numpy - Notions de base

numpy

Présentation

NumPy est une bibliothèque *Python* qui permet à la fois de manipuler des matrices (des tableaux multidimensionnels) mais aussi de faire des calculs sur ces objets. C'est l'outil indispensable pour le calcul numérique.

Numpy ajoute donc le type `array` qui est similaire à une liste (`list`) avec la condition supplémentaire que tous les éléments sont du même type. Nous concernant ce sera donc un tableau d'entiers, de flottants, de complexes voire de booléens.

Pour l'importer, on recommande d'utiliser :

```
import numpy as np
```

Toutes les fonctions *NumPy* seront alors préfixées par np. Par contre, on rappelle que pour une méthode il n'y a pas de préfixe à utiliser.

Construction de tableaux

Il est possible de construire des tableaux à partir de listes avec la fonction array. Pour une liste simple (dont chaque élément est un entier, un flottant…) on obtiendra un tableau à une dimension, pour une liste dont les m éléments sont eux même des listes "simples" à n éléments on obtiendra un tableau bidimensionnel m × n.

Exemples

```
import numpy as n
# les noms numpy sont accessibles avec le préfixe np.
# création d'un tableau 1d à l'aide d'une liste ...
T = np.array([0.1,3,-1,4,6.7])
# on mélange flottants et entiers => tableau de flottants
print(T)    # >>> [ 0.1 3. -1. 4. 6.7]
# création d'un tableau 2d à l'aide d'une liste (de listes) ...
U = np.array([[0.4, 0.9, 4],[-1.5, 2, 1./3]])
print(U)    # >>> [[ 0.4 0.94.] [-1.5 2. 0.33333333]]
type(T)     # >>> <type 'numpy.ndarray'>
type(U)     # >>> <type 'numpy.ndarray'>
```

Il y a plusieurs fonctions qui permettent de construire des tableaux types (elles ont souvent le même nom que les fonctions équivalentes de scilab ou matlab.) :

1 –« `linspace(a,b,n)` » permet de créer un tableau 1d dont les n composantes sont uniformément réparties entre a et b (attention, ici b est inclus) :

```
np.linspace(0,1,5)       #tableau à 5 termes équi-répartis entre 0 et 1
# cela donne : array([ 0. , 0.25, 0.5 , 0.75, 1. ])
```

2 –« `arange(a,b,inc)` » permet de créer un tableau 1d et fonctionne comme la fonction range (a,b,inc) sauf que les arguments peuvent être des flottants (attention, comme avec range, la valeur finale est exclue) :

```
np.arange(0,1,0.1)
# donne : array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

3 - les fonctions « `zéros` » et « `ones` » permettent de créer des tableaux contenants des 0 ou des 1. Attention pour créer des tableaux 2d et plus, les dimensions doivent être données sous la forme d'un seul argument tuple (contenant ces dimensions). Quelques exemples :

```
np.zeros(5) # tableau 1d de 0 : array([ 0., 0., 0., 0., 0.])
np.ones((2,4)) # tableau 2d de taille 2 x 4 contenant des 1
#      array([[ 1., 1., 1., 1.],
#             [ 1., 1., 1., 1.]])
```

```
np.zeros((2,2,2)) # tableau 3d de taille 2 x 2 x 2 contenant des 0
#      array([[[ 0., 0.],[ 0., 0.]],
#             [[ 0., 0.], [ 0., 0.]]])
```

Les fonctions `zéros_like(T)` et `ones_like(T)` sont très utiles pour initialiser des tableaux de 0 ou de 1 ayant les mêmes dimensions qu'un autre tableau T (évitant ainsi d'avoir à récupérer les dimensions de ce tableau si on devait utiliser zeros ou ones).

Le sous-module `random` de *numpy* permet d'obtenir des nombres aléatoires, random.rand pour générer des nombres aléatoires de loi uniforme U (0, 1), random.randn pour la loi normale centrée réduite (N(0,1)) et random.randint pour une loi uniforme sur un intervalle d'entiers. Exemples :

```
np.random.rand(4)      # pour un tableau 1d à 4 éléments
                       # array([ 0.90538856, 0.23389902, 0.17182099, 0.81463087])
np.random.randn(3,3)   # pour un tableau 2d 3 x 3
                       # array([[-0.15114084, 0.09718998, -0.81857774],
                       #        [-0.05252272, 0.02244689, 1.01026781],
                       #        [ 0.45428138, 0.64223423, 0.58867134]])
np.random.randint(0,9,80)
   # loi uniforme sur les entiers de [0,9[ (donc [0,8])
   # array([4, 2, 5, 4, 5, 2, 8, 8, 1, 7, 0, 1, 1, 5, 3, 7, 5, 2, 3, 2, 6, 5, 3,
   #        2, 1, 2, 6, 8, 0, 1, 4, 5, 0, 2, 6, 7, 5, 6, 6, 3, 5, 5, 6, 8, 0, 4, 2, 6,
   #        4, 3, 8, 2, 2, 8, 3, 8, 8, 3, 5, 2, 0, 4, 2, 6, 2, 6, 4, 4, 5, 3, 1, 0, 5,
   #        7, 3, 4, 4, 0, 2, 8])
np.random.randint(0,9,(3,3)) # pour obtenir un tableau 2d 3 x 3
                             # array([[3, 6, 8],[6, 4, 0], [5, 3, 4]])
```

Détermination des caractéristiques d'un tableau

Les méthodes « `ndim` », « `shape` » et « `dtype` » permettent de connaître respectivement le nombre de dimensions d'un tableau, le nombre d'éléments dans chaque dimension et le type des éléments du tableau. Exemples :

