Abderrahmane Fadil

Mathématiques et statistiques appliquées avec Python

Cours et exercices corrigés

ellipses

Mathématiques et statistiques appliquées avec Python

Cours et exercices corrigés

Références sciences

Mathématiques et statistiques appliquées avec Python

Cours et exercices corrigés

Abderrahmane Fadil

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-104280
Dépôt légal : mai 2025

www.editions-ellipses.fr

Avant-propos

Les mathématiques constituent les bases fondamentales des avancées scientifiques et technologiques de tous les temps. Elles ont permis le développement de modèles formels et de méthodes rigoureuses afin d'appréhender les situations réelles et la résolution des problèmes dans divers domaines : scientifiques, économiques, biologiques, logistiques et bien d'autres.

Le besoin croissant de mettre en œuvre ces méthodes à travers des disciplines, également basées sur les mathématiques, a permis ainsi le développement de l'algorithmique, des environnements de développement, de la gestion et l'analyse de données, etc. Dans cette dynamique, on aboutit à ce jour à des avancées considérables en data sciences, en Deep Learning et en intelligence artificielle.

Ce livre s'adresse à des étudiants et élèves-ingénieurs en :

- licence dans des filières scientifiques ;
- écoles de commerce ;
- cycle préparatoire des écoles d'ingénieurs ;
- formation en data sciences ;
- auto-formation.

Ce livre a pour objectif d'aborder des chapitres de mathématiques et de statistiques descriptives :

- rappels des bases et des concepts ;
- méthodes de résolution ;
- pratiques avec développement de scripts Python correspondants.

Le choix de Python est motivé par les facilités d'apprentissage et de mise en œuvre. Il contient des librairies riches et étoffées. Il s'est également et naturellement imposé dans les développements en data sciences et intelligence artificielle.

Table des matières

Chapitre 1

Notions de base du développement Python

1 Introduction

Avant d'aborder les outils et méthodes de visualisation de données (graphiques, courbes, …), de mise en application de fondements mathématiques et statistiques (algèbre linéaire, statistiques descriptives, …), d'importation/exportation de structures de données complexes (tableaux, matrices, *tensors*, …) ou encore l'introduction au *Machine Learning* (analyse de données, …) , il est utile de faire quelques rappels des notions de bases du développement Python.

2 Script Python

Un script informatique est une séquence d'instructions, écrites dans un langage dit de scripts et qui fait appel à un outil dit interpréteur afin d'exécuter les instructions. L'interpréteur est souvent associé à l'éditeur de code, qui permet d'exécuter la séquence d'instructions en interprétant les instructions l'une après l'autre de façon séquentielle. L'interpréteur ne crée pas un exécutable, contrairement à un compilateur et c'est là la différence majeure avec un langage compilé comme le C++ par exemple, et donc pour toute nouvelle exécution du script l'interpréteur est appelé et refait le même travail, à savoir lecture et interprétation des instructions l'une après l'autre de façon séquentielle.

Un programme écrit sous Python est un script et a besoin donc d'un programme auxiliaire, dit interpréteur, pour lire et exécuter au fur et à mesure les instructions du script.

Un script Python peut utiliser des fonctions et procédures prédéfinies, ce qui facilite la tâche du développeur. Ces fonctions sont classées par catégories et regroupées sous des librairies importables. Pour les éléments de langage, ces librairies sont dites aussi des modules, des bibliothèques ou encore des packages.

Pour pouvoir importer ces librairies et utiliser leurs fonctions prédéfinies il faut les avoir auparavant installées sous l'éditeur Python utilisé pour le développement.

Un script doit tenir compte également des contraintes d'environnement telles que la source de données lors d'importation de données ou encore le modèle de données lors d'une étape d'apprentissage par exemple en *Deep Learning*.

2.1 Représentation systémique

La *figure 1* ci-après est une représentation systémique d'un script. En effet, une séquence d'instructions est définie afin de réaliser un traitement de données.

Ce traitement agit sur des données en entrée e_1, e_2 , …, e_n où n est le nombre de ces données en entrée (c'est-à-dire les données qui alimentent le script).

Le traitement utilise ces données pour en produire d'autres. À la fin du traitement, ces données produites sont les données en sortie du script s_1, s_2, …, s_m où m est le nombre de ces données en sortie qui correspondent aux résultats escomptés du traitement (cf. *figure 1* ci-dessous).

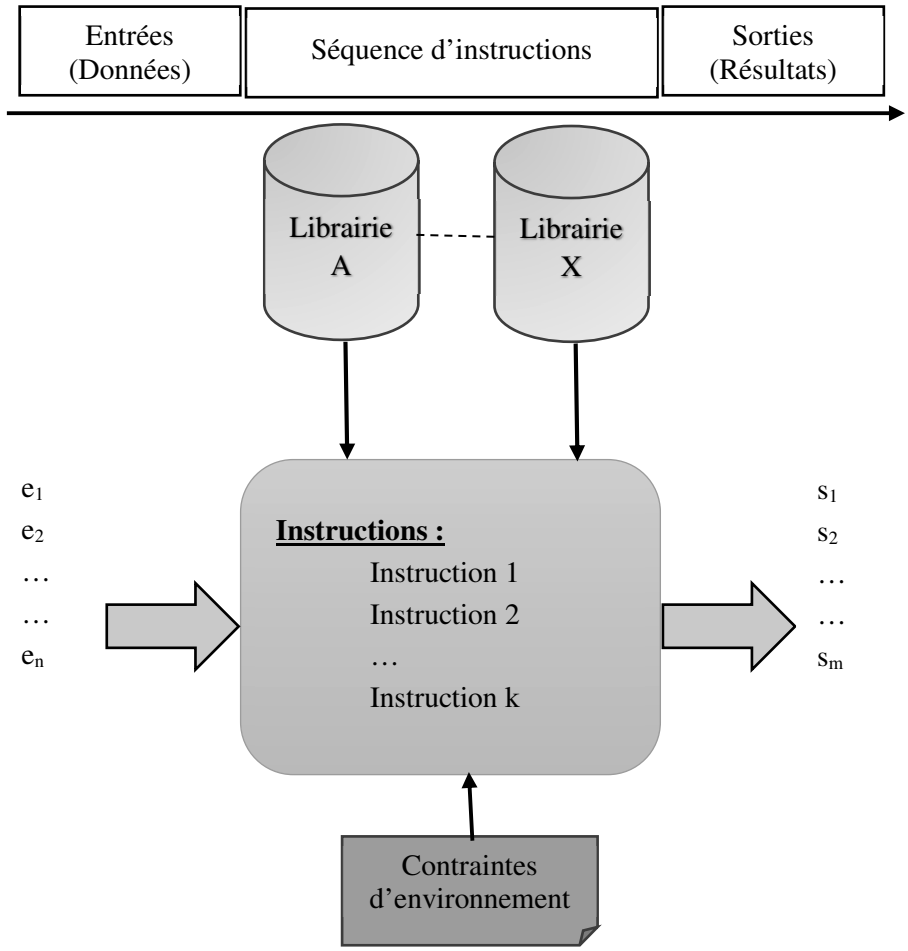


Figure 1. *Script modèle*

Voici, ci-dessous un exemple où on retrouve tous les éléments du script modèle, à savoir :

- importation de librairies ;
- contraintes d'environnement comme l'importation de données à partir d'un fichier texte ;
- les entrées / sorties ;
- les instructions de traitement.

2.2 Script Python

```
import numpy as np
import os

# Création de la matrice A
A=np.zeros(shape=(3,3))

# Chargement des données à partir d'une source externe
os.chdir("C:\\Exemples\Exemples Maths Data")
A=np.loadtxt("Matrice_ load.txt",delimiter="\t",dtype=float)
print("A = \n",A)

# taille et transpose d'une matrice
tA=np.zeros(shape=(A.shape[1],A.shape[0]))
tA=np.transpose(A)
print("tA = \n",tA)

# Déterminant d'une matrice carrée
d=np.linalg.det(A)
print("d = ",d)

# Valeurs propres et vecteurs propres
M=np.random.random(size=(2,2))*10
print("M = \n",M)
# vvp : vecteur des valeurs propres
# mvp : matrice des vecteurs propres (vecteurs colonnes)
vvp, mvp=np.linalg.eig(M)
```

```
print("vvp = ", vvp)
print("mvp = \n", mvp)

# M.v = l*v où v est un vecteur propre et l la valeur propre
correspondante
for i in range(M.shape[1]):
    vi=mvp[:,i]
    li=vvp[i]
    print("vérification n°", i+1, " : ",
          np.all(np.isclose(np.dot(M,vi),li*vi)))
```

2.3 Résultat

```
A =
 [[ 45.   62.  168. ]
  [-34.   15.5 187. ]
  [  1.   -1.2   0. ]]

tA =
 [[ 45.  -34.    1. ]
  [ 62.   15.5  -1.2]
  [168.  187.    0. ]]

d =  25942.4

M =
 [[2.45 1.87]
  [9.04 7.68]]

vvp =  [0.19 9.94]
mvp =
 [[-0.63 -0.24 ]
  [ 0.76 -0.97]]

vérification n° 1  :  True
vérification n° 2  :  True
```

3 Les instructions fondamentales

3.1 L'affectation

L'opération d'affectation permet d'affecter à une variable une valeur constante, ou le résultat d'une expression, ou le résultat d'une fonction, en utilisant l'opérateur = où le receveur est à gauche de l'opérateur et la valeur à affecter est à droite.

Exemples

- y = 20 ;
- y = (x**2 +1) ;
- y = numpy.sin((np.pi /3) * x) ;
- etc.

3.2 Les entrées / sorties

Les fonctions d'entrées / sorties qui permettent la lecture et l'écriture de données sont les suivantes :

Fonction	Syntaxe	Remarques et exemples
input()	v = input([*message*])	C'est l'instruction de saisie. v : est une variable qui reçoit la valeur saisie. [] : signifie facultatif. *message* : est un message à destination de l'utilisateur. Exemple : x = input("Saisir x : ")
print()	print(*arguments*)	C'est l'instruction d'affichage. *arguments* : la liste des arguments à afficher. Le séparateur des arguments est **,** (la virgule). Exemple : print("v = ", round(v, 2))

Exemple d'affectation et d'entrées / sorties

```
from math import sqrt

x=float(input("Saisir la valeur réelle de x : "))

# Instructions d'affectation
y=x**3
v=sqrt(y)

print("v = ",round(v,2))
```

Résultat

```
v =  111.18
```

3.3 Les structures de contrôle conditionnelles

Une structure de contrôle conditionnelle permet, selon le résultat de test d'une ou plusieurs conditions, à un script :

- de réaliser un bloc d'instructions lorsque une condition est vraie puis l'interpréteur passe à la suite du script ;
- de réaliser un bloc d'instructions lorsque une condition est vraie sinon un autre bloc d'instructions est exécuté puis l'interpréteur passe à la suite du script ;
    - de réaliser un bloc d'instructions parmi plusieurs blocs possibles (bifurcation).

Cela correspond à trois syntaxes du contrôle conditionnel.

3.3.1 Syntaxes

Syntaxe 1	**Remarque**
if Condition : *Bloc instructions*	Si *Condition* est vraie alors un bloc d'instructions est exécuté, sinon l'interpréteur passe à la suite du script.

Syntaxe 2	**Remarque**
if Condition : *Bloc instructions B1* *else :* *Bloc instructions B2*	Si *Condition* est vraie alors le premier bloc d'instructions est exécuté, sinon le deuxième bloc d'instructions est exécuté puis l'interpréteur passe à la suite du script.

Syntaxe 3	**Remarque**
if Condition$_1$: *Bloc instructions B_1* *elif Condition$_2$:* *Bloc instructions B_2* *...* *elif Condition$_n$:* *Bloc instructions B_n* *else :* *Bloc instructions B_{n+1}*	Si *Condition$_1$* est vraie alors le bloc d'instructions B_1 est exécuté, sinon si *Condition$_2$* est vraie alors le bloc d'instructions B_2 est exécuté, ainsi de suite, sinon le bloc d'instructions B_{n+1} est exécuté puis l'interpréteur passe à la suite du script.

3.3.2 Exemple

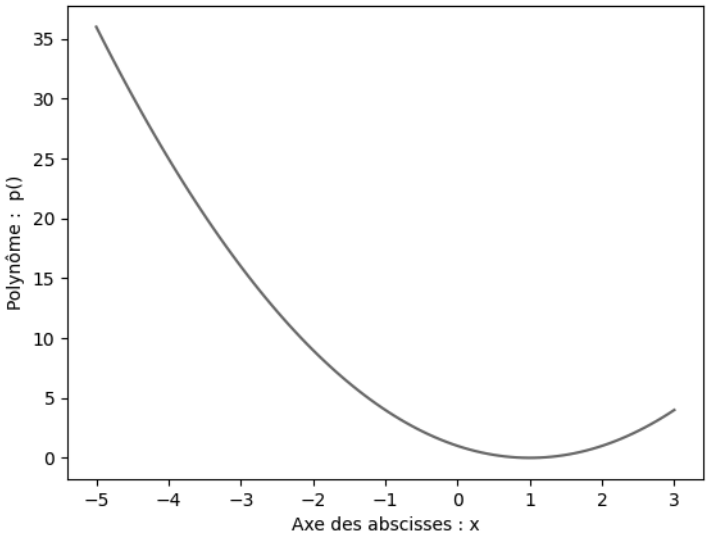
```
# Importation des librairies
import matplotlib.pyplot as plt
import numpy as np
# Définition des fonctions
def Est_Reel(valeur):
    try:
        float(valeur)
        return True
    except ValueError:
        return False

def Saisie_R():
            R=input("Saisir un réel : ")
```

```
        while(not Est_Reel(R)):
            R=input("Élément saisi n'est pas un réel. Recommencer : ")
        R=float(R)
        return(R)

# Saisie des coefficients
print("Coefficient a")
a=Saisie_R()
print("Coefficient b")
b=Saisie_R()
print("Coefficient c")
c=Saisie_R()

# Résolution
if a!=0:
        delta=b**2-4*a*c
        if delta<0:
                print("\nPas de racine réelle !")
        elif delta==0:
                x0=round(-b/(2*a),2)
                print("\nRacine unique x0 = ",x0)
        else:
                x1=round((-b-np.sqrt(delta))/(2*a),2)
                x2=round((-b+np.sqrt(delta))/(2*a),2)
                print("\nDeux racines : x1 = ",x1," ;  x2 = ",x2)
else:
        print("Ce n'est pas un polynôme de second degré")
# Graphique
p=np.poly1d([a,b,c])
print("Courbe du polynôme p(x) : ",p)
x=np.linspace(-5,3,100)
plt.plot(x,p(x))
plt.ylabel("Polynôme :  p()")
plt.xlabel("Axe des abscisses : x")
plt.show()
```

Résultat obtenu pour a = 1, b = −2 et c = 1

Coefficient a
Saisir un réel : 1
Coefficient b
Saisir un réel : -2
Coefficient c
Saisir un réel : 1

Racine unique x0 = 1.0
Courbe du polynôme p(x) : 1 x^2 - 2 x + 1



3.4 Les structures de contrôle répétitives

Une structure de contrôle répétitive permet à un script de répéter, c'est-à-dire itérer ou encore boucler, un bloc d'instructions un nombre n de fois :

- lorsque n est connu à l'avance, il s'agit alors de la boucle *for* ;
- lorsque n n'est pas connu à l'avance, il s'agit d'une boucle *while*.

3.4.1 La boucle for

La boucle *for* sous Python utilise un itérateur nommé *range()*. L'itérateur énumère des entiers dans un intervalle donné [start ;end[. Cette énumération se fait pas à pas et le pas est nommé *step*.

Syntaxe générale de la boucler *for* sous Python :

```
for i in range([start,] end [, step]) :
    bloc instructions
```

Remarque

Les crochets [] dans la syntaxe générale indiquent l'aspect facultatif et non obligatoire selon les situations considérées.

- *range(start, end, step)* # retourne les entiers de *start* à *end*−1, avec le pas *step* ;
- *range(start, end)* # retourne les entiers de *start* à *end*−1, avec le pas *step* = 1 ;
- *range(end)* # de même, avec *start* = 0 et avec le pas *step* = 1.

3.4.2 Exemple de la suite de Catalan

Ici, on utilise la boucle *for* afin de calculer les n premiers nombres de Catalan telle que la suite de Catalan est définie comme suit :

$$\begin{cases} C_0 = 1 \\ C_n = C_{n-1}\dfrac{2(2n-1)}{n+1} \quad pour\ n \geq 1 \end{cases}$$

Script

```
# Définition de la fonction de saisie
def Saisie_E(e):
    v=input("Saisir un entier : ")
    while (not v.isnumeric()) or (int(v)<e):
        v=input("Re-saisir un entier : ")
    v=int(v)
    return(v)

n = Saisie_E(1)
C=1
print("n =",0,"  , C = ",int(C))

# Boucle pour
```

```
for i in range(1,n+1):
    C=C*(2 * (2 * i - 1))/(i + 1)
    print("n =",i," , C = ",int(C))
```

Test du script

```
Saisir un entier : 10
n = 0   , C =  1
n = 1   , C =  1
n = 2   , C =  2
n = 3   , C =  5
n = 4   , C =  14
n = 5   , C =  42
n = 6   , C =  132
n = 7   , C =  429
n = 8   , C =  1430
n = 9   , C =  4862
n = 10   , C =  16796
```

3.4.3 La boucle while

La boucle *while* sous Python permet de répéter un bloc d'instructions. Le nombre d'itérations n'est pas connu à l'avance, cependant la répétition est conditionnée par la véracité d'une ou plusieurs conditions.

Syntaxe générale de la boucler *while* sous Python :

```
while (Condition) :
    bloc instructions
```

3.4.4 Exemple de script de test statistique avec while

```
# Importation des librairies
import random
import pandas as pd
import matplotlib.pyplot as plt

rPearson=0
matricule = pd.Series(range(101,111))
```

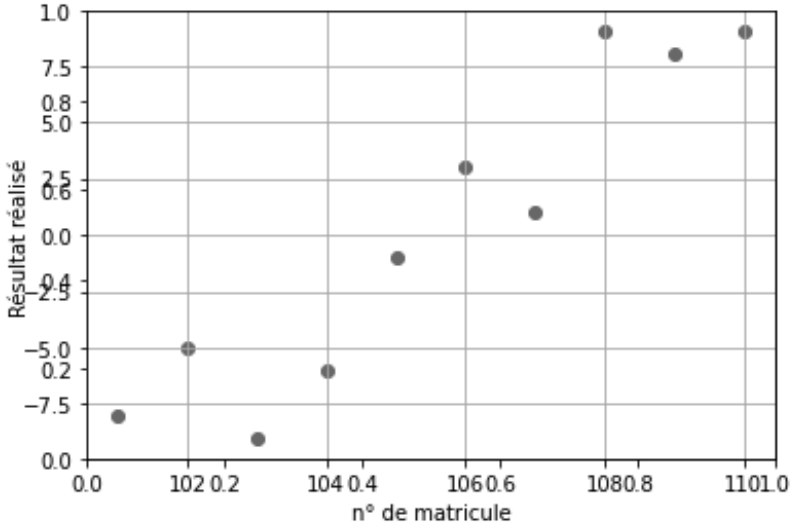
```
#Tant que le coefficient de corrélation est <0.75, regénérer une nouvelle
série
while(abs(rPearson)<0.90):
    result=pd.Series([random.randint(-10, 10) for i in range(10)])
    rPearson=matricule.corr(result)
    print("rPearson = ",round(rPearson,2),"\n","rPearson >= 0.90
?",rPearson >= 0.90)

# Construction d'un Data Frame
xy = pd.DataFrame({'Matricule': matricule, 'Résultat': result})
print(xy)
print("rPearson = ", round(rPearson,2))

# Graphique
print("Graphique")
plt.ylabel("Résultat réalisé")
plt.xlabel("n° de matricule")
axes = plt.axes()
axes.grid()
plt.scatter(matricule,result)
plt.show()
```

3.4.5 Test statistique

```
rPearson =  -0.08
 rPearson >= 0.90 ? False
rPearson =  0.21
 rPearson >= 0.90 ? False
rPearson =  0.19
 rPearson >= 0.90 ? False
rPearson =  0.94
 rPearson >= 0.90 ? True

   Matricule   Résultat
0        101        -8
```

1 102 -5
2 103 -9
3 104 -6
4 105 -1
5 106 3
6 107 1
7 108 9
8 109 8
9 110 9
rPearson = 0.94

Graphique



4 Les structures de données

Il est possible de regrouper plusieurs données élémentaires dans des structures sous Python. Ces structures sont dites des structures de données. Il en existe plusieurs :

- listes ;
- tableaux ;
- ensembles ;
- tuples ;
- dictionnaires ;
- mots (chaînes de caractères alphanumériques) ;

- piles ;
- files.

D'autres structures plus avancées sont disponibles sous Python, à condition d'importer les librairies adéquates, comme les *tensors* ou encore les *Data Frames* utilisés spécialement en *Machine Learning* et *Deep Learning*. On en utilisera certaines dans des exemples.

Le choix et l'utilisation d'une structure ou d'une autre dépend de plusieurs facteurs, comme :

- le contexte d'utilisation (applications scientifiques, mathématiques, statistiques, analyse textuelle, …) ;
- l'importance de l'ordre de stockage des données dans une structure (repérer un $i^{ème}$ élément de la structure, ordre d'arrivée et de sortie, …)
- le type de données dans la structure (numérique, alphanumérique, homogène, hétérogène, …).

4.1 Les listes

4.1.1 Définition et caractéristiques

Une liste est une collection de données telle que :

- une liste est de type *list* ;
- les données d'une liste peuvent être de types différents (données hétérogènes) ;
- les données d'une liste sont indexées (on peut repérer un ième élément dans la liste) ;
- le premier élément, c'est-à-dire la première donnée d'une liste se trouve à l'emplacement dont l'index est 0 ;
- les éléments sont accessibles et modifiables (lecture, écriture) ;
- la taille ou le nombre d'éléments de la liste est dynamique (on peut ajouter ou supprimer des éléments) ;
- l'ajout d'éléments se fait toujours à la fin de la liste ;
- on peut manipuler une liste de données élémentaires, ou bien une liste de listes ou bien une liste de listes et de données élémentaires.

4.1.2 Création et initialisation de listes

Voici, sous Shell, quelques créations/initialisations de listes :

```
# Création d'une liste vide
l=[]
```

```
# Initialisation d'une liste
l=[-1]*10
l
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

# Création et initialisation d'une liste
L=[5,'a','b']
L
[5, 'a', 'b']
La=list(range(3,39,3))
La
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]
S=['a','x50',20,True,15.5, L]
S
['a', 'x50', 20, True, 15.5, [5, 'a', 'b']]

# Type de données
type(S)
<class 'list'>
for i in range(len(S)):
    print("type de S[",i,"] : ",type(S[i]))
type de S[ 0 ] :  <class 'str'>
type de S[ 1 ] :  <class 'str'>
type de S[ 2 ] :  <class 'int'>
type de S[ 3 ] :  <class 'bool'>
type de S[ 4 ] :  <class 'float'>
type de S[ 5 ] :  <class 'list'>
```

4.1.3 Opérations sur liste

Voici quelques opérations sur les listes :

Opération	**Remarque**
liste.append(x)	Ajoute l'élément x à la fin de la liste
liste.sort()	Trie les éléments de la liste dans l'ordre croissant

liste.index(x)	Renvoie l'index de l'élément x dans la liste
liste.reverse()	Inverse l'ordre des éléments de la liste (image miroir)
liste.remove(x)	Supprime l'élément x de la liste
liste.pop()	Donne le haut de la liste puis l'enlève de la liste
liste.extend(liste2)	liste devient liste+liste2
"séparateur".join(liste)	Concatène les éléments de la liste avec *séparateur* entre les éléments
liste.count(x)	Donne le nombre d'occurrences de x dans la liste
len(liste)	Renvoie le nombre d'éléments dans la liste

4.2 Les tableaux

4.2.1 Définition et caractéristiques

Un tableau est une collection de données tel que :

- il faut importer une librairie qui permet de manipuler les tableaux ;
- un tableau est de type *array* ;
- les données d'un tableau sont toutes du même type (données homogènes, type uniforme) ;
- les données d'un tableau sont indexées (on peut repérer un ième élément dans le tableau) ;
- le premier élément, c'est-à-dire la première donnée d'un tableau se trouve à l'emplacement dont l'index est 0 ;
- les éléments sont accessibles et modifiables (lecture, écriture) ;
- la taille d'un tableau est connue dès sa création ;
- on peut manipuler des tableaux de dimensions différentes (par exemple les vecteurs lignes ou colonnes, les matrices 2D, les matrices 3D) ;
- on peut linéariser (transformer en un vecteur) un tableau à plusieurs dimensions ;
- etc.

Remarques

- Il convient d'utiliser les tableaux en algèbre linéaire ;
- on utilise la librairie *numpy* riche en opérations sur tableaux.

4.2.2 Création et initialisation de tableaux

Pour la suite, on utilisera la librairie *numpy*. Les tableaux sont désignés par la méthode *array()* du module *numpy*. Ainsi pour la référence à un tableau on utilise la syntaxe suivante :

```
Import numpy as np
...
T=np.array(vecteur(s), dtype, copy, order, subok, ndmin)
```

Où :

- *vecteur(s)* : ligne(s) qui compose(nt) le tableau ;
- dtype, copy, order, subok et ndmin sont facultatifs ;
- *dtype* : le type des données du tableau ;
- *ndmin* : le nombre minimum de dimensions ;
- tableau 1D : une seule ligne ou un seul vecteur ;
- tableau plusieurs dimensions : composé de plus d'une ligne ou plusieurs vecteurs.

Voici, sous Shell, quelques créations/initialisations de tableaux :

```
import numpy as np

# Création et initialisation d'un tableau avec la méthode array de numpy
T=np.array([1,2,3,4],dtype=float,ndmin=1)
T
array([1., 2., 3., 4.])
print(T)
[1. 2. 3. 4.]

# Type du tableau et type des éléments du tableau
type(T)
<class 'numpy.ndarray'>
T.dtype
dtype('float64')

# Création et initialisation d'un tableau avec la méthode zeros de numpy
M=np.zeros((2,2),dtype=float)
M
```

```
array([[0., 0.],
       [0., 0.]])
print(M)
[[0. 0.]
 [0. 0.]]

# Création et initialisation d'un tableau avec la méthode ones de numpy
A=np.ones((2,2),dtype=float)
A
array([[1., 1.],
       [1., 1.]])

# Création et initialisation d'un tableau avec la méthode eye de numpy
I=np.eye(3,dtype=float)
print(I)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

# Création et initialisation d'un tableau avec la méthode arange de
numpy
C=np.arange(-1.5,10,2.5,dtype=float)
print(C)
[-1.5  1.   3.5  6.   8.5]

# Création et initialisation d'un tableau avec la méthode linspace de
numpy
D=np.linspace(-5,10,15,dtype=int)
print(D)
[-5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8 10]
```

Remarque

Cette liste d'exemples n'est pas exhaustive. On verra, d'autres utilisations plus spécifiques et approfondies en calcul matriciel, en *Machine Learning* ou encore en *Deep Learning*.

4.2.3 Opérations sur les tableaux

Voici quelques opérations sur les tableaux :

Opération	**Remarque**
size	Le nombre d'éléments
shape	Taille du tableau
reshape	Modifie le format d'un tableau
min	Le min de toutes les données du tableau
max	Le max de toutes les données du tableau
T[a:b,c:d]	Le *slicing* ou tranchage d'un tableau de données (ici un tableau à 2 dimensions. cf. exemples suivants)

Remarque

Même remarque que précédemment : cette liste d'opérations n'est pas exhaustive. On en verra d'autres en calcul matriciel et dans les chapitres suivants.

```
import numpy as np

# Création et initialisation du tableau T
T=np.array([[1,2,3],[4,5,6],[-1,-2,-3],[-4,-5,-6]],dtype=float,ndmin=2)
print(T)
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [-1. -2. -3.]
 [-4. -5. -6.]]

# Opérations sur le tableau T
T.shape
(4, 3)
T.size
12
np.min(T)
-6.0
np.max(T)
6.0
Tl=T.reshape(T.size)
```

```
print(Tl)
[ 1.  2.  3.  4.  5.  6. -1. -2. -3. -4. -5. -6.]
Tm=Tl.reshape(2,T.size//2)
print(Tm)
[[ 1.  2.  3.  4.  5.  6.]
 [-1. -2. -3. -4. -5. -6.]]
T[0:]
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [-1., -2., -3.],
       [-4., -5., -6.]])
T[2:]
array([[-1., -2., -3.],
       [-4., -5., -6.]])
t=T[1:3,0:2]
t
array([[ 4.,  5.],
       [-1., -2.]])
ts=t.reshape(t.size)
ts
array([ 4.,  5., -1., -2.])
ts=sorted(ts,reverse=False)
ts
[-2.0, -1.0, 4.0, 5.0]
T[:,-1]
array([ 3.,  6., -3., -6.])
T[-1,:]
array([-4., -5., -6.])
```

4.3 Les tuples

4.3.1 Définition et caractéristiques

Un tuple est une collection de n données, dit aussi un n-uplet, tel que :

- n est un entier positif, c'est le nombre d'éléments du tuple ;
- un n-uplet est de type *tuple* ;

- les données d'un tuple peuvent être de types différents (données hétérogènes) ;
- les données d'un tuple sont indexées (on peut repérer un ième élément dans le tuple) ;
- le premier élément d'un tuple à l'index 0 ;
- les éléments sont accessibles (lecture) mais non modifiables (données immuables) ;
- la taille d'un tuple est connue dès sa création et elle n'est pas modifiable (pas d'ajout de nouveaux éléments et pas de suppression d'éléments);
- etc.

Remarque

On utilise un tuple, lorsque les données qu'il contient doivent être uniquement lisibles. Exemple : des constantes universelles, des dimensions normées, nombre de couches dans un réseau de neurones en *Deep Learning*, des clés de chiffrement, etc.

4.3.2 Création et initialisation d'un tuple

On peut créer en dur, c'est-à-dire dans le code, un tuple avec la syntaxe suivante :

```
tp = (éléments)
```

Le séparateur dans *éléments* est la virgule **,** .

4.3.3 Opérations sur les tuples

Voici quelques opérations sur les tuples :

Opération	**Remarque**
len()	Le nombre d'éléments du tuple
count()	Le nombre d'occurrence d'un élément dans le tuple
min()	Le min de toutes les données du tuple. Attention : toutes les données du tuple doivent être du même type régi par une relation d'ordre
max()	Le max de toutes les données du tuple. Attention : toutes les données du tuple doivent être du même type régi par une relation d'ordre
tp[a:b]	Le *slicing* ou tranchage d'un tuple de données
sorted()	Nouvelle collection triée dans l'ordre croissant. Attention : toutes les données du tuple doivent être du même type régi par une relation d'ordre
tuple()	La conversion d'une collection de données en tuple

Voici, sous Shell, quelques exemples

```
# Création et initialisation d'un tuple
tp1=(2,'math',['bases','dév','appro'])

# Type
type(tp1)
<class 'tuple'>
tp1[0]
2

# Opérations
tp1[len(tp1)-1]
['bases', 'dév', 'appro']
l=['bases', 'dév', 'appro']
type(l)
<class 'list'>
tpl=tuple(l)
tpl
('bases', 'dév', 'appro')
type(tpl)
<class 'tuple'>
tp2=(2,'math',tpl)
type(tp2)
<class 'tuple'>
tp2
(2, 'math', ('bases', 'dév', 'appro'))
tpnum=(2,3,2,5,-1)
min(tpnum)
-1
max(tpnum)
5
tpnum.count(2)
2
tpnum.index(-1)
4
```

```
min(tpl)
'appro'
max(tpl)
'dév'
tps=tuple(sorted(tpnum))
tps
(-1, 2, 2, 3, 5)
tps[1:3]
(2, 2)
tps[-1:]
(5,)
tps[:-1]
(-1, 2, 2, 3)
```

4.4 Les ensembles

4.4.1 Définition et caractéristiques

Un ensemble (*set*) est une collection de n données tel que :

- n est un entier positif, c'est le nombre d'éléments de l'ensemble dit aussi le cardinal ;
- un ensemble est de type *set* ;
- les données d'un ensemble peuvent être de types différents (données hétérogènes) ;
- les données d'un ensemble constituent une collection non ordonnée ;
- les éléments d'un ensemble peuvent être parcourus ou lus dans une structure itérative ;
- un ensemble de données, ne contient aucun doublon ;
- etc.

Remarque

On utilise un ensemble, lorsqu'on souhaite :

- éliminer des doublons d'une liste donnée ;
- et/ou faire des tests d'appartenance ;
- et/ou faire des opérations ensemblistes comme l'intersection ou l'union par exemples.

4.4.2 Création et initialisation d'un ensemble

On peut créer en dur, c'est-à-dire dans le code, un ensemble avec la syntaxe suivante :

```
E = {éléments}
…
F = set(éléments)
```

Le séparateur dans *éléments* est la virgule **,** .

Voici, sous Shell, quelques créations/initialisations d'ensembles :

```
l=[round(3.14159,2),round(2.71828,2),round(1.61803,2),round(0.57721,
2),round(1.41421,2)]
l
[3.14, 2.72, 1.62, 0.58, 1.41]
type(l)
<class 'list'>
CM=set(l)
CM
{0.58, 1.62, 2.72, 3.14, 1.41}
type(CM)
<class 'set'>
A={i%3 for i in range(1,13)}
A
{0, 1, 2}
B={(i**2)%6 for i in A}
B
{0, 1, 4}
Evide=set()
set()
type(Evide)
<class 'set'>
```

4.4.3 Opérations sur les ensembles

Voici, ci-après quelques opérations sur les ensembles.

Opération	Remarque
A.intersection(B)	Intersection de A et B
A.union(B)	Union de A et B
A.difference(B)	Ensemble des éléments appartenant à A et n'appartenant pas à B
A.symmetric_diference(B)	Ensemble des éléments appartenant à A mais pas à B union ensemble des éléments appartenant à B mais pas à A
A.update(B)	A est mise à jour et contient l'union de A et B
A.intersection_update(B)	A est mise à jour et contient l'intersection de A et B
A.issubset(B)	Vrai si A est un sous-ensemble de B
A.isdisjoint(B)	Vrai si l'intersection de A et B est vide

Voici un script avec quelques opérations en statistiques descriptives et ensemblistes :

```
import numpy as np
import statistics as stats

df=open("Control_Prod.csv")
T=np.loadtxt(df,delimiter=';')

c1=T[:,1]
c2=T[:,2]
E1=set(c1)
E2=set(c2)

n1=len(E1)
Moy1=round(stats.mean(E1),2)
Med1=round(stats.median(E1),2)
Var1=round(stats.variance(E1),2)
Stdev1=round(stats.stdev(E1),2)
n2=len(E2)
Moy2=round(stats.mean(E2),2)
Med2=round(stats.median(E2),2)
```

```
Var2=round(stats.variance(E2),2)
Stdev2=round(stats.stdev(E2),2)

print("Série 1 :\n", "Cardinal =",n1,", Moyenne =",Moy1," , Médiane
=",Med1,", Ecart-Type =",Stdev1)

print("Série 2 :\n", "Cardinal =",n2,", Moyenne =",Moy2," , Médiane
=",Med2,", Ecart-Type =",Stdev2)

C=E1.intersection(E2)
print(f"Intersection : {C}")

U=E1.union(E2)
print(f"Union : {U}")

D1=E1.difference(E2)
print(f"E1-E2 : {D1}")
D2=E2.difference(E1)
print(f"E2-E1 : {D2}")

print("D1 est sous ensemble de E1 : ",D1.issubset(E1))

df.close()
```

Résultat obtenu

```
Série 1 :
Cardinal = 13 , Moyenne = 13.15  , Médiane = 14.0 , Ecart-Type = 7.39

Série 2 :
Cardinal = 13 , Moyenne = 8.46  , Médiane = 7.0 , Ecart-Type = 7.1

Intersection : {0.0, 3.0, 17.0, 20.0, 21.0}

Union : {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0,
13.0, 14.0, 15.0, 16.0, 17.0, 20.0, 21.0, 26.0}
```

```
E1-E2 : {6.0, 8.0, 12.0, 13.0, 14.0, 15.0, 16.0, 26.0}
E2-E1 : {1.0, 2.0, 4.0, 5.0, 7.0, 9.0, 10.0, 11.0}

D1 est sous ensemble de E1 :  True
```

4.5 Les dictionnaires

4.5.1 Définition et caractéristiques

Un dictionnaire est une collection de données composées où chaque élément de cette composition est une association de (clé, valeur) tel que :

- les valeurs d'un dictionnaire sont modifiables ;
- chaque clé ne peut avoir qu'une valeur ;
- une valeur peut être associée à au moins une clé, c'est-à-dire l'association (clé, valeur) est surjective ;
- cette collection n'est pas ordonnée ;
- un dictionnaire est de type *dict* ;
- la valeur d'une clé donnée est modifiable ;
- on peut ajouter ou supprimer un élément (clé, valeur) ;
- etc.

