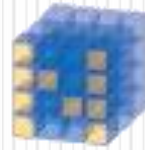


Manipulation des tableaux

Modules : NumPy, PIL, Matplotlib, SciPy

Les fichiers

Module NumPy



Vecteurs, matrices et tableaux

Documentation : <https://numpy.org/>

Introduction

- ❑ numpy est un package pour Python spécialisé dans la manipulation des tableaux (array), pour nous essentiellement les vecteurs et les matrices
- ❑ Les tableaux « numpy » ne gère que les objets de même type Les tableaux « numpy » sont plus performants (rapidité, gestion de la volumétrie) que les collections usuelles de Python
- ❑ Les tableaux « numpy » sont sous-jacents à de nombreux packages dédiés au calcul scientifique sous Python.
- ❑ Un vecteur est un tableau (array) à une seule dimension
- ❑ Une matrice est un tableau (array) à 2 dimensions
- ❑ Une image couleur RGB est un tableau à 3 dimensions
- ❑ Préalable important : importer le module « numpy » :

```
import numpy as np
```

Création d'un tableau (array)

création d'un vecteur V à partir d'une liste

`V = np.array( [ 2.1 , 3.6 , 7.2 ] )`

```
[ 2.1  3.6  7.2]
```

création d'une matrice A

`A = np.array([ [ 2 , 6 , 5] , [8 , 1 , 0] ] )`

```
[[2 6 5]
 [8 1 0]]
```

X : vecteur de 0 à 9

`X = np.arange(0,10)`

```
[0 1 2 3 4 5 6 7 8 9]
```

B : matrice 2 lignes , 5 colonnes

`B = np.arange(0,10).reshape(2,5)`

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

Y : vecteur de 10 réels

`Y = np.linspace( 0, 4.5 ,10 )`

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
```

C : matrice 3 lignes , 2 colonnes remplis de zéros

`C = np.zeros( (3,2) )`

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

I : matrice identité

`I = np.eye(3)`

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

```
[[5 0 0]
 [0 7 0]
 [0 0 2]]
```

M : matrice diagonale

`M = np.diag( [5 , 7 , 2] )`

Accès et slicing

a =

[	[	1.2	2.5]	]
[	3.2	1.8]	]	
[	1.1	4.3]	]	

#matrice de valeurs

```
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

L'accès à une valeur dans une matrice 2D passe par deux indices (ligne et colonne),

les lignes et les colonnes commencent de l'indice 0

```
print(a[0,0])
```

1.2

```
print(a[0][0])
```

Slicing : [début : fin+1 : { pas=1}]

#extraire la ligne d'indice 1

```
L1 = a[1]
```

```
L1 = a[1, :]
```

```
print(L1)
```

[	3.2	1.8]
---	-----	------

Pour extraire une ligne, soit on donne seulement l'indice de la ligne, soit on donne l'indice de la ligne et la plage des colonnes

Pour extraire une colonne entière, on passe la plage de toute les ligne et l'indice de la colonne.

#extraire la colonne d'indice 1

```
C1 = a[:, 1]
```

```
print(C1)
```

[	2.5	1.8	4.3]
---	-----	-----	------

Pour extraire une matrice bloc, On passe la plage de ligne et la plage de colonne

#extraire une matrice bloc

```
b = a[1:3, 0:2]
```

```
c = a[:, 0:2]
```

```
print(b)
```

```
print(c)
```

[	[	3.2	1.8]	]
[	1.1	4.3]	]	

[	[	1.2	2.5]	]
[	1.1	4.3]	]	

Parcours

a =

[	[	1.2	2.5]	]
[	3.2	1.8]	]	
[	1.1	4.3]	]	

#matrice de valeurs

```
a = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
```

Les propriétés des matrice sont :

- Taille : size <int>
- Dimension : ndim <int>
- Forme : shape <tuple>

```
print( a.size)
```

6

```
print(a.ndim)
```

2

pour récupérer la forme, il y a 2 façon de faire :

```
print(a.shape)
```

```
print(np.shape(a))
```

(3, 2)

Avec les indices, nous pouvons accéder aux valeurs de la matrice (ligne par ligne ou colonne par colonne) avec deux boucles imbriquées

#parcours de tous les éléments d'une matrice :

```
k = 1
```

```
for i in range(a.shape[0]) :
```

```
    for j in range(np.shape(a)[1]) :
```

```
        print(k , ' : ', a[i,j])
```

```
        k += 1
```

