Bibliothèques pour Python

Ce document ne prétend en aucun cas remplacer les manuels de référence des bibliothèques citées. Il s'agit simplement d'un aide-mémoire permettant de rapidement retrouver le nom et la syntaxe de quelques fonctions parmi les plus utiles. Ainsi, par exemple, il n'évoque pas la gestion des différents formats de nombres en virgule flottante, ni les multiples choix de méthode pour le calcul approché d'une intégrale ou la résolution d'une équation scalaire.

Les manuels de référence sont disponibles en ligne à ces adresses :

– pour NumPy et SciPy : http://docs.scipy.org/doc/

– pour Matplotlib : http://matplotlib.org/contents.html#

Leur lecture est vivement recommandée.

NumPy

Cette bibliothèque nous intéressera essentiellement pour deux objectifs : travailler avec de véritables tableaux, générer des nombres aléatoires. Elle propose également un module FFT et un module LINALG, auxquels on pourra préférer les modules équivalents de SciPy.

Les tableaux

Python propose non pas des tableaux mais des listes : les listes Python sont hétérogènes, et de taille variable. Ainsi on peut mélanger des entiers, des chaînes et des flottants dans une même liste. Et `append` permet par exemple de modifier la taille d'une liste en lui ajoutant des éléments.

NumPy propose de « vrais » tableaux : homogènes, et de taille fixe. Avantage : leur manipulation est optimisée et est beaucoup plus efficace que celle des listes.

Création de tableaux

Les tableaux peuvent être multidimensionnels. Leur forme ou dimension est un n-uplet d'entiers, appelée `shape` par NumPy.

```
import numpy as np
a = np.empty([3,2])   # crée un tableau de 3 lignes et 2 colonnes, sans
initialiser les valeurs
a = np.zeros([3,2])   # crée un tableau de 3 lignes et 2 colonnes, tous les
coefficients initialisés à 0
a = np.ones([3,2])    # crée un tableau de 3 lignes et 2 colonnes, tous les
coefficients initialisés à 1
a = np.identity(3)    # crée la matrice identité 3x3
a.shape               # renvoie la dimension du tableau

np.arange(5)          # renvoie array([0,1,2,3,4])
```

```
np.arange(3,6)        # renvoie array([3,4,5])
np.arange(1.,5.,0.1)  # renvoie array([1.0,1.1,1.2,...,4.8,4.9])
np.linspace(0.0,10.0) # par défaut 50 valeurs équiréparties de 0 à 10
bornes incluses
np.linspace(0.0,10.0,num=20)
```

On peut aussi transformer une liste Python en tableau à la NumPy.

```
a = np.array([1,3,2,4])
a = np.array([[1,3],[2,4]])
b = np.copy(a)        # renvoie une copie du tableau
```

Indexation des tableaux

Quelques exemples pour comprendre. Conformément à l'usage courant en Python, la borne inférieure est incluse, la borne supérieure exclue, comme pour les appels à la fonction `range`.

```
a = np.arange(12)
a                     array([ 0,1,2,3,4,5,6,7,8,9,10,11 ])
a[:]                  array([ 0,1,2,3,4,5,6,7,8,9,10,11 ])
a[3:]                 array([ 3,4,5,6,7,8,9,10,11 ])
a[:8]                 array([ 0,1,2,3,4,5,6,7 ])
a[3:8]                array([ 3,4,5,6,7 ])
b = np.reshape(np.arange(12),(3,4))
b                     array([[ 0,1,2,3 ], [ 4,5,6,7 ], [ 8,9,10,11 ]])
b[:,0]                array([ 0,4,8 ])
b[0,:]                array([ 0,1,2,3 ])
```

Statistiques élémentaires sur les tableaux

On peut, dans le cas des tableaux multidimensionnels, choisir un « axe » de calcul : si par exemple l'axe est 1, on travaillera ligne par ligne.

```
from math import *
a                     array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
b                     array([[ 0, 1, 2, 3 ],  [ 4,5,6,7 ], [ 8,9,10,11 ]])
np.amax(a)            11 # maximum
np.amin(b)            0  # minimum
np.mean(a)            5.5
np.mean(b)            5.5
np.mean(b,axis=0)     array([ 4.,  5.,  6.,  7.])
np.mean(b,axis=1)     array([ 1.5,  5.5,  9.5])
np.std(a)             3.4520525295346629
np.var(a)             11.916666666666666
sqrt(np.var(a))       3.452052529534663
np.std(b,axis=1)      array([ 1.11803399,  1.11803399,  1.11803399])
```