Remarque

On utilise un dictionnaire, lorsqu'on a une collection non ordonnée d'objets (clé, valeur) et la recherche et/ou l'accès à une valeur se fait par la clé comme référence.

4.5.2 Création et initialisation d'un ensemble

On peut créer des dictionnaires avec plusieurs syntaxes. Voici quelques exemples :

```
Dvide = dict()
D1={'a':1, 'c':3, 'b':2, 'alpha':99}
D2=D1
D2['layout']=100
```

Voici, sous Shell, d'autres exemples de créations/initialisations de dictionnaires :

```
D={'a':1,'c':3,'alpha':99}
type(D)
<class 'dict'>
len(D)
```

```
3
D.keys()
dict_keys(['a', 'c', 'alpha'])
D.values()
dict_values([1, 3, 99])
D.items()
dict_items([('a', 1), ('c', 3), ('alpha', 99)])
D.get('alpha')
99
D.get('ctrl',100)
100
D
{'a': 1, 'c': 3, 'alpha': 99}
D['ctrl']=1
D
{'a': 1, 'c': 3, 'alpha': 99, 'ctrl': 1}
min(D)
'a'
l=list(D)
type(l)
<class 'list'>
l
['a', 'c', 'alpha', 'ctrl']
d=list(D.values())
d
[1, 3, 99, 1]
```

4.5.3 Opérations sur les dictionnaires

Voici quelques opérations sur les dictionnaires :

Opération	**Remarque**
len(dico)	Le nombre d'éléments (clé, valeur) dans le dictionnaire dico
dict()	Le constructeur de dictionnaire. Cf. différents exemples ci-après

min(dico)	La clé correspondant à la 1ère valeur min du dictionnaire dico
max(dico)	La clé correspondant à la 1ère valeur max du dictionnaire dico
del dico[xclé]	La suppression de l'élément (xclé, xvaleur) du dictionnaire dico
sorted(dico)	Tri des clés par ordre croissant des clés du dictionnaire dico
sorted(dico, reverse=True)	Tri des clés par ordre décroissant des clés du dictionnaire dico
sorted(dico, key=dico.get)	Tri des clés par ordre croissant des valeurs du dictionnaire dico
sorted(dico, key=dico.get, reverse=True)	Tri des clés par ordre décroissant des valeurs du dictionnaire dico
dico.keys()	Toutes les clés du dictionnaire dico
dico.values()	Toutes les valeurs, non forcément distinctes, du dictionnaire dico
x=dico.pop(xkey)	*pop* récupère la valeur xvalue correspondant xkey, l'affecte à x puis supprime l'élément (xkey, xvalue) du dictionnaire dico
x=dico.pop(xkey, val)	Si xkey est une clé qui est dans le dictionnaire alors *pop* récupère la valeur xvalue correspondant xkey, l'affecte à x puis supprime l'élément (xkey, xvalue) du dictionnaire dico, sinon la valeur par défaut val est affectée à x

Voici un script qui permet de créer un dictionnaire à partir de deux séries de données puis de créer un *Data Frame* pandas à partir de ce dictionnaire :

```
# Importation de pandas utile à la création d'un data frame
import pandas as pd

dico={'Année':[2018, 2019, 2020, 2021, 2022, 2023],
    'Iprix':[1200, 1173, 1165, 1178, 1248, 1045]}

# Création d'un data frame
df=pd.DataFrame({cle:pd.Series(val) for cle, val in dico.items()})

print(df)
```

Résultat obtenu

	Année	Iprix
0	2018	1200
1	2019	1173
2	2020	1165
3	2021	1178
4	2022	1248
5	2023	1045

Chapitre 2

Calcul scientifique

1 Définition et généralités

1.1 Définition

Une librairie dite aussi bibliothèque, module ou encore paquet ou package, est une collection de fonctions, de méthodes, de classes d'objets, de constantes mises à disposition des développeurs pour ainsi simplifier et optimiser leur codage. Les éléments d'une librairie sont mobilisables et réutilisables dans les scripts développeurs.

Une librairie est constituée afin de répondre à un domaine ou à une thématique donnée comme : le calcul scientifique, l'algèbre linéaire, l'analyse statistique, la représentation et visualisation des données, les modèles prédictifs en *Machine Learning*, l'entraînement des modèles en *Deep Learning*, la manipulation des fichiers, etc.

Une librairie peut contenir aussi des sous-librairies. La classification des fonctions, méthodes, classes, et constantes est faite par spécialisation dans la même thématique.

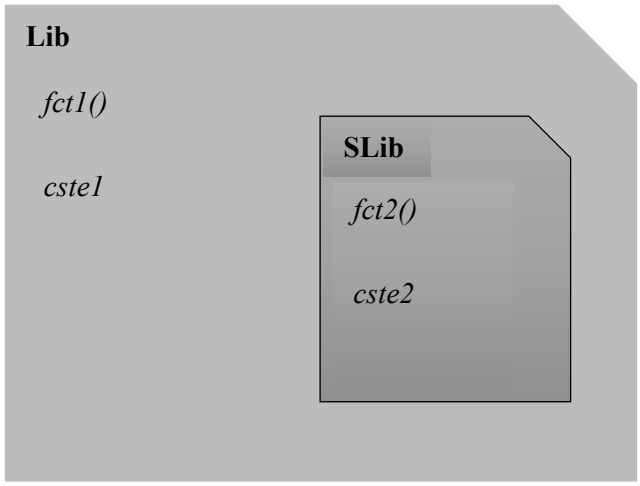


***Figure 2** : Librairies et sous-librairies*

Voici quelques librairies et sous-librairies qu'on utilise dans cet ouvrage :

- math ;
- cmath ;
- fractions ;

- numpy ;
- *linalg ;*
- *matplotlib ;*
- scipy ;
- os ;
- sympy ;
- *pandas*.

D'autres librairies comme *pytorch* et/ou *scikit-learn* (*sklearn*) permettent d'aborder plus tard, dans un autre ouvrage, le *Deep Learning* et les réseaux de neurones.

1.2 Importation

Avant l'utilisation d'une librairie dans un script, il faut l'importer en amont. L'importation suppose que la librairie concernée par cet import est déjà téléchargée dans l'éditeur utilisé pour le développement.

Voici également quelques syntaxes de l'instruction d'importation d'une librairie nommée *Lib* :

```
import Lib
from Lib import fct1
from Lib import *
import Lib.SLib
...
```

Exemples

```
# Librairies math et numpy
from math import *
import numpy
# Visualisation et représentations graphiques
import matplotlib
import matplotlib.pyplot
```

1.3 Utilisation

L'utilisation d'une fonction d'une librairie se fait dans le script développeur par appel de fonction. L'appel doit faire référence à la fonction appelée, à la librairie où se trouve cette fonction et au passage des arguments de la fonction s'ils sont obligatoires.

Voici un exemple pour attribuer un titre à un graphique en utilisant la fonction *title* de la librairie *matplotlib.pyplot* :

```
import matplotlib.pyplot
...
matplotlib.pyplot.title("Traitement data")
```

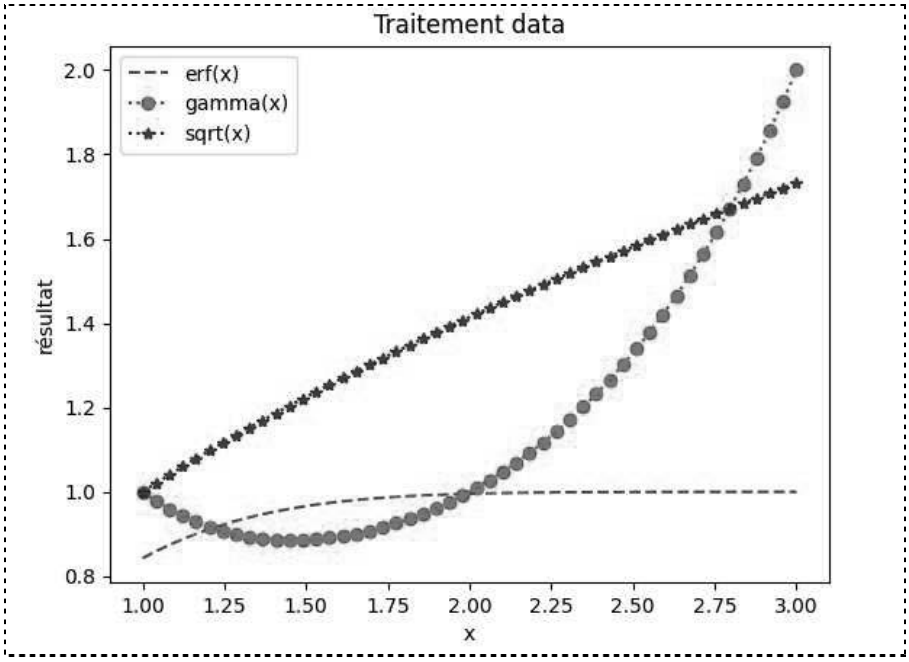
On peut aussi utiliser des allias afin de simplifier le nom de référence à la librairie dans le script développeur :

```
import matplotlib.pyplot as plt
...
plt.title("Traitement data")
```

1.4 Exemple

```
# Importation des librairies
import numpy as np
import matplotlib.pyplot as plt
import scipy.special as sci

# Visualisation des graphiques de fonctions
x = np.linspace(1, 3)

# Figure
fig, ax = plt.subplots()

ax.set_xlabel("x")
ax.set_ylabel("résultat")
plt.title("Traitement data")

plt.plot(x, sci.erf(x),"g--", label="erf(x)")
plt.plot(x, sci.gamma(x),"r:o", label="gamma(x)")
plt.plot(x, np.sqrt(x),"b:*", label="sqrt(x)")

plt.legend()
plt.show()
```

Résultat



2 Rappel sur les ensembles numériques

Dans la suite, nous considérons les nombres, les valeurs ou encore les données (data) numériques comme étant les éléments d'un ensemble. Ces données permettent la quantification, l'évaluation ou la mesure d'une entité ou d'une grandeur. Ils sont classés selon des critères ou des caractéristiques dans les ensembles : $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{Q}'$ et $\mathbb{C}$.

Ensemble	Désignation	Exemples	Lien ensembliste	Écriture / Utilisation
$\mathbb{N}$	Ensemble des nombres naturels	0, 1, 100, 1230001, …		Énumération, dénombrement $n \in \mathbb{N}$
$\mathbb{Z}$	Ensemble des nombres entiers	$-2, -1, 0,$ 1, 2, ...	$\mathbb{N} \subset \mathbb{Z}$	Nombres naturels et leurs opposés $k \in \mathbb{Z}$

Ensemble	Désignation	Exemples	Lien ensembliste	Écriture / Utilisation
$\mathbb{Q}$	Ensemble des nombres rationnels. C'est l'ensemble des fractions	5/7, -2, $-10/3$, …	$\mathbb{Z} \subset \mathbb{Q}$	$\forall\, q \in \mathbb{Q}$: $\exists\, a \in \mathbb{Z}\ et \exists\, b \in \mathbb{Z}^*$ $tel\ que\ q = \frac{a}{b}$
$\mathbb{Q}'$	Ensemble des nombres irrationnels	π, $\sqrt{17}$	$\mathbb{Q} \cap \mathbb{Q}' = \emptyset$	$\mathbb{Q}\ et\ \mathbb{Q}'$ sont mutuellement exclusifs.
$\mathbb{R}$	Ensemble des nombres réels	1, -2, $-10/3$, π	$\mathbb{R} = \mathbb{Q} \cup \mathbb{Q}'$	$\mathbb{Q}\ et\ \mathbb{Q}'$ représentent une partition de $\mathbb{R}$
$\mathbb{C}$	Ensemble des nombres complexes	3, π, i, $5-i\sqrt{21}$	$\mathbb{R} \subset \mathbb{C}$	$\forall\, z \in \mathbb{C}$: $\exists\, a \in \mathbb{R}\ et \exists\, b \in \mathbb{R}$ $tel\ que\ z = a + ib$ où : a est la partie réelle, b est la partie imaginaire et i nombre complexe tel que $i^2 = -1$

Voici, ci-après la présentation non exhaustive des librairies citées ci-dessus et de leurs principales fonctions.

3 Librairie *math*

Cette librairie fournit quelques constantes mathématiques connues comme la base du logarithme népérien et fournit des fonctions mathématiques prédéfinies, qu'on peut regrouper par thématique telles que l'arithmétique et la trigonométrie.

Les fonctions de la librairie *math* ne s'appliquent pas aux nombres complexes. Il y a une autre librairie *cmath* des fonctions mathématiques sur les nombres complexes.

Fonction, Constante	Description	Groupe thématique
ceil(x)	Le plus petit entier k tel que k ≥ x où x est un réel	Arithmétique
fabs(x)	La valeur absolue de x où x est un réel	Arithmétique
factorial(x)	La factorielle de x où x est un entier positif	Arithmétique
floor(x)	Le plus grand entier k tel que k ≤ x où x est un réel	Arithmétique
fmod(x, y)	Le reste de la division x / y où x et y sont des entiers relatifs	Arithmétique
gcd(x, y)	Le plus grand commun diviseur de x et y où x et y sont des entiers relatifs	Arithmétique
ldexp(x, i)	Renvoie x * (2**i) où x est un réel et i est un entier relatif	Arithmétique
modf(x)	Renvoie le binôme (partie fractionnaire, entière) de x	Arithmétique
trunc(x)	La partie entière de x où x est un réel	Arithmétique
e	La base du logarithme népérien	Constante
pi	La constante π	Constante
tau	la constante 2π	Constante
log(x[, b])	Le logarithme de x base b où x est un réel et b un entier strictement positif	Logarithmique
exp(x)	Renvoie e**x où x est un réel	Puissance
pow(x, y)	La puissance x**y où x et y sont des réels. Attention : x et y ne peuvent pas prendre en même temps la valeur 0	Puissance
sqrt(x)	La racine carrée de x où x est un réel positif	Puissance
erf(x)	La fonction d'erreur de x	Signal
gamma(x)	La fonction gamma de x	Signal
lgamma(x)	Le logarithme népérien de la fonction gamma de x	Signal
cos(x)	Le cosinus de x où x est un réel	Trigonométrique

Fonction, Constante	Description	Groupe thématique
degrees(x)	Convertit un angle exprimé en radians en un angle exprimé en degrés	Trigonométrique
radians(x)	Convertit un angle exprimé en degrés en un angle exprimé en radians	Trigonométrique
sin(x)	Le sinus de x où x est un réel	Trigonométrique
tan(x)	La tangente de x où x est un réel	Trigonométrique

4 Librairie c*math*

Dans la librairie cmath, on trouve des fonctions mathématiques appliquées sur les nombres complexes $z \in \mathbb{C}$.

Fonction	Syntaxe	Description
sqrt	cmath.sqrt(z)	Racine carrée
exp	cmath.exp(z)	Exponentiel
log	cmath.log(z[, base])	Logarithme
log10	cmath.log10(z)	Logarithme base 10
sin	cmath.sin(z)	Sinus
cos	cmath.cos(z)	Cosinus
tan	cmath.tan(z)	Tangente
sinh	cmath.sinh(z)	Sinus hyperbolique
cosh	cmath.cosh(z)	Cosinus hyperbolique
tanh	cmath.tanh(z)	Tangente hyperbolique
phase	cmath.phase(z)	Argument de z
polar	cmath.polar(z)	Résultat bi-uplet : (module, argument) de z
rect	cmath.rect(mod, arg)	Résultat le nombre complexe ayant mod pour module et arg pour argument

Sous Python standard, on dispose également de quelques fonctions et constructeurs tels que *complex()* et *conjugate()*.

```
z=complex(2, 3)
zc=z.conjugate()
# Affichage
z, zc
```

Résultat

```
((2+3j), (2-3j))
```

5 Librairie *fractions*

La librairie *fractions* permet l'écriture et la manipulation des nombres rationnels écrits sous forme : $\frac{a}{b}$ où $a \in \mathbb{Z}$ et $b \in \mathbb{Z}^*$.

Fonction	Description	Exemple
Fraction(a, b)	Le constructeur de fraction à partir de deux nombres relatifs a et b tel que b est non nul	*from fractions import ** *q=Fraction(7, 13)* *print(q)* *7/13* *type(q)* *<class 'fractions.Fraction'>*
q.numerator	Le numérateur du nombre rationnel q	*a=q.numerator* *print(a)* *7* *type(a)* *<class 'int'>*
q.denominator	Le dénominateur du nombre rationnel q	*b=q.denominator* *print(b)* *13* *type(b)* *<class 'int'>*

Voici, ci-après un exemple de script utilisant des fonctions et méthodes des bibliothèques *math* et *fractions*.

Cet exemple consiste à écrire une fraction sous forme de somme finie de fractions. C'est l'exemple des fractions égyptiennes.

```
from fractions import Fraction
from math import ceil

a = None
while a is None:
    try:
        a = int(input("Saisir un entier a : "))
    except ValueError:
        print("Erreur! recommencer")
b = None
while b is None or int(b)==0:
    try:
        b = int(input("Saisir un entier b non nul: "))
    except ValueError:
        print("Erreur! recommencer")
a=int(a)
b=int(b)

q0=Fraction(a,b)
q=q0

l = int(q)
q = q-l
LFractions = [l]
while(q.numerator > 1):
   l = Fraction(1, int(ceil(1/q)))
   LFractions.append(l)
   q = q-l
LFractions.append(q)

print(q0," = ",LFractions[0])
for i in range(1,len(LFractions)):
   print(" + ", LFractions[i])
```

Test du script

```
Saisir un entier a : 357
Saisir un entier b non nul: 232
357/232 = 1
 + 1/2
 + 1/26
 + 1/3016
```

6 Calcul scientifique ave la librairie *scipy*

La librairie *scipy* est l'une des bibliothèques, avec *numpy*, les plus étoffées en constantes, fonctions/méthodes et algorithmes pour réaliser des traitements scientifiques, mathématiques, statistiques et en ingénierie.

6.1 Les sous-librairies *scipy*

La librairie *scipy* contient plusieurs sous-librairies ou paquets (packages) thématiques :

Paquets	Description	Thèmes
cluster	Algorithmes de regroupement en analyse de données et apprentissage non supervisé	Analyse de données et *Machine Learning*
constants	Des constantes connues en mathématiques et en physique	Divers
fft	Contient des fonctions/méthodes relatives à la transformée de Fourier discrète	Physique / traitement de signal
fftpack	Contient des fonctions/méthodes relatives à la transformée de Fourier rapide (legacy)	Physique / traitement de signal
integrate	Contient un solveur d'équations différentielles et calcul intégral	Mathématiques
linalg	Contient des fonctions/méthodes en algèbre linéaire. On retrouve également un paquet *linalg* sous la librairie *numpy*	Mathématiques

Paquets	Description	Thèmes
ndimage	Contient des fonctions/méthodes pour le traitement d'image et analyse multidimensionnelles	Mathématiques, ingénierie (informatique, médecine, …)
odr	Contient des fonctions/méthodes pour la régression orthogonale	Statistiques, métrologie, biologie, …
optimize	Contient des fonctions/méthodes d'optimisation mathématiques de fonctions objectives	Mathématiques / recherche opérationnelle, ingénierie (production, économie et finance, agronomie, …)
signal	Contient des fonctions/méthodes de traitement de signal	Physique / traitement de signal
spatial	Contient des fonctions/méthodes pour le traitement de données spatiales	Systèmes d'information géographique, …
special	Contient des fonctions spéciales comme les fonctions de Bessel	Divers
stats	Contient des fonctions et distributions en statistiques et probabilités	Statistiques et probabilités

6.2 Les fonctions

Voici ci-dessous une liste, non exhaustive, de fonctions de différents paquets de *scipy* :

Fonctions (*scipy.*)	Description
cluster.hierarchy	Regroupement (*clustering*) hiérarchique
cluster.vq	Quantification vectorielle et k-means
integrate.dblquad	Intégration double sur un rectangle défini
integrate.odeint	Intégration d'équations différentielles
integrate.quad	Intégration de fonctions sur un intervalle donné
integrate.solve_ivp	Solveur d'équations différentielles
integrate.tplquad	Intégration triple sur un parallélépipède défini
linalg.det	Calcul du déterminant d'une matrice
linalg.eig	Calcul des valeurs et vecteurs propres

Fonctions (*scipy*.)	**Description**
linalg.inv	Calcul de l'inverse d'une matrice
linalg.svd	Décomposition en valeurs singulières
optimize.curve_fit	Ajustement de courbes non linéaires
optimize.linprog	Optimisation linéaire
optimize.minimize	Minimisation d'une fonction scalaire
optimize.root	Recherche des racines d'équations
stats.bernoulli	Distribution de Bernoulli
stats.binom	Distribution binomiale
stats.chi2	Distribution du chi carré
stats.expon	Distribution exponentielle
stats.linregress	Régression linéaire
stats.norm	Distribution normale (gaussienne)
stats.norm	Distribution normale (gaussienne)
stats.pearsonr	Calcul du coefficient de corrélation de Pearson
stats.poisson	Distribution de Poisson
stats.t	Distribution de Student
stats.uniform	Distribution uniforme continue
stats.f_oneway	Analyse de la variance à un facteur (ANOVA)
stats.kruskal	Test de Kruskal-Wallis
stats.pearsonr	Coefficient de corrélation de Pearson
stats.spearmanr	Coefficient de corrélation de Spearman
stats.linregress	Régression linéaire simple
stats.describe	Calcul des statistiques descriptives de base
stats.variation	Coefficient de variation

6.3 Résolution d'un programme linéaire par la méthode du simplex

Nous utiliserons le paquet *optimize* de *scipy* pour la résolution d'un programme linéaire.

Pour la résolution, on considère un programme linéaire écrit sous la forme suivante :

$$\begin{aligned} &min_x\ C^T x \\ &s.c \\ &\quad Ax \leq b \\ &\quad x \in \mathbb{R}^n \end{aligned}$$

Voici un exemple concret :

$$\begin{aligned} &min_x \quad 5x_1 - 30x_2 \\ &s.c \\ &\quad x_1 + 2x_2 \leq \ 101 \\ &\quad x_1 - x_2 \leq \ \ 40 \\ &\quad -3x_1 - 2x_2 \leq \ \ \ 0 \\ &\quad x_1 \geq 0 \\ &\quad x_2 \geq 0 \end{aligned}$$

6.3.1 Programme linéaire : script de résolution

```
from scipy import optimize

c=[5,-30]
A=[[1,2],
   [1,-1],
   [-3,-2]]
b=[101,40,0]

x1_dom=(0,None)
x2_dom=(0,None)
LP=optimize.linprog(c,A_ub=A,b_ub=b,bounds=[x1_dom,x2_dom])

#Message concernant la résolution
print(LP.message)
#Les valeurs optimales de x
print(LP.x)
#La valeur de la fonction objective à l'optimum
print(LP.fun)
#Respectivement les valeurs résiduelles et valeurs marginales
print(LP.ineqlin)
```

6.3.2 Résultat

```
Optimization terminated successfully. (HiGHS Status 7: Optimal)
[ 0.  50.5]
-1515.0
 residual: [ 0.000e+00  9.050e+01  1.010e+02]
 marginals: [-1.500e+01 -0.000e+00 -0.000e+00]
```

6.4 Résolution d'un programme d'affectation linéaire

Le programme d'affectation linéaire est un problème d'optimisation combinatoire à variables booléennes ou binaires (0/1).

On dispose de n ressources et de n travaux à réaliser. La réalisation du travail indice j par la ressource indice i engendre un coût de réalisation c_{ij}. L'ojectif est de trouver la meilleure affectation des ressources aux travaux qui minimise le coût global de réalisation engendré.

Notation

n : nombre de travaux à réaliser ;

i : i^{ème} ressource disponible ;

j : j^{ème} travail à réaliser ;

c_{ij} : les coûts d'affectation des ressources à la réalisation des travaux ;

x_{ij} : la variable d'affectation, est égale à 1 si le travail j est affecté à la ressource i, 0 sinon.

Le programme d'affectation linéaire est formulé comme suit :

$$min \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{i,j}x_{i,j}$$

$$s.c \quad \sum_{i=1}^{n} x_{i,j} = 1 \quad i = 1, \dots, n$$

$$\sum_{j=1}^{n} x_{i,j} = 1 \quad j = 1, \dots, n$$

$$x_{i,j} \in \{0,1\} \quad i = 1, \dots, n \quad et\ j = 1, \dots, n$$

où la fonction objective à minimiser est linéaire ainsi que les contraintes des ressources et des travaux. Les contraintes d'intégrité indiquent que la variable x_{ij} est booléenne.

En termes algorithmiques, la méthode hongroise permet de résoudre ce programme efficacement. Nous utiliserons la méthode *linear_sum_assignment* de la sous-librairie *scipy.optimize* pour la résolution de ce programme d'affectation linéaire.

linear_sum_assignment prend en argument la matrice des coûts et retourne les indices des colonnes retenues, c'est-à-dire les ressources affectées et les indices des lignes correspondant à leur affectation, c'est-à-dire les travaux affectés.

Remarques

- Pour la matrice des coûts, on utilise la librairie *numpy* ;
- les indices des tableaux sous Python commencent à 0.

6.4.1 Méthode hongroise : script de résolution

```
import numpy as np
from scipy.optimize import linear_sum_assignment

#Matrice des coûts. C'est une matrice carrée (n,n)
C = np.array([[5,1,3,4], [2,3,1,7], [2,5,7,8],[6,5,4,2]])
n=np.shape(C)[0]

#Résultat de linear_sum_assignment
#Les indices des des ressources afféctées aux travaux : Lignes, Colonnes
Lignes, Colonnes = linear_sum_assignment(C)
for k in range(n):
    i=Lignes[k]
    j=Colonnes[k]
    print("La ressource ",i,
          " Réalise le travail ",j,
          " avec un coût ", C[i][j])

print(Colonnes)
print(Lignes)

#La valeur de la fonction objective
```

```
print("Coût global minimum de l'affectation = ",C[Lignes,
Colonnes].sum())
```

6.4.2 Résultat

```
La ressource  0  Réalise le travail  1  avec un coût  1
La ressource  1  Réalise le travail  2  avec un coût  1
La ressource  2  Réalise le travail  0  avec un coût  2
La ressource  3  Réalise le travail  3  avec un coût  2
[1 2 0 3]
[0 1 2 3]
Coût global minimum de l'affectation =  6
```

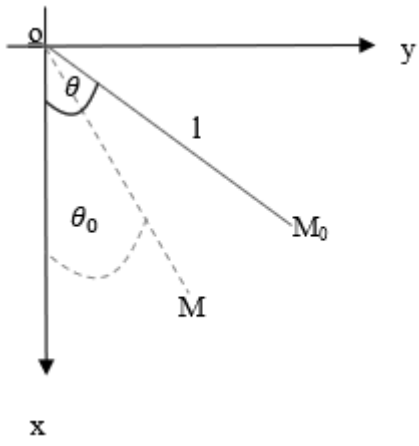
6.5 Résolution d'équations différentielles

Le mouvement de la pendule simple est un problème classique en physique et pour sa résolution on utilise la modélisation mathématique et plus spécialement les équations différentielles.

La description du problème est la suivante :

Une sphère ou un objet ponctuel représenté par le point M sur un plan orthogonal (xoy) vertical et de masse m. L'objet M est suspendu à un fil inextensible de longueur l. Cet objet est lâché à l'instant initial t_0 de la position initiale M_0 avec un angle initial θ_0 et sans vitesse initiale. L'oscillation de l'objet se fait avec l'hypothèse sans frottement.

On souhaite étudier/observer dans le temps t l'angle θ du mouvement de l'objet M dans le cas de petites oscillations.



6.5.1 Équation différentielle

$$\ddot{\theta} + \omega^2 \sin\theta = 0$$

où :

- $\omega = \sqrt{\frac{g}{l}}$;
- g est l'accélération de l'apesanteur (g = 9,81) ;
- $\theta(t)$ est l'angle d'oscillation, exprimé en radian, à l'instant t ;
- $u = \frac{d\theta}{dt}$;
- $\frac{du}{dt} = -\omega^2\theta$.

Pour réaliser cette observation dans le temps *t*, on utilise un intégrateur d'équations différentielles. La fonction utilisée est *odeint* de la sous-librairie *integrate* des fonctions et méthodes d'intégration dans le domaine scientifique. *integrate* fait partie en effet de la librairie *scipy*.

6.5.2 Implémentation

```
# Importation des librairies
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
import math

g=9.81
l=20
t=np.linspace(0,60,600)

# Eléments de l'équation du mouvement
def equamove(arguments,t):
    A, u = arguments
    edp=[u,-(g/l)*np.sin(A)]
    return(edp)

#-Angle initial converti du degré au radian
A0=math.radians(45)
```

```
# Dérivée de l'angle initial est égale à 0 radian
u0=math.radians(0)

# Appel à l'intégrateur d'équations différentielles
Theta=odeint(equamove,[A0,u0],t)

# Le tableau Theta est de taille (600, 2)
# 600 correspond au nombre d'itérations défini dans linspace
# et 2 correspond au nombre de colonnes
# chaque ligne contient un angle d'oscillation et sa dérivée

print("Le tableau Theta : \n",Theta)

plt.xlabel("instant t")
plt.grid(True)
plt.plot(Theta[:,0],label="$\\Theta(t)$")
plt.plot(Theta[:,1],label="$u$")

plt.legend()
print("Graphique")
plt.show()
```

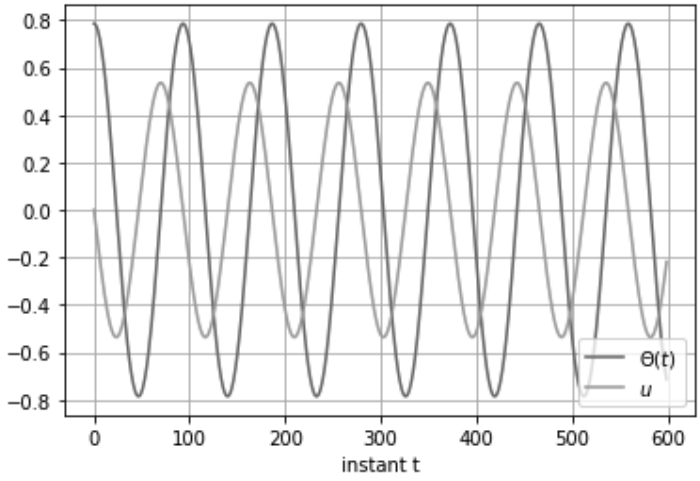
6.5.3 Test

Voici le résultat du test pour les données initiales suivantes :

- angle initial = 45° ;
- longueur l = 20 cm.

```
Le tableau Theta :
 [[ 0.78539816  0.        ]
 [ 0.7836587  -0.03472134]
 [ 0.77844635 -0.06932157]
 ...
 [-0.66403774 -0.28084803]
 [-0.69063531 -0.25004718]
 [-0.71409899 -0.21829747]]
```

Graphique



Chapitre 3

Visualisation graphique des données

Dans ce chapitre, on aborde les méthodes *Python* qui permettent la visualisation représentation graphiques de données brutes et/ou traitées. Plusieurs types de graphiques sont illustrés dans ce chapitre. Les représentations peuvent être statiques, animés ou interactives.

1 Visualisation des données

La librairie *matplotlib* dispose de fonctions et méthodes sous *pyplot* qui permettent de visualiser des courbes et des graphiques en deux (2D) ou trois (3D) dimensions :

- les courbes ;
- les nuages de points ;
- barres et histogrammes ;
- les diagrammes à moustaches ;
- circulaires ;
- en secteurs ;
- etc.

La visualisation en 2D nécessite deux séries ou deux tableaux de données numériques du même cardinal : x et y telle que y est fonction de x.

La visualisation en 3D nécessite trois séries ou trois tableaux de données numériques du même cardinal : x, y et z telle que z est fonction de x et y.

Des options sont aussi disponibles afin de personnaliser et de rendre la représentation graphique lisible et interprétable visuellement :

- couleur du rendu (courbe ou graphique) ;
- représentation sur une grille (un repère quadrillé) ;
- plusieurs courbes ou graphiques sur la même représentation ;
- titres et labels ;
- légende ;
- etc.

2 Les fonctions de *pyplot*

La librairie *pyplot* est une sous-librairie de *matplotlib* pour la visualisation des données et la représentation graphique.

2.1 Syntaxe

Dans la suite et pour la syntaxe d'importation des fonctions, on utilisera l'instruction suivante :

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

Les fonctions peuvent être classées par catégorie. Voici, ci-dessous quelques catégories ainsi qu'une liste non exhaustive de fonctions par catégorie.

2.2 Catégories de fonctions

- création ;
- type de graphique ;
- affichage ;
- étiquettes et labels ;
- etc.

3 Les graphiques 2D

Ici, on abordera d'abord les graphiques 2-dimensions (2D) où on représente graphiquement un traitement et/ou une prédiction y en fonction de x. On utilisera donc deux axes de représentation graphique (abscisses et ordonnées).

Certaines fonctions de *pyplot* utilisées ici seront appelées aussi dans les représentations 3-dimensions (3D).

3.1 Création

Les fonctions de création permettent tracer une représentation graphique / une figure.

Fonction	**Description**	**Syntaxe**
plt.plot()	Tracer la représentation graphique	*plt.plot(IX,IY,options)* où : IX : série de données numérique IY : l'image de IX par une fonction, un traitement ou une observation et *options* d'affichage
plt.figure()	Créer une figure	*fig=plt.figure()*

3.1.1 Exemple 1 de fonction définie par intégrale

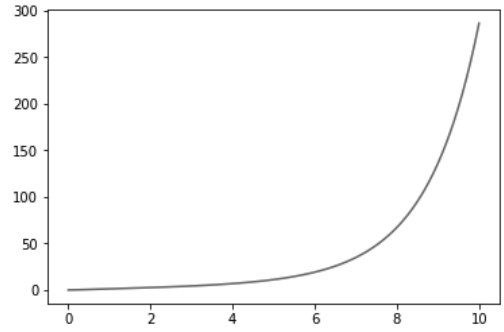
Tracer la courbe de fa fonction :

$$F(x) = \int_0^X \frac{e^t}{1+t^2} dt \quad pour\ x\ \in \mathbb{R}$$

```
# Importation des librairies
import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as integral

# Définition des fonctions
def f(t):
    return (np.exp(t)/(1+t**2))
def F(x):
    r=integral.quad(lambda x:f(x),0,x)
    return(r[0])

IX=np.linspace(0, 10,100)
LY=[F(x) for x in IX]
IY=np.array(LY)

plt.plot(IX,IY)
plt.show()
```

3.1.2 Visualisation de l'exemple 1

Ci-après on a la visualisation de la courbe et sans légende.