```
T.ndim   # donne : 1
U.ndim   # donne : 2
T.shape  # donne : (5,)
U.shape  # donne : (2, 3)
# ou np.ndim(T) pour la version fonction
# ou np.shape(T) pour la version fonction
T.dtype  # donne : dtype('float64')
```

Noter aussi que la fonction « `len` » retourne le nombre d'éléments de la première dimension d'un tableau ce qui permet d'écrire des codes qui peuvent fonctionner aussi bien sur une liste que sur un tableau 1d.

Copie d'un tableau

La copie d'un tableau s'obtient grâce à la méthode `copy`, exemple : `A = U.copy()`.

Référencer les éléments d'un tableau

Pour les tableaux 1d, la syntaxe est la même que pour les listes, i.e. T[1] désigne le 2ème élément du tableau, T[0:2] (ou T[:2]) désigne le sous-tableau formé par les éléments d'indice 0 et 1, etc...

Pour les tableaux 2d et plus, on utilise la syntaxe suivante : U[1,2] (Différente d'une liste de listes où on écrirait U[1][2] ! Attention cette syntaxe peut fonctionner mais peut désigner aussi bien $U_{1,2}$ que $U_{2,1}$ en fonction du stockage du tableau en mémoire ; en effet un tableau 2d numpy peut être "rangé" aussi bien colonne par colonne que ligne par ligne : bref il faut toujours utiliser la syntaxe U[1,2] sauf si vous savez exactement ce que vous faites !).

Si on veut désigner toute la colonne d'indice 1 on écrit U[:,1] et si on veut référencer la ligne d'indice 0 à partir de la colonne d'indice 1 on écrit U[0,1:], etc...

Opérations sur les tableaux

Combinaison linéaire

L'avantage des tableaux est que l'on peut utiliser des expressions linéaires comme en math telles que « αu + βv » où α et β sont des scalaires et u et v des vecteurs. Si α et β sont des scalaires et u et v des tableaux de même

forme (mêmes dimensions) alors « α*u + β*v » génère un tableau de même dimension en réalisant la combinaison **élément par élément**.

Multiplication de deux tableaux au sens élément par élément

L'opérateur « * » correspond à cette multiplication (et donc pas à la multiplication matricielle que vous verrez en math bientôt). Ainsi lorsque A et B sont deux tableaux de même forme, C = A*B est le tableau dont les éléments sont égaux à $C_i = A_i B_i$ dans le cas 1d et $C_{ij} = A_{ij} B_{ij}$ dans le cas 2d, etc... Cela fonctionne aussi avec la division et la puissance **.

Application de fonctions mathématiques aux éléments d'un tableau

Comme avec *matlab* ou *scilab*, toutes les fonctions élémentaires usuelles (exp, log, sin, cos, ...) s'appliquent sur les tableaux et renvoient un tableau de même taille que l'argument, la fonction ayant été appliquée élément par élément. Mais attention, il faut pour cela utiliser les fonction mathématique de *numpy* et pas celle de la librairie *math* : `np.sin(T)` et pas `m.sin(T)` !