1	:	1.2
2	:	2.5
3	:	3.2
4	:	1.8
5	:	1.1
6	:	4.3

Calcul matriciel (1)

a =

```
[[4 8 3 1]
 [1 0 5 3]
 [4 2 0 7]]
```

b =

```
[[ 2 -2 5 3]
 [-2 6 9 5]
 [ 5 9 0 9]]
```

a est une matrice d'entier

```
a = np.array([[4,8,3,1],[1,0,5,3],[4,2,0,7]] , dtype = int)
```

b est une matrice aléatoire d'entier

```
b = np.random.randint(-2,high=10,size=(3,4),dtype = int)
```

*Les opérations arithmétiques(+ , -
* , / , etc) entre matrices sont
faites terme à termes*

```
s = a + b
print(s)
p = a * b
print (p)
```

s =

```
[[ 6  6  8  4]
 [-1  6 14  8]
 [ 9 11  0 16]]
```

p =

```
[[ 8 -16 15  3]
 [-2  0 45 15]
 [20 18  0 63]]
```

*Les fonctions sin , cos , ... du
module numpy sont vectorisées*

```
C = np.cos( a )
print(np.round(C,2) )
```

```
[[-0.65 -0.15 -0.99  0.54]
 [ 0.54  1.    0.28 -0.99]
 [-0.65 -0.42  1.    0.75]]
```

*les calculs sont réalisés selon un
axe*

- 0 : traitement des
valeurs en ligne
- 1 : traitement des
valeurs en colonne

la sommes des valeurs de a

```
print (np.sum(a) )
```

38

la moyenne des lignes pour chaque colonne

```
moy = np.mean(a, axis = 0 )
print(np.round(moy,2))
```

```
[ 3.    3.33  2.67  3.67]
```

le maximum des colonnes pour chaque ligne

```
print(np.max(a , axis = 1 ) )
```

```
[8 5 7]
```

Calcul matriciel (2)

```
A =  
[[1 1 0 3]  
 [1 4 4 1]  
 [3 1 1 4]  
 [4 4 1 2]]
```

A est une matrice carrée

```
A = np.random.randint(0,high=5,size=(4,4),dtype = int)
```

*calcul de la matrice transposée
à l'aide de la fonction
transpose ou à l'aide de la
propriété T*

#ta : est la matrice transposée de A

```
ta = np.transpose(A)
```

```
ta = A.T
```

```
print(ta)
```

```
[[1 1 3 4]  
 [1 4 1 4]  
 [0 4 1 1]  
 [3 1 4 2]]
```

*Calcul du produit matricielle à
l'aide de la fonction dot ou du
raccourci @*

B est une matrice de forme (2,4)

```
B = np.array([[2,0],[1,3],[5,1],[0,6]],dtype =int)
```

C est le produit matricielle de A et B

```
C = np.dot(A,B)
```

```
C = A.dot(B)
```

```
C = A@B
```

```
[[ 3 21]  
 [26 22]  
 [12 28]  
 [17 25]]
```

```
B=  
[[2 0]  
 [1 3]  
 [5 1]  
 [0 6]]
```

*Calcul du déterminant d'une
matrice carrée*

déterminant de A :

```
print(np.linalg.det(A) )
```

```
-91.0
```

Calcul de la matrice inverse

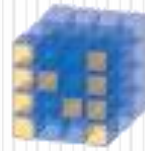
#matrice Inverse de A :

```
Ainv = np.linalg.inv(A)
```

```
print(np.round(Ainv,3))
```

```
[[ -0.495 -0.121  0.319  0.165]  
 [  0.407  0.055 -0.418  0.198]  
 [-0.374  0.22   0.33  -0.209]  
 [  0.363  0.022  0.033 -0.121]]
```

Module PIL



Traitement d'image

Documentation : <https://python-pillow.org>

Image niveau de gris

En zoomant sur l'image, ci-dessous, on s'aperçoit qu'elle est constituée d'une multitude de pixels organisés en lignes (H) et colonnes (L). La couleur de chaque pixel est codée sur un octet (c'est-à-dire 8 bits), et représente donc une nuance de gris parmi 255 valeurs possibles.

- la valeur 0 code la couleur noire,
- la valeur 255 code la couleur blanche.