3.2 Les options de la fonction plot()

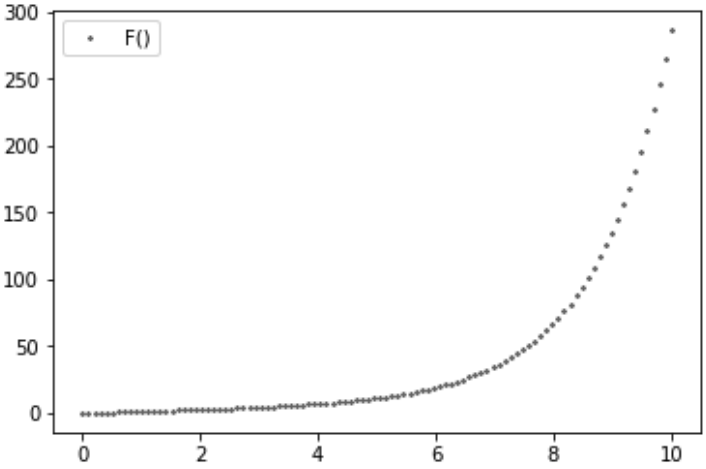
Les options permettent de personnaliser et d'améliorer l'affichage de la représentation graphique.

3.2.1 Exemple 2 de fonction définie par intégrale

Voici le même exemple précédent avec l'affichage des points seuls (style du marqueur, couleur et taille du point, label) et d'une légende.

```
# plot() avec l'option marqueur du point
plt.plot(IX,IY, 'm*',ms=2, label="F()")

# Insertion d'une légende
plt.legend()
# Visualisation
plt.show()
```

3.2.2 Visualisation de l'exemple 2



Style du marqueur

Marqueur	Signification
'-'	Ligne continue
'--'	Ligne en pointillés

Marqueur	Signification
'.'	Point
'o'	Point en forme de cercle
's'	Point en forme de carré
'+'	Point en forme du signe plus
'x'	Point en forme du signe x
'D'	Point en forme de diamant

Couleur du marqueur

Masque	Couleur
'b'	Bleu
'g'	Vert
'r'	Rouge
'c'	Cyan
'w'	Blanc
'm'	Magenta
'y'	Jaune
'k'	Noir

3.3 Autres fonctions

Autres fonctions permettent également la personnalisation de la représentation graphique :

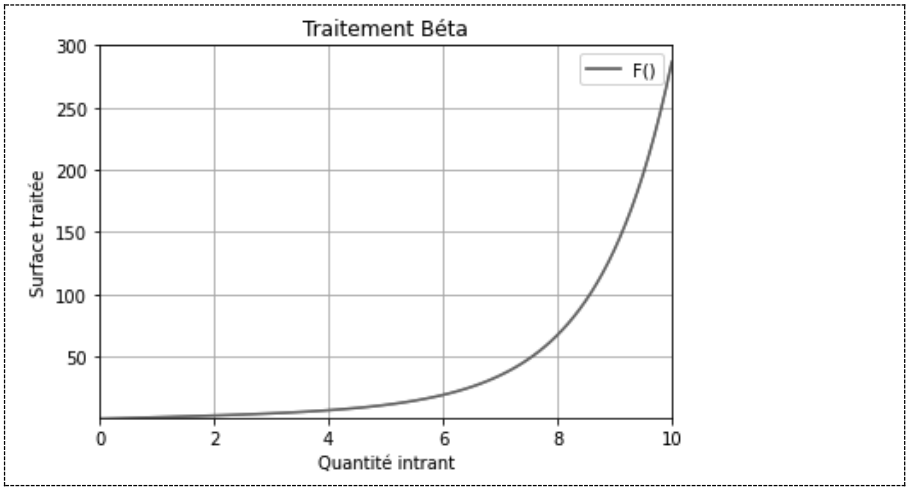
- limites des intervalles ;
- label des axes ;
- affichage de la grille et du titre ;
- etc.

3.3.1 Exemple précédent

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as integral
```

```
def f(t):
    return (np.exp(t)/(1+t**2))

def F(x):
    r=integral.quad(lambda x:f(x),0,x)
    return(r[0])

IX=np.linspace(0, 10,100)
LY=[F(x) for x in IX]
IY=np.array(LY)
plt.xlim(0,10)
plt.ylim(1,300)
plt.xlabel("Quantité intrant")
plt.ylabel("Surface traitée")
plt.plot(IX,IY, 'm-',ms=2, label="F()")
plt.grid(True)
plt.legend()
plt.title("Traitement Béta")
plt.show()
```

3.3.2 Résultat



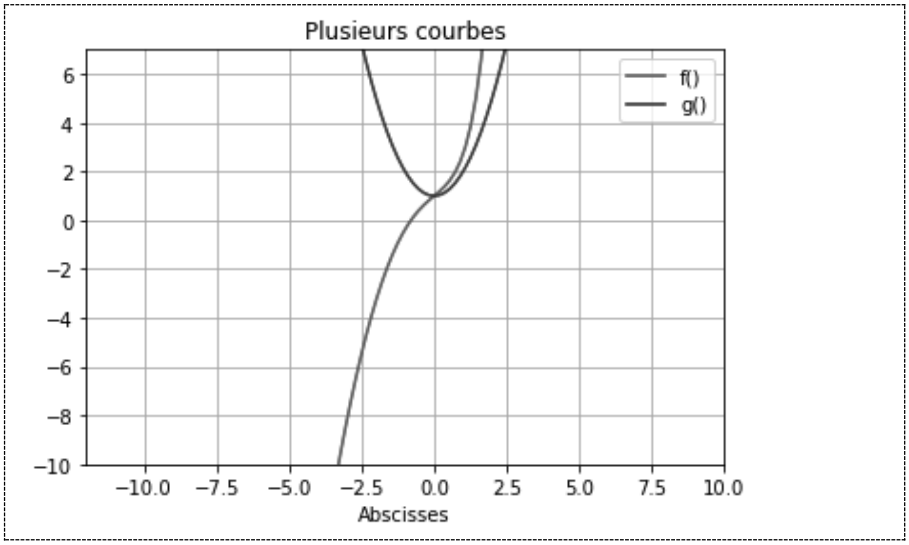
3.4 Plusieurs courbes

On peut également afficher plusieurs courbes sur la même figure en utilisant autant de fois la fonction plt.plot().

3.4.1 Exemple

```
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    return (t*np.exp(t)-t**2+1)

def g(t):
    return(1+t**2)

X=np.linspace(-120, 60,1000)
LY1=[f(x) for x in X]
IY1=np.array(LY1)
LY2=[g(x) for x in X]
IY2=np.array(LY2)
plt.xlabel("Abscisses")
plt.xlim(-12,10)
plt.ylim(-10,7)
plt.grid(True)

# plot() de la fonction f()
plt.plot(X,IY1, 'm-',label="f()")

# plot() de la fonction g()
plt.plot(X,IY2, 'b-',label="g()")

plt.legend()
plt.title("Plusieurs courbes")

plt.show()
```

3.4.2 Résultat

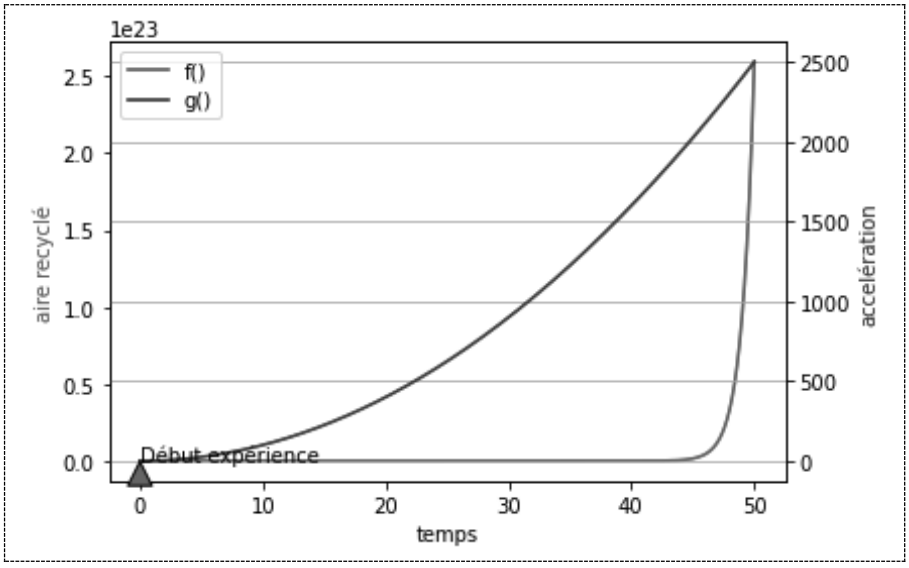


Il est possible d'ajouter aussi un axe supplémentaire et des annotations sur la figure.

```
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    return (t*np.exp(t)-t**2+1)

def g(t):
    return(1+t**2)

x=np.linspace(0, 50,250)
y1=f(x)
y2=g(x)

fig, ax=plt.subplots()

ax.plot(x,y1,'m-')
ax.set_xlabel("temps")
ax.set_ylabel("aire recyclé",color='m')
```

```
# Définition des axes
ax2=ax.twinx()
ax2.plot(x,y2,'b-')
ax2.set_ylabel("accelération", color='b')

# Anotations
ax2.annotate("Début expérience",xy=(0.0,1.0),xytext=(0, 3),
arrowprops=dict(facecolor='green', shrink=0.15))

# Axe supplémentaire
repere=[ax.get_lines()[0], ax2.get_lines()[0]]
plt.grid(True)
plt.legend(repere,["f()","g()"], loc="upper left")

# Visualisation
plt.show()
```

Résultat



4 Les graphiques 3D

Les graphiques 3-dimensions (3D) permettent la représentation graphique dans l'espace d'un traitement et/ou d'une prédiction z en fonction de deux données x et y : z = f(x,y). On utilisera donc trois axes de représentation graphique.

Sur un graphique 3D, on peut représenter des points, des courbes filaires, des surfaces, des polygones, des histogrammes, etc.

4.1 Courbe filaire en 3D

On souhaite représenter en 3D la fonction suivante :

$$z = f(x,y) = x^2(\cos x \sin y) + y^2(\cos y \sin x) \text{ où } x, y \text{ et } z \in \mathbb{R}$$

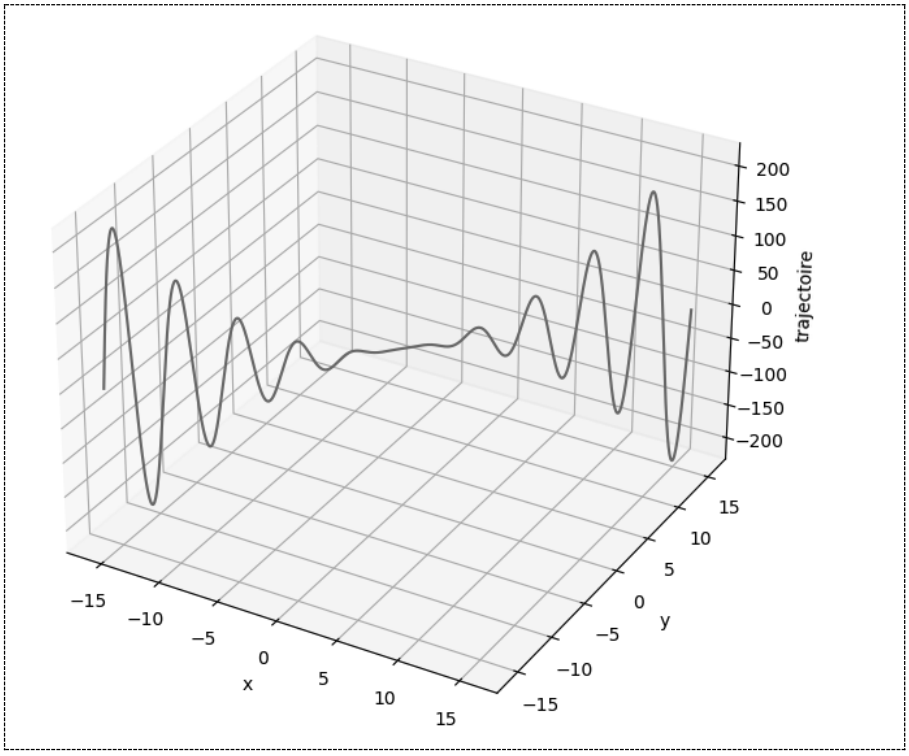
Pour l'implémentation, on définit :

$$x \in [-5\pi, 5\pi] \text{ et } y \in [-5\pi, 5\pi]$$

4.1.1 Exemple

```
import matplotlib.pyplot as plt
import numpy as np

def f(x,y):
   return(((x**2)*(np.cos(x))*np.sin(y))+((y**2)*(np.cos(y))*np.sin(x)))

X=np.linspace(-5*np.pi,5*np.pi,1000)
Y=np.linspace(-5*np.pi,5*np.pi,1000)
Z=f(X,Y)

fig=plt.figure(figsize=[6, 6])
print(fig.get_size_inches())
ax = fig.add_subplot(projection='3d')
ax.plot(X,Y,Z, label='Courbe 3D')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('trajectoire')
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=0.5)
plt.show()
```

4.1.2 Visualisation d'une courbe filaire



4.2 Surface en 3D

On dispose de deux fonctions supplémentaires qui facilitent la représentation et visualisation de surfaces en 3D :

- la fonction *plot_wireframe()* pour la représentation de graphiques 3D en treillis ;
- la fonction *plot_surface()* pour la représentation de graphiques 3D avec un rendu solide.

On peut également utiliser la fonction *meshgrid()* de la librairie *numpy* afin de créer une grille de données (x, y).

On souhaite avoir une visualisation 3D d'une représentation en treillis puis en surface de la fonction suivante :

$$z = \frac{(\cos x)^2 + (\sin y)^2}{1 + x^2} \text{ où } x, y \text{ et } z \in \mathbb{R}$$

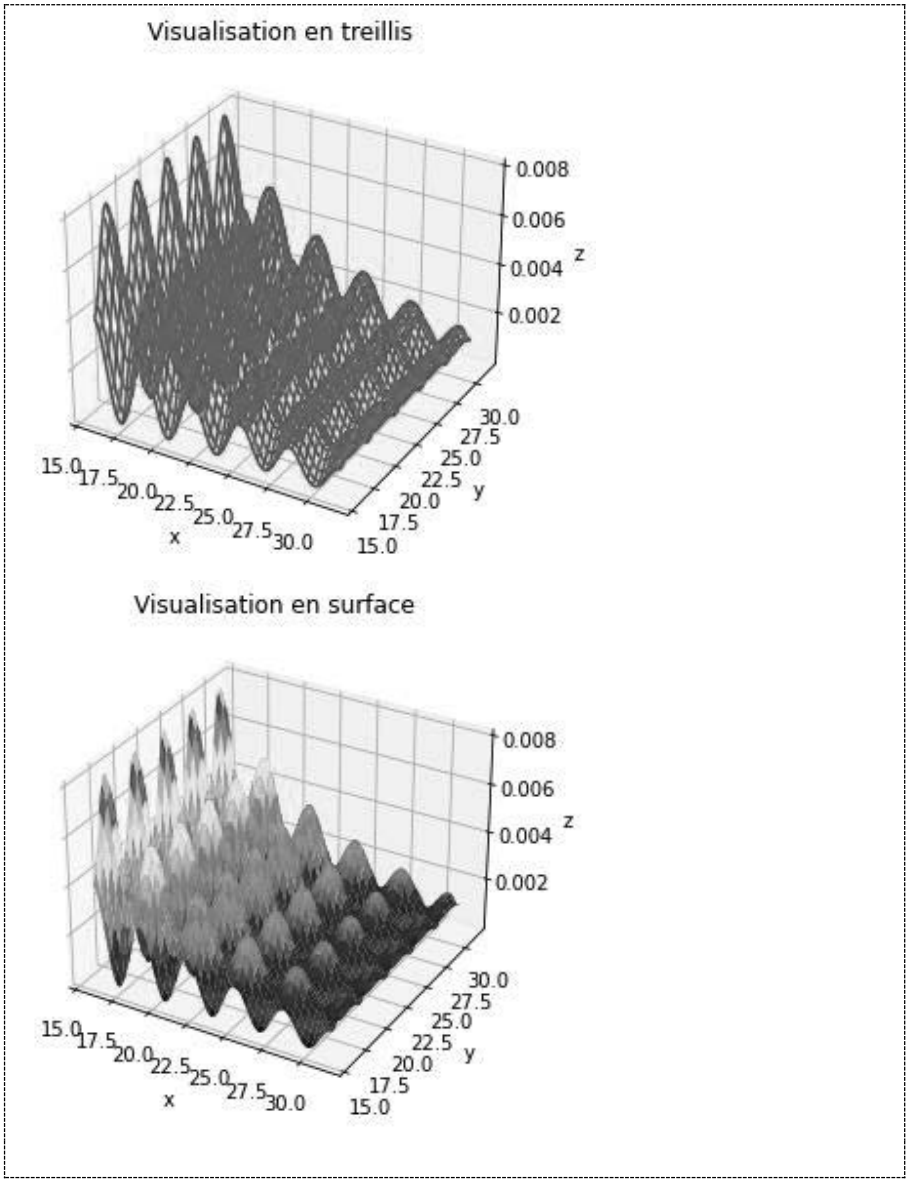
Pour l'implémentation, on définit :

$$x \in [5\pi, 10\pi] \; et \; y \; \in [5\pi, 10\pi]$$

4.2.1 Exemple

```
import numpy as np
import matplotlib.pyplot as plt

x, y = np.meshgrid(np.linspace(5*np.pi, 10*np.pi, 100),
np.linspace(5*np.pi, 10*np.pi, 100))
z = (((np.cos(x))**2) + ((np.sin(y))**2))/(1+x**2)

# Figure
fig = plt.figure(figsize=(10, 10))

# Graphique de gauche en treills
ax1 = fig.add_subplot(1, 2, 1, projection="3d")
ax1.plot_wireframe(x, y, z, color='m')
ax1.set_title("Visualisation en treillis",fontsize=12)
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_zlabel('z')

# Graphique de droite en surface
ax2 = fig.add_subplot(1, 2, 2, projection="3d")
ax2.plot_surface(x, y, z, cmap="nipy_spectral")
ax2.set_title("Visualisation en surface",fontsize=12)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_zlabel('z')

# Affichage du graphique
plt.show()
```

4.2.2 Visualisation d'une surface en 3D



5 Autres types de graphiques

D'autres types de graphiques peuvent être réalisés et visualisés à l'aide de la librairie *matplotlib* selon le contexte de traitement de données. Dans la suite nous allons

aborder quelques graphiques utilisés en statistiques, cartes géographiques ou encore les matrices de densité :

- nuage de points ;
- histogramme et diagramme et bâtons ;
- graphique circulaire ;
- diagramme en boîte ;
- lignes de niveau ;
- image d'une matrice dans une palette de couleurs ;
- etc.

5.1 Nuage de points

On utilise la fonction *scatter()* de *pyplot* afin de réaliser un graphique en nuage de points. Plusieurs options de la fonctions *scatter()* permettent de personnaliser la visualisation telle que la couleur, la forme du point, etc.

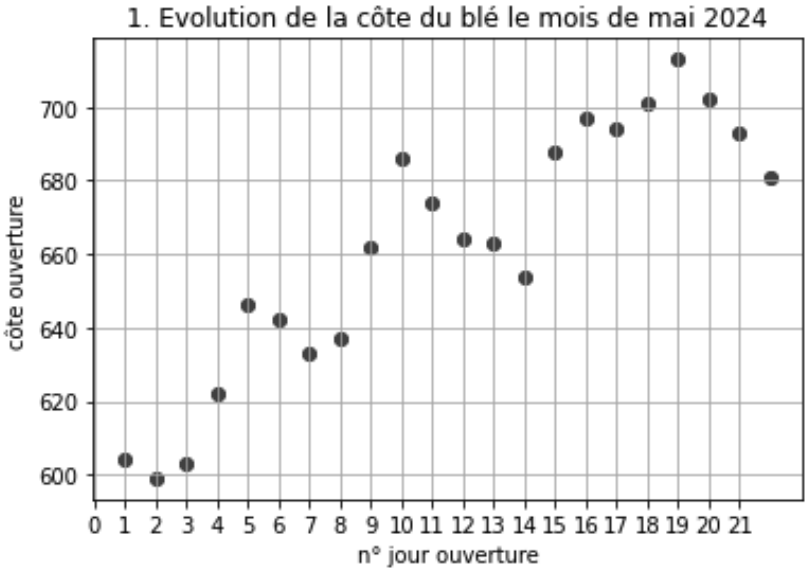
5.1.1 Exemple

On dispose d'un fichier *csv* qui répertorie l'évolution de la côte du blé sur le mois de mai (jour, ouverture, haut, bas, clôture). On souhaite représenter la côte à l'ouverture à l'aide d'un graphique en nuage de points.

```
import pandas as pd
import matplotlib.pyplot as plt

# Chargement de la data
data=pd.read_csv("datable052024.csv", sep=';')
print(data)

plt.scatter(data["jour"],data["ouverture"],c='b', marker='o')
plt.title('1. Evolution de la côte du blé le mois de mai 2024')
plt.xlabel('n° jour ouverture')
plt.xticks([i for i in range(data.shape[0])])
plt.ylabel('côte ouverture')
plt.grid()

plt.show()
```

5.1.2 Résultat

	jour	*ouverture*	*haut*	*bas*	*cloture*
0	*1*	*604*	*609*	*594*	*599*
1	*2*	*599*	*616*	*596*	*604*
2	*3*	*603*	*633*	*603*	*623*
3	*4*	*622*	*650*	*610*	*649*
...					
20	*21*	*693*	*694*	*673*	*681*
21	*22*	*681*	*692*	*674*	*679*



5.2 Histogramme

On dispose d'un jeu de données médicales concernant quelques centaines de personnes. On souhaite afficher graphiquement l'effectif par genre sur un histogramme simple. On utilise la fonction *hist()* de *pyplot* avec la possibilité de définir les options de l'histogramme. En effet, voici, ci-après quelques options de la fonctions *hist()*.

Paramètre	**Détails**	**Exemple**
label	Nom donné à la série	label='Effectif par genre'
log	Echelle logarithmique	log=True

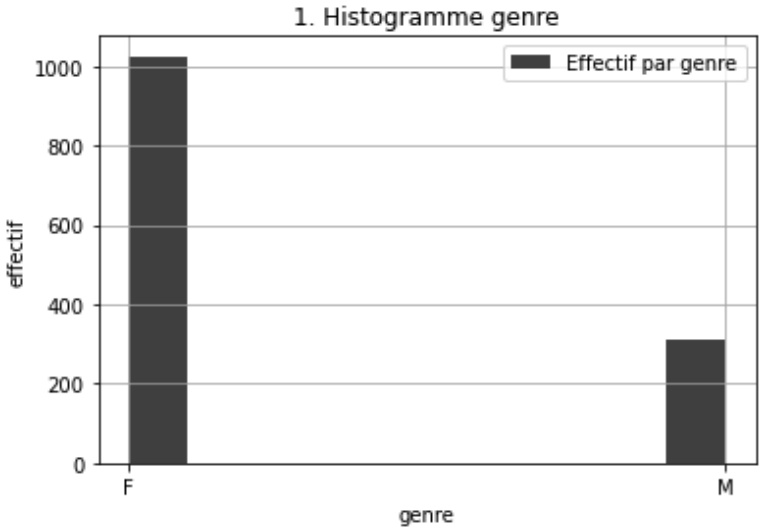
Paramètre	Détails	Exemple
align	Alignement de la barre par rapport au centre de l'intervalle : 'left', 'mid'ou 'right'.	align='left'
color	Couleur des barres	color='blue'
edgecolor	Couleur du cadre des barres.	edgecolor='red'
hatch	Hachuer les barres : '/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'	hatch='*'
histtype	'bar' histogramme avec des barres	histtype='bar'
	'barstacked' histogrammes barres empilés	
orientation	Orientation des barres : 'horizontal' ou 'vertical'.	orientation='vertical'
rwidth	Largeur de la barre par rapport à l'intervalle (fraction de 1. rwidth=0,5 correspond à 50%).	rwidth=0.5
stacked	Plusieurs séries de données empilées	stacked=True

5.2.1 Exemple

```
import matplotlib.pyplot as plt
import pandas as pd

data=pd.read_csv("DataMedical.csv", sep=';')
nbGenre=data[["genre","age"]].groupby("genre").count()
print(nbGenre)

plt.hist(data['genre'],rwidth=1,histtype='bar', align='mid',color='blue',
label='Effectif par genre')

plt.title('1. Histogramme genre')
plt.xlabel('genre')
plt.ylabel('effectif')
plt.grid()
plt.legend()

plt.show()
```

5.2.2 Résultat

```
        age
genre
F       1025
M        313
```



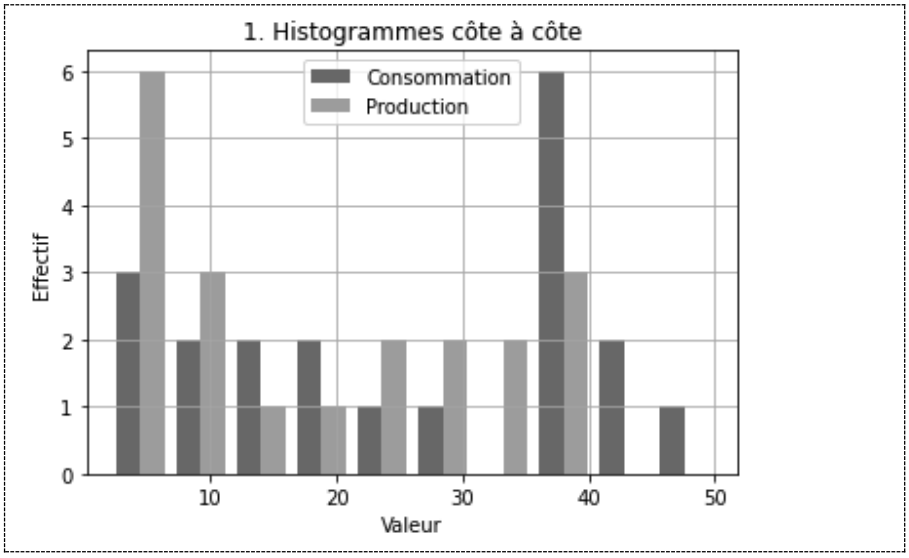
Histogrammes côte à côte

On dispose d'un jeu de données contenant deux séries de données consommation/production d'un système dynamique. On souhaite afficher les histogrammes de consommation/production côte à côte sur la même figure.

```
# Utiliser les mêmes librairies
import matplotlib.pyplot as plt
import numpy as np

# Figure
fig=plt.figure()

# Les séries de données
# Série y1
y1=np.array([24,13,36,37,50,40,10,40,45,5,16,40,30,21,4,20,7,44,39,2])
# Série y2
y2=np.array([2,6,33,25,4,10,30,37,3,7,6,20,29,4,33,24,36,15,11,38])
```

```
# Histogramme
plt.hist([y1,y2],stacked=False, label=['Consommation','Production'])

# Visualisation
plt.title('1. Histogrammes côte à côte')
plt.xlabel('Valeur')
plt.ylabel('Effectif')

# Quadrillage
plt.grid()

# Légende
plt.legend()

plt.show()
```

Résultat

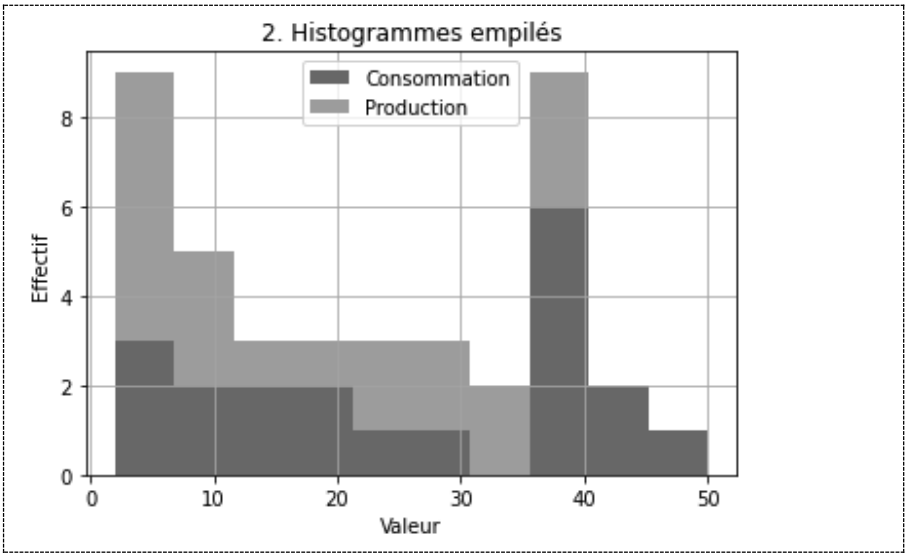


Histogrammes empilés

L'option *stacked* de la fonction *hist()* est à valeur booléenne. Lorsqu'elle prend la valeur *True* les histogrammes sont mis côte à côte et lorsqu'elle prend la valeur *False* les histogrammes sont empilés.

Pour le même exemple précédent consommation/production, on met *stacked = True*.

Résultat



5.3 Lignes de niveau

Définition

Soit *f* une fonction réelle à deux variables, et *h* un nombre réel. On appelle L_h ligne de niveau h de la fonction *f* telle que l'ensemble des points (x, y) du plan (Oxy) en lesquels *f* prend la valeur *h* :

$$L_h = \{(x, y) \in \mathbb{R}^2 \; tels \; que \; f(x, y) = h \}$$

5.3.1 Exemple : les cercles concentriques

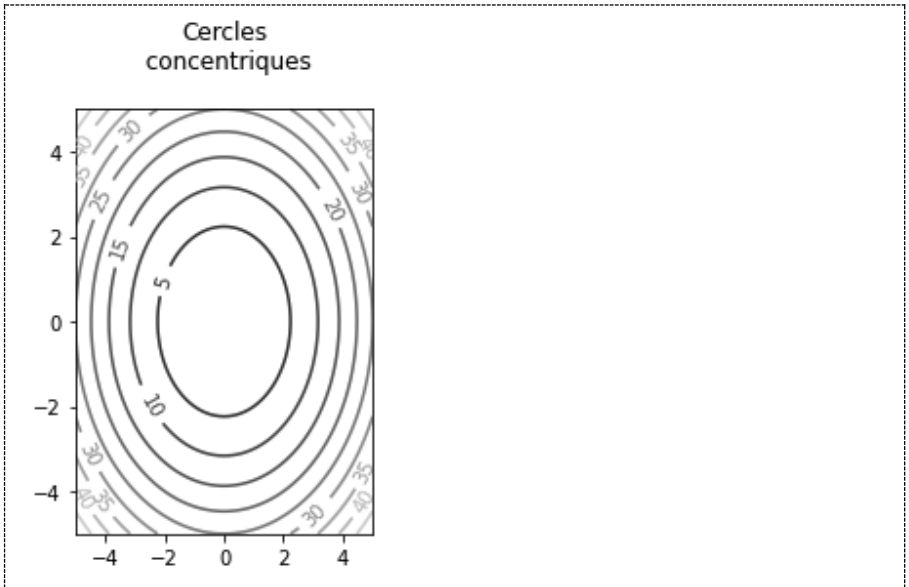
Dans cet exemple, la fonction à deux variables *f* définie comme suit :

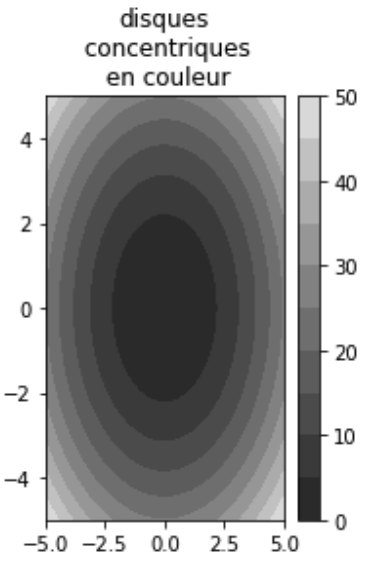
$$f(x, y) = x^2 + y^2$$

```
import numpy as np
import matplotlib.pyplot as plt

fig=plt.figure()
x=np.linspace(-5,5,100)
```

```
y=np.linspace(-5,5,100)
X,Y =np.meshgrid(x,y)
Z=X**2+Y**2

# 1er graphique
plt.subplot(1,2,1)
# 10 niveaux
c1=plt.contour(X,Y,Z,10)
plt.clabel(c1)
plt.title("Cercles\n concentriques\n")

# 2ème graphique
plt.subplot(1,2,2)
# 10 niveaux
c2=plt.contourf(X,Y,Z,10)
plt.colorbar()
plt.title("disques\n concentriques\n en couleur")

plt.show()
```

5.3.2 Résultat





Voici un autre exemple avec des lignes topographiques

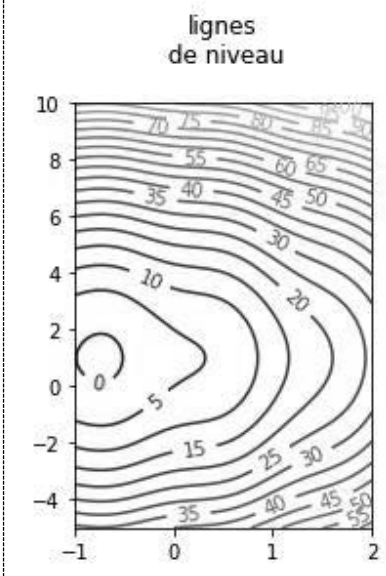
On souhaite représenter sur un plan les lignes topographiques d'un terrain. On utilise la modélisation mathématique afin de représenter les formes du terrain, particulièrement le dénivelé dans l'exemple suivant. Supposons que le dénivelé du terrain est théoriquement modélisable à l'aide la fonction à deux variables *f* définie comme suit :

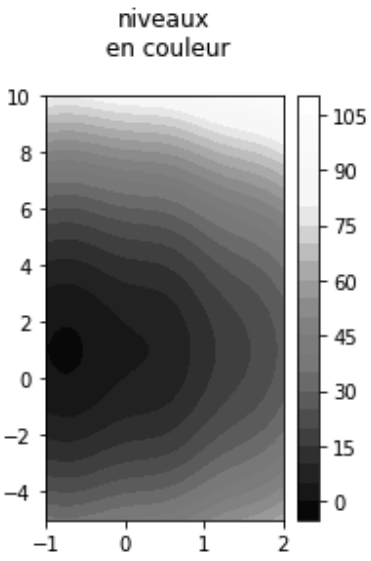
$$f(x,y) = 3(x+1)^2 + (y-1)^2 + \sin\left(\sqrt{\pi} + y\right)$$

```
import numpy as np
import matplotlib.pyplot as plt

fig=plt.figure()

x=np.linspace(-1,2,100)
y=np.linspace(-5,10,100)
X,Y =np.meshgrid(x,y)

# Fonction f
Z=3*(X+1)**2+(Y-1)**2+np.sin(np.sqrt(np.pi)+y)
```

```
# 1er graphique
plt.subplot(1,2,1)
# 25 niveaux
c1=plt.contour(X,Y,Z,25)
plt.clabel(c1)
plt.title("lignes\n de niveau\n")

# 2ème graphique
plt.subplot(1,2,2)
# 25 niveaux
c2=plt.contourf(X,Y,Z,25,cmap='hot')
plt.colorbar()
plt.title("niveaux\n en couleur\n")

# Graphique
plt.show()
```

Résultat





5.4 Graphique circulaire

Un graphique circulaire dit aussi graphique en secteurs, en anglais *pie chart*, est utilisé pour représenter l'importance relative d'une catégorie de données par rapport à d'autres catégories en étude. Cette représentation est plus adéquate pour la visualisation d'une proportion, d'un pourcentage ou encore de valeurs comparables comme la fréquence d'utilisation, le nombre de citations, etc.

On utilise la fonction *pie* de *pyplot* pour réaliser cette visualisation.

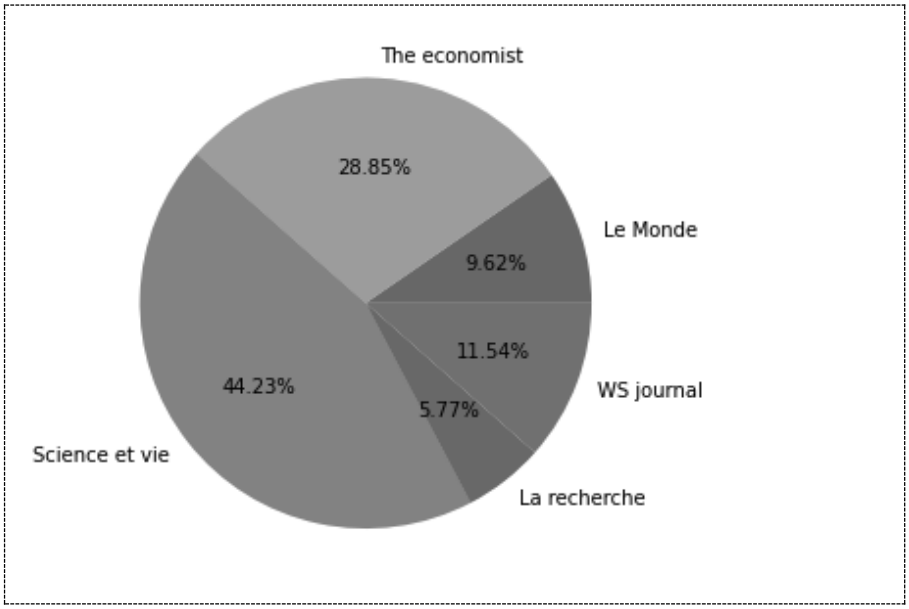
5.4.1 Exemple 1

Création d'un graphique circulaire à partir de deux listes de données (catégorie et % respectivement journal et citation).