Vous disposez aussi des constantes π et e. D'autre part ces fonctions élémentaires de *numpy* s'appliquent aussi sur les scalaires, il est donc inutile d'importer le module math dès que vous importez *numpy*.

Attention : lorsque A et B n'ont pas la même forme une erreur n'est pas forcément obtenue car l'opération peut être « étendue » (broadcasting) dans certains cas comme par exemple lorsque A est un tableau 1d de taille 3 et B un tableau 2d de taille 2 × 3 alors l'opération consiste à multiplier chaque ligne de B par A (au sens élément par élément).

Produit extérieur

Si *x* est un tableau 1d de taille *m* et *y* un tableau 1d de taille *n*, `A = np.outer(x,y)` renvoie un tableau 2d de taille m × n avec $A_{ij} = x_i y_j$.

matplotlib - Notions de base

Présentation

« *matplotlib* » est une librairie de python permettant de d'afficher une très grande variété de graphiques 2D ou 3D. « *matplotlib* » fonctionne sur *Linux*, *Windows* et *OSX*. « *matplotlib* » peut être utilisée dans des scripts ou bien directement dans la console (comme *matlab* par exemple).

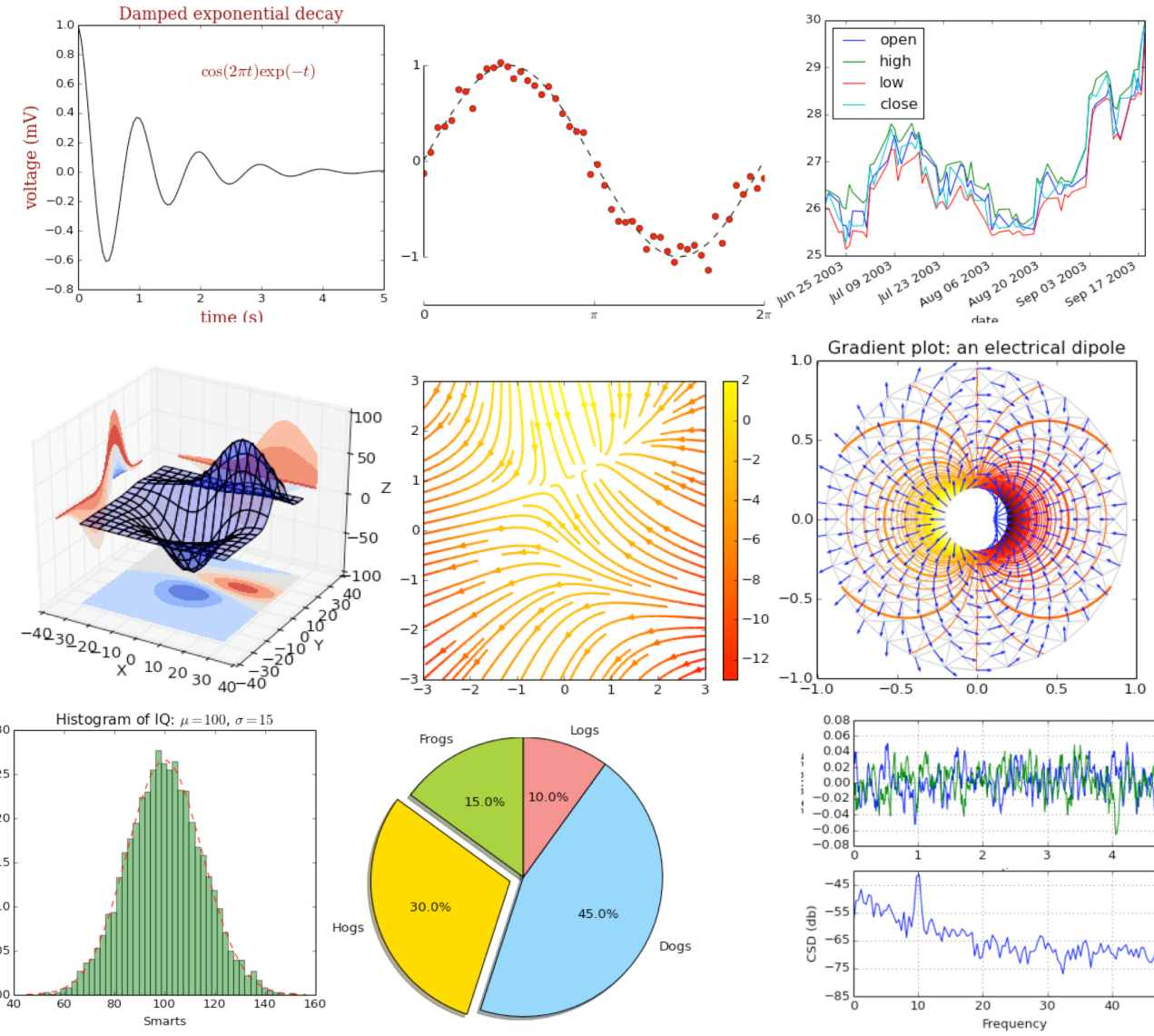
Ci-dessous sont présenté différents types de graphiques : tracé de fonction, affichage de points d'expériences, affichage de plusieurs courbes, surface en 3D, lignes isoclines, champs de vecteurs, histogrammes, diagrammes camembert…

Il est également possible de réaliser des graphes comportant plusieurs fenêtres (tracé de diagramme de Bode par exemple).

Il existe de très nombreuses options associées à chaque élément du graphique : le fond, le trait, les axes, la grille, la légende, les titres…

« Matplotlib tries to make easy things easy and hard things possible »

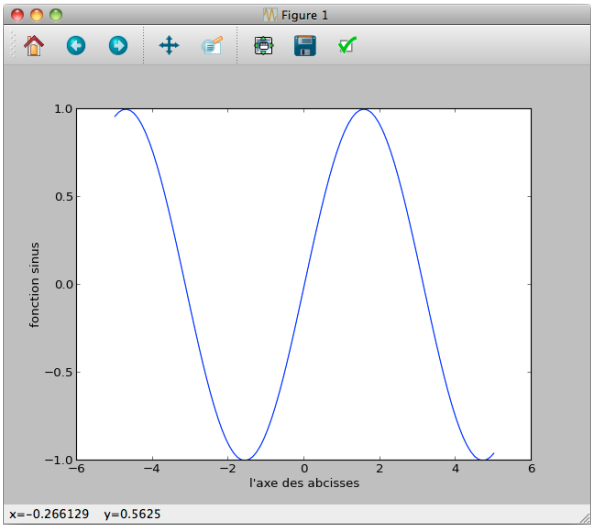
Exemples :



Prise en main élémentaire

Contrairement à *Mathematica*, *Maple*, voire même vos calculatrices, *matplotlib* ne travaille pas à partir de l'expression d'une fonction. Les tracés se font à partir de tableaux représentant les coordonnées de points du plan. Selon les options, ces points du plan peuvent être reliés entre eux de façon ordonnée par des segments. Le résultat ressemble alors une courbe.
L'importation du module graphique se fait classiquement de la manière suivante :

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



Commençons par un exemple élémentaire : le graphe de la fonction sinus.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-5,5,100)
y = np.sin(x) # on utilise la fonction sinus de numpy
plt.plot(x,y)
plt.ylabel('fonction sinus')
plt.xlabel("l'axe des abcisses")
plt.show()
```