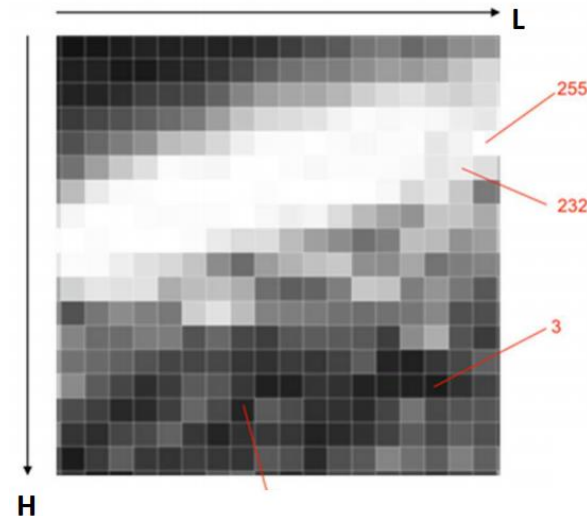
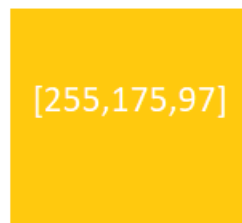


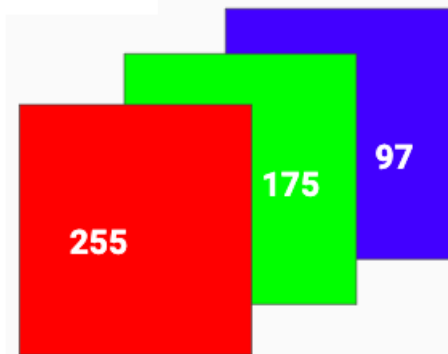
Image couleur RGB

En zoomant sur l'image, ci-dessous, on s'aperçoit qu'elle est constituée d'une multitude de pixels organisés en lignes (H) et colonnes (L). La couleur de chaque pixel est codée sur un 3 octet (c'est-à-dire 8 bits), et représente donc un mélange de 3 couleurs (Rouge, vert et bleu). Chaque couleur est codée sur 1 octet :

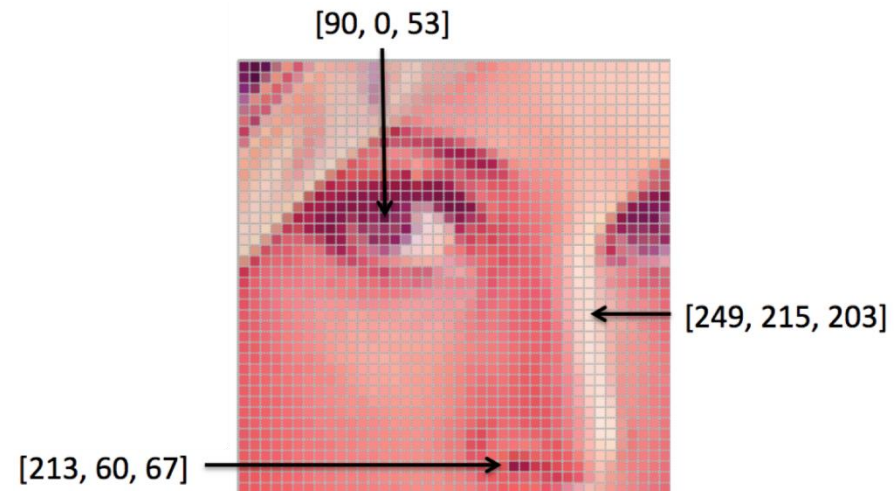
- R : code du rouge varie de 0 à 255
- G : code du vert varie de 0 à 255
- B : code du bleu varie de 0 à 255



=



Pixel RGB



Matrice Image en python

L'objectif est de transformer une image en tableau numpy, pour pouvoir ensuite la manipuler.

Lecture de l'image

PIL permet de lire une image enregistrée localement dans de nombreux format

Affichage de l'image

Image permet d'afficher une image (si c'est un tableau numpy)

Sauvegarde de l'image

Image permet sauvegarder un fichier image

importation de module

```
from PIL import Image  
import numpy as np
```

Lecture de l'image

```
imgpil = Image.open("monimage.png")
```

Transformation de l'image en tableau numpy

```
imgArray = np.array(imgpil)
```

Transformation du tableau numpy en image PIL

```
imgpil = Image.fromarray(imgArray)
```

Affichage de l'image

```
imgpil.show()
```

Sauvegarde des images

```
imgpil.save("resultat.jpg")
```

Matrice Image en python

```
import numpy as np
```

Pour certaines applications, on peut être amené à créer un tableau numpy contenant une image avec la fonction zeros et ayant le parametre dtype = uint8 (codage d'un entier 8 bits)

```
# créer un tableau numpy pouvant contenir une image  
imageRGB = np.zeros((100, 200, 3), dtype='uint8')
```

```
imageGray = np.zeros((100 , 200) , dtype = 'uint8')
```

on peut faire une copie d'une matrice image existante

```
# copie d'une matrice image  
image2RGB = np.copy(imageRGB)
```