```
import matplotlib.pyplot as plt

Journal = ['Le Monde', 'The economist', 'Science et vie', 'La recherche',
'WS journal']
Citation = [10,30,46,6,12]

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

# Visualisation et affichage
```

```
ax.pie(Citation, labels = Journal,autopct='%1.2f%%')
plt.show()
```

5.4.2 Résultat de l'exemple 1



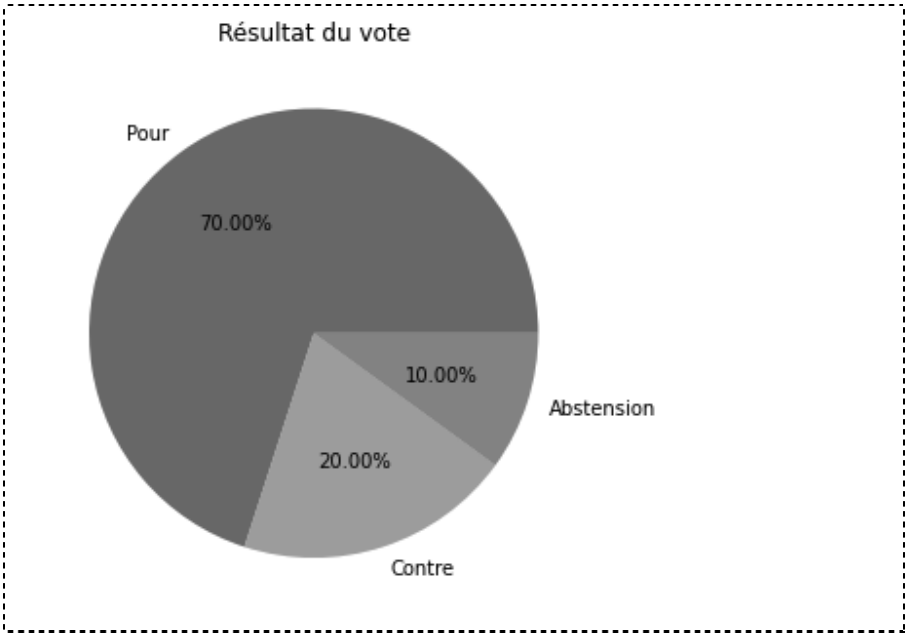
5.4.3 Exemple 2

Création d'un graphique circulaire à partir d'un dictionnaire de données et *DataFrame* de *pandas* (catégorie et %).

```
import matplotlib.pyplot as plt
import pandas as pd

# Figure et axes
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

# Data frame
data={'Vote':['Pour', 'Contre','Abstension'], 'NbVoix':[70,20,10]}
df=pd.DataFrame(data)

# Visualisation et affichage
```

```
ax.pie(df['NbVoix'], labels=df['Vote'], autopct='%1.2f%%')
plt.title(label="Résultat du vote")
plt.show()
```

5.4.4 Résultat de l'exemple 2



5.4.5 Exemple 3

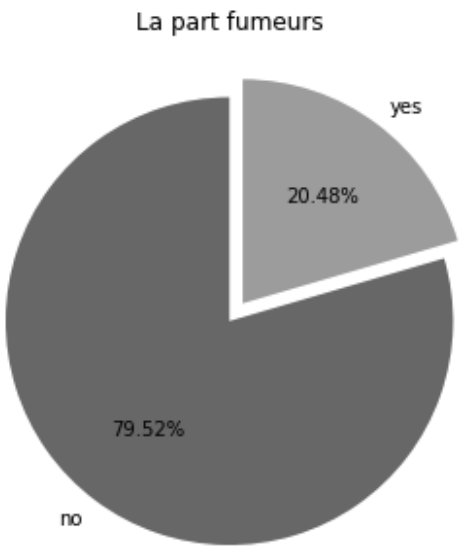
Création d'un graphique circulaire à partir d'un jeu de données enregistré dans un fichier csv et importé par *read_csv* de *pandas* (groupes et %).

```
import matplotlib.pyplot as plt
import pandas as pd

# Data Frame
data = pd.read_csv("DataMedicalPie.csv", sep=';')
fumeur = data['smoker'].value_counts()
print(fumeur)

groupes = fumeur.index
print(groupes)
```

```
# Figure et axes
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.axis('equal')

# La part à faire resortir, ici la 2ème part
part = (0, 0.1)

# Visualisation et affichage
ax.pie(fumeur, labels=groupes, explode=part, autopct='%1.2f%%',
startangle=90)
plt.title(label="La part fumeurs")
plt.show()
```

5.4.6 Résultat de l'exemple 3

```
no     1064
yes     274
Name: smoker, dtype: int64
Index(['no', 'yes'], dtype='object')
```



5.5 Graphique en beignet

Le graphique en beignet ou encore *donut chart* en anglais, est une version masterisée du graphique circulaire. En effet, on vient ajouter au centre un trou circulaire de rayon identique à toutes les parts du graphique circulaire.

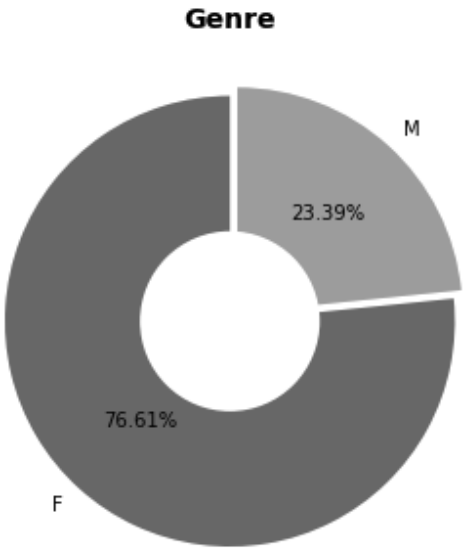
5.5.1 Exemple

Création d'un graphique en beignet à partir d'un jeu de données enregistré dans un fichier csv et importé par *read_csv* de *pandas* (genre et %).

```
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv("DataMedicalPie.csv", sep=';')
population = data['genre'].value_counts()
print(population)

groupes = population.index
print(groupes)

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.axis('equal')

# La part à faire resortir, ici la 2ème part
part = (0, 0.05)

ax.pie(population, labels=groupes, explode=part, autopct='%1.2f%%',
startangle=90)

titre="Genre"
plt.title(titre, loc='center', size=14, weight="bold")

trou = plt.Circle((0, 0), 0.4, facecolor='white')
ax.add_artist(trou)

plt.show()
```

5.5.2 Résultat

```
F    1025
M     313
Name: genre, dtype: int64
Index(['F', 'M'], dtype='object')
```



5.5.3 Image d'une matrice

Un graphique *« image d'une matrice »* permet de représenter des valeurs d'une matrice à double entrée et leur importance dans une échelle de couleurs comme *hot* ou *seismic*. Ces valeurs peuvent être des relevés d'observation de terrain ou d'expérimentation, mais peuvent être aussi générées par une fonction à deux variables.

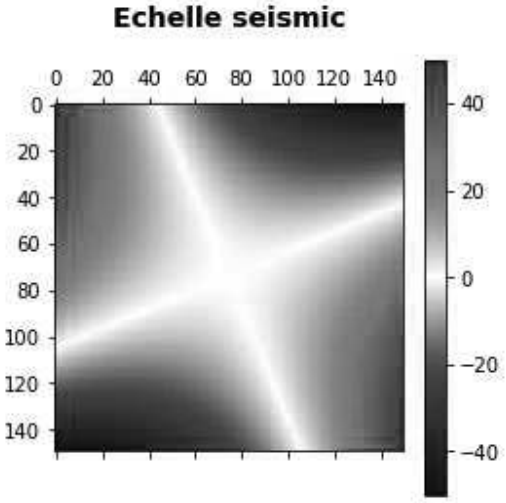
5.5.4 Exemple

Pour générer des valeurs dans cet exemple, on utilise la fonction à deux variables *f* définie comme suit :

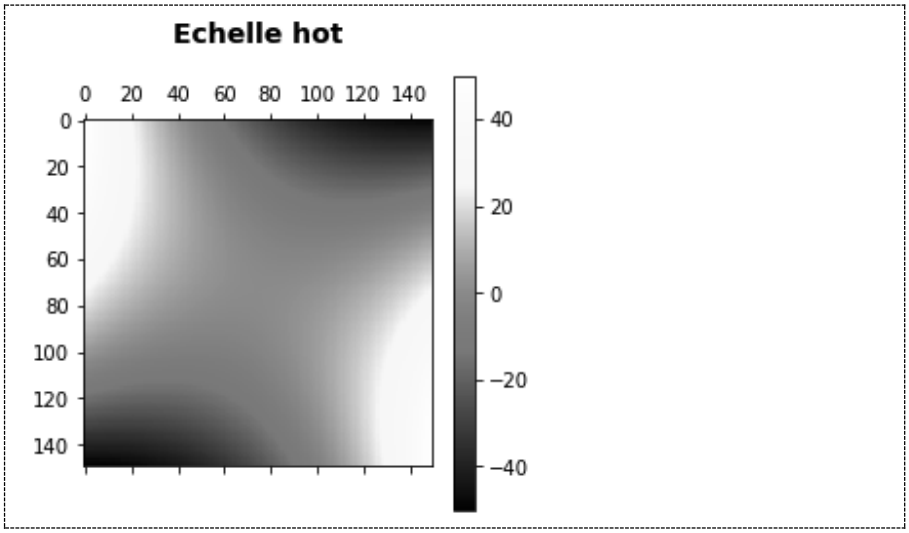
$$f(x,y) = x^2 - y^2 + 2xy \quad o\grave{u}\ \ x,y\ [-5{,}5]$$

```
import matplotlib.pyplot as plt
import numpy as np

def T(t,i):
    r=(t**2-i**2+2*t*i)
    return(r)

x=np.linspace(-5,5,150)
y=np.linspace(-5,5,150)
X, Y = np.meshgrid(x,y)
Z=T(X,Y)

plt.matshow(Z,cmap='seismic')
plt.colorbar()
plt.title("Echelle seismic\n", loc='center', size=14, weight="bold")

plt.show()
```

5.5.5 Résultat



Voici, ci-après le graphique *« image d'une matrice »* sur une échelle de couleurs *hot* (*cmap='hot'* dans *matshow*).



5.6 Graphique en radar

Un graphique en radar dit aussi toile d'araignée (*polygon radar chart* en anglais) est utilisé pour représenter et comparer des valeurs d'une ou plusieurs variables selon plusieurs critères.

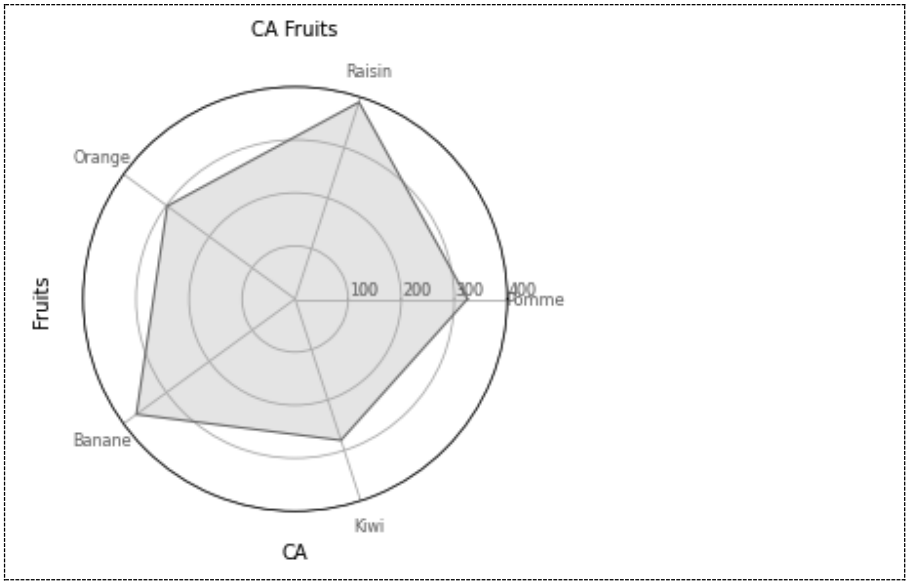
5.6.1 Exemple

Visualiser sous forme d'un graphique radar le chiffre d'affaires CA de vente de fruits (pomme, raisin, orange, banane et kiwi) sur 4 périodes successives (en l'occurrence 4 trimestres T1, T2, T3 et T4).

```
# Importation des librairies utiles
import matplotlib.pyplot as plt
import pandas as pd
from math import pi

# Définition du Data Frame
df = pd.DataFrame({
# Périodes
'CA': ['T1','T2','T3','T4'],
# Fruits
'Pomme': [325, 145, 350, 245],
'Raisin': [390, 110, 390, 234],
```

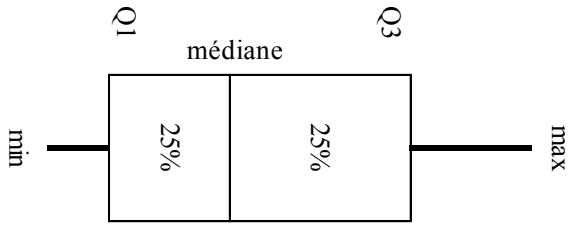
```
'Orange': [298, 139, 223, 124],
'Banane': [370, 320, 150, 140],
'Kiwi': [280, 115, 230, 250]
})

# Catégories
categories=list(df)[1:]
N = len(categories)

# Valeurs
valeurs=df.loc[0].drop('CA').values.flatten().tolist()
valeurs += valeurs[:1]

theta = [n / float(N) * 2 * pi for n in range(N)]
theta += theta[:1]

# Graphique
ax = plt.subplot(111,label='CA', polar=True)
plt.xticks(theta[:-1], categories, color='green', size=12)
ax.set_rlabel_position(0)

plt.yticks([100,200,300,400], ["100","200","300","400"], color="red",
size=10)
plt.ylim(0,400)

ax.plot(theta, valeurs, linewidth=1, linestyle='solid')
ax.fill(theta, valeurs, 'b', alpha=0.1)

plt.xlabel('CA',loc='center')
plt.ylabel('Fruits\n')

plt.title("CA Fruits\n", verticalalignment='center',
horizontalalignment='center', size=14)

plt.show()
```

5.6.2 Résultat



5.7 Diagramme en boîte

Un diagramme en boîte dit aussi boîte à moustaches ou encore *box plot* en anglais, est utilisé pour visualiser graphiquement quelques caractéristiques d'une série de données numériques, à savoir le minimum, le maximum, la médiane, les 1er et le 3ème quartiles.



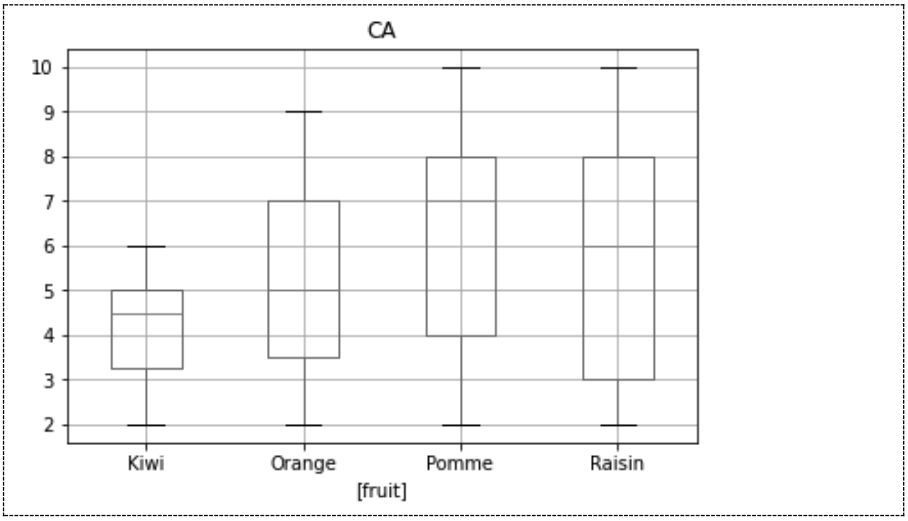
5.7.1 Exemple 1

On dispose d'un tableau de vente de fruits (Kiwi, Orange, Pomme, Raisin) où on peut trouver le chiffre d'affaires en K€ (CA), le volume de production en Kt

(Production) et le mois de vente. On souhaite visualiser les diagrammes à boîtes des fruits par chiffre d'affaires.

```
import matplotlib.pyplot as plt
import pandas as pd

df=pd.read_csv('datavente.csv',sep=';')
df.boxplot(by=['fruit'], column=['CA'], grid=True)

plt.suptitle("")
plt.show()
```

5.7.2 Résultat de l'exemple 1



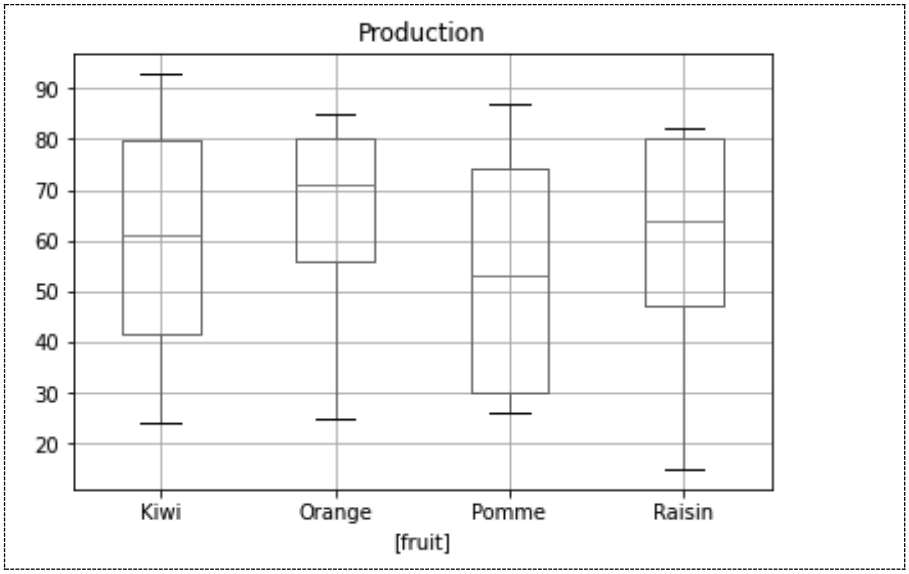
5.7.3 Exemple 2

Même exemple que précédent. On souhaite visualiser les diagrammes à boîtes des fruits par production.

```
import matplotlib.pyplot as plt
import pandas as pd

df=pd.read_csv('datavente.csv',sep=';')
```

```
df.boxplot(by=['fruit'], column=['Production'], grid=True)

plt.suptitle("")
plt.show()
```

5.7.4 Résultat de l'exemple 2



Chapitre 4

Nombres complexes et mathématiques symboliques

Commençons par quelques rappels mathématiques sur les structures algébriques.

1 Structure algébrique

1.1 Produit cartésien

Soient E et F deux ensembles. On appelle produit cartésien de E par F l'ensemble, noté E x F, des couples (x, y) tels que x appartient à E et y appartient F.

$$\mathrm{E\ x\ F} = \{(x, y) |\ x\ \in E, y\ \in F\}$$

1.2 Loi de composition interne

Soit E un ensemble. $\oplus$ est une loi de composition interne de E si pour tous les éléments x et y de E, l'élément x $\oplus$ y est un élément de E.

$$\forall\, x, y\ \in E, \qquad x \oplus \mathrm{y}\ \in \mathrm{E}$$

Exemple

$E = \mathbb{R}$ et $\oplus$ est l'addition de deux réels

$$\forall\, x, y\ \in\ \mathbb{R}, \qquad x + \mathrm{y}\ \in \mathbb{R}$$

1.3 Groupe

Un ensemble *E* muni d'une loi de composition interne $\oplus$ est un groupe si :

- $\oplus$ est associative

  $\forall\, (x, y, z)\ \in\ E^3 \quad x \oplus (\mathrm{y} \oplus \mathrm{z}) = (x \oplus \mathrm{y}) \oplus \mathrm{z}$;
- $\oplus$ admet un élément neutre, noté 0_E, dans *E*

  $\exists!\ 0_E\ \in E, \forall\, x\ \in E \quad x \oplus 0_E = 0_E \oplus \mathrm{x} = \mathrm{x}$;
- tout élément x de *E* admet un élément symétrique x' dans *E* par la loi de composition interne $\oplus$

  $\forall\, x\ \in E, \exists\, x' \in E\ \ x \oplus x' = x' \oplus x = 0_E$.

(E, $\oplus$) est un groupe.

Si en plus la loi $\oplus$ est commutative, c'est-à-dire : $\forall\ (x, y) \in\ E^2, x \oplus \mathrm{y} = \mathrm{y} \oplus \mathrm{x}$ alors (E, $\oplus$) est un groupe commutatif dit aussi un groupe abélien.

Exemple

$E = \mathbb{N}$ et $\oplus$ est l'addition de deux entiers naturels.

1.4 Corps commutatif

Un ensemble K muni de deux lois internes $\oplus$ et $\otimes$ est un corps commutatif si :

- (K, $\oplus$) est un groupe commutatif
- $\otimes$ est distributive par rapport à $\oplus$

$$\forall\ (x, y, z)\ \in\ E^3 \quad x \otimes (\mathrm{y} \oplus \mathrm{z}) = (x \otimes \mathrm{y}) \oplus (x \otimes \mathrm{z})$$

- L'ensemble $K^* = K - \{0_K\}$ muni de la loi interne $\otimes$ est un groupe commutatif.

Exemple

($\mathbb{R}$, +, x) est un corps commutatif où + est l'addition de deux réels et x est la multiplication de deux réels.

2 Corps des nombres réels

L'ensemble $\mathbb{R}$ mini d'une addition (+) interne et d'une multiplication (*) interne est un corps commutatif.

De plus :

$$\forall\ (x, y) \in\ \mathbb{R}^2, \quad \begin{array}{l} \forall\ a\ \in\ \mathbb{R}\ \ x \leq y \iff x + a\ \leq y + a \\ \forall\ a\ \in\ \mathbb{R}^+\ \ x \leq y \iff x * a\ \leq y * a \end{array}$$

Dans la suite, on utilisera la notation * pour désigner la multiplication.

2.1 Partie entière

2.1.1 Définition

Pour tout réel a, il existe un unique entier relatif *e* tel que :

$$e\ \leq a\ < e + 1$$

Notation :

$$e = [a] = E(a)$$

2.1.2 Propriétés

- $\forall\,(a,b) \in \mathbb{R}^2 : \ [a]+[b] \leq [a+b]$;
- $\forall\, a \in \mathbb{R}, \forall\, n \in \mathbb{Z} \ : \ [a+n] = [a]+n$;
- $\forall\,(a,b) \in \mathbb{R}^2 : a \leq b \implies [a] \leq [b]$.

2.2 Valeur absolue

2.2.1 Définition

Pour tout réel *a*, le plus grand des deux réels *a* et *-a* s'appelle la valeur absolue de *a*, notée |a|.

$$\forall\, a \in \mathbb{R} : |a| = \max(a, -a)$$

2.2.2 Propriétés

- $\forall\, a \in \mathbb{R} : \begin{cases} |a| \in \mathbb{R}^+ \\ |a| = 0 \iff a = 0 \end{cases}$;
- $\forall\,(a,b) \in \mathbb{R}^2 : |a * b| = |a| * |b|$;
- $\forall\,(a,b) \in \mathbb{R}^2 : \big||a| - |b|\big| \leq |a+b| \leq |a| + |b|$ (inégalité triangulaire).

3 Corps des nombres complexes

3.1 Définition

Tout nombre complexe z admet une écriture unique sous la forme suivante :

z = a + i*b (ou encore z = a + ib)

où a et b sont deux réels et i est le nombre complexe spécial qui vérifie : $i^2 = -1$.

a + i*b est l'écriture cartésienne de z.

L'ensemble des nombres complexes est noté : $\mathbb{C}$.

Remarque

L'unité imaginaire en mathématiques est notée i et sous Python est notée j.

Exemples

```
# Exemples de nombres complexes sous Python
z1=3+2j
z2=-3j
```

```
z1
z2
```

(3+2j)
(-0-3j)

```
# En utilisant un constructeur
z=complex(3, 2)


z
```

(3+2j)

3.2 Propriété

(ℂ, +, *) est un corps commutatif.

3.3 Partie réelle, partie imaginaire

3.3.1 Définitions

Pour tout nombre complexe z = a + i*b :

- a est la partie réelle de z, notée : a = Re(z) ;
- b est la partie imaginaire de z, notée b = Im(z) ;
- si z = Re(z) alors z est un réel ;
- si z = Im(z) alors est un imaginaire pur ;
- Le conjugué de z est $\overline{z}$ = a – i*b.

Exemples

```
# Partie réelle et partie imaginaire
a1, b1 = z1.real, z1.imag


a1, b1
```

(3.0, 2.0)

```
# Le conjugué d'un nombre complexe
z = round(3*np.pi, 2)-2j
zc = z.conjugate()
```

```
z, zc
```

((9.42-2j), (9.42+2j))

```
# Conjugué en utilisant numpy
import numpy as np
zc=np.conjugate(z)


z, zc
```

((9.42-2j), (9.42+2j))

3.3.2 Propriétés

- $\overline{z + z'} = \overline{z} + \overline{z'}$;
- $\overline{z * z'} = \overline{z} * \overline{z'}$;
- $\overline{\overline{z}} = z$;
- z + $\overline{z}$ = 2*Re(z) ;
- z − $\overline{z}$ = 2*Im(z)*i ;
- z * $\overline{z}$ = $a^2 + b^2$;
- z est un réel ⇔ z = $\overline{z}$;
- z est un imaginaire pur ⇔ z = $-\overline{z}$.

Vérifications

```
np.conjugate(z1+z2) == np.conjugate(z1)+np.conjugate(z2)
```

True

```
np.conjugate(z1*z2) == np.conjugate(z1)*np.conjugate(z2)
```

True

```
z*np.conjugate(z) == z.real**2 + z.imag**2
```

True

3.4 Module et argument

3.4.1 Définition : module

On appelle module du nombre complexe z = a + i*b, le réel $\sqrt{a^2 + b^2}$ noté :

$$|z| = \sqrt{z * \overline{z}} = \sqrt{a^2 + b^2}$$

Cas particulier : si z est un réel alors son module est égal à sa valeur absolue.

3.4.2 Propriétés : module

- $\forall z \in \mathbb{C} : |z| \in \mathbb{R}^+$;
- $\forall z \in \mathbb{C} : |z| = 0 \Leftrightarrow z = 0$;
- $\forall (z, z') \in \mathbb{C} : |z * z'| = |z| * |z'|$;
- $\forall (z, z') \in \mathbb{C} : \big||z| - |z'|\big| \leq |z + z'| \leq |z| + |z'|$.

3.4.3 Définition : argument

Soient z un nombre complexe non nul et r son module r = |z|. Il existe un réel θ défini, à $2k\pi$ où $k \in \mathbb{Z}$, comme suit :

$$z = r(\cos\theta + i\sin\theta)$$

où θ est l'argument de z, noté : $\theta = \arg(z)$.

On note que le module du complexe 0 est égal à 0 et le complexe 0 n'a pas d'argument.

On note également que $(\cos\theta + i\sin\theta) = e^{i\theta}$.

Ainsi, tout nombre complexe z admet les trois formes d'écriture suivantes :

- forme cartésienne : $a + ib \quad (a, b) \in \mathbb{R}^2$;
- forme trigonométrique : $r(\cos\theta + i\sin\theta) \quad r \in \mathbb{R}^{+*}, \theta \in \mathbb{R}$;
- forme exponentielle : $re^{i\theta} \quad r \in \mathbb{R}^{+*}, \theta \in \mathbb{R}$.

```
# Module et argument d'un nombre complexe z
# Représentation de z en coordonnées polaires →
# (module, argument) avec polar() de cmath
import cmath
z=complex(2,3)
pol=cmath.polar(z)

pol
```

(3.605551275463989, 0.982793723247329)

```
# Module de z
r=round(pol[0], 2)
# Argument de z
theta=round(pol[1], 2)

r, theta
```

(3.61, 0.98)

```
# Forme cartésienne de z tels que r et theta sont connus
import cmath
r=4
theta=3*cmath.pi/4
cmath.rect(r,theta)
```

(-2.82842712474619+2.8284271247461903j)

```
a = round(cmath.rect(r,theta).real, 2)
b = round(cmath.rect(r,theta).imag,2)
z=complex(a, b)
z
```

(-2.83+2.83j)

```
# Forme exponentielle d'un nombre complexe z
from cmath import *
z=complex(3,5)
pol=polar(z)
z == pol[0]*exp(complex(0,pol[1]))
```

True

3.4.4 Propriétés : argument

- $z = re^{i\theta}$ $\quad \overline{z} = re^{-i\theta}$;
- $z = r(\cos\theta + i\sin\theta)$ $\quad \overline{z} = r(\cos\theta - i\sin\theta)$;
- $z.z' = r.r'e^{i(\theta+\theta')}$;
- $z^{-1} = r^{-1}e^{-i\theta}$;
- $z^n = r^n e^{in\theta}$.

Formule de Moivre :

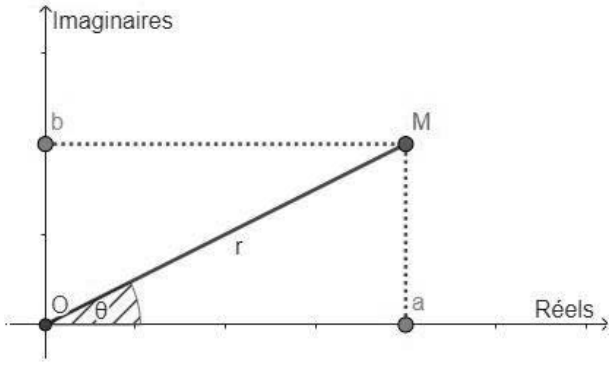
$$z^n = r^n(\cos n\theta + i \sin n\theta)$$

Formule d'Euler :

$$\cos\theta = \frac{1}{2}\left(e^{i\theta} + e^{-i\theta}\right) \text{ et } \sin\theta = \frac{1}{2i}\left(e^{i\theta} - e^{-i\theta}\right)$$

3.5 Interprétation géométrique

Soit O l'origine d'un repère orthonormé du plan. À tout point M du plan de coordonnées (a, b), on associe le nombre complexe z = a + ib. z est dit l'affixe de M ou encore M a pour affixe z.



Le nombre complexe z et son affixe M vérifient les propriétés suivantes :

- $|z| = \|\overrightarrow{OM}\|$;
- arg (z) $= \left(\vec{\imath}\ ,\ \widehat{\overrightarrow{OM}}\right)$;
- Im z = ordonnée de M ;
- Re(z) = abscisse de M.

3.6 Racine n^{ième} d'un nombre complexe

Soient :

- $n \in \mathbb{N}^*$;
- $Z \in \mathbb{C}^*$;
- $z \in \mathbb{C}$;
- $r \in \mathbb{R}^{+*}$ *module* z ;
- $R \in \mathbb{R}^{+*}$ *module* Z

- $\theta\ argument\ de\ z$;
- $\varphi\ argument\ de\ Z$.

3.6.1 Propriété

L'équation $Z = z^n$ (éq.) admet n racines complexes distinctes.

3.6.2 Résolution de l'équation

$z = re^{i\theta}$

$Z = Re^{i\varphi}$

$Z = z^n \Leftrightarrow Re^{i\varphi} = \left(re^{i\theta}\right)^n \Leftrightarrow Re^{i\varphi} = r^n e^{in\theta} \Leftrightarrow$

$$\begin{cases} R = r^n \\ n\theta = \varphi + k2\pi \text{ où } k \in \mathbb{Z} \end{cases} \Leftrightarrow \begin{cases} r = \sqrt[n]{R} = R^{\frac{1}{n}} \text{ sachant } r \text{ et } R \text{ positifs} \\ \theta = \frac{\varphi}{n} + k\frac{2\pi}{n} \text{ où } k \in \mathbb{Z} \end{cases}$$

Les racines de Z sont donc :

$$z_k = R^{\frac{1}{n}} e^{i\left(\frac{\varphi}{n} + k\frac{2\pi}{n}\right)} \text{ où } k \in \{0, 1, \ldots, n-1\}$$

À partir de k = n on retrouve les mêmes racines par périodicité.

3.7 Racines cubiques de 1

3.7.1 Définition

L'équation $z^3 = 1$ où z est un nombre complexe, admet trois racines distinctes :

- forme algébrique

  $z_0 = 1,\ z_1 = -\frac{1}{2} + i\frac{\sqrt{3}}{2},\ z_2 = -\frac{1}{2} - i\frac{\sqrt{3}}{2}$;
- forme trigonométrique

  $z_0 = 1,\ z_1 = \cos\left(\frac{2\pi}{3}\right) + i\sin\left(\frac{2\pi}{3}\right),\ z_2 = \cos\left(-\frac{2\pi}{3}\right) + i\sin\left(-\frac{2\pi}{3}\right)$;
- forme exponentielle

  $z_0 = 1, z_1 = e^{i\frac{2\pi}{3}},\ z_2 = e^{-i\frac{2\pi}{3}}$.

3.7.2 Remarques

- z_0 est une racine cubique triviale de 1 ;

- les deux racines complexes z_1 et z_2 sont dites deux racines cubiques non triviales de 1 ;
- $z_2 = \overline{z_1}$.

3.8 Fonctions complexes

3.8.1 Définition d'une fonction complexe

Soit *f* une fonction complexe à une variable complexe :

$$\begin{array}{rccc} f: & \mathbb{C} & \rightarrow & \mathbb{C} \\ & z & \rightarrow & f(z) \end{array}$$

z s'écrit comme : z = x + iy où $(x, y) \in \mathbb{R}^2$.

Par conséquent $f(z)$ peut s'écrire comme :

$$f(z) = R(x, y) + i\, I(x, y)$$

où $R(x, y)\ et\ I(x, y)$ deux fonctions réelles à deux variables.

De la même façon $f(z)$ peut s'écrire sous forme exponentielle ou encore sous forme trigonométrique.

3.8.2 Visualisation d'une fonction complexe

Le choix de la représentation graphique pour la visualisation de $f(z)$ est fait sur la bijection établie entre un code couleur unique HSL et l'image de z par *f* sous forme :

- polaire, c'est-à-dire en écriture exponentielle :
  $f(z) = f(x + iy) = re^{i\theta}$ où $r\ et\ \theta$ sont respectivement le module et l'argument de z ;
- cartésienne, c'est-à-dire :
  $f(z) = R(x, y) + i\, I(x, y)$ où $R(x, y)\ et\ I(x, y)$ deux fonctions réelles à deux variables.

Exemple 1

$$f(z) = f(x + iy) = (3x^2 - y^2) + i(-2x^2 + y^2)$$

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Partie Réelle de f()
def R(a, b):
    return(3*a**2-b**2)
# Partie imaginaire de f()
def I(a,b):
    return(-2*x**2+y**2)

x = np.linspace(-3, 3, 51)
y = np.linspace(-3, 3, 51)

X, Y = np.meshgrid(R(x,y), I(x,y))
C = np.angle(X + 1j *Y)

plt.pcolormesh(X, Y, C, shading="auto",  cmap=plt.cm.hsv)
plt.colorbar()
plt.xlabel("Partie réelle")
plt.ylabel("Partie imaginaire")
plt.title("Visualisation exemple 1")

plt.show()
```