Le module random

Tirages aléatoires : ces fonctions admettent en argument la « forme » du tableau demandé. On peut réaliser des tirages uniformes entre 0 et 1 (`rand`), suivant la loi normale centrée réduite (`randn`), de nombres entiers dans une plage précise (`random_integers`).

```
import numpy.random as alea
alea.rand(4)          array([ 0.61809,0.10437,0.46354,0.45504 ])
alea.rand(2,3)        array([[0.307,0.0118,0.5550],[0.8115,0.0038,0.8366]])
alea.randn(4)         array([ 0.760152, -0.25892946, 1.231548, 0.578457 ])
alea.random_integers(3,10,size=7) array([3, 7, 9, 4, 7, 6, 8])
```

NumPy propose également tout un jeu de distributions standards. Cette fois, si on veut plusieurs valeurs, on utilisera l'argument optionnel `size`.

```
alea.binomial(15,0.3)        3
alea.binomial(15,0.3,size=10)array([3, 4, 5, 5, 4, 5, 5, 4, 5, 5])
alea.exponential(2.5,size=4) array([ 0.11846, 0.90558, 1.2499, 4.71623 ])
alea.geometric(0.32)         4
alea.geometric(0.32,size=4)  array([2, 8, 1, 1])
mu = 12
sigma = 5
alea.normal(mu,sigma,size=3) array([ 9.7544804, 18.4279264, 15.378136 ])
alea.poisson(2.5,size=6)     array([1, 0, 4, 1, 1, 2])
```

SciPy

Nous ne présenterons ici que quelques exemples, issus des sous-modules `integrate`, `optimize`, `linalg` et `stats`.

Le module integrate

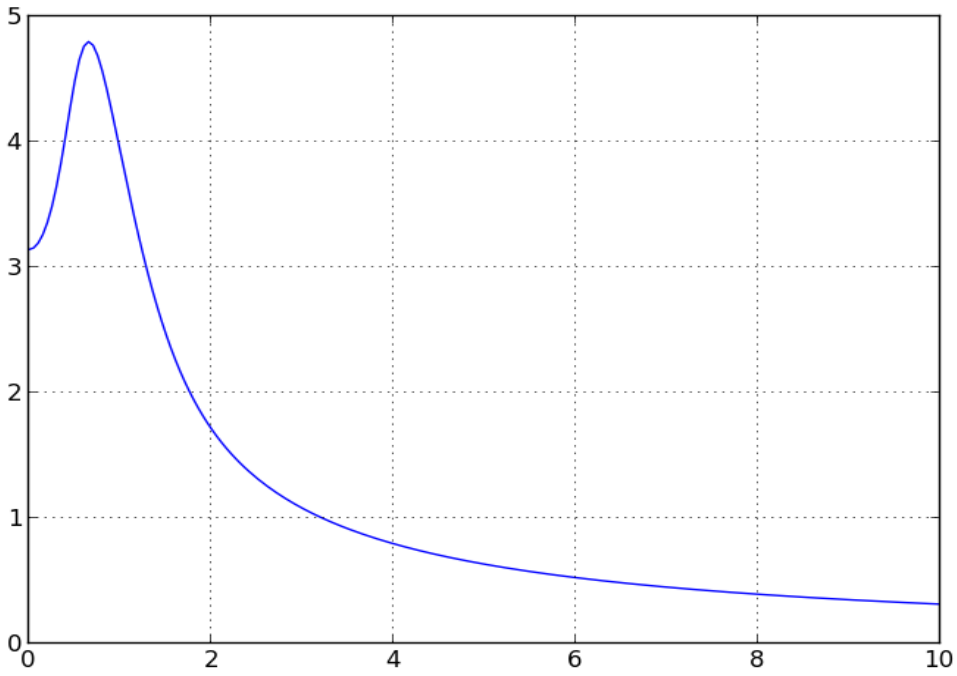
La fonction `quad` permet de calculer de façon approchée des intégrales définies, la fonction `odeint` permet de résoudre des équations différentielles ordinaires. SciPy propose un grand choix d'autres fonctions utilisant différentes méthodes de quadrature, et une fonction, nommée `ode`, permettant de paramétrer très finement la méthode d'approximation utilisée.