Une matrice image couleur est à 3 dimensions :

- Nombre de ligne
- Nombre de colonne
- 3 : R, V et B

```
# Propriétés d'une matrice image couleur
```

```
imageRGB.ndim      # 3
```

```
ImageRGB.shape     # (100 , 200 , 3 )
```

Une matrice image niveau de gris est à 2 dimensions :

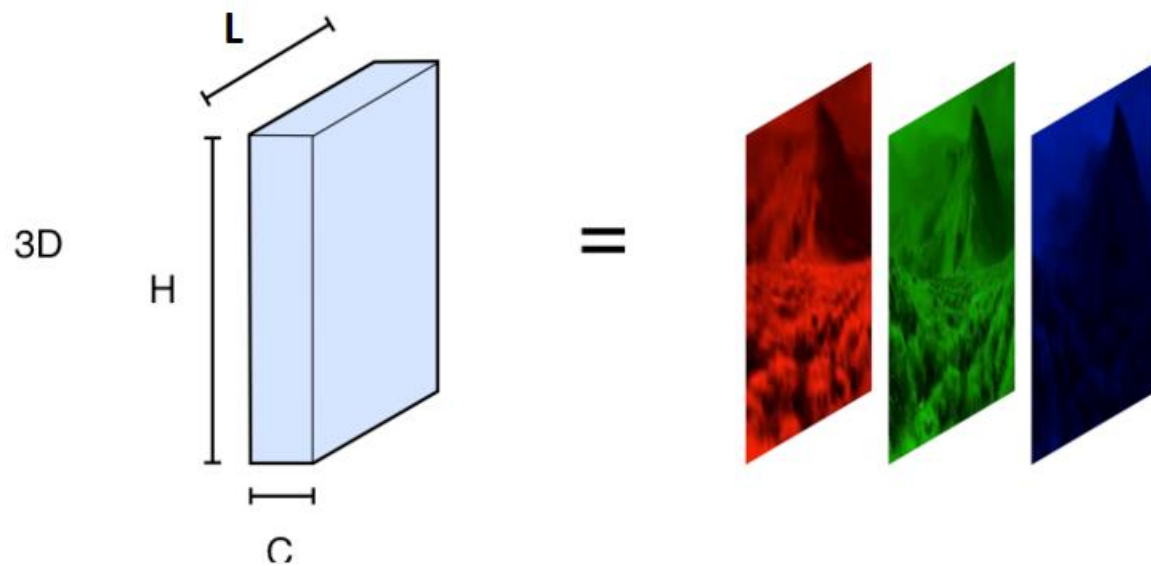
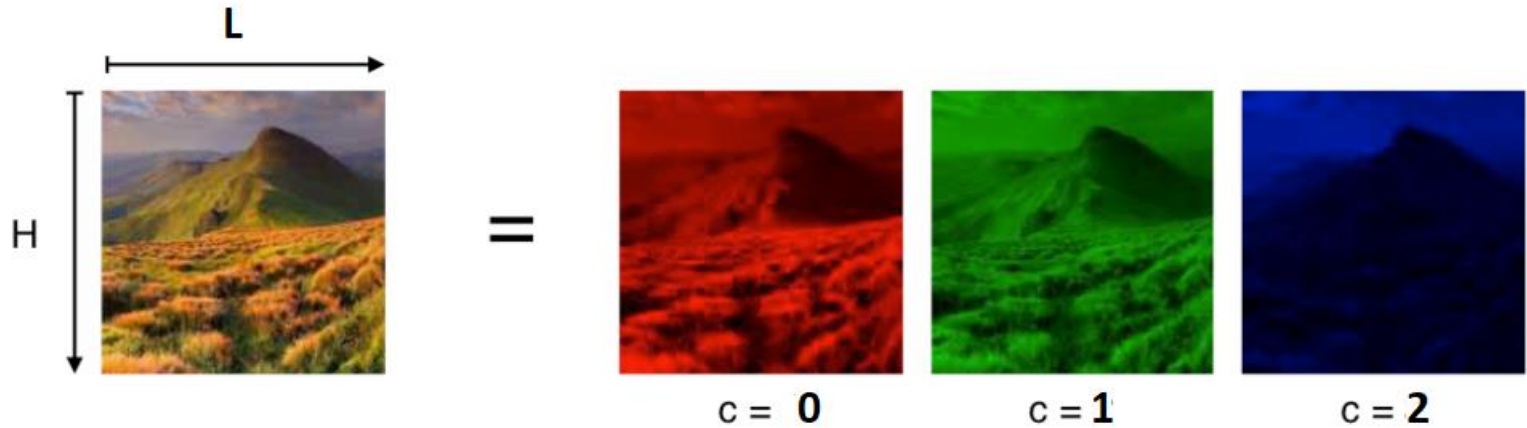
- Nombre de ligne
- Nombre de colonne

```
# Propriétés d'une matrice niveau de gris
```

```
imageGray.ndim     # 2
```

```
ImageGray.shape    # (100 , 200 )
```