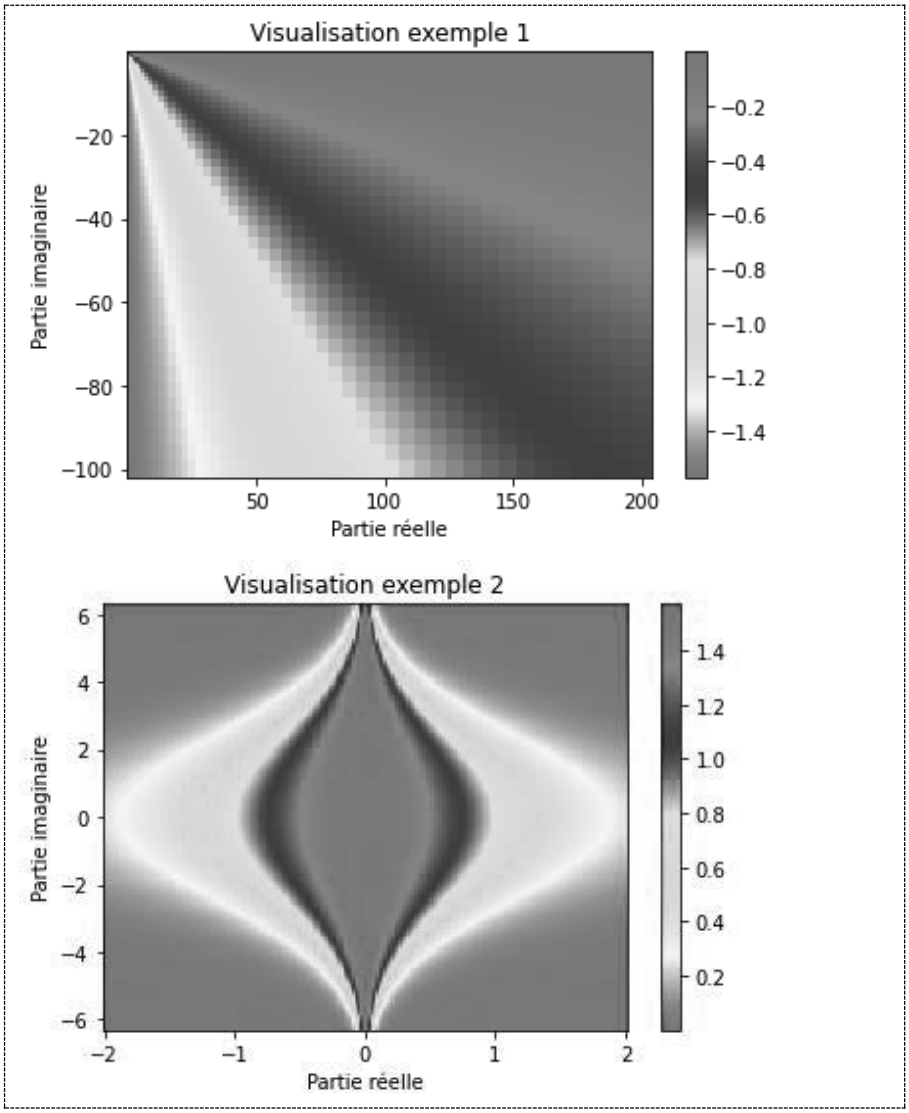
Exemple 2

$$f(z) = f(x + iy) = x^2 + ie^{\frac{-y^2}{6}}$$

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 200)
y = np.linspace(-2*np.pi, 2*np.pi, 200)

X, Y = np.meshgrid(x, y)
C = np.angle(X**2 + 1j *np.exp(-Y**2/6))

plt.pcolormesh(X, Y, C, shading="auto",  cmap=plt.cm.hsv)
```

```
plt.colorbar()
plt.xlabel("Partie réelle")
plt.ylabel("Partie imaginaire")
plt.title("Visualisation exemple 2")

plt.show()
```

Visualisation de l'exemple 1 puis de l'exemple 2



4 Mathématiques symboliques et calcul polynomial

Les mathématiques symboliques permettent la définition et l'utilisation d'expressions mathématiques en vue de manipulation et/ou de résolution de problèmes mathématiques et calcul formel tels que la résolution d'équations ou encore le calcul de dérivées.

Ces manipulations, y compris la création de variables et de fonctions, sont possibles avec les méthodes de la librairie *sympy* de Python.

On abordera ici une liste non exhaustive de manipulations :

- création de variables ;
- écriture d'une expression ;
- instanciation d'une variable ;
- développement d'une expression ;
- factorisation d'une expression ;
- simplification d'une expression ;
- résolution d'équation ;
- définition d'une fonction ;
- dérivée symbolique ;
- limite ;
- intégrale ;
- un exemple du calcul matriciel ;
- etc.

4.1 Importation de *sympy*

On peut importer quelques méthodes, comme *diff()* pour le calcul de la dérivée symbolique ou encore *lambdify()* pour la définition d'une fonction, de la librairie *sympy* :

```
from sympy import diff, lambdify
```

On peut aussi importer toute la librairie *sympy* avec un alias :

```
import sympy as sp
```

ou sans alias :

```
from sympy import *
```

4.2 Création d'une variable

Voici, ci-après la définition de *x* comme une variable réelle à l'aide de la méthode *symbols()*.

```
import sympy as sp
x=sp.symbols('x',Real=True)

print(x)
x
```

Voici, ci-dessous la définition de *x* comme une variable entière relative à l'aide de la méthode *symbols()*.

```
x=sp.symbols('x',Integer=True)

print(x)
x
```

Voici, ci-dessous la définition de *x* comme une variable numérique positive (réelle positive) à l'aide de la méthode *symbols()*.

```
x=sp.symbols('x',Positive=True)

print(x)
x
```

Voici, ci-dessous la définition de plusieurs variables réelles *x, y* et *z* à l'aide de la méthode *symbols()*.

```
x,y,z=sp.symbols('x y z', Real=True)
```

4.3 Écriture d'une expression

On définit une expression symbolique en fonction de variables déjà définies, mobilisées dans des expressions et fonctions mathématiques.

Exemple

```
import sympy as sp

x=sp.symbols('x',Real=True)
exp=x**6-2*x**2+sp.sqrt(2)*x+1
print(exp)
x**6 - 2*x**2 + sqrt(2)*x + 1
```

```
exp
x**6 - 2*x**2 + sqrt(2)*x + 1
```

Afin d'obtenir un affichage en format *LaTex* d'une expression, on fait appel à la méthode *init_printing()* :

```
exp=x**6-2*x**2+sp.sqrt(2)*x+1
sp.init_printing(use_latex=True)

print(exp)
x**6 - 2*x**2 + sqrt(2)*x + 1

exp
```

$$x^6 - 2x^2 + \sqrt{2}x + 1$$

4.4 Évaluer une expression

On utilise la méthode *evalf(nbre_chiffres, ensemble_des_valeurs)* pour évaluer, c'est-à-dire avoir la valeur numérique d'une expression.

Exemple précédent

```
sp.init_printing(use_latex=True)

v=exp.evalf(5,subs={x:2})
print(v)
59.828

v=exp.evalf(5,subs={x:3.14})
print(v)
944.19

# On remplace la constante de l'expression par un paramètre
# Le paramètre a entier relatif puis on substitue a par la valeur pi
import sympy as sp

x=sp.symbols('x',Real=True)
a=sp.symbols('a', Real=True)
```

```
exp=x**6-2*x**2+sp.sqrt(2)*x+a.subs(a,round(sp.pi,2))

exp
```

$$x^6 - 2x^2 + \sqrt{2}x + 3.14$$

Exemple avec plusieurs variables

```
x,y,z=sp.symbols('x y z', Real=True)

y=2*x**4+sp.exp(x)
z=2*y-1

v=z.evalf(6,subs={x:2.1})
print(v)

93.1247
```

4.5 Développement d'une expression

Pour développer une fonction à une variable ou une expression factorisée, on utilise la méthode *expand()* de *sympy*.

```
import sympy as sp

sp.init_printing(use_latex=True)

x,y=sp.symbols('x y', Real=True)
y=(sp.sqrt(x**2+1))*(x-1)**2*(x+1)**2
yd=sp.expand(y)

print(yd)
x**4*sqrt(x**2 + 1) - 2*x**2*sqrt(x**2 + 1) + sqrt(x**2 + 1)

yd
```

$$x^4\sqrt{x^2+1} - 2x^2\sqrt{x^2+1} + \sqrt{x^2+1}$$

De la même façon pour développer une fonction à plusieurs variables ou une expression factorisée à plusieurs variables, on utilise la méthode *expand()* de *sympy*.

```
x,y,z=sp.symbols('x y z', Real=True)
z=(x**2*y**2 - x**2*y + y**2 - y)/(x**2+1)

z
```

$$x^2y\sqrt{x^2+1}\left(\sqrt{x^2+1}-1\right)^2$$

```
zd=sp.expand(z)

zd
```

$$x^4y\sqrt{x^2+1}-2x^4y+2x^2y\sqrt{x^2+1}-2x^2y$$

4.6 Factorisation d'une expression

Pour factoriser une fonction ou une expression à une variable, on utilise la méthode *factor()* de *sympy*.

```
y=sp.sin(x)**2 + 2*sp.sin(x)*sp.cos(x) + sp.cos(x)**2
yf=sp.factor(y)
print(yf)
(sin(x) + cos(x))**2

yf
```

$$\left(sin(x)+cos(x)\right)^2$$

```
y=x**9 - 11*x**8 + 40*x**7 - 32*x**6 - 110*x**5 + 178*x**4 +
96*x**3 - 216*x**2 - 27*x + 81

yf=sp.factor(y)
print(yf)
(x - 3)**4*(x - 1)**2*(x + 1)**3

yf
```

$$(x-3)^4(x-1)^2(x+1)^3$$

De la même façon, pour factoriser une fonction ou une expression à plusieurs variables, on utilise également la méthode *factor()* de *sympy*.

```
x,y,z=sp.symbols('x y z', Real=True)
z=x**2*y**2+y**3+y**2-x**2*y-y**2-y


z
```

$$x^2y^2 - x^2y + y^3 - y$$

```
zf=sp.factor(z)


zf
```

$$y(y-1)(x^2+y+1)$$

4.7 Simplification d'une expression

Pour simplifier une expression à une variable, on utilise la méthode *simplify()* de *sympy*.

```
y=(x**6 + x**4 - x**2 - 1)/(x**2+1)**2


y
```

$$\frac{x^6 + x^4 - x^2 - 1}{(x^2+1)^2}$$

```
ys=sp.simplify(y)


print(ys)
x**2 - 1
ys
```

$$x^2 - 1$$

```
y=sp.sin(x)**2 + sp.cos(x)**2


y
```

$$\sin^2(x) + \cos^2(x)$$

```
ys=sp.simplify(y)

print(ys)
1
```

De la même façon, pour simplifier une expression à plusieurs variables, on utilise également la méthode *factor()* de *sympy*.

```
z=(x**2*y**2 - x**2*y + y**2 - y)/(x**2+1)

z
```

$$\frac{x^2y^2 - x^2y + y^2 - y}{x^2 + 1}$$

```
zs=sp.factor(z)

zs
```

$$y(y - 1)$$

4.8 Séparation de variables

À l'instar de *simplify()*, on peut utiliser la méthode *separatevars()* de *sympy* pour factoriser une expression à plusieurs variables en séparant, lorsque cela est possible, les variables.

```
x,y,z=sp.symbols('x y z', Real=True)
z=x**2*y**2+2*x*y**2+y**2-x**2-2*x-1

z
```

$$x^2y^2 - x^2 + 2xy^2 - 2x + y^2 - 1$$

```
zs=sp.separatevars(z)

zs
```

$$(x + 1)^2(y - 1)(y + 1)$$

```
z=x**3+2*x**2+x**2*y+x+2*x*y+y

z
```

$$x^3 + x^2y + 2x^2 + 2xy + x + y$$

```
zs=sp.separatevars(z)

zs
```

$$(x + 1)^2(x + y)$$

4.9 Division polynomiale

Pour effectuer la division d'un polynôme *P(x)* par un polynôme *g(x)*, on utilise la méthode *div()* de *sympy* telle que :

- $x \in \mathbb{R}$;
- $g(x) \neq 0$;
- $P(x) = g(x)q(x) + r(x)$;
- où $g(x), q(x), r(x)$ *des polynômes et* $\deg(r) < \deg(g)$.

```
# Exemple 1
x,y=sp.symbols('x y', Real=True)

P=x**2-1
g=x-1

sp.div(P,g)
```

$$(x + 1, 0)$$

```
q=sp.div(P,g)[0]

q
```

$$x + 1$$

```
r=sp.div(P,g)[1]
```

0

```
# Exemple 2
P=x**2-x*y+y
g=x-y

sp.div(P,g)
```

$$(x, y)$$

```
q=sp.div(P,g)[0]

q
```

$$x$$

```
r=sp.div(P,g)[1]
```

$$y$$

```
# Exemple 3
P=2*x**5-3*x**3+x-1

P
```

$$2x^5 - 3x^3 + x - 1$$

```
g=x**2+1

g
```

$$x^2 + 1$$

```
sp.div(P,g)
```

$$(2x^3 - 5x, 6x - 1)$$

```
q=sp.div(P,g)[0]

q
```

$$2x^3 - 5x$$

```
r=sp.div(P,g)[1]
```

$$6x - 1$$

4.10 Définition de fonction

Sympy dispose également de méthodes permettant ainsi :

- la définition de fonction symbolique ;
- la transformation d'une expression en fonction ;
- l'évaluation d'une fonction ;
- etc.

4.10.1 Transformation d'une expression en fonction

La méthode *lambdify()* de *sympy* transforme une expression en fonction.

```
import sympy as sp
sp.init_printing(use_latex=True)
x, y =sp.symbols('x y', Real=True)
y=(sp.sqrt((x**4+1))/(x**2+1))

y
```

$$\frac{\sqrt{x^4 + 1}}{x^2 + 1}$$

```
f=sp.lambdify(x, y)
f(0)
```

1.0

```
round(f(1),2), round(f(-1),2)
```

(1.0, 1.0)

4.10.2 Définition de fonction symbolique

La méthode *Lambda()* de *sympy* permet la définition d'une fonction symbolique à une variable.

```
# Exemple :g function symbolique
x, y =sp.symbols('x y', Real=True)
y=x**4-sp.exp(x)
```

```
g=sp.Lambda(x, y)
g(x)
```

$$x^4 - e^x$$

```
g(-1), g(0), g(sp.sqrt(2))
```

$$\left(1 - e^{-1}, -1, 4 - e^{\sqrt{2}}\right)$$

De la même façon, la méthode *Lambda()* de *sympy* permet la définition d'une fonction symbolique à plusieurs variables.

```
x,y,z=sp.symbols('x y z', Real=True)
z=x**2+y**2+x*y
h=sp.Lambda((x,y),z)


h(x,y)
```

$$x^2 + xy + y^2$$

```
h(0,2), h(-2,2), h(-2,0)
```

(4, 4, 4)

4.11 Dérivée d'une fonction ou d'une expression

La méthode *Derivative()* de *sympy* permet d'écrire la dérivée nième symbolique d'une expression ou d'une fonction à une variable.

```
import sympy as sp
sp.init_printing(use_latex=True)
x,y=sp.symbols('x y', Real=True)
# Dérivée d'une expression
y=5*x**4 - x**3 +3*x**2 - x - 1


y
```

$$5x^4 - x^3 + 3x^2 - x - 1$$

```
sp.Derivative(y,x)
```

$$\frac{d}{dx}(5x^4 - x^3 + 3x^2 - x - 1)$$

```
sp.Derivative(y,x,2)
```

$$\frac{d^2}{dx^2}(5x^4 - x^3 + 3x^2 - x - 1)$$

```
# Dérivée d'une fonction
f=sp.Lambda(x, y)
sp.Derivative(f(x),x)
```

$$\frac{d}{dx}(5x^4 - x^3 + 3x^2 - x - 1)$$

```
sp.Derivative(f,x)
```

$$\frac{d}{dx}\left((x \mapsto 5x^4 - x^3 + 3x^2 - x - 1)\right)$$

La méthode *Derivative()* de *sympy* permet d'écrire la dérivée partielle nième et symbolique d'une expression ou d'une fonction à plusieurs variables.

```
# Dérivée d'une fonction
x,y,z=sp.symbols('x y z', Real=True)
z=x**2*y**2+y**3+y**2-x**2*y-y**2-y


z
```

$$x^2y^2 - x^2y + y^3 - y$$

```
sp.Derivative(z,x)
```

$$\frac{\partial}{\partial x}(x^2y^2 - x^2y + y^3 - y)$$

```
sp.Derivative(z,x,2)
```

$$\frac{\partial^2}{\partial x^2}(x^2y^2 - x^2y + y^3 - y)$$

En effet, le calcul de dérivées de fonctions alimente le calcul différentiel et permet ainsi l'étude de variations locales de fonction.

La méthode *diff()* de *sympy* permet de calculer la dérivée nième d'une expression ou d'une fonction à une variable.

Reprenons l'exemple de l'expression : $5x^4 - x^3 + 3x^2 - x - 1$

```
sp.diff(y, x)
```

$$20x^3 - 3x^2 + 6x - 1$$

```
sp.diff(y, x, 2)
```

$$6(10x^2 - x + 1))$$

La méthode *diff()* de *sympy* permet également de calculer la dérivée partielle nième d'une expression ou d'une fonction à plusieurs variables.

Reprenons l'exemple de l'expression : $x^2y^2 - x^2y + y^3 - y$

```
x, y, z = sp.symbols('x y z', Real=True)
z=x**2*y**2+y**3+y**2-x**2*y-y**2-y
sp.diff(z,x)
```

$$2xy^2 - 2xy$$

```
sp.diff(z,x,2)
```

$$2y(y - 1)$$

On utilise les mêmes méthodes pour le calcul de dérivées et dérivées partielles de fonctions.

```
h=sp.Lambda((x,y),z)
h(x, y)
```

$$x^2y^2 - x^2y + y^3 - y$$

```
sp.diff(h(x,y),x,2)
```

$$2y(y - 1)$$

4.12 Primitive et intégrale d'une fonction

Voici, ci-après quelques rappels, définitions et propriétés.

4.12.1 Définition de primitive

Soient *f* et *F* deux fonctions réelles définies toutes les deux sur I un intervalle ou une réunion d'intervalles de $\mathbb{R}$:

$f : I \rightarrow \mathbb{R}$
$F : I \rightarrow \mathbb{R}$

La fonction *F* est une fonction primitive de la fonction *f* si et seulement si *F* est dérivable sur I **et** $\forall x \in I : F'(x) = f(x)$.

Notation

$$F(x) = \int f(x)dx$$

4.12.2 Propriété

Si *F* est une primitive de *f* alors : $\forall\, c \in \mathbb{R} : F + c \; est\; une\; primitive\; de\; f$.

4.12.3 Définition intégrale

Soient :

- *f* une fonction continue et positive définie sur un intervalle fermé I = [a ; b] de $\mathbb{R}$;
- *F* une fonction primitive de *f*.

L'intégrale de *f* sur I, notée $\int_a^b f(x)dx$, est l'aire de la surface située ou délimitée par :

- la courbe *f* ;
- l'axe des abscisses x ;
- les deux droites (x = a) et x = b).

Cette aire est égale à :

$$\int_a^b f(x)dx = [F(x)]_a^b = F(b) - F(a)$$

Aire algébrique

Si *f* est négative sur I alors cette aire est égale à : $-\int_a^b f(x)dx$.

4.12.4 Méthode integrate()

La méthode *integrate()* de *sympy* permet d'écrire la primitive d'une fonction lorsqu'elle existe ou de calculer son intégrale sur un intervalle donné.

Syntaxe de calcul d'une primitive :

```
integrate(f, x)
```

Syntaxe de calcul d'une intégrale :

```
integrate(f, (x, a, b))
```

Exemple

```
import sympy as sp
sp.init_printing(use_latex=True)
x, y=sp.symbols('x y', Real=True)
y=(sp.exp(x))/(1 + x**2)

y
```

$$\frac{e^x}{1+x^2}$$

```
# Primitive
sp.integrate(y, x)
```

$$\int \frac{e^x}{1+x^2}\,dx$$

```
# Intégrale
sp.integrate(y, (x, 0, 4))
```

$$\int_0^4 \frac{e^x}{1+x^2}\,dx$$

```
# Fonction primitive
F=sp.lambdify(x,y)

# Interval [0; 4]
```

```
a, b = 0, 4
round(F(0), 2), round(F(4),2)
```

(1.0, 3.21)

```
# Calcul d'aire
aire=round(F(4)-F(0), 2)

aire
```

2.21

4.13 Limite d'une fonction

Pour calculer la limite d'une fonction *f* lorsque la variable *x* tend vers une valeur x_f, on utilise la méthode *limit()* de *sympy*.

$$\lim_{x \to x_f} f(x)$$

Exemples

```
import sympy as sp
sp.init_printing(use_latex=True)
x, y=sp.symbols('x y', Real=True)

# Exemple 1
y=(sp.exp(x))/(1 + x**2)

y
```

$$\frac{e^x}{1+x^2}$$

```
from math import *

sp.limit(y,x,inf)
```

∞

```
sp.limit(y,x,-inf)
```

0

```
# Exemple 2
y=(sp.sin(x)/(x-1))


y
```

$$\frac{\sin(x)}{x-1}$$

```
sp.limit(y, x, 1, dir='-')
```

$$-\infty$$

```
sp.limit(y, x, 1, dir='+')
```

$$\infty$$

```
# Exemple 3
y = (sp.exp(-x)) * ((x**3 -1) / (x**2))


y
```

$$\frac{(x^3-1)e^{-x}}{x^2}$$

```
sp.limit(y, x, 0)
```

$$-\infty$$

```
sp.limit(y, x, -inf)
```

$$-\infty$$

```
sp.limit(y, x, +inf)
```

0

4.14 Résolution d'équations

Les méthodes *solve* de *sympy* permettent la résolution d'une équation ou d'un système d'équations :

- algébriques où chaque terme est un polynôme ;
- équations différentielles ;

- équations linéaires en écriture matricielle ;
- etc.

4.14.1 Résolution d'équations algébriques

Pour résoudre une équation algébrique de type *y=P(x)=0* où *P(x)* est un polynôme, on utilise la méthode *solve()* de *sympy*. On pourra spécifier lors de la déclaration des variables symboliques si on cherche des racines complexes ou uniquement des racines réelles.

```
import sympy as sp
sp.init_printing(use_latex=True)
x, y = sp.symbols('x y', Real=True)

# Définition de l'expression y
y = -2*x**2 + (2+sp.sqrt(2))*x -sp.sqrt(2)

y
```

$$-2x^2 + x\left(\sqrt{x} + 2\right) - \sqrt{2}$$

```
# s est la liste des racines réelles, si elles existent, de l'équation y = 0
s=sp.solve(y, x)

s
```

$$\left[1, \sqrt{2}/2\right]$$

```
s[0]
```

1

```
s[1]
```

$$\sqrt{2}/2$$

Si on cherche des racines complexes, on affecte la valeur booléenne *True* à l'option *complex* dans la méthode *symbols* :

```
x, y = sp.symbols('x y', complex=True)
y = 2*x**2 +1
```

```
s=sp.solve(y, x)

y
```

$$2x^2 + 1$$

```
s=sp.solve(y, x)

s
```

$$[-\sqrt{2}i/2, \sqrt{2}i/2,]$$

Si toutefois on cherche que des racines réelles si elles existent, on affecte la valeur booléenne *False* à l'option *complex* dans la méthode *symbols* :

```
# Même équation que la précédente
x, y = sp.symbols('x y', complex=False)
y = 2*x**2 +1
s=sp.solve(y, x)

s
```

[ ]

Remarque : on remarque ici que la liste des racines réelles est vide [ ].

De la même façon, pour résoudre un système d'équations algébriques à plusieurs variables, par exemple pour deux variables x et y :

$$P_1(x, y) = 0$$
$$P_2(x, y) = 0$$
$$\ldots$$

où P(x, y) sont des polynômes à deux indéterminées x et y, on utilise la méthode *solve()* de *sympy*. On pourra spécifier lors de la déclaration des variables symboliques si on cherche des racines complexes ou uniquement des racines réelles.

```
# Exemple 1
x, y, p1, p2 = sp.symbols('x y p1 p2', complex=True)
```

```
p1 = 2*x + y +1
p2 = -x + 3*y

p1, p2
```

$$(2x + y + 1, -x + 3y)$$

```
d=sp.solve([p1, p2], dict=True)

d
```

$$[\{x: -3/7\,, y: -1/7\}]$$

```
# Exemple 2
x, y, p1, p2 = sp.symbols('x y p1 p2', complex=True)
p1 = x**2 -x*y + x
p2 = 4*(x**2)*y + 2*x*y + y

p1
```

$$x^2 - xy + x$$

```
p2
```

$$4x^2y + 2xy + y$$

```
d=sp.solve([p1, p2], dict=True)

d
```

$$\left[\begin{array}{c}\{x: -1, y: 0\}, \{x: 0, y: 0\}, \\ \left\{x: \frac{\sqrt{3} + 3i}{2\left(\sqrt{3} - 3i\right)}, y: \frac{3}{4} + \frac{\sqrt{3}i}{4}\right\}, \left\{x: \frac{\sqrt{3} - 3i}{2\left(\sqrt{3} + 3i\right)}, y: \frac{3}{4} - \frac{\sqrt{3}i}{4}\right\}\end{array}\right]$$

4.14.2 Résolution d'équations différentielles

Définition

Soient :

- I : un intervalle ou une réunion d'intervalles de $\mathbb{R}$;

- $x \in \mathbb{R}$;
- y une fonction de x, n fois dérivable sur I telles que $y', y'', \ldots, y^{(n)}$sont ses n dérivées successives.

Une équation qui met en relation $x, y', y'', \ldots, y^{(n)}$ est une équation différentielle (ED) sur I d'ordre n. La résolution d'une équation différentielle (ED) consiste à trouver toutes les fonctions y qui vérifient cette équation.

En mathématiques, il existe plusieurs formes d'équations différentielles (ED) telles que les (ED) d'ordre 1, d'ordre 2, ordinaires, linéaires, non linéaires, etc.

Dans la suite de ce paragraphe, on va résoudre quelques équations différentielles ordinaires (ODE).

Exemples de (ODE)

- $y'(x) + 2y(x) = -\pi e^x$;
- $xy'(x) - y(x) = \frac{\pi}{2}\sin(x)$;
- $2y''(x) + y'(x) - y(x) = 0$;
- etc.

Pour l'implémentation Python, on travaillera sur l'ODE suivante avec conditions initiales (IC).

La résolution passe par plusieurs étapes :

- création des variables symboliques ;
- création d'une fonction objet ;
- création de l'équation différentielle ;
- résolution sans les conditions initiales ;
- instanciation des constantes ;
- création d'un dictionnaire des conditions initiales (ics) ;
- résolution avec les conditions initiales ;
- vérification des valeurs initiales du problème (ivp).

Exemple 1

$$4xy''(x) + y'(x) = x^2 - 1$$
$$y(1) = 1 \; et \; y'(0) = -1$$

```
import sympy as sp
sp.init_printing(use_latex=True)
x=sp.symbols('x')
```

```
# Fonction objet : y()
y=sp.Function("y")(x)

y
```

$$y(x)$$

```
# L'équation différentielle
diff_eq=sp.Eq(4*x*y.diff(x,x)+y.diff(x),x**2-1)

diff_eq
```

$$4x\frac{d^2}{dx^2}y(x)+\frac{d}{dx}y(x)=x^2-1$$

```
# Résolution sans les conditions initiales
s=sp.dsolve(diff_eq,y)

s
```

$$y(x)=C_1+C_2x^{3/4}+x^3/27-x$$

```
# Le terme de droite
td=s.rhs

td
```

$$C_1+C_2x^{3/4}+x^3/27-x$$

```
# tuple des symboles de l'ODE C1, C2 et x
tuple(td.free_symbols)
```

$$(C_1,C_2,x)$$

```
# Instanciation des constantes
C1, C2, _ = tuple(td.free_symbols)
td.subs(C1,0).subs(C2,1)
```

$$x^{3/4}+\frac{x^3}{27}-x$$

```
# Dictionnaire des conditions initiales (ics)
ics={y.subs(x,1):1,y.diff().subs(x,0):-1}

# Résolution avec les conditions initiales (ivp)
ivp=sp.dsolve(diff_eq, ics=ics)

ivp
```

$$y(x) = \frac{x^3}{27} - x + \frac{53}{27}$$

```
# Vérification des valeurs initiales du problème (ivp)
ivp.subs(x,1)
```

$$y(1) = 1$$

```
ivp.rhs.diff().subs(x,0)
```

-1

```
(4*x*ivp.rhs.diff(x,x)+ivp.rhs.diff()).simplify()
```

$$x^2 - 1$$

Exemple 2

Dans cet exemple, on considère l'équation différentielle qui modélise la variation de température, dans le temps, d'un liquide dans un milieu à température ambiante constante. La modélisation est basée sur le modèle de Newton. On a l'hypothèse de l'expérimentation suivante :

- ta : température ambiante, ta = 25 °C ;
- t : temps en minutes ;
- y(t) : température du liquide à l'instant t ;
- y(0) : condition initiale de l'équation différentielle, y(0) = 5 °C ;
- cp : coefficient de proportionnalité de l'équation différentielle, cp = 0,02.

Le modèle de l'équation différentielle est suivant :

$$y'(t) = cp\big(ta - y(t)\big)$$

```
import sympy as sp
sp.init_printing(use_latex=True)
t = sp.symbols('t')
cp=0.02
ta=25
y = sp.symbols('y',cls=sp.Function)
diff_eq = sp.Eq(sp.Derivative(y(t),t),-cp*y(t)+cp*ta)
ics = {y(0):5}

diff_eq
```

$$\frac{d}{dt}y(t) = 0.5 - 0.02y(t)$$

```
ivp=sp.dsolve(diff_eq,ics=ics)

ivp
```

$$y(t) = 25.0 - 20.0e^{-0.02t}$$

```
# Vérification de la condition initiale
ivp.subs(t,0)
```

$$y(0) = 5.0$$

4.15 Résolution d'équations linéaires en écriture matricielle

La librairie *sympy* dispose aussi de méthodes permettant le calcul matriciel telles que la transposée, la somme, le produit, le déterminant, etc. Cependant, pour le calcul matriciel, on préfère utiliser les outils de la librairie *numpy* plus adaptée à la manipulation de matrices numériques.

Dans ce paragraphe, on aborde la résolution d'une équation matricielle sous forme A.X = B où :

- A : est une matrice symbolique, c'est-à-dire qui contient au moins un symbole, d'ordre n ($n \in \mathbb{N}^*$) ;
- X : vecteur colonne qui contient les inconnues de l'équation matricielle ;
- Y : vecteur colonne qui contient les constantes de l'équation matricielle, c'est le 2ème terme.

Notons qu'une équation matricielle, abordée ici, correspond à un système de n équations linéaires et à n inconnues.

On utilisera le constructeur *Matrix()* de la sous-librairie *sympy.matrices* pour définir une matrice symbolique.

Pour résoudre l'équation matricielle, on utilisera la méthode *solve()* qui est basée sur la méthode mathématique *Gauss-Jordan* ou bien par *LUsolve()* également. Si la matrice A n'est pas une matrice carrée alors *sympy* renvoie un message d'erreur.

Construction d'une matrice symbolique

```
# Exemple 1 : matrice symbolique d'ordre 2
import sympy as sp
from sympy import init_printing
init_printing(use_unicode=True)
import sympy.matrices as spm
a, y, z = sp.symbols('x y z')
# Construction de la matrice symbolique M2
M2=spm.Matrix([[x, sp.pi], [y, z]])

M2
```

$$\begin{bmatrix} x & \pi \\ y & z \end{bmatrix}$$

```
# Exemple 2 : matrice symbolique d'ordre 3
M=sp.MatrixSymbol('m',3,3)
M=spm.Matrix(M)

M
```

$$\begin{bmatrix} m_{0\,0} & m_{0\,1} & m_{0\,2} \\ m_{1\,0} & m_{1\,1} & m_{1\,2} \\ m_{2\,0} & m_{2\,1} & m_{2\,2} \end{bmatrix}$$

```
# Exemple d'opérations sur matrice symbolique
# Transposée
M.T
```

$$\begin{bmatrix} m_{0\,0} & m_{1\,0} & m_{2\,0} \\ m_{0\,1} & m_{1\,1} & m_{2\,1} \\ m_{0\,2} & m_{1\,2} & m_{2\,2} \end{bmatrix}$$

```
import numpy as np

# Affectation d'un tableau à une matrice sympy
M=sp.MatrixSymbol('m',3,3)
M=spm.Matrix(M)
S=np.arange(9)
S.shape=(3,3)
M=spm.Matrix(S)

M
```

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Exemple 1 : résolution d'une équation matricielle avec la méthode *solve()*

La méthode *solve()* permet d'obtenir une solution, lorsqu'elle existe, sous forme de matrice.

Prenons comme exemple le système d'équations linéaires suivant avec 3 inconnues *x, y, z* et 3 équations :

$$\begin{cases} 3x & -y & +az & = & 15 \\ x & +2y & -z & = & -3 \\ ax & +y & -3z & = & 10 \end{cases}$$

où : $a, x, y, z \in \mathbb{R}$ et a est un paramètre du système.

L'écriture matricielle de ce système est la suivante :

$$A.X = B$$

où :

$$A = \begin{bmatrix} 3 & -1 & a \\ 1 & 2 & -1 \\ a & 1 & -3 \end{bmatrix}$$

$$X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

et

$$B = \begin{bmatrix} 15 \\ -3 \\ 10 \end{bmatrix}$$

```
# Résolution
import sympy as sp
from sympy import init_printing
init_printing(use_unicode=True)
import sympy.matrices as spm
a, x, y, z = sp.symbols('a x y z')
A=spm.Matrix([[3,-1,a],[1,2,-1],[a,1,-3]])


A
```

$$\begin{bmatrix} 3 & -1 & a \\ 1 & 2 & -1 \\ a & 1 & -3 \end{bmatrix}$$

```
B=spm.Matrix([15,-3,10])


B
```

$$\begin{bmatrix} 15 \\ -3 \\ 10 \end{bmatrix}$$

```
X=A.solve(B)


X
```

$$\begin{bmatrix} \frac{23a + 56}{2a^2 - 2a + 18} \\ \frac{-3a^2 + 5a - 102}{2a^2 - 2a + 18} \\ \frac{27a - 94}{2a^2 - 2a + 18} \end{bmatrix}$$

Si le paramètre $a = 1$ alors la solution, lorsqu'elle existe, est :

```
X=X.subs(a,1)


X
```

$$\begin{bmatrix} \frac{79}{18} \\ \frac{-50}{9} \\ \frac{67}{18} \end{bmatrix}$$

Exemple 2 : résolution d'une équation matricielle avec la méthode *LUsolve()*

La méthode *LUsolve()* permet également d'obtenir une solution, lorsqu'elle existe, d'un système d'équations linéaires.

Prenons comme exemple le système d'équations linéaires suivant avec 3 inconnues *x, y, z* et 3 équations :

$$\begin{cases} 3x & -y & +z & = & 5 \\ x & +2y & 2z & = & -3 \\ ax & +ay & 2az & = & 3 \end{cases}$$

où : $a, x, y, z \in \mathbb{R}$ et a un paramètre du système.

L'écriture matricielle de ce système est la suivante :

$$A.X = B$$

où :

$$A = \begin{bmatrix} 3 & -1 & 1 \\ 1 & 2 & 2 \\ a & a & 2a \end{bmatrix}$$

$$X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

et

$$B = \begin{bmatrix} 5 \\ -3 \\ 3 \end{bmatrix}$$