On notera que le résultat est *a priori* un nombre complexe.

```
from math import *
import numpy as np
import scipy as sp
import scipy.integrate as integr
def f(x):
   return sin(x*x)

integr.quad(f,0,pi)   (0.7726517126900656, 1.6493474541321405e-12)
```

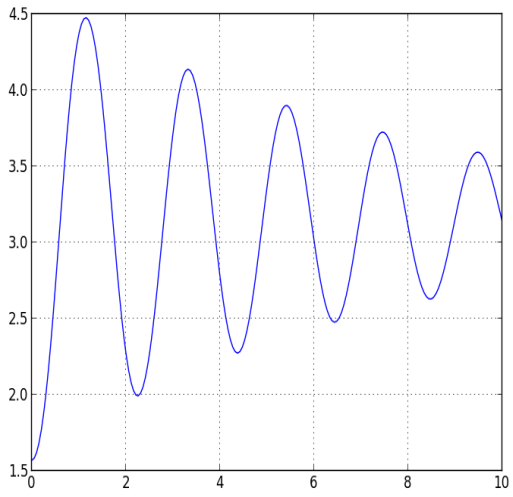
Pour résoudre une équation différentielle du premier ordre, scalaire ou vectorielle, on utilisera la fonction `odeint`. On suppose que l'équation s'écrit x'(t) = f(x(t),t), où x est à valeurs scalaires ou vectorielles.

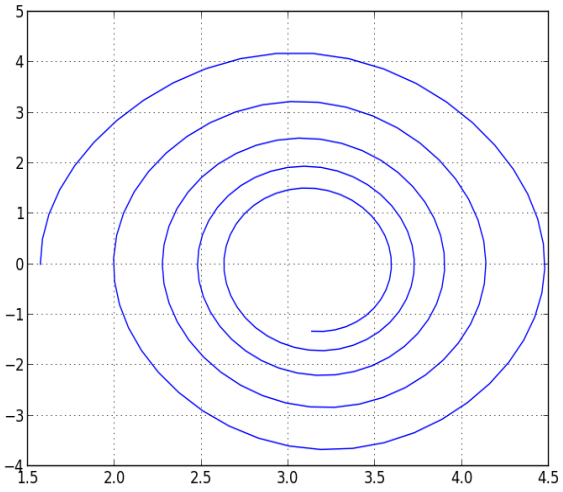
```
from math import *
import numpy as np
import scipy as sp
import scipy.integrate as integr
import matplotlib as mp
import matplotlib.pyplot as plt
def f(x,t):
  return (t+x)*sin(t*x)

t = np.linspace(0,10,num=200)
sol = integr.odeint(f, pi, t)
plt.grid()
plt.plot(t,sol)
plt.show()
```



Voici le cas classique du pendule pesant amorti. On trace d'abord l'angle en fonction du temps, puis le diagramme de phase.

```
from math import *
import numpy as np
import scipy as sp
import scipy.integrate as integr
import matplotlib as mp
import matplotlib.pyplot as plt
def f(xy,t):
   return [xy[1],10*sin(xy[0])-xy[1]/4]

t = np.linspace(0.0, 10.0, num=200)
sol = integr.odeint(f, [pi/2,0], t)
plt.grid()
plt.plot(t, sol[:,0]) # theta en fonction de t
plt.show()
plt.grid()
plt.plot(sol[:,0], sol[:,1]) # diagramme de phase
plt.show()
```





Le module optimize

Ce module propose en particulier deux fonctions qui permettent de trouver des valeurs approchées des zéros d'une fonction : `root` et `fsolve`. On renvoie au manuel de référence pour plus de détails.

```
from math import *
import numpy as np
import scipy as sp
import scipy.optimize
def f(x):
   return x-cos(x)

scipy.optimize.fsolve(f,0)        array([ 0.73908513])
def g(x):
   return x-tan(x)

scipy.optimize.root(g,4.4)        # renvoie un ensemble d'informations :
  status: 1
 success: True
     qtf: array([  1.91077716e-07])
    nfev: 9
       r: array([-20.19015893])
     fun: array([  5.38058487e-12])
       x: array([ 4.49340946])
 message: 'The solution converged.'
    fjac: array([[-1.]])
```