Matrice Image en python



Module Matplotlib



Traçage des courbes, des histogrammes, etc.

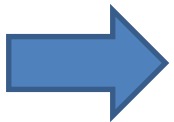
Documentation : <https://matplotlib.org/>

Introduction

- ❑ le module **matplotlib** est chargé principalement de tracer des courbes.
- ❑ Préalable important : importer le module « matplotlib » :

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

- ❑ **plt.plot** est la fonction principale pour tracer des courbes.
 - ❑ Elle prend en paramètres des vecteurs (points du plan).
 - ❑ Elle accepte encore des paramètres optionnelles
 - ❑ Elle relie les points du plan entre eux de façon ordonnée par des segments.



le résultat est une courbe

Exemple 1

*Importation des modules
numpy et matplotlib*

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

*La fonction np.linspace
génère un vecteur*

```
# ici : 100 designe le nombre de points  
x = np.linspace(-2*np.pi , 2*np.pi , 100)
```

*On passe deux vecteurs à
la fonction plt.plot*

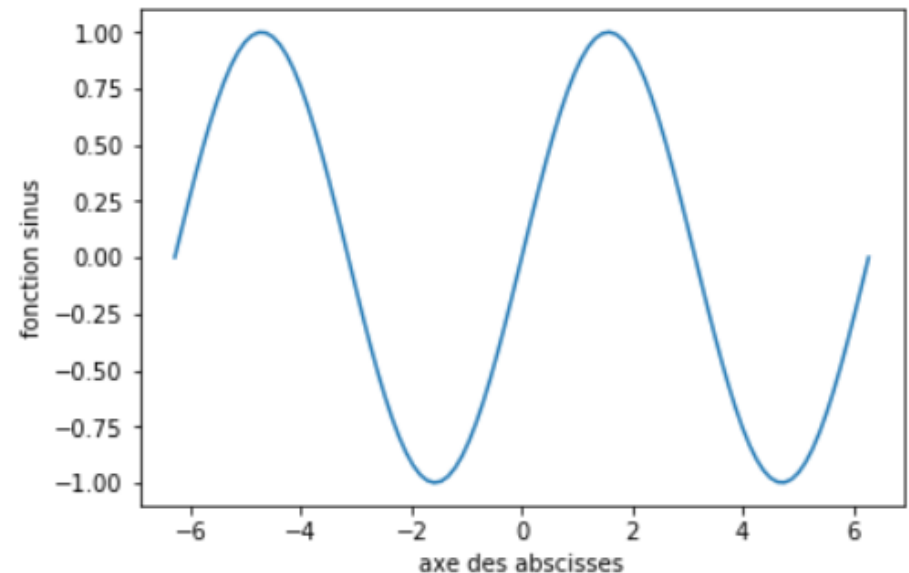
```
# la fonction np.sin est vectorisée  
plt.plot(x , np.sin(x))
```

*On ajoute des étiquette
sur les axes*

```
plt.ylabel('fonction sinus')  
plt.xlabel('axe des abscisses')
```

*On peut afficher la figure
avec la fonction plt.show,
ou bien sauvegarder le
graphique avec la fonction
plt.savefig('nomFig.ext'),
où ext est 'png', 'pdf', ...*

```
plt.show()
```



Exemple 2

Des options peuvent être envisagées :

- Bornes :

`plt.axis([xmin, xmax, ymin,ymax])`

- Titre : `plt.title(' titre')`

- Couleur de trait : (color)

- g : vert
- r : red
- k : noir
- y : jaune
- etc

- Symbole : (marker)

`[ + , * , . , ^ , o , ... ]`

- Style de trait : (linestyle ou ls)

`[ -- , - , : , -. , etc ]`

- Epaisseur du trait : (linewidth ou lw)

- Etc

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

```
import numpy as np
```

```
# spécifier un rectangle de représentation
```

```
plt.axis([-3, 0, -1,-0.75])
```

```
x = np.linspace(-2*np.pi , 2*np.pi , 100)
```