Si tout se passe bien, une fenêtre doit s'ouvrir avec la figure ci-dessus. Il est possible de jouer avec les fonctions associées aux icones du haut de cette fenêtre : zoomer, déplacer la figure…, et surtout sauvegarder dans un format *PNG*, *PDF*, *EPS*, etc.
Pour sauvegarder votre figure, il est possible d'utiliser la commande :

`plt.savefig('nomdufichier.extension').`

Par exemple `plt.savefig("mongraphe.png")` sauve sous le nom "mongraphe.png" le graphique au format *PNG*.

Par défaut le format est *PNG*. Il est possible d'augmenter la résolution, de choisir la couleur de fond, l'orientation (portrait ou paysage), la taille (A0, A1, lettertype, etc) ainsi que le format de l'image. Si aucun format n'est spécifié, le format est celui de l'extension dans "nomfigure.ext" (où "ext" est "eps", "png", "pdf", "ps" ou "svg"). Il est toujours conseillé de mettre une extension aux noms de fichier ; si vous y tenez `plt.savefig('toto',format='pdf')` sauvegarder l'image sous le nom "toto" (sans extension !) au format "pdf".

Une autre commande très utile : `plt.clf()` qui permet d'effacer la fenêtre graphique active.

Comportement en mode interactif

En mode interactif Python ou IPython, une caractéristique est le mode interactif de cette fenêtre graphique. Si vous avez tapé l'exemple précédent, et si cette fenêtre n'a pas été fermée alors la commande `plt.xlabel("cequevousvoulez")` modifiera l'étiquette sous l'axe des abscisses. Si vous fermez la fenêtre alors la commande `plt.xlabel("cequevousvoulez")` se contentera de faire afficher une fenêtre graphique avec axe des abscisses, des ordonnées allant de 0 à 1 et une étiquette "ce que vous voulez" sous l'axe des abscisses. L'équivalent "non violent" de fermer la fenêtre est la commande `plt.close()`.
L'inconvénient, une fois un premier graphique fait, est le ré-affichage ou l'actualisation de cette fenêtre graphique au fur et à mesure des commandes graphiques : lenteur éventuelle si le graphique comporte beaucoup de données. Il est donc indispensable de pouvoir suspendre ce mode interactif. Heureusement tout est prévu !

`plt.isinteractive()` : Retourne True ou False selon que la fenêtre graphique est interactive ou non.

`plt.ioff()` : Coupe le mode interactif.

`plt.ion()` : Met le mode interactif.

`plt.draw()` : Force l'affichage (le "retraçage") de la figure.

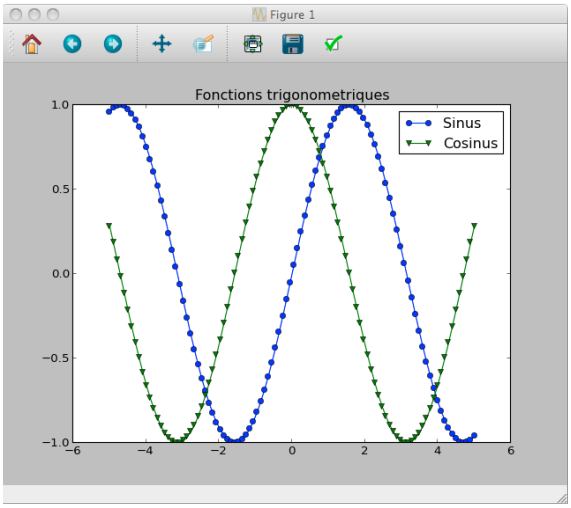
Ainsi une fois la première figure faite pour revenir à l'état initial, les deux commandes `plt.close()` et `plt.ioff()` suffisent.

Options graphiques

Pour connaître toutes les options, le mieux est de se référer à la documentation de *matplotlib*. Voyons ici quelques unes d'entre elles :

- **la couleur du trait** : pour changer la couleur du tracé une lettre : g vert (green), r rouge (red), k noir, b bleu, c cyan, m magenta, y jaune (yellow), w blanc (white) :
  - `plt.plot(np.sin(x),'r')` tracera notre courbe sinus en rouge.
  - Les amateurs de gris sont servis via l'option : `color = 'unflottantentre0et1'`.
  - Enfin pour avoir encore plus de couleurs, la séquence `color = '#eeefff'` donnera la couleur attendue dont les composantes rouge, vert et bleu (décrites sous 8 bits et donc donnant 256 nuances différentes) sont données en hexadécimal (ee, ef et ff pour l'exemple)
  - les amateurs de RGB sont servis par `color = (R,G,B)` avec trois paramètres compris entre 0 et 1.
- **le style du trait** : pointillés, absences de trait, etc se décident avec `linestyle`. Au choix
  - '-' ligne continue,
  - '--' tirets,
  - '-.'points-tirets,
  - ':' pointillés, sachant que 'None', '', '' donnent "rien-du-tout".
- **l'épaisseur du trait** : `linewidth = flottant` (comme `linewidth = 2`) donne un trait, ou pointillé (tout ce qui est défini par style du trait) d'épaiseur« flottant » en points. Il est possible d'utiliser `lw` en lieu et place de `linewidth`.
- **les symboles des points** : mettre des symboles aux points tracés se fait via l'option `marker`. Les possibilités sont nombreuses parmi :
  ['+' '*' ',' '.' '1' '2' '3' '4' '<' '>' 'D' 'H' '^' '_' 'd' 'h' 'o' 'p' 's' 'v' 'x' '|' 'TICKUP|TICKDOWN|TICKLEFT|TICKRIGHT' 'None'...].
- **la taille des symboles** (markers) : `markersize = flottant` comme pour l'épaisseur du trait. D'autres paramètres sont modifiables :
  - `markeredgecolor` : la couleur du trait du pourtour du marker,
  - `markerfacecolor` : la couleur de l'intérieur (si le marker possède un intérieur comme 'o'),
  - `markeredgsize = flottant` : l'épaisseur du trait du pourtour du marker.