Le module linalg

Le module est riche de très nombreuses fonctions. Nous n'en présentons qu'un sous-ensemble très restreint. On pourrait évoquer également le calcul de différentes normes matricielles à l'aide de la fonction `norm`, les multiples factorisations standard (LU, Cholesky, QR, etc.)…

Le calcul matriciel est facilité dès qu'on utilise le type `matrix` de NumPy. Il suffit alors d'utiliser les symboles * et ** pour la multiplication et l'élévation à la puissance.

```
import numpy as np
import scipy as sp
import scipy.linalg as LA
a = np.matrix([[1,2],[3,4]])
b = np.matrix([[7,-2],[0,4]])
a                       matrix([[1, 2], [3, 4]])
a*a                     matrix([[ 7, 10], [15, 22]])
a ** 3                  matrix([[ 37,  54], [ 81, 118]])
a ** (-1)               matrix([[-2. ,  1. ], [ 1.5, -0.5]])
3*a-4*b                 matrix([[-25,  14], [  9,  -4]])
a+np.identity(2)        matrix([[ 2.,  2.], [ 3.,  5.]])
LA.inv(a)               # inverse de a
LA.det(a)               # déterminant de a
# résolution du système AX = B
LA.solve(a,[4,-6])      array([-14., 9.])
LA.eigvals(a)           array([-0.37228132+0.j,  5.37228132+0.j])
LA.eig(a)               (array([-0.37228132+0.j,  5.37228132+0.j]),
          array([[-0.82456484, -0.41597356],[ 0.56576746, -0.90937671]]))
```

Le module stats

Ce module distingue les distributions continues (du type `rv_continuous`) et les distributions discrètes (du type `rv_discrete`). Parmi les premières, on peut citer `norm` (normale), `expon` (exponentielle), `uniform` (uniforme) ou `triang` (triangulaire). Parmi les deuxièmes, `bernoulli`, `binom` (binomiale), `geom` (géométrique), `hypergeom` (hypergéométrique), `poisson`, `randint` (loi discrète uniforme).

Le fonctionnement est un peu particulier. On commence par créer un objet `va` représentant la variable aléatoire, suivant la distribution de son choix. On dispose alors de la fonction de densité de la probabilité en appelant `va.pdf` *(probability density function)*, de sa primitive en appelant `va.cdf` *(cumulative density function)*, de la réciproque de cette dernière en appelant `va.ppf` *(percent point function)*. On peut réaliser des tirages à l'aide de `va.rvs` *(random variable)*.

Voici un premier exemple avec la loi normale centrée.

```
import numpy as np
import scipy as sp
from scipy.stats import norm
va = norm()           # loi normale réduite centrée
va.rvs(size=10)       array([ 0.965682, -0.638357, -0.827413,  1.287467,
          0.498022, -0.553815,  0.148200, -1.495644, -0.97127,  0.02743 ])
va.pdf(0)             0.3989422804014327
va.pdf(-1)            0.24197072451914337
va.pdf(1)             0.24197072451914337
va.pdf(2.5)           0.01752830049356854
va.cdf(0)             0.5
va.cdf(2)             0.97724986805182079
va.cdf(2)-va.cdf(-2)  0.95449973610364158
va.ppf(0.95)          1.6448536269514722
```

On peut modifier les paramètres de la loi normale : `loc` pour la moyenne et `scal` pour l'écart-type.

```
va = norm(loc=10,scal=4) # on a modifé moyenne et écart-type
va.rvs(size=4)        array([10.7891974,9.0883591,9.8668062,10.0332539])
va.pdf(10)            0.3989422804014327
va.cdf(10)            0.5
va.ppf(0.95)          11.644853626951472
```

On fait de façon similaire pour la loi exponentielle, par exemple. Les paramètres sont `loc` (0 par défaut) et `scal` (1 par défaut, c'est le classique 1/λ).

```
from scipy.stats import expon
va = expon(loc=0,scale=2.5)
va.pdf(0)             0.40000000000000002
va.cdf(2.5)           0.63212055882855767
va.cdf(5)             0.8646647167633873
va.ppf(0.5)           1.7328679513998633
```

Le mode d'emploi est analogue pour les variables discrètes.