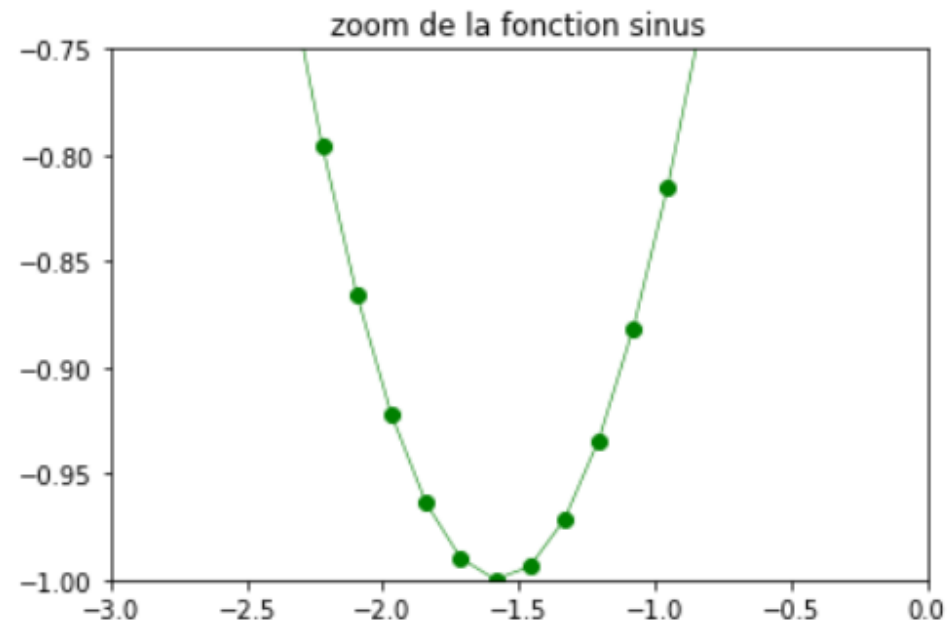
```
# spécifier des options graphiques
```

```
plt.plot(x , np.sin(x), color = 'g', marker = 'o',ls = '-',lw='0.6')
```

```
# ajouter un titre pour la courbe
```

```
plt.title ('zoom de la fonction sinus')
```

```
plt.show()
```



Exemple 3

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

```
plt.title ('Fonctions trigonometriques')  
x = np.linspace(-2*np.pi , 2*np.pi , 100)
```

Pour superposer plusieurs graphes, il est possible de faire une succession de la fonction plt.plot

La fonction plt.legend ajoute une légende dans le graphique qui correspond à l'option label de la fonction plt.plot

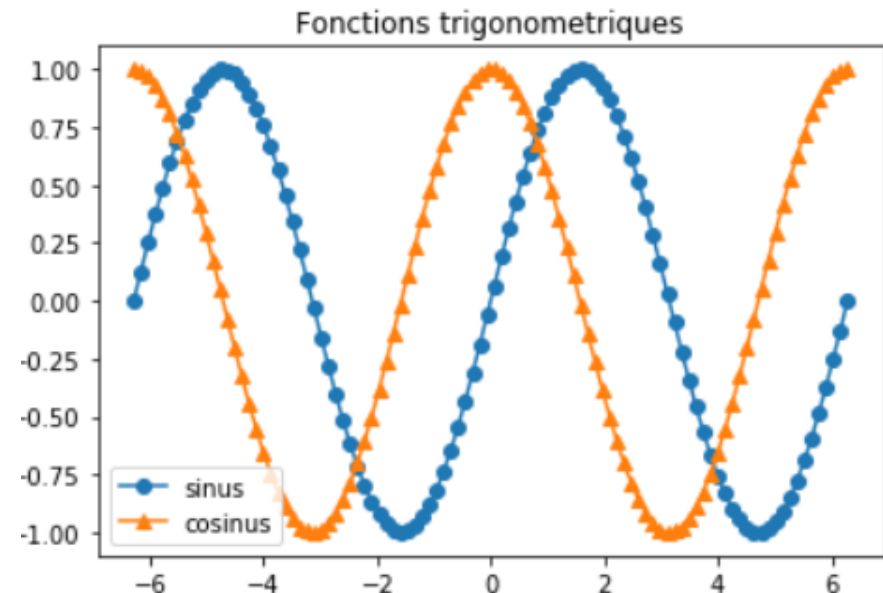
```
# une succession de plt.plot
```

```
plt.plot(x , np.sin(x), marker = 'o', label='sinus')  
plt.plot(x , np.cos(x), marker = '^', label='cosinus')
```

```
# afficher la légende
```

```
plt.legend()
```

```
plt.show()
```