```
# Résolution
import sympy as sp
from sympy import init_printing
init_printing(use_unicode=True)
import sympy.matrices as spm
a, x, y, z = sp.symbols('a x y z')
```

```
A=spm.Matrix([[3,-1,1],[1,2,2],[a,a,2*a]])

A
```

$$\begin{bmatrix} 3 & -1 & 1 \\ 1 & 2 & 2 \\ a & a & 2a \end{bmatrix}$$

```
B=spm.Matrix([5,-3,3])

B
```

$$\begin{bmatrix} 5 \\ -3 \\ 3 \end{bmatrix}$$

```
X=A.LUsolve(B)

X
```

$$\begin{bmatrix} 1-\frac{4(a+3)}{5a} \\ -2-\frac{a+3}{a} \\ \frac{7(a+3)}{5a} \end{bmatrix}$$

Si le paramètre $a = 1$ alors la solution, lorsqu'elle existe, est :

```
X=X.subs(a,1)

X
```

$$\begin{bmatrix} -11/5 \\ -6 \\ 28/5 \end{bmatrix}$$

Exemple 3 : résolution d'une équation matricielle avec la méthode *LUsolve()*

Prenons comme exemple le système d'équations linéaires suivant avec 3 inconnues *x, y, z* et 3 équations.

$$\begin{cases} 3x & -y & 6z & = & 15 \\ x & +2y & 2z & = & -3 \\ ax & +ay & 2az & = & 6 \end{cases}$$

où : $a, x, y, z \in \mathbb{R}$ et a un paramètre du système.

L'écriture matricielle de ce système est la suivante :

$$A.X = B$$

où :

$$A = \begin{bmatrix} 3 & -1 & 6 \\ 1 & 2 & 2 \\ a & a & 2a \end{bmatrix}$$

$$X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

et

$$B = \begin{bmatrix} 15 \\ -3 \\ 6 \end{bmatrix}$$

```
# Résolution
import sympy as sp
from sympy import init_printing
init_printing(use_unicode=True)
import sympy.matrices as spm
a, x, y, z = sp.symbols('a x y z')
A=spm.Matrix([[3,-1,6],[1,2,2],[a,a,2*a]])

A
```

$$\begin{bmatrix} 3 & -1 & 6 \\ 1 & 2 & 2 \\ a & a & 2a \end{bmatrix}$$

```
B=spm.Matrix([15,-3,6])

B
```

$$\begin{bmatrix} 15 \\ -3 \\ 6 \end{bmatrix}$$

```
X=A.LUsolve(B)

X
```

$$\begin{bmatrix} 1 - \frac{4(a+3)}{5a} \\ -2 - \frac{a+3}{a} \\ \frac{7(a+3)}{5a} \end{bmatrix}$$

Si le paramètre *a* = 1 alors la solution, lorsqu'elle existe, est :

```
X=X.subs(a,1)

# Message d'erreur affiché
NonInvertibleMatrixError: Matrix det == 0; not invertible
```

La matrice A est non inversible. En effet, le déterminant de A est égal à 0 :

```
A.det()
```

0

Chapitre 5

Éléments de statistiques descriptives

Ici on abordera des méthodes Python permettant l'utilisation des éléments pratiques en termes de calcul statistique à une ou plusieurs variables, comme la moyenne, l'écart-type, la variance, la corrélation, etc. Plusieurs éléments de ce chapitre sont basés sur des références dans le domaine comme *Statistiques descriptives* de Joseph Salmon (cf. Bibliographie).

1 Définitions

Voici, ci-dessous quelques définitions de terminologie. Pour les calculs théoriques et formulations, on conseille quelques références dans la bibliographie de cet ouvrage.

1.1 Population

Une population est un ensemble d'éléments appelés aussi individus. Dans une population, chaque individu est unique et a donc une donnée correspondant à un identifiant.

Exemple de population : l'ensemble des étudiants d'une école d'ingénieurs.

1.2 Échantillon

Un échantillon est un sous-ensemble, au sens large, d'une population donnée. Le but dans une étude statistique est de faire des analyses sur un échantillon représentatif d'une population donnée.

Exemple d'échantillon : les étudiants en M1 d'une école d'ingénieurs.

1.3 Variable statistique

Une variable statistique, dite aussi attribut en bases de données relationnelles, permet de décrire un caractère d'un individu. L'ensemble des valeurs d'une variable constitue donc les modalités. Une valeur correspond à une observation dans le cadre d'une expérimentation ou d'un relevé ou d'une étude. Une variable peut être quantitative ou qualitative, selon l'organigramme ci-après.

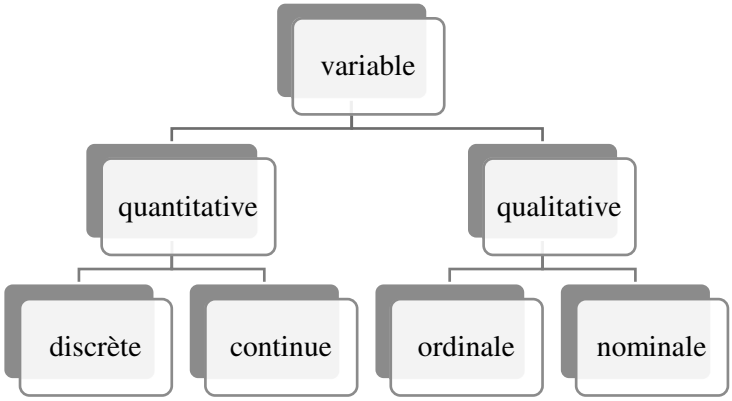
1.4 Variable quantitative

Une variable quantitative, dite également variable numérique, est une variable statistique dont les valeurs sont de type numérique. Une variable quantitative décrit ainsi une caractéristique quantifiable d'un individu.

1.4.1 Variable quantitative continue

Une variable quantitative continue est une variable quantitative dont l'ensemble des valeurs est un intervalle donné qui contient une infinité de valeurs réelles possibles.

Exemple : le volume d'eau récupéré par captation de la rosée dans un contenant d'un litre maximum. Ce volume peut, théoriquement prendre une infinité de valeurs dans l'intervalle [0 ; 1] litres.



1.4.2 Variable quantitative discrète

Une variable quantitative discrète est une variable quantitative dont l'ensemble des valeurs est un ensemble fini de réelles.

Exemple : le nombre de frères et sœurs d'un étudiant, qui sont inscrits en études supérieures. Ce nombre est un entier positif appartenant à l'intervalle [0... 12] où l'on suppose que 12 est théoriquement le nombre maximum de personnes dans la même fratrie.

1.5 Variable qualitative

Une variable qualitative, dite aussi variable catégorielle ou encore facteur, est une variable statistique dont les valeurs ne sont pas de type numérique. Ces valeurs

décrivent une qualité de chaque individu dans un langage alphanumérique où les valeurs sont des mots du langage et où l'ensemble des valeurs est un ensemble fini.

1.5.1 Variable qualitative ordinale

Une variable qualitative ordinale est une variable qualitative tel que l'ensemble des valeurs est fini, est strictement ordonné et la relation d'ordre est significative dans une étude statistique.

Exemple, la mention obtenue par un étudiant en M1 en fin d'année.

1.5.2 Variable qualitative nominale

Une variable qualitative nominale est une variable qualitative tel que l'ensemble des valeurs est fini et où ces valeurs ne sont régies par aucune relation d'ordre.

Exemple : le sexe d'un individu.

1.6 Statistiques descriptives

Les statistiques descriptives offrent des outils pour l'organisation, la description et la représentation d'ensembles de données.

Lorsque la description se base sur une seule variable statistique, il s'agit alors de statistiques descriptives **univariées**.

Lorsque la description se base sur deux variables statistiques simultanément, il s'agit alors de statistiques descriptives **bivariées**.

De façon générale, lorsque la description se base sur plusieurs variables statistiques simultanément, il s'agit alors de statistiques descriptives **multivariées**.

2 Mesures

Voici, ci-dessous une liste non exhaustive de mesures en statistiques descriptives. Certaines sont abordées dans les implémentations Python suivantes.

Dans la suite, on désigne par :

- x_i : la valeur de la i^{ème} observation ;
- n : nombre d'observations ;
- x_{max} : valeur maximale ;
- x_{min} : valeur minimale ;
- $x_{(k)}$: k^{ième} valeur dans l'ensemble des valeurs trié ;
- Q_1 : 1er quartile ;
- Q_3 : 3^{ième} quartile ;
- moyenne arithmétique des données : $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$;

- mesure de la dispersion des données dite variance : $\sigma^2 = \frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}$;
- l'écart-type : $\sigma = \sqrt{\sigma^2}$;
- l'écart entre le 3ème et le 1er quartile, dit aussi l'écart l'interquartile ou encore l'étendue interquartile :
  $IQR = Q_3 - Q_1$;
- valeur centrale dite médiane des données où l'ensemble des données est trié :
  $$Me = \begin{cases} x_{(\frac{n}{2})} & si\ n\ est\ impair \\ \frac{x_{(\frac{n}{2})}+x_{((\frac{n}{2})+1)}}{2} & si\ n\ est\ pair \end{cases} ;$$
- etc.

3 Implémentation

Ici on implémentera des exemples qui illustrent certaines de ces mesures et d'autres mesures à deux variables. Pour les données, on utilisera un jeu de données téléchargé en *open data* sur des mesures de santé.

3.1 Moyenne, variance et écart-type

3.1.1 Script

```
import os
import numpy as np
import statistics as st

# Nombres d'observations
def NData(lv):
    t=len(lv)
    return(t)

# Fonction moyenne d'une liste de valeurs numériques lv
def MoyArith(lv):
    m=(1/len(lv))*sum(lv)
    return m

# Fonction variance
def Variance(lv):
```

```
    m=MoyArith(lv)
    var=0
    n=len(lv)
    for v in lv:
        var += (v-m)**2
    var=(1/n)*var
    return(var)

# Fonction Ecart_Type
def Ecart_Type(lv):
    et=Variance(lv)**(1/2)
    return(et)

# Charger les données dans mesure
os.chdir("C:\\Users\\a.fadil\\OneDrive - Groupe-ESA\\Bureau\\Bureau
Courant\\F-à-F 2022-2023\\IL2\Exemples Maths AL")

mesure= np.loadtxt("mesure.txt",delimiter="\t",dtype=float)
print("Les mesures : ", mesure)
n=NData(mesure)
print("n = ",n,"\n")
# mean() pour la moyenne arithmétique
m1=round(MoyArith(mesure),2)
m2=round(st.mean(mesure),2)
print("Moyenne programmée = ",m1)
print("Moyenne prédéfinie dans le module statistics =",m2)
print("Egalité ? : ",m1==m2,"\n")
# pvariance() pour une population et variance() pour un échantillon
v1=round(Variance(mesure),2)
print("Variance programmée = ",v1)
v2=round(st.pvariance(mesure),2)
print("Variance prédéfinie dans le module statistics =",v2)
print("Egalité ? : ",v1==v2,"\n")
# pstdev() pour ecart-type population et stdev() pour un échantillon
et1=round(Ecart_Type(mesure),2)
```

```
et2=round(st.pstdev(mesure),2)
print("Ecart-type programmé = ",et1)
print("Ecart-type prédéfini dans le module statistics =",et2)
print("Egalité ? : ",et1==et2)
```

3.1.2 Résultat

```
Les mesures :  [ 0. 10. 20. 15.  5. 13.  7.  0. 20. 18. 16.  4.  2.]
n =  13

Moyenne programmée =  10.0
Moyenne prédéfinie dans le module statistics = 10.0
Egalité ? :  True

Variance programmée =  51.38
Variance prédéfinie dans le module statistics = 51.38
Egalité ? :  True

Ecart-type programmé =  7.17
Ecart-type prédéfini dans le module statistics = 7.17
Egalité ? :  True
```

3.2 Résumé statistique et matrice de corrélation

3.2.1 Script

```
# Importation des librairies
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Charger le fichier de données sur le même répertoire que le script
df=pd.read_csv("Medical_Cost.csv",delimiter=',')

# Synthèse
taille=df.shape
```

```
print("Taille : ", taille)
entetes=df.columns
print("En-têtes : ", entetes)
print("Récapitulatif des infos ")
df.info()
print("\n")
print("Les 3 1ères lignes")
print(df.head(3).T)
print("\n")
print("Les 5 dernières lignes")
print(df.tail(5).T)
print("\n")
print("Combien de valeurs nulles par attribut ?")
print(df.isna().sum())
print("\n")

print("Résumé statistique : (count, mean, std, min, 25%, 50%, 75%,
max)")
res=np.zeros((4,8))
res=df.describe().T
print(df.describe().T)
print(res)
print("Nombre de valeurs par attribut : ")
print(df.sex.value_counts())
f=df.sex.value_counts()[0]
print("Filles = ",f)
g=df.sex.value_counts()[1]
print("Garçons = ",g,"\n")
print("\n")

# Graphique : boite à moustache
# La fonction subplots() pour afficher ou superposer plusieurs tracés sur
le même graphique
print("Graphique : boite à moustache \n")
fig, axes = plt.subplots(1,1)
```

```
sns.boxplot(df['age'])
plt.title('Boites à moustache')
plt.show()

# Nuage de points : effet de l'âge
print("Nuage de points : effet de l'âge")
sns.relplot(x='age', y='charges', hue= 'sex', data=df, palette='husl')
plt.title("Effect de l'âge sur la charge")
plt.show()

# Histogramme des âges
print("Histogramme des âges")
sns.histplot(data=df,x='age')
plt.title("Histogramme des âges")
plt.show()

# Histogramme des charges
print("Histogramme des charges")
plt.hist(df['charges'])
plt.title("Histogramme des charges")
plt.show()

# Etude de corrélation
print("Heatmap corrélation")
plt.figure(figsize=(8,5))
corr=df.corr()
axe=sns.heatmap(corr,vmin=-1,vmax=1,center=0,annot=True)
plt.title("Matrice de corrélation")
plt.show()
```

3.2.2 Résultat

```
Taille :  (1338, 7)

En-têtes :  Index(['age', 'sex', 'bmi', 'children', 'smoker', 'region',
'charges'], dtype='object')
```

Récapitulatif des infos
RangeIndex: 1338 entries, 0 to 1337

Data columns (total 7 columns):

#	Column	Non-Null Count	Dtype
0	age	1338 non-null	int64
1	sex	1338 non-null	object
2	bmi	1338 non-null	float64
3	children	1338 non-null	int64
4	smoker	1338 non-null	object
5	region	1338 non-null	object
6	charges	1338 non-null	float64

dtypes: float64(2), int64(2), object(3)

Les 3 1ères lignes

	0	1	2
age	19	18	28
sex	female	male	male
bmi	27.9	33.77	33.0
children	0	1	3
smoker	yes	no	no
region	southwest	southeast	southeast
charges	16884.924	1725.5523	4449.462

Les 5 dernières lignes

	1333	1334	1335	1336	1337
age	50	18	18	21	61
sex	male	female	female	female	female
bmi	30.97	31.92	36.85	25.8	29.07
children	3	0	0	0	0
smoker	no	no	no	no	yes
region	northwest	northeast	southeast	southwest	northwest
charges	10600.5483	2205.9808	1629.8335	2007.945	29141.3603

Combien de valeurs nulles par attribut ?

age	0
sex	0
bmi	0
children	0
smoker	0
region	0
charges	0

dtype: int64

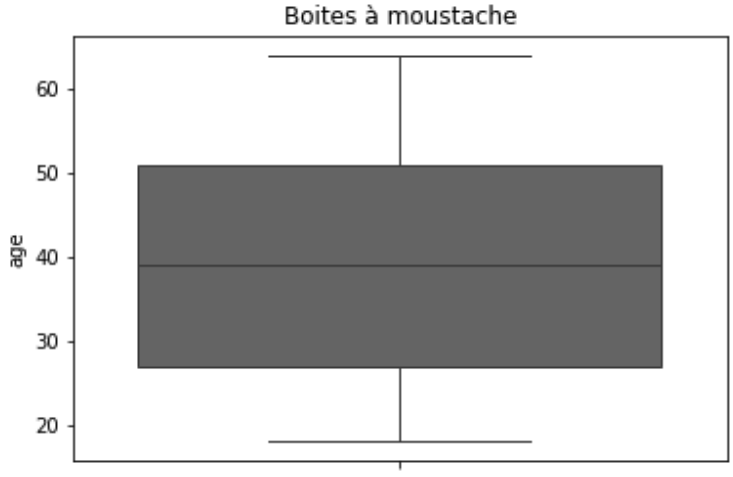
Résumé statistique : (count, mean, std, min, 25%, 50%, 75%, max)

```
            count          mean  ...           75%          max
age        1338.0     39.207025  ...     51.000000     64.00000
bmi        1338.0     30.663397  ...     34.693750     53.13000
children   1338.0      1.094918  ...      2.000000      5.00000
charges    1338.0  13270.422265  ...  16639.912515  63770.42801

[4 rows x 8 columns]
```

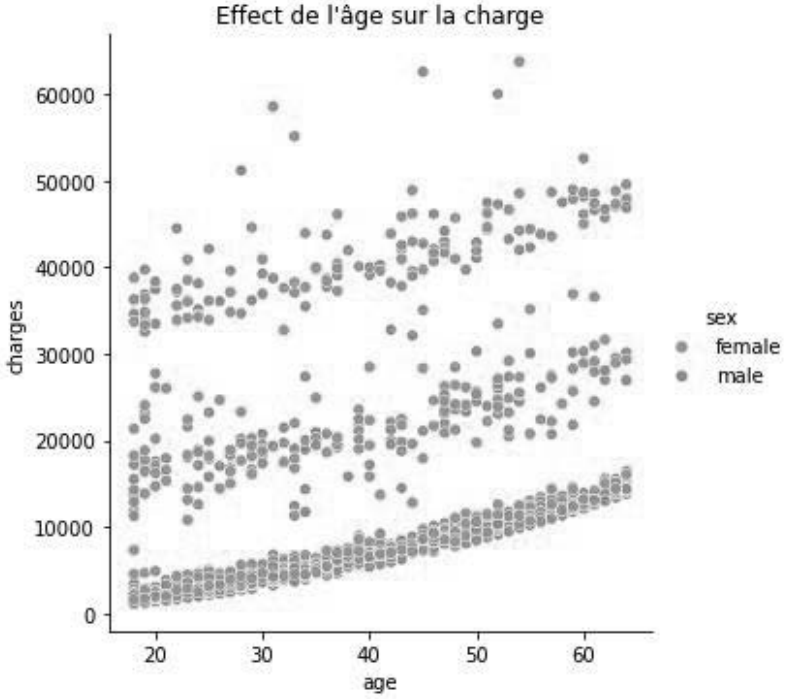

```
            count          mean  ...           75%          max
age        1338.0     39.207025  ...     51.000000     64.00000
bmi        1338.0     30.663397  ...     34.693750     53.13000
children   1338.0      1.094918  ...      2.000000      5.00000
charges    1338.0  13270.422265  ...  16639.912515  63770.42801

[4 rows x 8 columns]
```


```
Nombre de valeurs par attribut :
male      676
female    662
Name: sex, dtype: int64
Filles =  676
Garçons =  662
```

Graphique : boite à moustache

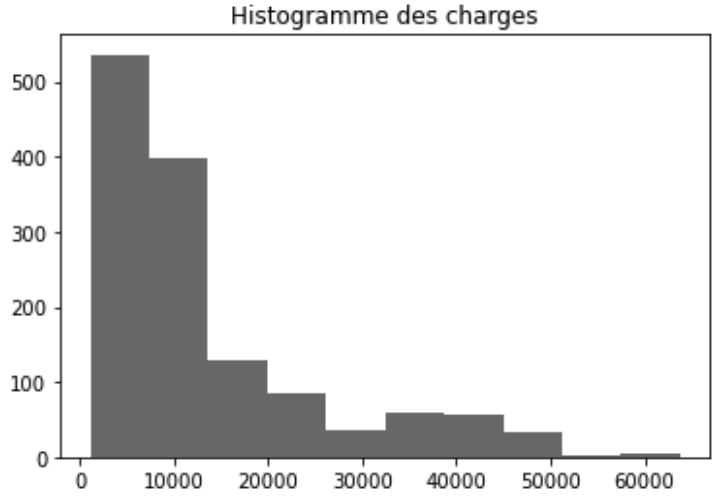


Nuage de points : effet de l'âge

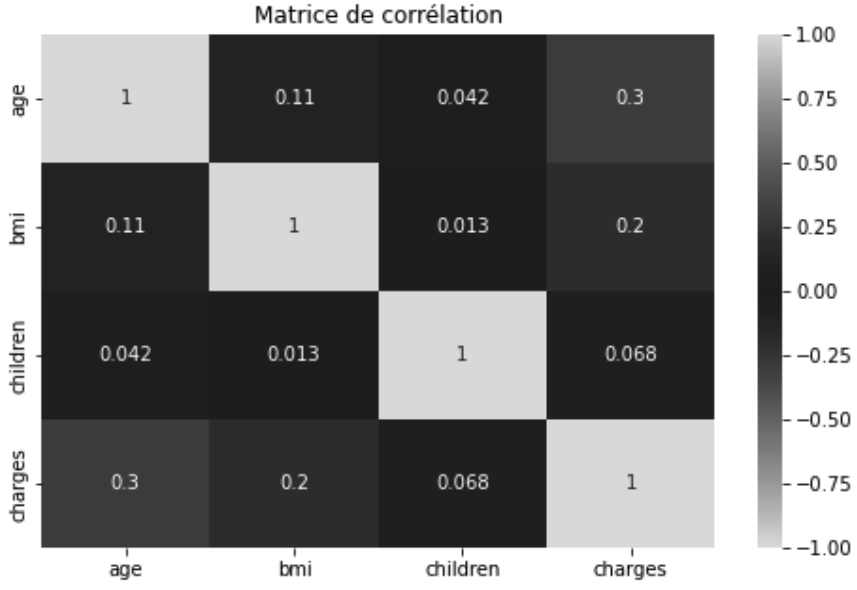


Histogramme des âges

Histogramme des charges



Heatmap corrélation



Chapitre 6

Éléments d'algèbre linéaire

1 Introduction

L'algèbre linéaire fait partie des outils et méthodes mathématiques. Elle permet d'appréhender plusieurs situations réelles et de les modéliser. Le point commun à tous les modèles issus de l'algèbre linéaire est l'abstraction d'une correspondance ou application linéaire entre d'un côté les paramètres, variables et constantes de base, c'est-à-dire du modèle, et de l'autre côté les variables de sortie représentant le résultat escompté, le résultat d'un traitement ou encore l'objectif réalisable dans le cadre d'un modèle prédictif.

Les objets formels les plus utilisés en algèbre linéaire pour la représentation des applications linéaires sont « les matrices ». Un modèle matriciel peut être une représentation en une (1D) ou deux (2D) ou plusieurs dimensions.

Le modèle matriciel est utilisé dans plusieurs domaines scientifiques pour l'abstraction et la modélisation des situations étudiées :

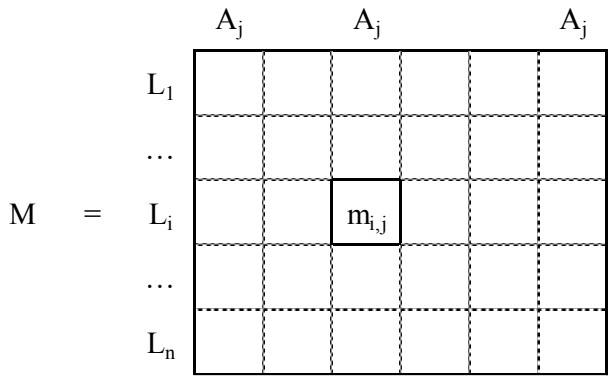
- modèle de LEONTIEF en économie (prix Nobel d'économie en 1973) ;
- modèle d'évolution d'une population en démographie et matrice de Leslie ;
- matrice de diffusion en physique des particules ;
- matrice tarifaire en gestion des flux de transport ;
- matrices des coûts en optimisation dans la production ;
- matrices progressives en psychologie ;
- factorisation matricielle pour la génomique des populations ;
- matrices dans l'informatique quantique ;
- etc.

2 Matrice des liaisons en trafic urbain

Ci-après un exemple de modélisation linéaire, dans le domaine du transport, en utilisant une matrice des liaisons. Supposons que dans un réseau de transport urbain, il y a n lignes de transport et m arrêts desservis par des bus urbains. On souhaite modéliser les lignes et les arrêts desservis à l'aide d'une matrice des liaisons $M = (m_{i,j})_{n,m}$ telle que :

$$m_{i,j} = \begin{cases} 1 \; si \; l'arrêt \; j \; est \; deservis \; par \; la \; ligne \; i \\ 0 \; sinon \end{cases}$$

On obtient la matrice des liaisons suivante :



Voici, ci-dessous quelques notions et rappels mathématiques avant d'aborder l'utilisation de la librairie *numpy* dans le calcul matriciel.

3 Calcul matriciel

Dans la suite de ce titre, on donne de façon non exhaustive quelques définitions, théorèmes et propriétés concernant le calcul matriciel en 2D dans le corps des réels.

3.1 Définition matrice

Soit A une matrice de taille (n, m). A est un tableau de réels $a_{i,j}$ où :

- n : nombre de lignes, $n \in \mathbb{N}^*$;
- m : nombre de colonnes, $m \in \mathbb{N}^*$;
- i : numéro (ou indice) de lignes, i = 1, 2, …, n. Pour rappel, les indices sous Python commencent à 0 : i = 0, 1, …, n − 1 ;
- j : numéro (ou indice) de colonnes, j = 1, 2, …, m. Pour rappel, les indices sous Python commencent à 0 : j = 0, 1, …, m − 1 ;
- $a_{i,j}$: les $a_{i,j}$ sont les coefficients de la matrice . Un coefficient $a_{i,j}$ est la valeur correspondant à l'intersection de la ligne i et la colonne j, $a_{i,j} \in \mathbb{R}$.

Notation

A : matrice de taille (n, m) :

$$A = A_{n,m} = \left(a_{i,j}\right)_{n,m}$$

où i = 1, 2, …, n et j = 1, 2, …, m. Pour rappel, les indices sous Python commencent à 0 : i = 0, 1, …, n − 1 et : j = 0, 1, …, m – 1.

3.2 Exemple

Dans un contexte de production manufacturière, on dispose de 3 machines multi-tâches et de 3 travaux différents. Les lignes correspondent aux travaux et les colonnes correspondent aux machines. La matrice *A* représente les coûts de production en €. Ici n = 3 et m = 3. Il s'agit d'une matrice carrée (n = m).

$$A = \begin{bmatrix} 2 & 5 & 1 \\ 1 & 1 & 3 \\ 3 & 2 & 5 \end{bmatrix}$$

3.3 Taille d'une matrice

La taille d'une matrice *A* est le binôme (n, m) où n et m sont des entiers strictement positifs et correspondent respectivement au nombre de lignes et au nombre de colonnes de la matrice *A*.

Attention : certaines références utilisent le mot *dimension* d'une matrice pour désigner le binôme (n, m). Dans cet ouvrage, on utilisera plutôt le terme *taille* d'une matrice ; tandis que la *dimension* correspond à la dimension de l'espace vectoriel dans lequel on travaille 1D, 2D, etc.

3.4 Vecteur

Un vecteur est un tableau linéaire qui contient une seule ligne (vecteur ligne) ou une seule colonne (vecteur colonne).

L est un vecteur ligne où $l_j \in \mathbb{R}$ pour j = 1, 2, …, m. Pour rappel, les indices sous Python commencent à 0 : j = 0, 1, …, m − 1.

$$L = [l_1 \quad l_2 \quad \dots \quad l_m]$$

L peut être considéré comme une matrice de taille (1, m). Dans ce cas, sous Python, il faudra mettre à jour la taille par une fonction prédéfinie.

C est un vecteur colonne où $c_i \in \mathbb{R}$ pour i = 1, 2, …, n. Pour rappel, les indices sous Python commencent à 0 : j = 0, 1, …, n − 1.

$$C = \begin{bmatrix} c_1 \\ c_2 \\ \dots \\ c_n \end{bmatrix}$$

C peut être considéré comme une matrice de taille (n, 1). Dans ce cas sous Python, il faudra mettre à jour la taille par une fonction prédéfinie.

3.5 Somme de deux matrices

Soient *A* et *B* deux matrices de même taille (n, m) à coefficients réels $(a_{i,j})_{n,m}$ et $(b_{i,j})_{n,m}$. La somme de *A* et *B* est une matrice *S* de même taille (n, m) et de coefficients $(s_{i,j})_{n,m}$ tel que :

$$s_{i,j} = a_{i,j} + b_{i,j}$$

pour i = 1, 2, .., n (sous Python i = 0, 1, .., n – 1) et pour j = 1, 2, …, m (sous Python j = 0, 1, …, m − 1).

Propriétés

Soient *A*, *B* et *C* trois matrices de même taille (n, m) ; l'addition matricielle vérifie :

- L'addition est commutative : *A + B = B + A* ;
- L'addition est associative : *(A + B) + C = A + (B + C)*.

3.6 Produit matriciel

Soient *A* une matrice de taille (n, p) et *B* une matrice de taille (p, m), toutes les deux à coefficients réels $(a_{i,k})_{n,p}$ et $(b_{k,j})_{p,m}$. Le produit de A par B est une matrice P de taille (n, m) et à coefficients réels $(p_{i,j})_{n,m}$ tels que :

$$p_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j} \ \ pour\ i = 1, \dots, n\ et\ j = 1, \dots, m$$

Même remarque que précédemment concernant les indices sous Python qui commencent à 0.

Le produit matriciel est dit aussi multiplication matricielle.

Propriétés

Soient *A, B* et *C* trois matrices de tailles respectivement (n, p), (p, m) et (n, m) ; la multiplication matricielle vérifie :

- La multiplication matricielle n'est pas commutative : en général, $AB \neq BA$;
- La multiplication matricielle est associative : $A(BC) = (AB)C$;
- La multiplication matricielle est distributive par rapport à l'addition : $A(B + C) = AB + AC \quad et \quad (B + C)A = BA + CA$.

3.7 Transposée d'une matrice

Soit A une matrice de taille (n, m) à coefficients réels $(a_{i,k})_{n,p}$. La transposée de la matrice A, notée A^t est une matrice T de taille (m, n) à coefficients réels $(t_{j,i})_{m,n}$ tels que :

$$pour\ i = 1, \dots, n$$
$$pour\ j = 1, \dots, m$$
$$t_{j,i} = a_{i,j}$$

Même remarque que précédemment concernant les indices sous Python qui commencent à 0.

Exemple

$$A = \begin{bmatrix} 2 & 5 & 1 \\ 1 & 1 & 3 \\ 3 & 2 & 5 \end{bmatrix}$$

et

$$A^t = \begin{bmatrix} 2 & 1 & 3 \\ 5 & 1 & 2 \\ 1 & 3 & 5 \end{bmatrix}$$

3.8 Produit scalaire

Soient L un vecteur ligne à n coefficients réels et C un vecteur colonnes à n coefficients réels. Le produit scalaire $\langle L, C \rangle$ est un réel tel que :

$$\langle L, C \rangle = \sum_{i=1}^{n} l_i c_i$$

Même remarque que précédemment concernant les indices sous Python qui commencent à 0.

3.9 Mineur

Soit M une matrice carrée d'ordre n (c'est-à-dire de taille (n, n)) à coefficients réels où $M = (m_{i,j})_{n,n}$. On appelle mineur du couple (i, j) le déterminant de la matrice obtenue à partir de M où on a enlevé la ligne indice i et la colonne indice j.

Cette matrice est notée $M_{i,j}$ et le mineur du couple (i, j) est noté : $det(M_{i,j})$ ou encore $|M_{i,j}|$.

Exemple

Soit la matrice *M* d'ordre 3 à coefficients réels :

$$M = \begin{bmatrix} -1 & 2 & 5 \\ 1 & 2 & 3 \\ -2 & 1 & 0 \end{bmatrix}$$

Le mineur du couple (1, 3) :

$$\left|M_{1,3}\right| = \begin{vmatrix} -1 & 2 & 5 \\ 1 & 2 & 3 \\ -2 & 1 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 2 \\ -2 & 1 \end{vmatrix} = 5{,}0$$

3.10 Cofacteurs et comatrice

Soit *M* une matrice carrée d'ordre n et $|M_{i,j}|$ le mineur du couple (i, j). On appelle cofacteur du couple (i, j), noté $Cof_{i,j}(M)$ le réel :

$$Cof_{i,j}(M) = (-1)^{i+j}\left|M_{i,j}\right|$$

La matrice *C* des cofacteurs $(Cof_{i,j})_{n,n}$ de M est appelée comatrice de *M*, notée *C* ou encore *Comat(M),* où :

$$c_{i,j} = Cof_{i,j}(M)\ pour\ i = 1, \dots, n\ et\ j = 1, \dots, n$$

et

$$Comat(M) = C = \left(c_{i,j}\right)_{n,n} = \left(Cof_{i,j}(M)\right)_{n,n}$$

Exemple

Pour l'exemple précédent, la matrice des cofacteurs est la suivante :

$$Comat(M) = \begin{bmatrix} -3 & -6 & 5 \\ 5 & 10 & -3 \\ -4 & 8 & -4 \end{bmatrix}$$

3.11 Déterminant d'une matrice

Soit M une matrice d'ordre n à coefficients réels $M = (m_{i,j})_{n,n}$. Le déterminant est calculé selon une ligne ou une colonne donnée. Dans la suite, nous le calculons selon la ligne indice k :

$$\det(M) = \sum_{j=1}^{n} m_{k,j} * c_{k,j}$$

Pour l'exemple précédent, on fixe k = 1, ce qui donne :

$$det(M) = (-1)*(-3) + (2)*(-6) + (5)*(5) = 16.$$

Théorème

Soit M une matrice d'ordre n à coefficients réels $M = (m_{i,j})_{n,n}$. Le déterminant de M est égal au déterminant de sa transposée :

$$det(M) = det(M^t)$$

En effet, le *det(M)* selon la ligne k est égal au déterminant de M selon la colonne k et ce dernier est égal au déterminant de M^t selon la colonne k :

$$\det(M) = \sum_{j=1}^{n} m_{k,j} * c_{k,j} = \sum_{j=1}^{n} m_{j,k} * c_{j,k} = det(M^t)$$

3.12 Matrice inverse

Soit M une matrice d'ordre n à coefficients réels $M = (m_{i,j})_{n,n}$. M est inversible si et seulement si son déterminant est un réel non nul.

Soit M une matrice d'ordre n à coefficients réels $M = (m_{i,j})_{n,n}$ et M est inversible. Son inverse noté M^{-1} vérifie la formule suivante :

$$M^{-1} = \frac{1}{det(M)} C^t$$

où C est la comatrice de M.

Caractérisation

Soit A une matrice inversible d'ordre n à coefficients réels et B est son inverse. On a :

$$AB = BA = I_n \text{ où } I_n \text{ est la matrice identité d'ordre } n$$

En plus :

$$det(AB) = det(A)det(B) = det(I_n) = 1$$

Exemple

Pour l'exemple précédent :

$$M = \begin{bmatrix} -1 & 2 & 5 \\ 1 & 2 & 3 \\ -2 & 1 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} -3 & -6 & 5 \\ 5 & 10 & -3 \\ -4 & 8 & -4 \end{bmatrix}$$

Calcul du déterminant selon la ligne k = 1 : *det(M) = 16*

Puis vérification :

$$M^{-1} = \frac{1}{det(M)} C^t$$

$$M^{-1} = \frac{1}{16} \begin{bmatrix} -3 & 5 & -4 \\ -6 & 10 & 8 \\ 5 & -3 & -4 \end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} -3/16 & 5/16 & -1/4 \\ -3/8 & 5/8 & 1/2 \\ 5/16 & -3/16 & -1/4 \end{bmatrix}$$

$$M.M^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I_3$$

3.13 Matrice élémentaire

Une matrice élémentaire $E = (e_{i,j})_{n,n}$ est une matrice d'ordre n à coefficients réels, obtenue de la matrice identité I_n en effectuant une seule opération par ligne.

Type d'opération ligne	**Opération**	**Matrice élémentaire**	**Remarque**	**Exemple**
Expression linéaire	$L_{i,i'}(k)$	$E_{i,i'}(k)$	i et i' : indices de lignes. La ligne i (L_i) est remplacée par $L_i + k*L_{i'}$ de la matrice identité I_n	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \xrightarrow{L_{21}(-3)} \begin{bmatrix}1 & 0\\-3 & 1\end{bmatrix} = E_{21}(-3)$
Permutation	$L_{i,i'}$	$E_{i,i'}$	Permutation de L_i avec $L_{i'}$	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \xrightarrow{L_{12}} \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} = E_{12}$
Multiplication	$L_i(k)$	$E_i(k)$	La ligne i (L_i) est remplacée par $k*L_i$	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \xrightarrow{L_1(\pi)} \begin{bmatrix}\pi & 0\\0 & 1\end{bmatrix} = E_1(\pi)$

Propriété

Soient *A* une matrice de taille (n, m) à coefficients réels et E une matrice élémentaire d'ordre n et à coefficients réels. La matrice *B* de taille (n, m) obtenue de *A* par une « opération élémentaire ligne » est égale au produit *EA*.

Exemple

$$A = \begin{bmatrix}-1 & 2\\5 & 3\end{bmatrix} \xrightarrow{L_{21}(-3)} \begin{bmatrix}-1 & 2\\8 & -3\end{bmatrix} = B$$

$$I_2 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \xrightarrow{L_{21}(-3)} \begin{bmatrix}1 & 0\\-3 & 1\end{bmatrix} = E$$

et

$$EA = \begin{bmatrix}1 & 0\\-3 & 1\end{bmatrix}\begin{bmatrix}-1 & 2\\5 & 3\end{bmatrix} = \begin{bmatrix}-1 & 2\\8 & -3\end{bmatrix} = B$$

Propriété

Toute matrice élémentaire est inversible.

Exemple

Soit la matrice E de réels d'ordre 2. Voici, ci-dessous les instructions Python pour calculer :

- le déterminant $det(E)$;
- l'inverse de E si elle existe E^{-1} ;
- le produit matriciel $E.E^{-1}$.

$$E = \begin{bmatrix} 1 & 0 \\ -3 & 1 \end{bmatrix}$$

```
Import numpy as np
np.linalg.det(E)
```

1.0

```
np.linalg.inv(E)
```

$$\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}$$

```
np.dot(E,np.linalg.inv(E))
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

3.14 Matrice triangulaire

Une matrice A d'ordre n à coefficients réels $A = (a_{i,j})_{n,n}$ est une matrice triangulaire supérieure lorsque tous ses coefficients au-dessous de la diagonale sont nuls :

$$a_{i,j} = 0, \forall\ (i,j)\ \ tels\ que\ 1 < i < n, 1 < j < n\ et\ i\ > j$$

Exemple

$$A = \begin{bmatrix} 1 & 1/2 & 1/2 \\ \mathbf{0} & 1 & 3/5 \\ \mathbf{0} & \mathbf{0} & 1 \end{bmatrix}$$

Une matrice A d'ordre n à coefficients réels $A = (a_{i,j})_{n,n}$ est une matrice triangulaire inférieure lorsque tous ses coefficients au-dessus de la diagonale sont nuls :

$$a_{i,j} = 0, \forall\ (i,j)\ \ tels\ que\ 1 < i < n, 1 < j < n\ et\ i\ < j$$

Exemple

$$A = \begin{bmatrix} 2 & \mathbf{0} & \mathbf{0} \\ \sqrt{3} & -1 & \mathbf{0} \\ -2 & \pi/2 & 4 \end{bmatrix}$$

3.15 Matrice diagonale

Une matrice A d'ordre n à coefficients réels $A = (a_{i,j})_{n,n}$ est une matrice triangulaire supérieure lorsque tous ses coefficients au-dessous de la diagonale sont nuls :

$$a_{i,j} = 0, \forall\ (i,j)\ tels\ que\ 1 \leq i \leq n, 1 \leq j \leq n\ et\ i\ \neq j$$

Exemple

$$A = \begin{bmatrix} 2 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 4 \end{bmatrix}$$

3.16 Autres notions

D'autres notions et méthodes importantes en algèbre linéaire et calcul matriciel peuvent être abordées si besoin dans la suite en développement Python. Dans la littérature il existe plusieurs références pour le lecteur, plus orientées mathématiques fondamentales pour réviser ou apprendre ces notions telles que :

- valeurs propres et vecteurs propres ;
- matrice de passage ;
- matrice semblable ;
- matrice singulière ;
- trace d'une matrice ;
- système d'équations linéaires ;
- diagonalisation ;
- pivot de Gauss ;
- etc.