  Remarque : si la couleur n'est pas spécifiée, pour chaque nouvel appel la couleur des "markers" change de façon cyclique.
- **les étiquettes sur les axes des abcisses/ordonnées** : *matplotlib* décide tout seul des graduations sur les axes. Tout ceci se modifie via `plt.xticks(tf)`, `plt.yticks(tl)` où `tf`, `tl` sont des tableaux de flottants ordonnés de façon croissante.
- **le titre** : `plt.title("Montitre")`
- **les légendes** : dans un premier temps, pour chaque courbe, donner le texte de la légende avec l'option « `label` » puis utiliser la commande `plt.legend()`
- **les bornes du graphique** : spécifier un rectangle de représentation, ce qui permet un zoom, d'éviter les grandes valeurs des fonctions par exemple, se fait via la commande `plt.axis([xmin,xmax,ymin,ymax])`

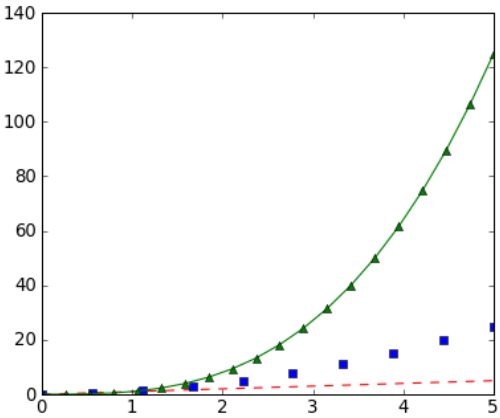



```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
p1                                                    =
plt.plot(x,np.sin(x),marker='o',label='Sinus')
p2 = plt.plot(x,np.cos(x),marker='v',label='Cosinus')
plt.title("Fonctions trigonometriques")
plt.legend()
plt.show()
```

Quelques exemples

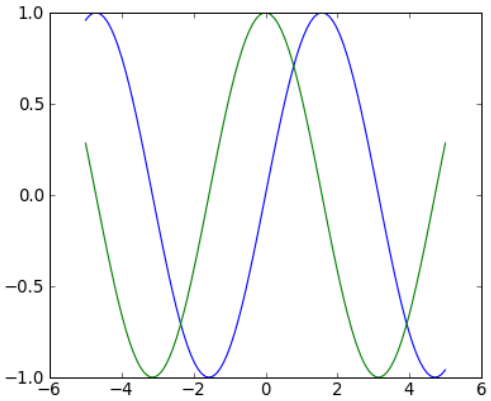
Pour superposer plusieurs graphes de fonctions, il est possible de faire une succession de commandes `plt.plot` ou encore en une seule commande. Remarquez aussi que pour des choses simples il est possible de se passer des `ls`, `color` et `marker`.

```
import matplotlib.pyplot as plt
import numpy as np
t1 = np.linspace(0,5,10)
t2 = np.linspace(0,5,20)
plt.plot(t1, t1, 'r--', t1, t1**2, 'bs', t2, t2**3, 'g^-')
```



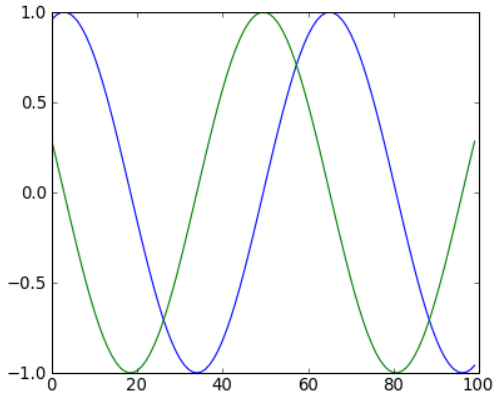
Donner comme deuxième argument (abscisses) une matrice qui a autant de ligne que l'argument des abscisses est possible