Exemple 4

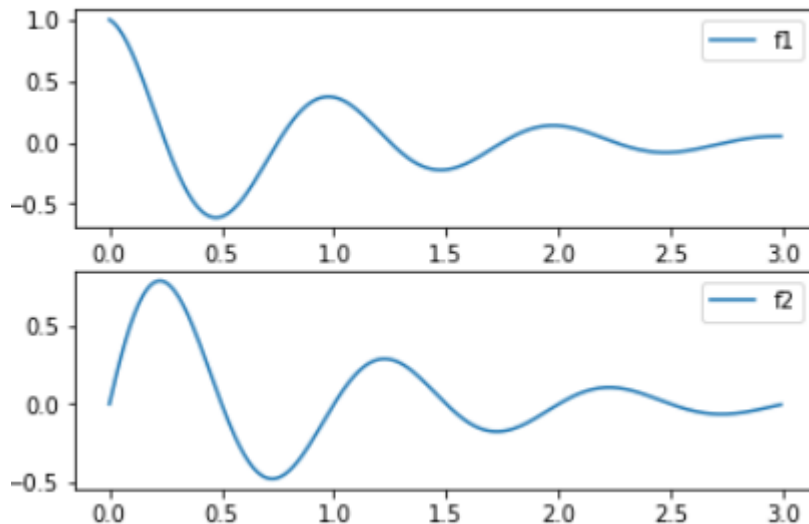
Pour juxtaposer différents graphiques, on utilise une succession des fonctions subplot et plot.

La fonction subplot crée une grille de sous figure. Elle prend paramètres soit un entier de la forme : lcp, soit 3 entier l,c,p

Avec l : nombre de lignes

c : nombre de colonnes

p : la position de la sous figure



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

Définition de deux fonctions

```
def f1(t):
```

```
    return np.exp(-t) * np.cos(2*np.pi*t)
```

```
def f2(t):
```

```
    return np.exp(-t) * np.sin(2*np.pi*t)
```

```
t = np.arange(0.0, 3.0, 0.01)
```

La première sous figure

```
ax1 = plt.subplot(211)
```

```
ax1.plot(t, f1(t), label='f1')
```

```
ax1.legend()
```

La deuxième sous figure

```
ax2 = plt.subplot(2, 1, 2)
```

```
ax2.plot(t, f2(t), label='f2')
```

```
ax2.legend()
```

```
plt.show()
```

Module SciPy



Calcul intégrale, équations différentielles, etc.

Documentation : <https://www.scipy.org/>

Introduction

- ❑ le module *scipy* manipule des tableaux comme le module *numpy*
- ❑ Il propose des fonctions beaucoup plus complexes et élaborées sur les objets tableaux telles :
 - ❑ méthodes d'intégration : `scipy.integrate`
 - ❑ méthodes de résolution d'équation : `scipy.optimize`
 - ❑ méthodes d'algèbre linéaire : `scipy.linalg`
 - ❑ méthodes d'interpolation : `scipy.interpolate`
 - ❑ méthodes de Fourier : `scipy.fftpack`
 - ❑ etc.
- ❑ Préalable important : importer le module « `scipy` » :

scipy.optimize

from scipy import optimize

Fonctions utiles du module scipy.optimize :

- **brentq(f,a,b)** : détermine une racine de la fonction f dans l'intervalle $[a, b]$ par la méthode de Brent
- **bisect(f,a,b)** : détermine une racine de f dans $[a, b]$ en effectuant une dichotomie.
 $f(a)$ et $f(b)$ ont des signes opposés.
- **newton(f,x0)** : détermine une racine approchée de f en partant de $x0$, par la méthode de Newton.

définition de la fonction $f : x \rightarrow x^2 - 1$

```
def f(x):  
    return (x**2 - 1)
```

```
root = optimize.brentq(f, 0, 2)  
print('solution Brent : ',root)
```

```
solution Brent : 1.0
```

```
root = optimize.bisect(f, 0, 2)  
print('solution dichotomie : ',root)
```

```
solution dichotomie : 1.0
```

```
root = optimize.newton(f, 1.5)  
print('solution Newton : ',root)
```

```
solution Newton : 1.00000000000000042
```

scipy.integrate(1)

Fonctions utiles pour le calcul numérique d'intégrale :

- `quad(f,a,b)` : calcule une valeur approchée de $\int_a^b f(x) dx$
- `dblquad(f,a,b)` et `tplquad(f,a,b)` : même chose pour des intégrales doubles ou des intégrales triples.

