)$ (les flèches pointant vers $(0,0)$ indique la présence de directions dans lesquelles $f(x,y) < f(0,0)$ au voisinage de $(0,0)$, tandis que les flèches partant de $(0,0)$ indiquent la présence de directions dans lesquelles $f(x,y) > f(0,0)$ au voisinage de $(0,0)$) :



G.8. Histogrammes

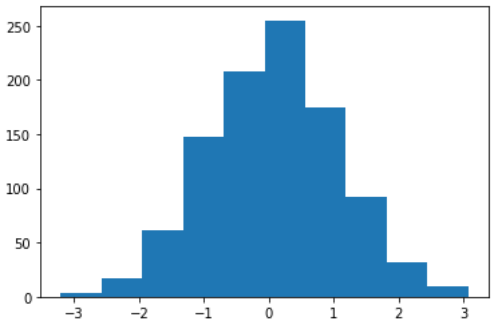
Si l'on dispose d'une série statistique comportant un certain nombre de relevés $x_1, \dots, x_n$ (non forcément distincts), on peut chercher à représenter cette série statistique par un histogramme. Rappelons que si l'on se donne un intervalle $[a,b]$ et des classes $[a_0,a_1],[a_1,a_2[,\dots,[a_{n-2},a_{n-1}[,[a_{n-1},a_n]$ (avec $a = a_0 < a_1 < \cdots < a_n = b$), un histogramme sera obtenu en représentant les rectangles dont les bases sont les différentes classes et dont les aires sont proportionnelles à l'effectif de la classe (c'est-à-dire au nombre de relevés appartenant à la classe) ; notons également que, si les classes ont toutes la même amplitude, il est alors équivalent de dire que la hauteur de chaque rectangle est proportionnelle à l'effectif de la classe.

Pour représenter un histogramme, on peut utiliser la fonction `hist` de la bibiothèque `matplotlib.pyplot`.

Par exemple, l'exécution du programme suivant :

```
import matplotlib.pyplot as plt
import numpy.random as rd
X=rd.normal(0,1,1000)
plt.hist(X)
```

permet de faire une simulation de 1000 variables aléatoires indépendantes et de même loi normale centrée réduite puis d'afficher l'histogramme de la série ainsi obtenue (permettant d'avoir l'allure de la distribution empirique de la loi normale centrée réduite sur un échantillon de 1000 valeurs) :

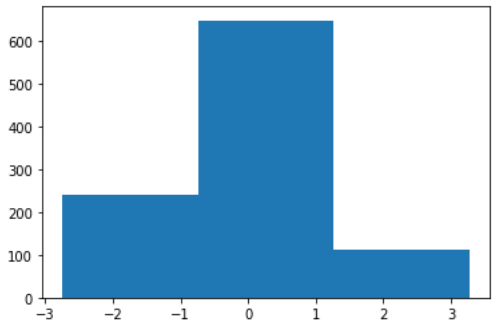


Sans autre précision, Python choisit automatiquement les classes dans lesquelles les données seront réparties et la somme des aires des différents rectangles obtenus est égale à l'effectif total. Il est possible de changer cela en indiquant :

- le nombre de classes, avec l'option `bins=n` (où n est un entier naturel non nul), Python choisissant automatiquement les extrémités ; par exemple, en exécutant

```
import matplotlib.pyplot as plt
import numpy.random as rd
X=rd.normal(0,1,1000)
plt.hist(X,bins=3)
```

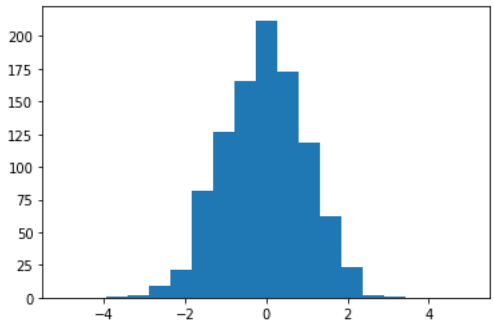
  on obtient :



- la liste `x` des extrémités des classes, avec l'option `bins=x` où `x` est un tableau contenant la liste des extrémités des classes ; par exemple, en exécutant

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rd
X=rd.normal(0,1,1000)
plt.hist(X,bins=np.linspace(-5,5,20))
```

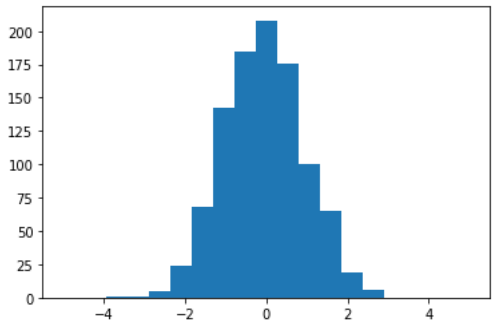
  on obtient :



- la normalisation, avec l'option `density=True` la somme des aires sera égale à 1, ce qui permet de mieux rendre compte de l'aspect probabiliste du graphique ; par exemple, en exécutant

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rd
X=rd.normal(0,1,1000)
plt.hist(X,bins=np.linspace(-5,5,20),
    density=True,edgecolor='black',
    color='lightgray')
```

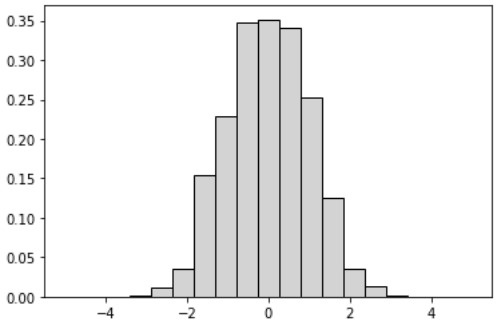
on obtient :