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-5,5,100)
y = np.zeros((100,2))
y[:,0] = np.sin(x)
y[:,1] = np.cos(x)
plt.plot(x,y)
plt.show()
```



Si un seul vecteur (ou une seule matrice) est donné, on trace le graphe avec comme abscisse l'indice des éléments :

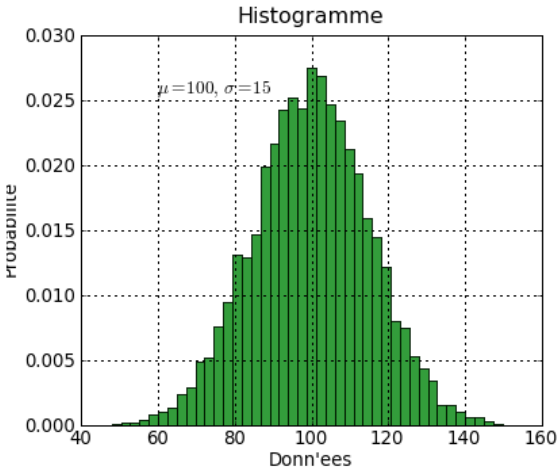
```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-5,5,100)
y = np.zeros((100,2))
y[:,0] = np.sin(x)
y[:,1] = np.cos(x)
plt.plot(y)
plt.show()
```



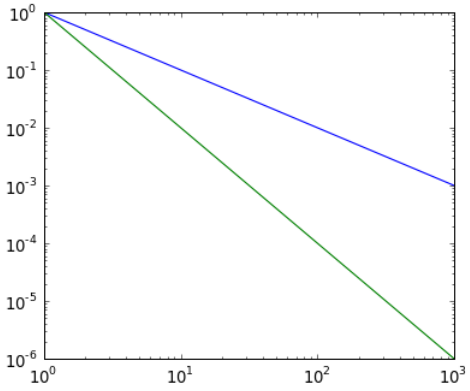
Dans le même registre, ouvrir plusieurs fenêtres graphiques est possible. Si vous avez déjà une fenêtre graphique, la commande `plt.figure(2)` en ouvre une seconde et les instructions `plt.plot` qui suivent s'adresseront à cette seconde figure. Pour revenir et modifier la première fenêtre graphique, `plt.figure(1)` suffit.

Pour terminer, un histogramme et un affichage de texte sur le graphique

```
import numpy as np
import matplotlib.pyplot as plt
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
#  histogramme des données
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)
plt.xlabel('Donn\'ees')
plt.ylabel('Probabilite')
plt.title('Histogramme')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
```



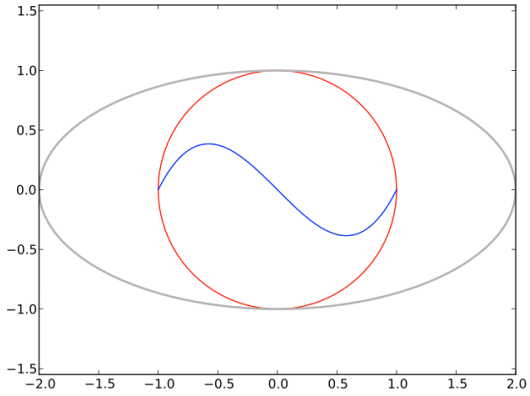
Certains graphiques demandent une échelle logarithmique (simple ou double). Les commandes `plt.semilogx()`, `plt.semilogy()` et `plt.loglog()` mettent respectivement le graphe à l'échelle logarithmique simple en `x`, logarithmique simple en `y` et double échelle logarithmique.



```
X = np.linspace(1,1000,50)
plt.loglog()
plt.plot(x,1./x)
plt.plot(x,1./x**2)
plt.show()
```

Echelle isométrique

Parfois on a besoin d'une échelle isométrique, celle-ci s'obtient en utilisant axis("equal"). Exemple si on veut tracer sur une même figure, le cercle $x^2 + y^2 = 1$ en rouge, l'ellipse $(x/2)^2+y^2 = 1$ en gris clair avec une épaisseur de 2 et la fonction $f(x) = (x - 1)x(x + 1)$ pour x ∈ [−1, 1] en bleu, on peut procéder ainsi :



```
t = np.linspace(0,2*pi,100)
x1 = cos(t)
y1 = sin(t)
x2 = 2*cos(t)
y2 = sin(t)
x3 = np.linspace(-1,1,100)
y3 = (x3-1)*x3*(x3+1)
plt.plot(x1,y1,"r",x3,y3,"b")
plt.plot(x2,y2,color="0.7",linewidth=2)
plt.axis("equal")
```

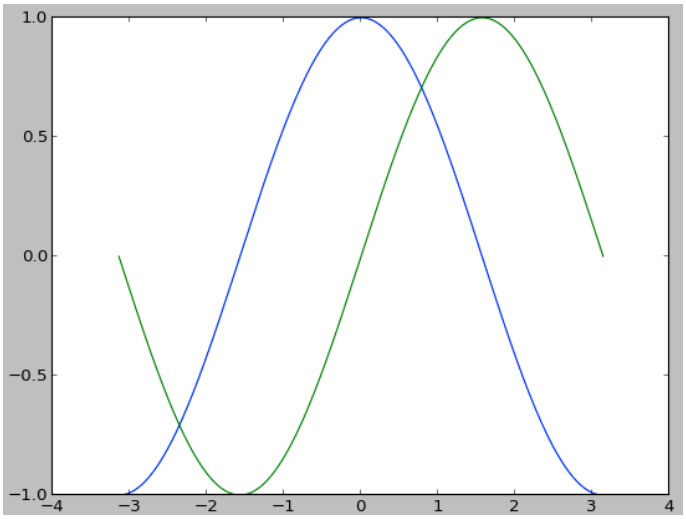
Pour faire joli

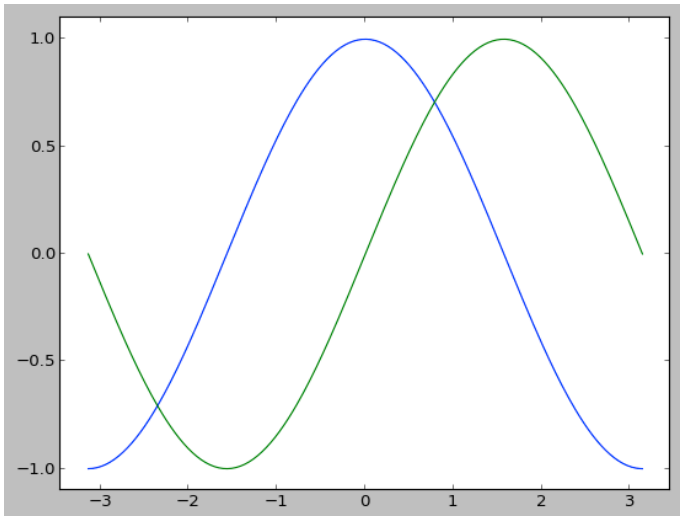
De base, l'affichage est réalisé en utilisant les valeurs min et max du vecteur x (de même pour y). On peut modifier cela avec les commandes « `xlim` » et « `ylim` ». On peut également imposer les valeurs placées sur les axes avec « `xticks` » et « `yticks` » :