4 Les tableaux et matrices avec la librairie *numpy*

Nous utiliserons les tableaux sous *numpy* pour travailler avec des tableaux, des matrices, des vecteurs et appliquer les notions d'algèbre linéaire sur des cas pratiques et le traitement de situations réelles.

4.1 Syntaxe

On utilise ici le terme tableau qui désigne également vecteur ou encore matrice au sens informatique du terme.

Les tableaux sont désignés par la méthode *array()* du module *numpy*. Ainsi pour adresser un tableau on utilise la syntaxe suivante :

```
import numpy as np
...
T=np.array(vecteur(s), dtype, copy, order, subok, ndmin)
```

où :

- *vecteur(s)* : ligne(s) qui compose(nt) le tableau ;
- dtype, copy, order, subok et ndmin sont facultatifs ;
- *dtype* : le type des données du tableau ;
- *ndmin* : le nombre minimum de dimensions.

4.2 Création de tableaux

Pour la création d'un tableau 1D, 2D ou 3D, plusieurs situations peuvent se présenter :

- création d'un tableau avec des données saisies dans le code ;
- création d'un tableau vide avec une taille donnée qui sera renseigné lors de l'exécution du code ;
- création d'un tableau avec des méthodes de *numpy* comme *ones()* ou *zeros()* qui sera renseigné lors de l'exécution du code ;
- transformation d'une liste avec des données homogènes en un tableau ;
- importation des données stockées dans un fichier, dans un tableau ;
- création d'un tableau par copie d'un autre tableau ;
- création d'une matrice identité d'ordre n avec la méthode *eye()* de *numpy* ;
- etc.

Dans ce chapitre, on utilisera *spyder* et/ou sa console *IPython*.

4.2.1 Création d'un tableau avec des données saisies dans le code

```
# T un tableau 1D de réels de taille 4
import numpy as np

T=np.array([1,2,3,4],dtype=float,ndmin=1)
T
```

array([1., 2., 3., 4.])

```
type(T)
```

numpy.ndarray

```
np.shape(T)
```

(4,)

```
print(T)
```

[1. 2. 3. 4.]

```
# T un tableau 2D de réels de taille (2, 2)
import numpy as np

T=np.array([[1,2],[3,4]],dtype=float,ndmin=2)
type(T)
```

numpy.ndarray

```
print(T)
```

[[1. 2.]
 [3. 4.]]

```
np.shape(T)
```

(2, 2)

```
# T un tableau 3D de taille (2, 2, 3) c-à-d. (plans, lignes, colonnes)
import numpy as np
```

```
T=np.array([[[1,2,3],[4,5,6]],[[-1,-2,-3],[-4,-5,-6]]], dtype=float,
ndmin=3)
type(T)
```

numpy.ndarray

```
print(T)
```

```
[[[ 1.  2.  3.]
  [ 4.  5.  6.]]
 [[-1. -2. -3.]
  [-4. -5. -6.]]]
```

```
np.shape(T)
```

(2, 2, 3)

4.2.2 Création d'un tableau vide

Ici on utilisera la méthode *empty()* pour créer un tableau vide, avec une taille donnée, qui sera renseigné lors de l'exécution du code.

```
import numpy as np

T=np.empty((2,2), dtype=float)
type(T)
```

numpy.ndarray

4.2.3 Transformation d'une liste en un tableau

Comme précédemment, on donnera des exemples pour illustrer les méthodes *numpy* de création de tableaux.

```
# Création d'un tableau 1D à partir d'une liste numérique
import numpy as np

l=[i for i in range(1,5)]
type(l)
```

list

```
T=np.array(l,dtype=float, ndmin=1)
type(T)
```

numpy.ndarray

```
print(T)
```

[1. 2. 3. 4.]

```
# Création d'un vecteur ligne de taille (1,4) à partir d'une liste numérique
VL=np.array(l,dtype=float, ndmin=2)
print(VL)
```

[[1. 2. 3. 4.]]

```
np.shape(VL)
```

(1, 4)

```
# Création d'un vecteur colonne de taille (4,1) à partir d'une liste numérique
# l : une liste numérique
l=[[i] for i in range(1,5)]
# VC : un vecteur colonne
VC=np.array(l,dtype=float, ndmin=2)
np.shape(VC)
```

(4, 1)

```
print(VC)
```

[[1.]
 [2.]
 [3.]
 [4.]]

```
# Création d'un tableau de taille (3,3) à partir de deux listes numériques
import numpy as np
import random
```

```
l1=[round(random.randint(-5,5)*i,0) for i in range(1,4)]
l2=[round(i*np.sqrt(i**2+2),0) for i in range(1,4)]
l3=[round(3*i*np.sqrt(i+2),0) for i in range(1,4)]
l=[l1,l2,l3]

T=np.array(l,dtype=float, ndmin=2)
type(T)
```

numpy.ndarray

```
T.dtype
```

dtype('float64')

```
print(T)
```

```
[[-3.  8. 12.]
 [ 2.  5. 10.]
 [ 5. 12. 20.]]
```

```
np.shape(T)
```

(3, 3)

D'autres méthodes de création de tableaux seront abordées dans les paragraphes suivants et plus généralement dans cet ouvrage.

4.2.4 Création d'un tableau avec ones() ou zeros()

Ici on utilisera la méthode *zeros()* pour créer un tableau initialisé avec la valeur numérique 0 puis la méthode *ones ()* pour créer un tableau initialisé avec la valeur numérique 1, avec une taille du tableau donnée. Le tableau sera renseigné lors de l'exécution du code.

```
import numpy as np

d1=4
d2=3
M0=np.zeros((d1,d2),float)
M1=np.ones((d1,d2),float)
```

```
type(M0)
```

numpy.ndarray

```
type(M1)
```

numpy.ndarray

```
print(M0)
```

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

```
print(M1)
```

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

4.2.5 Création d'un tableau par copie d'un autre tableau

Ici on utilisera la méthode *copy()* pour la création d'un tableau par copie d'un autre tableau déjà existant. Les deux tableaux auront la même taille et le même type de données. Lorsque la copie est réalisée, les deux tableaux sont indépendants.

```
import numpy as np

A=np.array([[1,0,0,0,1],[0,1,0,-2,2],[3,0,1,1,0],[0,4,0,1,-1],[5,0,0,0,-1]],
dtype=float)
n=A.shape[0]
m=A.shape[1]
B=np.copy(A)

# A et B sont de même taille
print(n==B.shape[0])
```

True

```
print(n==B.shape[1])
```

True

```
print(A)
```

```
[[ 1.  0.  0.  0.  1.]
 [ 0.  1.  0. -2.  2.]
 [ 3.  0.  1.  1.  0.]
 [ 0.  4.  0.  1. -1.]
 [ 5.  0.  0.  0. -1.]]
```

```
print(B)
```

```
[[ 1.  0.  0.  0.  1.]
 [ 0.  1.  0. -2.  2.]
 [ 3.  0.  1.  1.  0.]
 [ 0.  4.  0.  1. -1.]
 [ 5.  0.  0.  0. -1.]]
```

```
print(A==B)
```

```
[[ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]
```

```
#A et B sont indépendantes
B[0][0]=25
print(A==B)
```

```
[[False  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]
```

4.2.6 Importation d'un tableau sauvegardé dans un fichier

Il existe plusieurs librairies Python, et donc plusieurs méthodes, qui permettent l'importation de données à partir d'un fichier comme les librairies *pandas*, ou *torch* ou encore *os*. Chacune peut être utilisée dans certains contextes spécifiques. Par exemple, l'importation de données à l'aide de la librairie *pandas* peut être utilisée dans les cas des représentations graphiques de données ou encore l'importation de données à l'aide de la librairie *torch* peut être utilisée dans le cas d'analyse de données en *Machine Learning*.

Dans ce paragraphe, nous utiliserons la librairie *os*, plus simple pour l'importation de tableaux et matrices à partir d'un fichier texte. De la même façon, nous l'utiliserons pour sauvegarder des tableaux et matrices dans un fichier texte.

```
import numpy as np
import os

#Importation de MatriceIn.txt dans A
os.chdir("C:/ Documents/Fichiers data")
A=np.loadtxt("MatriceIn.txt",delimiter=",",dtype=float)
type(A)
```

numpy.ndarray

```
print(A)
```

```
[[ 1.  0.  0.  0.  1.]
 [ 0.  1.  0. -2.  2.]
 [ 3.  0.  1.  1.  0.]
 [ 0.  4.  0.  1. -1.]
 [ 5.  0.  0.  0. -1.]]
```

```
#B est une matrice résultat à la suite d'un traitement sur A
B=A[0:3,0:4]

print(B)
```

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

```
# Sauvegarde de B dans le fichier MatriceOut.txt
np.savetxt('MatriceOut',B,delimiter=",", fmt='%3.1f')
```

4.2.7 Création d'un tableau à partir d'un range

Ici on utilisera la méthode *arange()* pour créer un tableau initialisé avec un range, c'est-à-dire avec une séquence finie de valeurs entières générées, par incrémentation ou décrémentation, à partir d'une valeur entière donnée. Un tableau crée avec la méthode *arange()* est un tableau à une dimension et de taille égale au cardinal du range.

Syntaxe

```
numpy.arange([start, ]stop, [step, ]dtype)
```

Exemples

```
import numpy as np

T=np.arange(10, dtype=int)
type(T)
```

numpy.ndarray

```
print(T)
```

[0 1 2 3 4 5 6 7 8 9]

```
import numpy as np

A=np.arange(1, 20, 2, dtype=float)
print(A)
```

[1. 3. 5. 7. 9. 11. 13. 15. 17. 19.]

```
A.shape
```

(10,)

4.2.8 Création d'un tableau par linspace

La méthode *linspace()* est utilisée pour créer un tableau, de valeurs réelles, à une dimension. Parmi les paramètres de *linspace()* il y a :

- *start* : valeur de début ;
- *stop* : valeur de fin ;
- *num* : nombre de valeurs générées à partir de l'intervalle réel *[start, stop]*. Les valeurs générées sont équidistantes.

```
import numpy as np

X=np.linspace(-10,10,5)
X
```

array([−10., −5., 0., 5., 10.])

```
Y=np.linspace(3,11,5)
Y
```

array([3., 5., 7., 9., 11.])

4.3 Opérations sur les tableaux

La librairie *numpy* propose des méthodes permettant ainsi la manipulation des tableaux tels que la lecture, l'écriture, la taille et le changement de taille, le *slicing*, l'utilisation de tableaux spécifiques, le calcul matriciel, etc.

Nous allons en aborder quelques-unes ici. Pour d'autres opérations telles que la lecture, l'écriture, l'adressage d'un tableau, le tri ou encore la recherche dans un tableau, le lecteur peut se référer à l'ouvrage *Algorithmique et développement Python* du même auteur, paru aux éditions Ellipses.

4.3.1 Taille d'un tableau

Rappel : on utilise ici le terme tableau qui désigne également vecteur ou encore matrice au sens informatique du terme.

La taille d'un tableau donne explicitement le nombre d'éléments par dimension. Pour calculer la taille d'un tableau sous *numpy* on utilise la méthode *shape()*.

Exemple

```
import os
import numpy as np
```

```
# V : tableau à une dimension
V=np.arange(-1,4, dtype=float)

V
```

```
array([−1.,  0.,  1.,  2.,  3.])
```

```
# M : matrice deux dimensions
os.chdir("C:/ Documents/Fichiers data ")
M=np.loadtxt("Matrice4x5.txt", delimiter=",",dtype=float)
M
```

```
array([[ 1.,  0.,  0.,  0.,  1.],
       [ 0.,  1.,  0., -2.,  2.],
       [ 3.,  0.,  1.,  1.,  0.],
       [ 0.,  4.,  0.,  1., -1.]])
```

```
# E : matrice à trois dimensions (plans, lignes, colonnes)
E=np.array([[[1,2,3],[4,5,6]],[[-1,-2,-3],[-4,-5,-6]]], dtype=float,
ndmin=3)
E
```

```
array([[[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]],
       [[−1., −2., −3.],
        [−4., −5., −6.]]])
```

```
a=np.shape(V)
print(V.shape == a)
```

```
True
```

```
M.shape
```

```
(4, 5)
```

```
# Vérification taille par dimension
(M.shape ==(M.shape[0], M.shape[1]))
```

```
True
```

```
# Même vérification pour E en 3D
(E.shape ==(E.shape[0], E.shape[1], E.shape[2]))
```

True

4.3.2 Changement de taille d'un vecteur ou d'une matrice

On peut transformer un tableau à 1D de taille n en une matrice à 2D de taille (a, b) avec conservation des données et où n, a et b sont des entiers strictement positifs tels que n = a*b, on utilisant la méthode *reshape()* de *numpy*.

```
import numpy as np
import random

T=np.array([round(random.randint(-5,22)*i,0) for i in range(1,25)],
dtype=float, ndmin=1)
T
```

```
array([ −5.,   2.,  −6., −12., −25., −30., 154.,  24., 153., 110.,  77.,24., −52.,
       196., −15., 304., 221., 396.,  76., 360.,  63., 154., 161., −24.])
```

```
T.shape[0]
```

24

```
M=T.reshape(4,6)
M
```

```
array([[ -5.,   2.,  -6., -12., -25., -30.],
       [154.,  24., 153., 110.,  77.,  24.],
       [-52., 196., -15., 304., 221., 396.],
       [ 76., 360.,  63., 154., 161., -24.]])
```

```
M.shape
```

(4, 6)

```
# Vérification
T.shape[0]==M.shape[0]*M.shape[1]
```

True

De la même façon, on peut transformer une matrice de taille (n, m) en une matrice de taille (a, b) avec conservation des données et où n, m, a et b sont des entiers strictement positifs tels que n*m = a*b, en utilisant la méthode *reshape()* de *numpy*.

```
# Suite de l'exemple précédent
N=M.reshape(8,3)
N
```

```
array([[ −5.,  2., −6.],
       [−12., −25., −30.],
       [154., 24., 153.],
       [110., 77., 24.],
       [−52., 196., −15.],
       [304., 221., 396.],
       [ 76., 360., 63.],
       [154., 161., −24.]])
```

```
# Vérification
M.shape[0]*M.shape[1]==N.shape[0]*N.shape[1]
```

```
True
```

4.3.3 Transformation d'une matrice en un tableau 1D

Aplatir une matrice à deux dimensions en un tableau à une dimension revient à créer un tableau qui contient les données de cette matrice ligne par ligne à la queue leu-leu ou bien colonne par colonne à la queue leu-leu. On utilise la méthode *flatten()* de *numpy* avec l'argument *order = 'C'*, c'est-à-dire ligne par ligne à la queue leu-leu ou bien *order = 'F'*, c'est-à-dire colonne par colonne à la queue leu-leu.

```
# Suite de l'exemple précédent
A=N.flatten(order='C')
A
```

```
array([ −5.,  2., −6., −12., −25., −30., 154., 24., 153., 110., 77., 24., −52.,
196., −15., 304., 221., 396., 76., 360., 63., 154., 161., −24.])
```

```
A.shape[0]
```

```
24
```

```
# Vérification
A==T
```

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True, True,
        True,  True,  True,  True,  True,  True,  True,  True, True,  True,  True,
        True,  True,  True])
```

```
B=N.flatten(order='F')
B
```

```
array([ −5., −12., 154., 110., −52., 304.,  76., 154.,   2., −25.,  24., 77., 196.,
        221., 360., 161.,  −6., −30., 153.,  24., −15., 396., 63., −24.])
```

```
B.shape[0]
```

```
24
```

```
B==T
```

```
array([ True, False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False, False, False,
        False, False,  True])
```

4.3.4 Le slicing d'une matrice

Le *slicing* d'une matrice, littéralement trancher une matrice, revient à extraire une « tranche » de données de cette matrice allant d'un indice de début jusqu'à un indice de fin selon chaque dimension.

Soit M une matrice de réels de taille *(n, m)* où $n \in \mathbb{N}^*$ et $m \in \mathbb{N}^*$.

Soient *a, b, c* et *d* des entiers tels que :

- $b \in \mathbb{N}^* \; et \; 0 \leq a < n$;
- $a \in \mathbb{N}^* \; et \; 0 \leq a \leq b < n$;
- $d \in \mathbb{N}^* \; et \; 0 \leq d < m$;
- $c \in \mathbb{N}^* \; et \; 0 \leq c \leq d < m$.

On souhaite extraire les données de la matrice M situées entre les indices de lignes *a* et *b* incluses et entre les indices des colonnes *c* et *d* incluses.

L'instruction Python correspondante est :

```
M[a :b+1, c :d+1]
```

Exemple

```
# Suite de l'exemple précédent
print(N)
```

```
[[ −5.   2.  −6.]
 [−12. −25. −30.]
 [154.  24. 153.]
 [110.  77.  24.]
 [−52. 196. −15.]
 [304. 221. 396.]
 [ 76. 360.  63.]
 [154. 161. −24.]]
```

```
N[1:4,0:2]
```

```
array([[−12., −25.],
       [154.,  24.],
       [110.,  77.]])
```

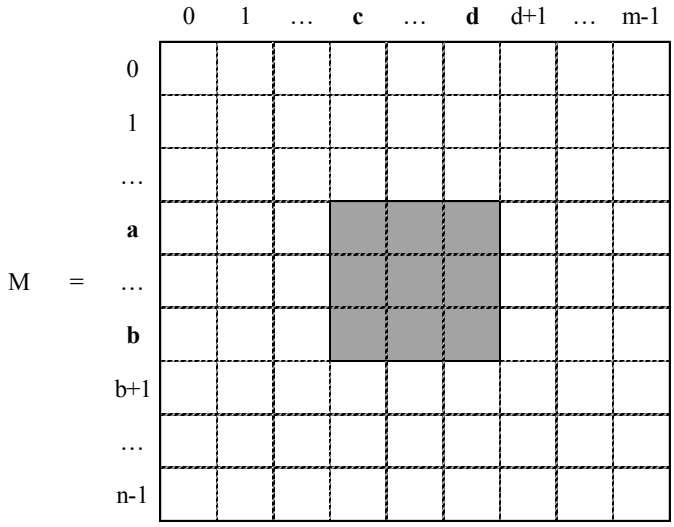
```
S=N[2:5,1:]
S
```

```
array([[ 24., 153.],
       [ 77.,  24.],
       [196., −15.]])
```

Voici quelques explications complémentaires de la syntaxe *slicing* avec les mêmes contraintes de définition des entiers *n, m, a, b, c* et *d* :

- M[a : b+1, c : d+1] : données situées entre les indices de lignes a et b incluses et entre les indices des colonnes c et d incluses ;
- M[: b+1, c : d+1] : données situées entre les indices de lignes 0 et b incluses et entre les indices des colonnes c et d incluses ;
- M[a :, c : d+1] : données situées entre les indices de lignes a et n - 1 incluses et entre les indices des colonnes c et d incluses ;
- M[a, c : d+1] : données situées sur la ligne indice a et entre les indices des colonnes c et d incluses ;
- M[-1 :, c : d+1] : données situées sur la dernière ligne et entre les indices des colonnes c et d incluses ;

- M[: -1, c : d+1] : données situées sur toutes les lignes sauf la dernière ligne et entre les indices des colonnes c et d incluses ;
- etc.



Exemple

```
# Exemple de script permettant une permutation de lignes d'une matrice

import numpy as np
import os

os.chdir("C:/ Documents/Fichiers data ")
A=np.loadtxt("MatriceIn.txt",delimiter=",",dtype=float)

print("Matrice initiale : \n",A,"\n")

#Indice dernière ligne
d=A.shape[0]-1

# Permutation de la première ligne avec la dernière ligne
print("\nPermutation de la première ligne avec la dernière ligne\n")
L=np.copy(A[0,:])
```

```
A[0, :] = A[d, :]
A[d, :] = L
print("Matrice modifiée : \n",A)
```

Résultat

```
Matrice initiale :
 [[ 1.  0.  0.  0.  1.]
 [ 0.  1.  0. -2.  2.]
 [ 3.  0.  1.  1.  0.]
 [ 0.  4.  0.  1. -1.]
 [ 5.  0.  0.  0. -1.]]

Permutation de la première ligne avec la dernière ligne
Matrice modifiée :
 [[ 5.  0.  0.  0. -1.]
 [ 0.  1.  0. -2.  2.]
 [ 3.  0.  1.  1.  0.]
 [ 0.  4.  0.  1. -1.]
 [ 1.  0.  0.  0.  1.]]
```

4.3.5 Image miroir d'une matrice

L'image d'une matrice 2D selon un miroir vertical, consiste à inverser l'ordre des éléments selon l'**axe 1** de gauche à droite, c'est-à-dire inverser l'ordre des éléments de cette matrice ligne par ligne. On utilisera la méthode *fliplr()* de *numpy* pour réaliser cette inversion.

```
import numpy as np

M=np.arange(20, dtype=float).reshape(4,5)
print(M)
```

```
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]]
```

```
N=np.fliplr(M)
print(N)
```

```
[[ 4.  3.  2.  1.  0.]
 [ 9.  8.  7.  6.  5.]
 [14. 13. 12. 11. 10.]
 [19. 18. 17. 16. 15.]]
```

L'image d'une matrice 2D selon un miroir horizontal, consiste à inverser l'ordre des éléments selon l'**axe 0** de haut en bas, c'est-à-dire inverser l'ordre des éléments de cette matrice colonne par colonne. On utilisera la méthode *flipud()* de *numpy* pour réaliser cette inversion.

```
#Suite exemple précédent
print(M)
```

```
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]]
```

```
H=np.flipud(M)
print(H)
```

```
[[15. 16. 17. 18. 19.]
 [10. 11. 12. 13. 14.]
 [ 5.  6.  7.  8.  9.]
 [ 0.  1.  2.  3.  4.]]
```

4.3.6 Scinder un tableau

Pour scinder un tableau en une liste de tableaux, on peut utiliser la méthode *split()* de *numpy*. Cette méthode permet de scinder un tableau en une liste de tableaux selon l'axe 0, c'est-à-dire selon les colonnes, par défaut.

Voici la syntaxe générale de *split()* :

```
numpy.split(T, nombre_ou_morceaux [, axis = 0])
```

où :

- *T* : le tableau à scinder ;
- *nombre_ou_morceaux* : si c'est un nombre entier strictement positif alors c'est le nombre de morceaux de tailles égales obtenus. Si ce n'est pas possible de scinder le tableau en nombre de morceaux alors un message d'erreur s'affiche ;
- *nombre_ou_morceaux* : si c'est un tableau d'entiers strictement positifs, triés dans l'ordre croissant, alors ce tableau donne les indices supérieurs exclus de chaque morceau (cf. exemples suivants) ;
- *axis = 0* : par défaut, le tableau est scindé selon les colonnes.

```
# Suite de l'exemple précédent
T=np.resize(M,(20,))
T
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14., 15., 16.,
17., 18., 19.])
```

```
# Scinder T en deux tableaux
TS=np.split(T,2)
TS
```

```
[array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]), array([10., 11., 12., 13., 14., 15.,
16., 17., 18., 19.])]
```

```
type(TS)
```

```
list
```

```
TS[0]
```

```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
TS[1]
```

```
array([10., 11., 12., 13., 14., 15., 16., 17., 18., 19.])
```

```
np.split(T,3)
# Message d'erreur
ValueError: array split does not result in an equal division
```

```
TI=np.split(T,[6,11])
TI
```

```
[array([0., 1., 2., 3., 4., 5.]),
 array([ 6.,  7.,  8.,  9., 10.]),
 array([11., 12., 13., 14., 15., 16., 17., 18., 19.])]
```

```
# TI est également une liste
type(TI)
```

```
list
```

```
# Chaque élément de la liste TI est un tableau
type(TI[0])
```

```
numpy.ndarray
```

```
# Il est possible d'avoir des tableaux vides dans la liste
TE=np.split(T,[6,11,18,22])
TE
```

```
[array([0., 1., 2., 3., 4., 5.]),
 array([ 6.,  7.,  8.,  9., 10.]),
 array([11., 12., 13., 14., 15., 16., 17.]),
 array([18., 19.]),
 array([], dtype=float64)]
```

```
# Exemple de matrice 2D scindée en plusieurs morceaux.
# Par défaut, les morceaux correspondent à des regroupements de lignes successives
# Reprenons la matrice M précédente de taille (4, 5)

M
```

```
array([[ 1.,  0.,  0.,  0.,  1.],
       [ 0.,  1.,  0., -2.,  2.],
       [ 3.,  0.,  1.,  1.,  0.],
       [ 0.,  4.,  0.,  1., -1.]])
```

```
MS=np.split(M,2)
MS
[array([[ 1.,  0.,  0.,  0.,  1.],
        [ 0.,  1.,  0., -2.,  2.]]),
 array([[ 3.,  0.,  1.,  1.,  0.],
        [ 0.,  4.,  0.,  1., -1.]])]
```