- les options de couleurs (bords, couleur des rectangles,...) ; par exemple, en exécutant

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rd
X=rd.normal(0,1,1000)
plt.hist(X,bins=np.linspace(-5,5,20),
    density=True,edgecolor='black',
    color='lightgray')
```

on obtient :

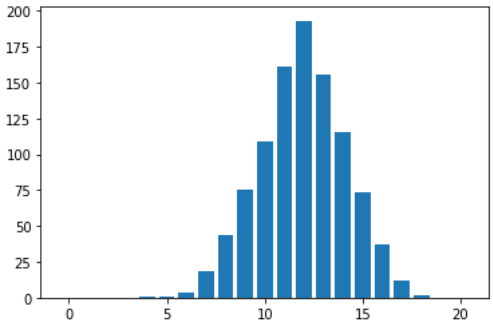


G.9. Diagrammes en bâtons

Si l'on dispose d'une série statistique $(x_i, n_i)_{1\leqslant i\leqslant p}$ où $x_1, \ldots, x_p$ sont les relevés distincts et $n_1, \ldots, n_p$ sont les effectifs, on peut représenter cette série par un diagramme en bâtons en utilisant la fonction `bar` de `matplotlib.pyplot`. Par exemple, en exécutant

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rd
X=rd.binomial(20,0.6,1000)
N=np.arange(0,21)
Effectif=np.zeros(21)
for k in range(0,21):
    Effectif[k]=sum(X==k)
plt.bar(N,Effectif)
```

on obtient un diagramme en bâtons représentant les relevés et les effectifs dans une suite de 1000 épreuves indépendantes et de même loi binomiale $\mathcal{B}(20, 0.6)$.



H. Systèmes linéaires et diagonalisation

H.1. Systèmes linéaires

Si A est une matrice inversible de $\mathcal{M}_n(\mathbb{R})$ et B un élément de $\mathcal{M}_{n,1}(\mathbb{R})$, Python permet de déterminer l'unique solution de l'équation $AX = B$ dans $\mathcal{M}_{n,1}(\mathbb{R})$ à l'aide de la fonction `solve` de la bibliothèque `numpy.linalg` : l'exécution de la commande `al.solve(A,B)` renvoie alors la solution recherchée.
Par exemple, si l'on exécute le programme suivant :

```
import numpy.linalg as al
import numpy as np
A=np.array([[1,2],[2,1]])
B=np.array([1,1])
print(al.solve(A,B))
```

on obtient le résultat suivant :

```
[0.33333333 0.33333333]
```

H.2. Valeurs propres et vecteurs propres d'une matrice carrée

La fonction `eig` de la bibliothèque `numpy.linalg` permet de déterminer les valeurs propres et des vecteurs propres d'une matrice carrée. Si A est une matrice carrée d'ordre n, alors la commande `S,P=al.eig(A)` renvoie un tableau `S` dont les coefficients sont les valeurs propres de A et une matrice carrée P dont les colonnes sont des vecteurs propres de A, respectivement associés aux coefficients correspondants de `S`. On se souviendra que :

- si A est diagonalisable, alors le nombre d'occurrences d'une valeur propre dans `S` est égal à la dimension du sous-espace propre associé et `P` représente la matrice de passage de la base canonique à une base formée de vecteurs propres de A et, en notant D la matrice dont `S` est la diagonale, on a : $A = PDP^{-1}$,
- si A n'est pas diagonalisable, la matrice `P` obtenue n'est pas inversible et le nombre d'occurrences d'une valeur propre dans `S` peut être supérieur à la dimension du sous-espace propre associé.

Par exemple, si l'on exécute le programme suivant :

```
import numpy.linalg as al
import numpy as np
A=np.zeros([3,3])
A[np.arange(0,3),np.arange(0,3)]=np.arange(1,4)
S,P=al.eig(A)
print(S)
print(P)
```

on obtient le résultat suivant, ce qui n'est guère surprenant puisqu'on cherche ici les valeurs propres et des vecteurs propres de la matrice $A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$:

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

De même, en exécutant le programme suivant :

```
import numpy.linalg as al
import numpy as np
A=np.ones([3,3])
S,P=al.eig(A)
print(S)
print(P)
```

on obtient le résultat suivant :

```
[-2.22044605e-16 3.00000000e+00 0.00000000e+00]
[[-0.81649658 0.57735027  0. ]
 [ 0.40824829 0.57735027 -0.70710678]
 [ 0.40824829 0.57735027  0.70710678]]
```

En tenant compte des erreurs d'arrondis, il semble donc raisonnable de conjecturer que les valeurs propres de A sont 0 et 3 et que leurs sous-espaces propres associés respectifs sont :

$$E_0(A) = \text{Vect}\left(\begin{pmatrix} -2 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}\right) \quad \text{et} \quad E_3(A) = \text{Vect}\left(\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\right)$$

Enfin, en exécutant le programme suivant :

```
import numpy.linalg as al
import numpy as np
A=np.zeros([4,4])
A[np.arange(0,3),np.arange(1,4)]=np.ones(3)
S,P=al.eig(A)
print(S)
print(P)
```

on obtient le résultat suivant :

```
[0. 0. 0. 0.]
[[ 1.00000000e+000 -1.00000000e+000  1.00000000e+000 -1.00000000e+000]
 [ 0.00000000e+000  4.00833672e-292 -4.00833672e-292  4.00833672e-292]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000 -0.00000000e+000]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000  0.00000000e+000]]
```

En tenant compte des erreurs d'arrondis, cela signifie que 0 est la seule valeur propre de de la matrice

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

et que son sous-espace propre associé est la droite vectorielle engendrée par le vecteur $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$.

Sommaire