```
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)
plt.plot(X,C)
plt.plot(X,S)
plt.xlim(-4.0,4.0)
plt.xticks(np.linspace(-4,4,9,endpoint=True))
plt.ylim(-1.0,1.0)
plt.yticks(np.linspace(-1,1,5,endpoint=True))
```

si l'espace a droite, à gauche en haut et en bas n'est pas bon, on peut utiliser :

```
xlim(X.min()*1.1, X.max()*1.1)
plt.xticks(np.linspace(-3,3,7))
ylim(C.min()*1.1, C.max()*1.1)
```



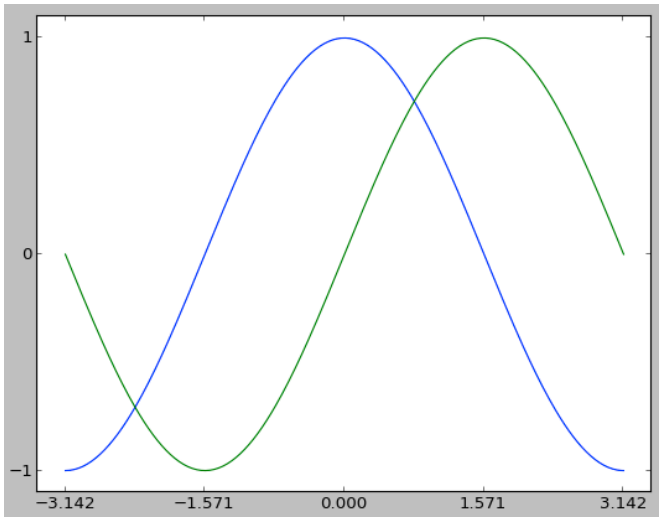


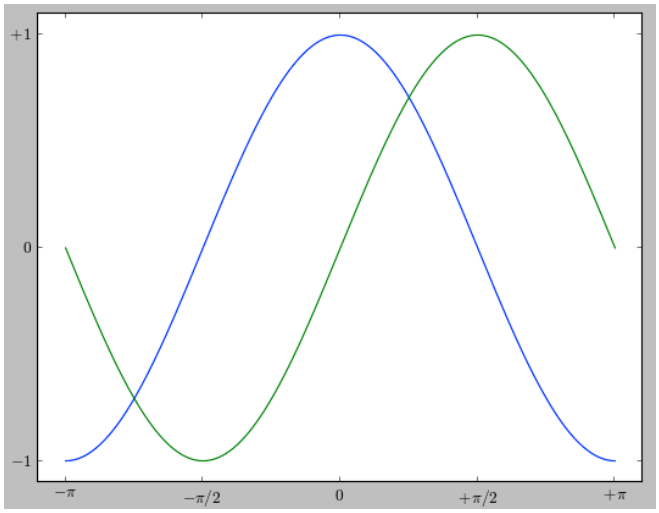
Peut mieux faire : pour utiliser des valeurs intéressantes pour nos 2 fonctions on peut écrire :

```
plt.xticks( [-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
plt.yticks([-1, 0, +1])
```

Voir même pour les plus pointilleux (figure de droite) :

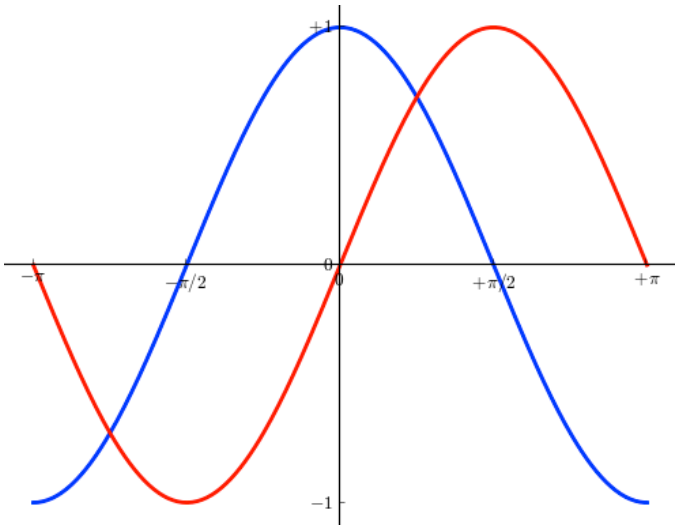
```
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
[r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
plt.yticks([-1, 0, +1], [r'$-1$', r'$0$', r'$+1$'])
```

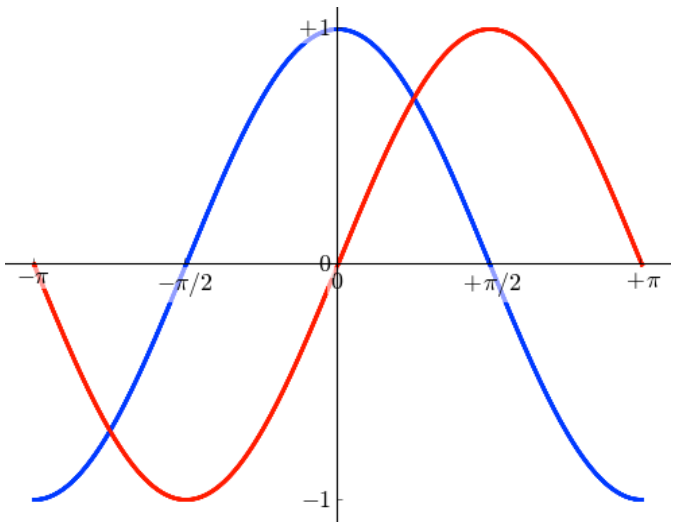




Enfin pour faire « pro »