- `romberg(f,a,b)` : calcule une valeur approchée de à l'aide de la méthode de Romberg

- `simps(y)` : calcule une intégrale cumulée de y à l'aide de la méthode de Simpson.
y : est un tableau de valeurs des $f(x_i)$

- `cumtrapz(y)` : calcule une intégrale cumulée de y à l'aide de la méthode des trapèzes .

la fonction ressort un tableau, constituée des valeurs approchées prises par la primitive de la fonction aux mêmes points.

```
import numpy as np
```

```
from scipy import integrate
```

```
f = lambda x: x**2
```

```
# Calculer  $\int_0^4 x^2 dx$  et la comparer avec un calcul  
# analytique
```

```
sq, err = integrate.quad( f , 0, 4)
```

```
print('solution approchée : ', sq)
```

```
solution approchée : 21.333333333333336
```

```
sr = integrate.romberg( f , 0 , 4 )
```

```
print('solution approchée romberg : ',sr)
```

```
solution approchée romberg : 21.3333333333
```

```
x = np.linspace(0,4,num = 6)
```

```
y = f (x)
```

```
ss = integrate.simps(y, x)
```

```
print('solution Simpson : ',ss)
```

```
solution Simpson : 21.4186666667
```

```
st = integrate.cumtrapz(y, x)
```

```
print('solution trapeze : \n' , st)
```

```
solution trapeze :
```

```
[ 0.256  1.536  4.864 11.264 21.76 ]
```

```
print('solution analytique : ',4**3 / 3)
```

```
solution analytique : 21.333333333333332
```

scipy.integrate(2)

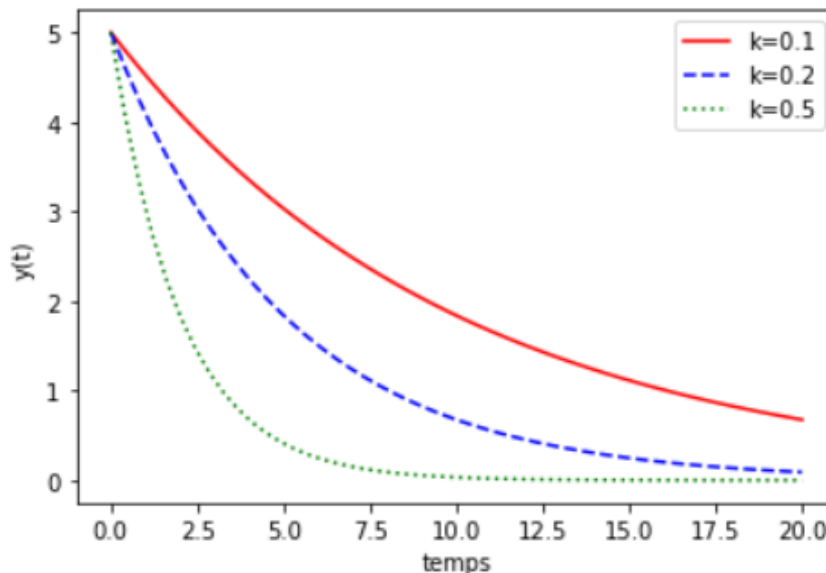
Fonction utile pour la résolution approchée d'équations différentielles :

En fait, c'est très simple puisque l'unique fonction **odeint** sert à résoudre toutes les équations différentielles.

On lui donne :

- une fonction représentant l'équation (ou le système)
- une condition initiale y_0
- un tableau de nombres correspondant aux valeurs t_i .
- args de type tuple contient les arguments extra du système.

La fonction **odeint** renvoie un tableau de valeurs qu'on peut évidemment représenter graphiquement si on le souhaite.



```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# fonction système qui renvoie dy/dt
dydt = lambda y,t,k : -k * y
# condition initiale
y0 = 5
# time points
t = np.linspace(0,20)
# résolution des équations
k = 0.1
y1 = odeint(dydt ,y0,t,args=(k,))
k = 0.2
y2 = odeint(dydt ,y0,t,args=(k,))
k = 0.5
y3 = odeint(dydt ,y0,t,args=(k,))
# traçage des résultats
plt.plot(t,y1,'r-',label='k=0.1')
plt.plot(t,y2,'b--',label='k=0.2')
plt.plot(t,y3,'g:',label='k=0.5')
plt.xlabel('temps')
plt.ylabel('y(t)')
plt.legend()
plt.show()
```