Par exemple pour une loi binomiale :

```
from scipy.stats import binom
va = binom(30,0.34)   # binomiale, 30 essais, p=0.34
va.rvs(size=10)       array([ 8, 13, 12, 11, 12, 11,  7,  7,  8, 14])
va.pmf(3)             0.0021411596194044454
va.pmf(10)            0.15255986875856462
va.cdf(10)            0.55403316254290469
va.cdf(10.1)          0.55403316254290469
va.ppf(0.55)          10.0
va.ppf(0.56)          11.0
```

Ou encore une loi de Poisson :

```
from scipy.stats import poisson
va = poisson(2.4)
va.rvs(size=10)       array([2, 1, 3, 6, 1, 2, 4, 3, 2, 3])
va.pmf(3)             0.20901416437880638
va.pmf(10)            0.00015850487634501214
va.cdf(4)             0.90413140966360084
va.cdf(4.1)           0.90413140966360084
va.ppf(0.8)           4.0
va.ppf(0.7)           3.0
```

Matplotlib

Le module est riche, et foisonnant. Nous nous attacherons à montrer, sur quelques exemples, les éléments les plus couramment utilisés.

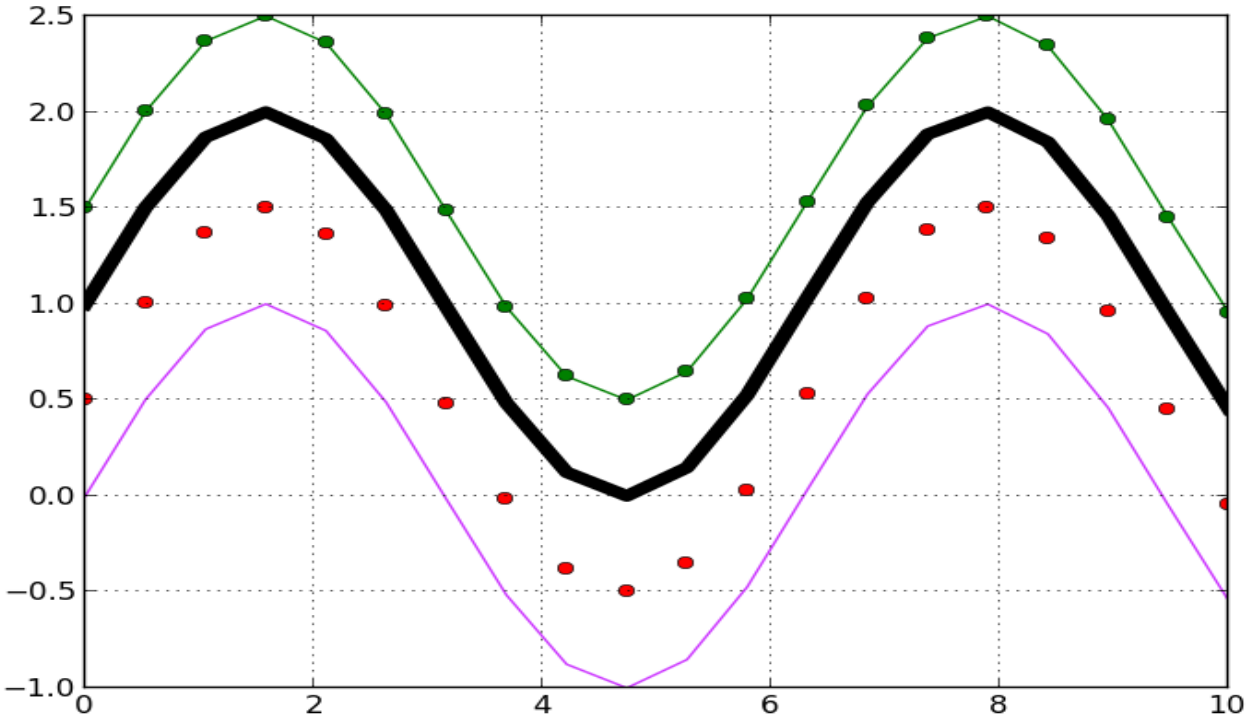
Commençons par tracer quelques courbes, en variant leur représentation : le troisième argument permet de préciser la couleur et le type de tracé. Le premier caractère désigne la couleur (`r` pour rouge, `b` pour bleu, `k` pour noir, `g` pour vert, `c` pour cyan, `m` pour magenta, `y` pour jaune, `w` pour blanc), les suivants précise le type du tracé (`-` pour des segments de droite, `.` pour des points isolés, `o` pour des « gros points » isolés, `o-` pour des gros points reliés par des segments, etc.), et on peut ajouter des arguments optionnels pour préciser son intention, comme par exemple la largeur des traits avec `linewidth`.