```
ax = gca()             # gca means : get current axis (instance)
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
```






```
for label in ax.get_xticklabels() + ax.get_yticklabels():
label.set_fontsize(16)
label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65))
```

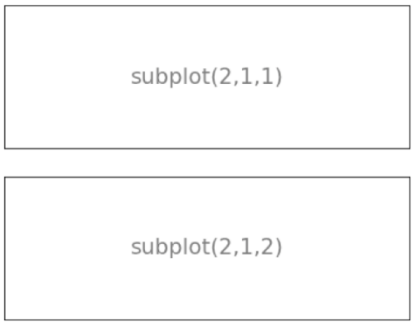
Plusieurs graphiques sur une même figure

La commande « `subplot` » permet d'insérer plusieurs graphes dans une même figure. Cela peut permettre entre autre de dessiner un diagramme de bode en gain et phase.

`subplot` attend 3 arguments : le nombre de lignes de graphes, le nombre de graphe par ligne et le numéro du graphe sur lequel on va effectuer les commande qui suivent :

`subplot(nb_lignes,nb_colonnes, num_graphe)`.

Il y a une condition à respecter : le nombre de lignes multiplié par le nombre de colonnes est supérieur ou égal au nombre de figure. Ensuite *matplotlib* place les figures au fur et à mesure dans le sens des lignes.



2 graphes l'un au dessus de l'autre

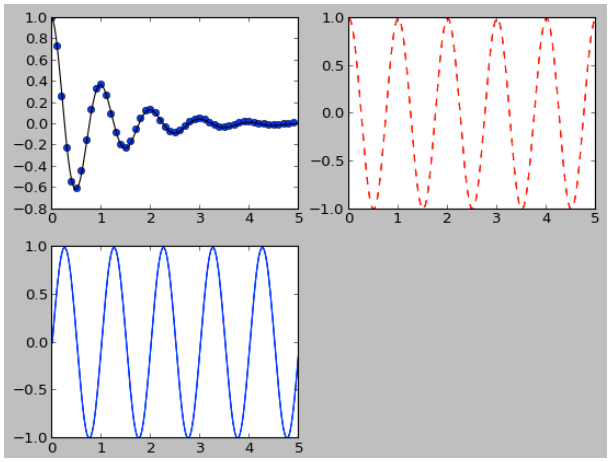
2 graphes cote à cote

4 graphes

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

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure(1)
plt.subplot(221)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(222)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.subplot(223)
plt.plot(t2, np.sin(2*np.pi*t2), 'b-')
```



Création d'animations

Le module `animation` est disponible à partir de la version 1.1 de *matplolib*. Nous allons utiliser la fonction `FuncAnimation()` du module `animation`.

Dans notre script, nous allons définir une fonction `init()` et une fonction `animate()`. La fonction `init()` servira à créer l'arrière de l'animation qui sera présent sur chaque image. La fonction `animate()` met à jour la courbe pour chaque image.

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

# définition des paramètres caractéristiques de la courbe à tracer

Pi = np.pi
k = 2*pi
w = 2*pi
dt = 0.01
xmin = 0
xmax = 3
nbx = 100
x = np.linspace(xmin, xmax, nbx)

# on définit une figure avec des axes, une échelle et une ligne vide

fig = plt.figure()
line, = plt.plot([],[])
plt.xlim(xmin, xmax)
plt.ylim(-1,1)
```

```
# fonction à définir quand blit=True
# créé l'arrière de l'animation qui sera présent sur chaque image

def init():
    line.set_data([],[])
    return line,            # il est indispensable de renvoyer la ligne à modifier
def animate(i):
    t = i * dt
    y = np.cos(k*x - w*t)
    line.set_data(x, y)
    return line,

ani = animation.FuncAnimation(fig, animate, init_func=init, frames=50, blit=True,
interval=20, repeat=False)
plt.show()

# blit = True : option qui impose au module d'animation de ne retracer que
# les objets qui ont été modifiés (ici : line,)
# interval : temps en ms entre 2 images
```

Conclusion

Vous pouvez faire bien d'autres choses avec *matplotlib*. La documentation fait plus de 1500 pages…. Les notions présentées dans ce cours sont normalement suffisantes pour vos besoins futurs en TP d'IpT, TP de physique, SI… et TIPE.