```
MS=np.split(M,3)
# Message d'erreur
ValueError: array split does not result in an equal division
```

Il y a aussi d'autres méthodes de division de tableau :

- *dsplit()* : deep split ;
- *hsplit()* : horizontal split, équivalent *split()* avec axis = 1 ;
- *vsplit()* : vertical split ; équivalent *split()* avec axis = 0.

5 Algèbre linéaire et calcul matriciel avec *numpy*

Ici, on trouvera un tableau récapitulatif et non exhaustif d'opérations du calcul matriciel ainsi que des exemples d'applications. Ces exemples d'applications se basent sur le schéma méthodologique ci-après (cf. figure 3).

5.1 Opérations et calcul matriciel avec *numpy*

Dans la suite des opérations déjà abordées sur les tableaux (matrices), voici un tableau récapitulatif d'autres opérations et méthodes de la librairie *numpy* (*import numpy as np*) et *numpy.linalg* (*import numpy.linalg as la*) mobilisables en algèbre linéaire.

Remarques

Pour une matrice d'ordre (n, m) :

- mathématiquement l'indice des lignes varie de 1 à n mais sous Python l'indice des lignes varie de 0 à n - 1. Nous apporterons cet ajustement plus loin dans les scripts Python ;
- mathématiquement l'indice des colonnes varie de 1 à m tandis que sous Python l'indice des colonnes varie de 0 à m - 1. Nous apporterons également cet ajustement plus loin dans les scripts Python.

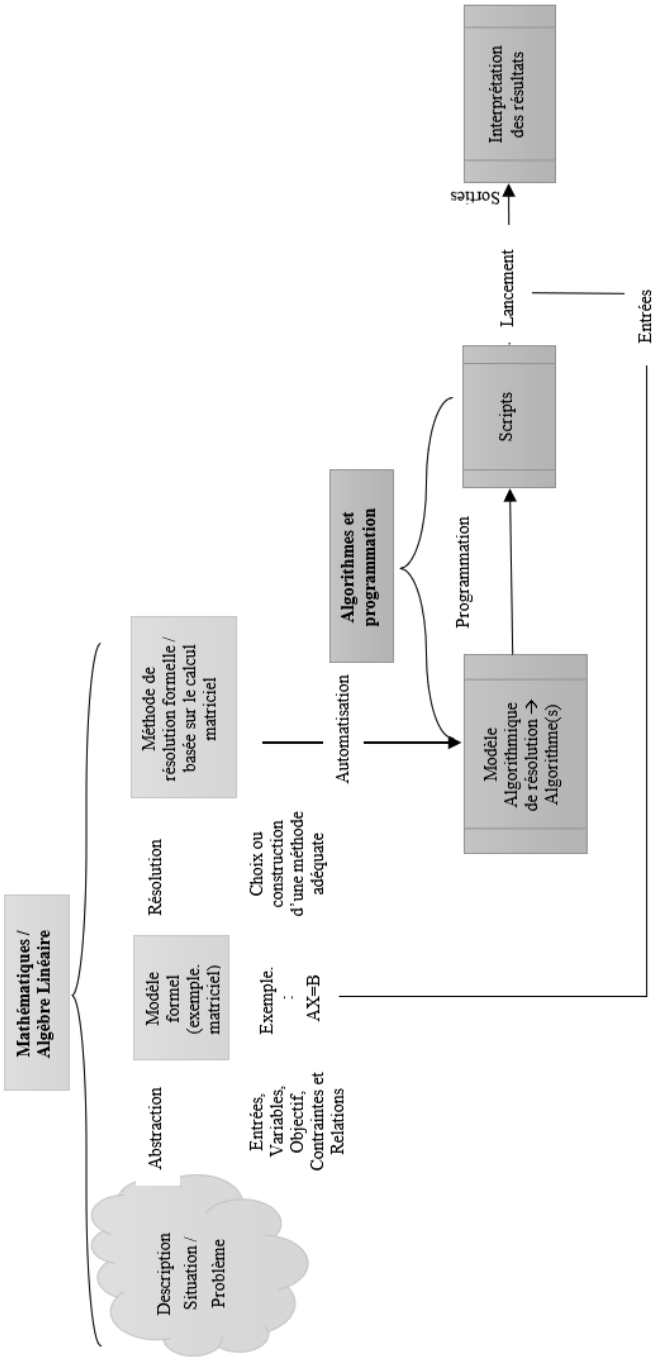
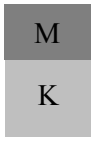


***Figure 3**. Schéma méthodologique*

Description	**Syntaxe**	**Librairie / Remarque / Exemple**
Addition terme-à-terme de deux matrices M et K	*M + K*	*Les deux matrices M et K sont de même taille. Addition terme-à-terme* *A = M + K*
Arrondir toutes les valeurs d'une matrice M à d décimales	*around(M, d)*	*import numpy as np* *# Fonction d'arrondi* *M = np.array([round(np.pi,2)*i for i in range(1,4)])* *M* *array([3.14, 6.28, 9.42])* *print(np.around(M,1))* *[3.1, 6.3, 9.4]*
Déterminant d'une matrice carrée M d'ordre n	*det(M)*	*import numpy.linalg as la* *d=la.det(M)*
Diagonale d'une matrice carrée M d'ordre n	*diag(M)*	*import numpy as np* *# D est un tableau de taille n* *D = np.diag(M)*
Élément d'un tableau, d'une matrice	*Tableau 2D :* *M[i][j]* *Matrice 2D :* *M[i,j]*	*# Élément à l'intersection de la ligne indice i et la colonne indice j* *M[0][1]* *M[0, 1]*
Fusion horizontale : concaténation horizontale de deux matrices M et K (Condition : M et K ont le même nombre de lignes)	*concatenate((M,K), axis=1)* 	*import numpy as np* *# Condition : M et K ont le même nombre de lignes* *M = np.array([[0, -1, 3],[2, -2, 0],[4, 0, -1]], dtype=float)* *K = np.array([[1, 1, 0], [-1, -1, 2], [3, -1, 3]], dtype=float)* *H=np.concatenate((M,K),axis=1)* *H* *array([[0., -1., 3., 1., 1., 0.],* *[2., -2., 0., -1., -1., 2.],* *[4., 0., -1., 3., -1., 3.]])*

Description	Syntaxe	Librairie / Remarque / Exemple
Fusion verticale : concaténation verticale de deux matrices M et N (Condition : M et N ont le même nombre de colonnes)	*concatenate((M,N), axis=0)* 	*import numpy as np* *# Condition : M et K ont le même nombre de colonnes* *M=np.array([[0,-1,3],[2,-2,0],[4,0,-1]], dtype=float)* *N=np.array([[10,10,0],[-10,-10,20]], dtype=float)* *V=np.concatenate((M,N), axis=0)* *V* *array([[0., -1., 3.],* *[2., -2., 0.],* *[4., 0., -1.],* *[10., 10., 0.],* *[-10., -10., 20.]])*
Inverse d'une matrice M carrée d'ordre n	*inv(M)*	*import numpy as np* *import numpy.linalg as la* *# M est inversible si et seulement si son determinant est non nul* *M=np.array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]], dtype=float)* *if la.det(M) <> 0 :* *IM=la.inv(M)* *print(MI)* *[[0.2 1.2 -0.8]* *[0.4 -0.6 0.4]* *[-0.3 0.7 -0.3]]*
Matrice (exemple $M_{(3,3)}$)	*matrix(data, dtype)*	*import numpy as np* *# Le constructeur de matrices* *Mx=np.matrix([[0, -1, 3], [2, -2, 0], [4, 0, -1]], dtype=float)* *Mx* *matrix([[0., -1., 3.],* *[2., -2., 0.],* *[4., 0., -1.]])*

Description	Syntaxe	Librairie / Remarque / Exemple
Matrice identité d'ordre n	*identity(n, dtype)*	*import numpy as np* *I=np.identity(3,float)* *array([[1., 0., 0.],* *[0., 1., 0.],* *[0., 0., 1.]])*
Multiplication d'une matrice par un scalaire	*s*M*	*import numpy as np* *# Multiplication de la matrice M par le scalaire s.* *# Exemple où s = round(pi, 2)* *M=np.array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *s = round(np.pi, 2)* *Ms=s*M* *Ms* *array([[3.14, 6.28, 0.],* *[0. , 9.42, 12.56],* *[-3.14, 15.7 , 18.84]])*
Norme de U	*norm()*	*import numpy as np* *import numpy.linalg la* *# Si V est un vecteur --> norme euclidienne* *V=np.array([4., 0., 3.])* *d=la.norm(V)* *print(d)* *5.0* *# Si U est une matrice --> norme de Frobenius* *U=np.array([[0., 3.],[2., 0.],[4., -1.],[4., 1.]])* *f=la.norm(U)* *print(round(f, 2))* *6.86*

Description	**Syntaxe**	**Librairie / Remarque / Exemple**
Plus grand élément (pge) d'une matrice	*max(M)*	*import numpy as np* *M = np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *pge = np.max(M)* *pge* *6*
Plus grand élément sur chaque colonne d'une matrice (tableau PGEC)	*Formule avec max() et range()*	*import numpy as np* *M = np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *PGEC=np.array([np.max(A[:,i]) for i in range(np.shape(A)[1])])* *PGEC* *array([1, 5, 6])*
Plus grand élément sur chaque ligne d'une matrice (tableau PGEL)	*Formule avec max() et range()*	*import numpy as np* *M = np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *PGEL=np.array([np.max(A[i,:]) for i in range(np.shape(A)[0])])* *PGEL* *array([2, 4, 6])*
Plus petit élément (ppe) d'une matrice	*min(M)*	*import numpy as np* *M = np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *ppe = np.min(M)* *ppe* *-1*
Plus petit élément sur chaque colonne d'une matrice (tableau PPEC)	*Formule avec min()*	*import numpy as np* *M = np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* # *Utilisation de l'itérateur range()* *PPEC=np.array([np.min(A[:,i]) for i in range(np.shape(A)[1])])* *PPEC* *array([-1, 2, 0])*

Description	**Syntaxe**	**Librairie / Remarque / Exemple**
Plus petit élément sur chaque ligne d'une matrice (tableau PPEL)	*Formule avec min()*	*import numpy as np* *M = np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *# Utilisation de l'itérateur range()* *PPEL=np.array([np.min(A[i,:]) for i in range(np.shape(A)[0])])* *PPEL* *array([0, 0, -1])*
Produit matriciel de deux matrices M1 et M2	*dot(M1, M2)*	*import numpy as np* *# $M1_{(n, m)}$ et $M2_{(m, l)}$* *M1=np.array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *M2=np. array([[0, -1], [-3, -3], [1, 1]])* *P = np.dot(M1, M2)* *P* *array([[-6, -7],* *[-5, -5],* *[-9, -8]])*
Produit de deux vecteurs	*dot(V1,V2)*	*import numpy as np* *# $M1_{(n, m)}$, $M2_{(m, l)}$, V1 et V2 ont le même nombre d'éléments* *M1=np.array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *M2=np. array([[0, -1], [-3, -3], [1, 1]])* *V1=M1[0,:]* *V2=M2[:,0:1]* *a=float(np.dot(V1,V2))* *V1, V2, v* *(array([1, 2, 0]),* *array([[0],* *[-3],* *[1]]),* *array([-6]))*

Description	Syntaxe	Librairie / Remarque / Exemple
Puissance matricielle (M^n où n entier positif)	*matrix_power(M,k)*	*import numpy as np* *import numpy.linalg la* *M=np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *M0 = la.matrix_power(M, 0)* *M0* *array([[1, 0, 0],* *[0, 1, 0],* *[0, 0, 1]])* *M4 = la.matrix_power(M, 4)* *M4* *array([[-87, 584, 744],* *[-372, 2357, 3028],* *[-571, 3599, 4628]])* *if la.det(M) != 0 :* *MI = la.matrix_power(M, -1)* *print("Matrice inverse :\n",* *la.matrix_power(M, -1))* *Matrice inverse :* *[[0.2 1.2 -0.8]* *[0.4 -0.6 0.4]* *[-0.3 0.7 -0.3]]*
Rang d'une matrice $M_{(n,m)}$	*matrix_rank(M)*	*# Le nombre de lignes linéairement indépendantes* *import numpy as np* *import numpy.linalg as la* *M=np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *print(la.matrix_rank(M))* *3* *N=np. array([[1, 2, 0], [0, 3, 4], [-2, -4, 0], [0, -6, -8]])* *print(la.matrix_rank(N))* *2*

Description	Syntaxe	Librairie / Remarque / Exemple
Résolution de A.X=B	*solve()*	*import numpy as np* *import numpy.linalg as la* *# $A_{(n,n)}$; $X_{(n,1)}$ et $B_{(n,1)}$* *A=np.array([[0,-1,3],[2,-2,0],[1,0,-1]],dtype=float)* *B=np.array([[2],[1],[1]])* *X=la.solve(A,B)* *X* *array([[2.25],* *[1.75],* *[1.25]])*
Somme de tous les éléments d'une matrice M	*sum(M)*	*import numpy as np* *M=np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *print("total = ",np.sum(M))* *total = 20*
Somme des colonnes d'une matrice	*Formule avec les fonctions sum() et range()*	*import numpy as np* *M=np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *SC = np.array([sum(M[i,:] for i in range(np.shape(M)[1]))])* *SC* *array([[0, 10, 10]])*
Somme des lignes d'une matrice M	*Formule avec les fonctions sum() et range()*	*SL=np.array([sum(M[:,i] for i in range(np.shape(M)[0]))])* *SL* *array([[3, 7, 10]])*
Sous-matrice	*M[i:i',j:j']*	*# Les lignes indices i à (i'-1) et les colonnes indices j à (j'-1)* *import numpy as np* *M= np. array([[1, 2, 0], [0, 3, 4], [-1, 5, 6]])* *M[1:3,0:2]* *array([[0, 3],* *[-1, 5]])*

Description	**Syntaxe**	**Librairie / Remarque / Exemple**
Supprimer une colonne donnée d'une matrice M	*delete(M, col, axis=1)*	*# Sauvegarder, si besoin, la matrice initiale avant le delete()* *# Exemple : matrice obtenue par suppression de la colonne indice 1 de la matrice M* *M = np.array([[0,-1,3],[2,-2,0],[4,0,-1],[4,2,1]], dtype=float)* *MC = np.delete(M, 1, axis=1)* *MC* *array([[0., 3.],* *[2., 0.],* *[4., -1.],* *[4., 1.]])*
Supprimer une ligne donnée d'une matrice M	*delete(M, obj, axis=0)*	*# Sauvegarder, si besoin, la matrice initiale avant le delete()* *# Exemple : matrice obtenue par suppression de la ligne indice 1 de la matrice M* *M = np.array([[0,-1,3],[2,-2,0],[4,0,-1],[4,2,1]],dtype=float)* *ML = np.delete(M, 1, axis=0)* *ML* *array([[0., -1., 3.],* *[4., 0., -1.],* *[4., 2., 1.]])*
Trace d'une matrice	*trace(M)*	*import numpy as np* *# La somme des éléments de la diagonale principale d'une matrice carrée $M_{(n,n)}$* *M = np.array([[0,-1,3],[2,-2,0],[4,0,-1],[4,2,1]],dtype=float)* *t=np.trace(M)* *t* *-3*

Description	**Syntaxe**	**Librairie / Remarque / Exemple**
Transposée d'une matrice A	*transpose(A)*	*import numpy as np* *# $At_{(m, n)}$ est la transposée de $A_{(n, m)}$* *A = np.array([[0., -1., 3.], [2., -2., 0.], [4., 0., -1.], [4., 2., 1.]])* *print(np.transpose(A))* *[[0., 2., 4., 4.],* *[-1., -2., 0., 2.],* *[3., 0., -1., 1.]]*
Tri des éléments d'une matrice M	*sort(M)*	*import numpy as np* *# Le tri est par défaut croissant* *# Les éléments de chaque ligne sont triés par ordre croissant* *M = np.array([[1, 2, 0], [3, 0, 4], [5, -1, 6]])* *M1 = np.sort(M)* *M1* *array([[0, 1, 2],* *[0, 3, 4],* *[-1, 5, 6]])* *# Idem en ordre décroissant* *M1 = np.sort(M)[:,::-1]* *M1* *array([[2, 1, 0],* *[4, 3, 0],* *[6, 5, -1]])* *# Les éléments de chaque colonne sont triés par ordre croissant* *M2 =np.sort(M, axis=0)* *M2* *array([[1, -1, 0],* *[3, 0, 4],* *[5, 2, 6]])*

Description	Syntaxe	Librairie / Remarque / Exemple
Tri des éléments d'une matrice M (suite)	*sort(M)*	*# Idem en ordre décroissant* *M2 = np.sort(M, axis=0)[::-1]* *M2* *array([[5, 2, 6],* *[3, 0, 4],* *[1, -1, 0]])*
Valeurs et vecteurs propres d'une matrice	*eig(M)*	*import numpy as np* *import numpy.linalg as la* *# V contient les vecteurs propres (vecteurs colonnes) et l les valeurs propres associées* *M = np. array([[1, 2, 0], [3, 0, 4], [5, -1, 6]])* *l, V=la.eig(M)* *# Exemple : le vecteur V[:,0] et valeur propre associée l[0]* *l[0]* *9.0* *V[:,0]* *array([-0.89442719, 0.4472136])* *# NB. la.eigvals(M) retourne uniquement les valeurs propres de M*
Vecteur colonne	*M[:,j]*	*import numpy as np* *M = np. array([[1, 2, 0], [3, 0, 4], [5, -1, 6]])* *# La colonne indice j=1 de M* *M[:,1]* *array([2, 0, -1])*
Vecteur ligne	*M[i,:]*	*import numpy as np* *M = np. array([[1, 2, 0], [3, 0, 4], [5, -1, 6]])* *# La ligne indice i = 1 de M* *M[1,:]* *array([3, 0, 4])*

5.2 Résolution de A.X = B par pivot de Gauss

Ici on va développer les éléments algorithmiques puis implémenter une méthode de résolution d'un système d'équations à n équations et n inconnues où $n \in \mathbb{N}^*$. Puis on comparera les résultats obtenus par le développement de cet algorithme avec les résultats obtenus en appliquant les méthodes de la librairie *numpy* et sa sous-librairie d'algèbre linéaire *linalg*.

5.2.1 Méthode de résolution

On utilise la méthode de Gauss appelée aussi méthode du pivot de Gauss pour transformer un système de Cramer A.X = Y en un système équivalent, par transvection et échange, $A'.X = Y'$ où A' est une matrice triangulaire supérieure.

Soient :

$A \in \mathcal{M}_{n,n}(\mathbb{R}), X \in \mathcal{M}_{n,1}(\mathbb{R})\ et\ Y \in \mathcal{M}_{n,1}(\mathbb{R})$

Système de Cramer :

(S) :AX=Y

$$(S) : \begin{cases} L_1 \\ \dots \\ L_n \end{cases}$$

Les étapes de transvection et échange

— on choisit un pivot non nul (exemple : le pivot dont la valeur absolue est la plus grande) dans la première colonne, sur la ligne i ;

— on échange la première et la i ligne ;

— On effectue les opérations suivantes sur les lignes $2 \leq k \leq n$:

$L_k = L_k - \frac{a_{k,1}}{a_{1,1}} L_1$;

— on passe à la colonne suivante.

À la colonne i :

— on choisit un pivot non nul dans la colonne i dans une ligne d'indice supérieur ≥ i ;

— on place le pivot sur la ligne i en effectuant si besoin un échange de lignes ;

— on effectue les opérations suivantes sur les lignes $i+1 \leq k \leq n$:
$L_k = L_k - \frac{a_{k,i}}{a_{i,i}} L_i$.

On obtient un système triangulaire à diagonale principale :

$(S^*) \Leftrightarrow (S)$
$(S^*) : A'X = Y'$

$A' = \begin{pmatrix} & \cdots & \neq 0 \\ \vdots & \ddots & \vdots \\ = 0 & \cdots & \end{pmatrix}$ Matrice triangulaire supérieure.

Algorithme de transvection

Pour j = 0 à n-2 faire
  Trouver k entre j et n-1 tel que $|a_{k,j}|$ *est max* > 0
  Echanger L_k *et* L_j
  Pour i = j+1 à n-1 faire
    $L_i = L_i - \frac{a_{i,j}}{a_{j,j}} L_j$
  Fin pour
Fin pour

La remontée

On obtient X par remontée : on remonte de la dernière ligne à la première. En remontant, à chaque ligne *i* il y a une seule inconnue x_i telle que :

$$x_i = \frac{1}{a_{i,i}} \left(y_i - \sum_{j=i+1}^{n-1} a_{i,j} x_j \right)$$

Avant la remontée, on peut continuer avec l'adaptation Gauss-Jordan pour transformer le système $A'.X = Y'$ en un système équivalent $A".X = Y"$ où $A"$ est une

matrice diagonale ne comporte que des 1 (c'est-à-dire $A^{"} = I_n$ et dans ce cas la solution est immédiate, c'est-à-dire $X = Y^{"}$.

5.2.2 Script de résolution

```
import numpy as np

# Fonctions et procédures utiles
# Procédure échange deux lignes
def echange(A, i, j):
    # Permutation mutuelle des lignes A[i, :] et A[j, :]
    # en utilisant un vecteur ligne temporaire L
    L=np.copy(A[i,:])
    A[i, :] = A[j, :]
    A[j, :] = L

# Procédure de transvection
def transvection(A, i, j, mu):
    # Remplacement de la ligne indice i de la matrice A i.e. A[i,:]
    # par A[i, :] + mu*A[i, :]
    A[i, :] += mu * A[j, :]

# Procédure ligne équivalente
def Ligne_Equivalente(A, i, r):
    # La ligne A[i, :] est remplacée par  r*A[i, :]
    A[i, :] *= r

# Fonction recherche indice de pivot
def indice_pivot(A, j):
    # Rechercher l'indice k de la ligne de A
    # parmi les lignes qui suivent la ligne courante j
    # pour laquelle la valeur absolue |A[k,j]|
    # soit le max(|A[k,j]|) pour tout k > j
    indice = j
    for k in range(j + 1, A.shape[0]):
        if abs(A[k, j]) > abs(A[indice, j]):
```

```
            indice = k
    print("\nIndice de Pivot : ",indice,"\n")
    return indice

# Fonction résolution A.X=Y avec triangularisation
# puis diagonalisation de A
def pivot(A, Y):
    # A doit être une matrice (carrée) inversible
    n = A.shape[0]
    m = A.shape[1]
    # la suite n'est possible que si A est une matrice carrée
    assert n == m

    # création la matrice C=(A | Y) par concaténation de A et Y
    # les éléments de la nouvelle matrice C doivent être aussi
    # des réels de type float
    C = np.concatenate((A.astype(float), Y.astype(float)), axis=1)
    print("Matrice C=(A | Y) initiale : \n",C)

    # Traitement
    for j in range(n - 1):
        k = indice_pivot(C, j)
        if k != j:
            echange(C, j, k)
        # pas de else car k = j le pivot est déjà trouvé et donc ne
        # fait pas l'échange
        for i in range(j + 1, n):
            # la suite n'est pas possible s'il y a une division par 0
            assert C[j, j] != 0
            mu = - C[i, j] / C[j, j]
            transvection(C, i, j, mu)
        print("Matrice C=(A | Y) intermédiaire : \n",C)

    # Lignes équivalentes pour lesquelles les coefficients
     #  diagonaux correspondant à 1.
```

```
    for i in range(n):
        Ligne_Equivalente(C, i, 1 / C[i, i])

    # Matrice triangulaire supérieure
    print("\nMatrice C=(A | Y) finale telle que A est triangulaire
supérieure : \n",C)

    # Obtention de C finale telle A est diagonale
    for j in range(n - 1, 0, -1):
        for i in range(j):
            transvection(C, i, j, -C[i,j])
    print("\nMatrice C=(A | Y) finale telle que A est diagonale : \n",C)

    # X final est dans la dernière colonne de C
    X=np.array(C[:, n:],float)
    return X

# PROGRAMME PRINCIPAL
A=np.array([[2,1,-4],[3,3,-5],[4,5,-2]],float)
print("Matrice A initiale : \n", A)
print("Taille de la matrice A : ",A.shape)
Y=np.array([[8],[14],[16]],float)
print("Vecteur Y initial : \n", Y)
print("\nRésolution de A.X=Y par la méthode du Pivot de Gauss \n" )
print("Solution X = \n",pivot(A, Y))
# Comparaison avec le solveur de numpy.linalg
print("\n\nSolution obtenue par le solveur de linalg :
\n",np.linalg.solve(A,Y))
```

5.2.3 Exemple

```
Matrice A initiale :
 [[ 2.  1. -4.]
 [ 3.  3. -5.]
 [ 4.  5. -2.]]
```

```
Taille de la matrice A :  (3, 3)
Vecteur Y initial :
[[ 8.]
[14.]
[16.]]

Résolution de A.X=Y par la méthode du Pivot de Gauss

Matrice C=(A | Y) initiale :
[[ 2.  1. -4.  8.]
[ 3.  3. -5. 14.]
[ 4.  5. -2. 16.]]

Indice de Pivot :  2

Matrice C=(A | Y) intermédiaire :
[[ 4.    5.   -2.   16.  ]
[ 0.   -0.75 -3.5   2.  ]
[ 0.   -1.5  -3.    0.  ]]

Indice de Pivot :  2

Matrice C=(A | Y) intermédiaire :
[[ 4.   5.  -2.  16. ]
[ 0.  -1.5 -3.   0. ]
[ 0.   0.  -2.   2. ]]

Matrice C=(A | Y) finale telle que A est triangulaire supérieure :
[[ 1.    1.25 -0.5   4.  ]
[-0.    1.    2.   -0.  ]
[-0.   -0.    1.   -1.  ]]

Matrice C=(A | Y) finale telle que A est diagonale :
[[ 1.  0.  0.  1.]
[ 0.  1.  0.  2.]
```

```
 [-0. -0.  1. -1.]]
Solution X =
 [[ 1.]
 [ 2.]
 [-1.]]

Solution obtenue par le solveur de linalg :
 [[ 1.]
 [ 2.]
 [-1.]]
```

5.3 Calcul du déterminant et matrice inverse

Ici on va développer les éléments algorithmiques puis implémenter la méthode de calcul du déterminant d'une matrice basée sur les mineurs et les cofacteurs. Puis on comparera les résultats obtenus par le développement de cet algorithme avec les résultats obtenus en appliquant des méthodes de la sous-librairie d'algèbre linéaire *numpy.linalg*.

5.3.1 Méthode de calcul

Soient $n \in \mathbb{N}^*$ et M une matrice de réels d'ordre n : $M \in \mathcal{M}_{n,n}(\mathbb{R})$.

Notons également :

- i : indice des lignes (i = 0, 1, …, n-1) ;
- j : indice des colonnes (j = 0, 1, …, n-1) ;
- $M = M_{n,n} = \left(m_{i,j}\right)_{(n,n)}$;
- M^{-1} : matrice inverse de M (si elle existe) ;
- $M_{i,j}$: est la matrice obtenue à partir de *M* sans la ligne i et sans la colonne j ;
- $\det(M_{i,j})$: est le déterminant de la matrice $M_{i,j}$. C'est le mineur du couple (i, j) ou encore le mineur du terme $m_{i,j}$ de la matrice M ;
- $c_{i,j}$: le cofacteur du couple (i, j) et $c_{i,j} = (-1)^{i+j} det\left(M_{i,j}\right)$;
- I_n : matrice identité d'ordre n.

Le déterminant

Le déterminant de *M* selon la ligne i est calculé comme suit :

$$det(M) = \sum_{j=0}^{n-1} m_{i,j}(-1)^{i+j} det\left(M_{i,j}\right) = \sum_{j=0}^{n-1} m_{i,j} c_{i,j}$$

Dans l'implémentation Python suivante, on calculera le déterminant de *M* selon la ligne indice 0 :

$$det(M) = \sum_{j=0}^{n-1} m_{0,j}(-1)^{j} det\left(M_{0,j}\right) = \sum_{j=0}^{n-1} m_{0,j} c_{0,j}$$

La comatrice

La comatrice de M est la matrice de ses cofacteurs :

$$\text{Comat}(M)=(C_{i,j})_{(n,\,n)}$$

La matrice inverse

$$det(M) \neq 0 \iff M \text{ est inversible et son inverse } M^{-1}$$
$$= \frac{1}{\det(M)}\ {}^{t}\left(Comat(M)\right)$$

Propriété de la matrice inverse :

$$M.M^{-1} = M^{-1}.M = I_n$$

5.3.2 Script de calcul

```
import numpy as np
import os

# Fonction déterminant matrice (2,2)
def detM22(A):
    d=(A[0][0]*A[1][1])-(A[0][1]*A[1][0])
    return d

# Fonction déterminant matrice (1,1)
def detM11(A):
    d=A[0][0]
    return d
```

```
# Fonction sous-matrice de A sans la ligne indice i et la colonne indice j
def sousMat(A,i,j):
    n=A.shape[0]
    m=A.shape[1]
    SA=np.zeros((n-1,m-1),dtype=float)
    # Parcours en Z des blocs
    #Copie bloc 1
    for k in range(i):
        for l in range(j):
            SA[k][l]=A[k][l]
    #Copie bloc 2
    for k in range(i):
        for l in range(j+1,m):
            SA[k][l-1]=A[k][l]
    #Copie bloc 3
    for k in range(i+1,n):
        for l in range(j):
            SA[k-1][l]=A[k][l]
    #Copie bloc 4
    for k in range(i+1,n):
        for l in range(j+1,n):
            SA[k-1][l-1]=A[k][l]
    return(SA)

# Fonction Cofacteur
def cofac(A,i,j):
    SA=sousMat(A,i,j)
    c=(-1)**(i+j)*detMat(SA)
    return(c)

# Fonction déterminant de la matrice A, fait appel à cofac()
# qui elle-même fait appel à nouveau de detMat()
def detMat(A):
    n=A.shape[0]
```

```
    m=A.shape[1]
    if n==1:
        d=detM11(A)
    elif n==2:
        d=detM22(A)
    else:
        d=0
        # Développement selon la ligne indice 0 i.e. la 1ère ligne
        for l in range(m):
            d=d+A[0][l]*cofac(A,0,l)
    return(d)

# Les fonctions suivantes pour le calcul de l'inverse de A
# Matrice des cofacteurs de A
def Comat(A):
    n=A.shape[0]
    m=A.shape[1]
    CO=np.zeros((n,m),dtype=float)
    for i in range(n):
        for j in range(m):
            CO[i][j]=cofac(A,i,j)
    return(CO)

# Matrice transposée de A
def Trans(A):
    n=A.shape[0]
    m=A.shape[1]
    T=np.zeros((n,m),dtype=float)
    for i in range(n):
        for j in range(m):
            T[i][j]=A[j][i]
    return(T)

# Script principal
# Calcule le déterminant puis la matrice inverse quand c'est possible
```

```
# Changement du répertoire
os.chdir("C:/ Documents/Fichiers data ")

# Importation de la matrice test
A=np.loadtxt("Matrice4x4.txt",delimiter=",",dtype=float)
n=A.shape[0]
m=A.shape[1]
if n==m:
   print("Taille de A : (",n,",",m,")")
   print("Matrice A de départ : \n",A)
   d=detMat(A)
   print("Le déterminant de A = ",d)
   if d!=0:
      CO=Comat(A)
      print("La matrice des cofacteurs de A est : \n",CO)
      np.savetxt("Comatrice5x5.txt",CO,delimiter=",")
      invA=(1/d)*Trans(Comat(A))
      print("La matrice inverse de A est :\n",invA)
      np.savetxt("Inverse5x5.txt",invA,delimiter=",")
      print("Vérification A*invA=Id -- > \n",np.dot(A,invA))
   else:
      print("Matrice non inverssible")
else:
   print("Ce n'est pas une matrice carrée !")
```

5.3.3 Exemples

```
# 1er exemple

Taille de A : ( 4 , 4 )
Matrice A de départ :
 [[ 2.  5. -3. -2.]
 [-2. -3.  2. -5.]
```

```
 [ 1.  3. -2.  2.]
 [-1. -6.  4.  3.]]
Le déterminant de A =  -4.0
La matrice des cofacteurs de A est :
 [[  0.  -8. -12.  -0.]
 [  7. -17. -23.  -1.]
 [ 13. -31. -41.  -3.]
 [  3. -13. -19.  -1.]]
La matrice inverse de A est :
 [[-0.   -1.75 -3.25 -0.75]
 [ 2.    4.25  7.75  3.25]
 [ 3.    5.75 10.25  4.75]
 [ 0.    0.25  0.75  0.25]]
Vérification A*invA=Id -- >
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
# 2ème exemple
Taille de A : ( 5 , 5 )
Matrice A de départ :
 [[ 1.  0.  0.  0.  1.]
 [ 0.  1.  0.  0.  2.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  0.  0.  1. -1.]
 [ 0.  0.  0.  0. -1.]]
Le déterminant de A =  -1.0
La matrice des cofacteurs de A est :
 [[-1. -0.  0. -0.  0.]
 [-0. -1. -0.  0. -0.]
 [ 0. -0. -1. -0.  0.]
 [-0.  0.  1. -1. -0.]
 [-1. -2. -1.  1.  1.]]
La matrice inverse de A est :
 [[ 1.  0. -0.  0.  1.]
```

```
[ 0.  1.  0. -0.  2.]
[-0.  0.  1. -1.  1.]
[ 0. -0.  0.  1. -1.]
[-0.  0. -0.  0. -1.]]
Vérification A*invA=Id -- >
[[1. 0. 0. 0. 0.]
[0. 1. 0. 0. 0.]
[0. 0. 1. 0. 0.]
[0. 0. 0. 1. 0.]
[0. 0. 0. 0. 1.]]
```

5.4 Suites de matrices et suites numériques

Cet exercice permet de manipuler des matrices, des suites de matrices et des suites numériques.

5.4.1 Énoncé

Soit (A_m) une suite de matrices carrées de réels d'ordre 3 définie comme suit :

$$\forall\, m \in \mathbb{N} :$$

$$A_m = \begin{pmatrix} (-1)^m(2^m) & 0 & 1 \\ 0 & (-1)^{m+1}(2^m+1) & 0 \\ 1 & 0 & (-1)^m(2^m-1) \end{pmatrix}$$

Données de l'exercice :

1. supposons que m un entier positif déjà connu, écrire une fonction *TerMat(A, m)* qui calcule et retourne à l'algorithme appelant une matrice A carrée d'ordre 3 telle que $A=A_m$;
2. soient *B* et *C* deux matrices carrées de réels toutes deux d'ordre 3. Supposons que B et C sont connues et qu'on dispose de la fonction *ProdMat(B, C)* qui calcule et retourne à l'algorithme appelant une matrice *P* carrée d'ordre 3 telle que $P = B.C$;
3. supposons que *n* un entier strictement positif déjà connu et *A* la matrice carrée d'ordre 3 telle que $A=A_m$, écrire une fonction **récursive** *PuisMat(A, n)* qui calcule et renvoie à l'algorithme appelant A^n ;
4. écrire le script principal qui permet de saisir *m* un entier positif et *n* un entier strictement positif et de calculer et d'afficher A_m^n .

Pour la suite de l'exercice et pour le calcul de A_m, on fixe $m = 2$. On essayera d'autres tests avec $m > 2$.

Soient les suites numériques suivantes (u_h), (v_h) et (w_h) telles que :

$u_0 = 1$, $v_0 = -2$, $w_0 = 3$

et pour $h \in \mathbb{N}^*$

$$\begin{aligned} u_h &= 4u_{h-1} + 0 + w_{h-1} \\ v_h &= 0 + -5v_{h-1} + 0 \\ w_h &= u_{h-1} + 0 + 3w_{h-1} \end{aligned}$$

Écrire un algorithme (bloc de traitement) qui pour h données, calcule et affiche les termes u_h, v_h et w_h.

Écriture matricielle

$$A = A_2 = \begin{pmatrix} 4 & 0 & 1 \\ 0 & -5 & 0 \\ 1 & 0 & 3 \end{pmatrix}$$

et

$$\begin{pmatrix} u_0 \\ v_0 \\ w_0 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix} et \begin{pmatrix} u_h \\ v_h \\ w_h \end{pmatrix} = \begin{pmatrix} 4 & 0 & 1 \\ 0 & -5 & 0 \\ 1 & 0 & 3 \end{pmatrix} . \begin{pmatrix} u_{h-1} \\ v_{h-1} \\ w_{h-1} \end{pmatrix} = A . \begin{pmatrix} u_{h-1} \\ v_{h-1} \\ w_{h-1} \end{pmatrix}$$

$$\begin{pmatrix} u_h \\ v_h \\ w_h \end{pmatrix} = A . \begin{pmatrix} u_{h-1} \\ v_{h-1} \\ w_{h-1} \end{pmatrix} = A^2 . \begin{pmatrix} u_{h-2} \\ v_{h-2} \\ w_{h-2} \end{pmatrix} = \cdots = A^h . \begin{pmatrix} u_0 \\ v_0 \\ w_0 \end{pmatrix} = A^h . \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}$$

5.4.2 Script de résolution

```
import numpy as np

# Fonction TerSuites(...)
def TerSuites(A,h,S):
    return(np.dot(PuisMat(A,h),S))

#Contôle de saisie un entier >=e
def Saisie_E(e):
```

```
    v=input("saisir : ")
    while(not v.isnumeric() or int(v)<e):
        v=input("Recommencer : ")
    v=int(v)
    return(v)

# Fonction TerMat(...)
def TerMat(A,m):
    for i in range(3):
        for j in range(3):
            if i==j:
                if i==0:
                    A[i][j]=((-1)**m)*(2**m)
                elif i==1:
                    A[i][j]=((-1)**(m+1))*((2**m)+1)
                else:
                    A[i][j]=((-1)**m)*((2**m)-1)
            elif (i==0 and j==2) or (i==2 and j==0):
                A[i][j]=1
            else:
                A[i][j]=0
    return(A)

# Fonction PuisMat(...)
def PuisMat(A,n):
    if n==1:
        return(A)
    else:
        return(np.dot(PuisMat(A,n-1),A))

# Script principal
print("\nSaisie de m >=0 ")
m=Saisie_E(0)
print("\nSaisie de n >= 1\n")
n=Saisie_E(1)
```

```
A=np.zeros((3,3))
P=np.zeros((3,3))
A=TerMat(A,m)
print(" A = ","\n",A,"\n")
P=PuisMat(A,n)
print("A puissance ",n," = ","\n",P, "\n")
print("Saisie de h >= 1")
h=Saisie_E(1)
S=np.array([[1],[-2],[3]])
print("Termes initiaux : ","\n",S, "\n")
S=TerSuites(A,h,S)
print("S final = ","\n",S)
```

5.4.3 Exemples

```
#Exemple 1 : m=2, n=3 et h=3
Saisie de m >=0
saisir : 2

Saisie de n >= 1
saisir : 3
 A =
 [[ 4.  0.  1.]
 [ 0. -5.  0.]
 [ 1.  0.  3.]]

A puissance  3  =
 [[  75.    0.   38.]
 [   0. -125.    0.]
 [  38.    0.   37.]]

Saisie de h >= 1
saisir : 3
Termes initiaux u0, v0 et w0 :
 [[ 1]
```

```
 [-2]
 [ 3]]

Termes finaux uh, vh et wh
 [[189.]
 [250.]
 [149.]]

#Exemple 2 : m=3, n=3 et h=3
Saisie de m >=0
saisir : 3

Saisie de n >= 1
saisir : 3
 A =
 [[-8.  0.  1.]
 [ 0.  9.  0.]
 [ 1.  0. -7.]]

A puissance  3  =
 [[-535.    0.  170.]
 [   0.  729.    0.]
 [ 170.    0. -365.]]

Saisie de h >= 1
saisir : 3
Termes initiaux u0, v0 et w0 :
 [[ 1]
 [-2]
 [ 3]]

Termes finaux uh, vh et wh
 [[  -25.]
 [-1458.]
 [ -925.]]
```

```
#Exemple 2 : m=4, n=3 et h=5
Saisie de m >=0
saisir : 4

Saisie de n >= 1
saisir : 3

 A =
[[ 16.   0.   1.]
[  0. -17.   0.]
[  1.   0.  15.]]

A puissance  3  =
 [[ 4143.     0.   722.]
 [    0. -4913.     0.]
 [  722.     0.  3421.]]

Saisie de h >= 1
saisir : 5

Termes initiaux u0, v0 et w0 :
 [[ 1]
 [-2]
 [ 3]]

Termes finaux uh, vh et wh
 [[1961948.]
 [2839714.]
 [2678189.]]
```

Bibliographie

OUVRAGES

Méthodes mathématiques pour l'informatique.
Jacques Vélu. Éditions Dunod, 1994.

Apprendre à programmer avec Python 3.
Gérard Swinnen, Éditions Eyrolles, 3ème édition, 2012.

Python for Data Analysis.
Wes McKinney, O'REILLY, 2012.

Informatique pour tous en classes préparatoires et aux grandes écoles.
Benjamin Wack *et al*. Éditions Eyrolles, 2013.

Apprenez à programmer en Python.
Vincent Le Goff. Éditions Eyrolles, 2014.

Doing Math with Python: Use Programming to Explore Algebra, Statistics, Calculus, and More.
Amit Saha. Edition No Starch Press, Inc., 2015.

Machine Learning with Python.
Tutorials Point, 2019.

Mise en œuvre des probabilités et des statistiques - Cours, exercices et problèmes de synthèse corrigés, programmation en Matlab et Python.
Mansour Ali, Osswald Christophe. Éditions Ellipses, 2019.

Programmation en Python pour les mathématiques.
Alexandre Casamayou-Boucau, Pascal Chauvin, Guillaume Connan. Éditions Dunod, 2022.

Algorithmique et développement Python - Cours et exemples d'applications.
Abderrahmane FADIL. Éditions Ellipses, 2021.

Statistique et probabilités - Exercices d'application et problèmes corrigés avec rappels de cours.

Abdesselam Rafik. Éditions Ellipses, 2021.

Algèbre et calcul formel.

Fleury Odile, Foissy Loic, Ninet Alain. Éditions Ellipses, 2023.

WEBOGRAPHIE

https://www.Python.org/

https://www.developpez.net/

https://josephsalmon.eu/Courses/enseignement/Montpellier/HLMA408/StatDescriptives.pdf

Index

M

N

P

S

T

U

V