Si on veut paramétrer plus finement les couleurs, on peut utiliser un triplet de valeurs de l'intervalle [0,1] pour représenter les composantes RGB d'une couleur.

Par défaut, les points sont reliés par des segments, et la couleur change pour chaque nouvelle courbe.

Voici un premier exemple de tracé.

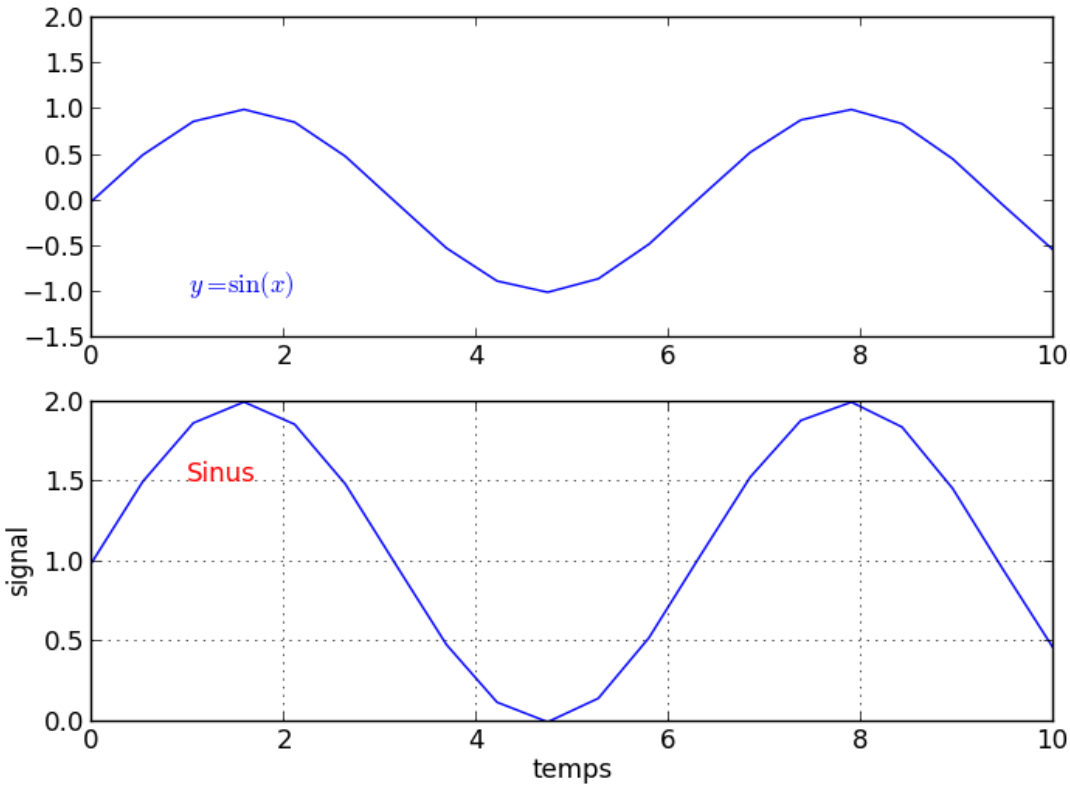
```
from math import *                  # dans les exemples ultérieurs,
import numpy as np                  # ces quatre lignes d'import
import matplotlib as mp             # seront supposées écrites
import matplotlib.pyplot as plt     # mais ne seront pas reproduites

y = []
t = np.linspace(0.,10.,20)
for k in range(4):
    y.append([k/2.0+sin(x) for x in t])

plt.plot(t,y[0],color=(0.8,0.2,1.0))
plt.plot(t,y[1],'ro')
plt.plot(t,y[2],'k-',linewidth=6)
plt.plot(t,y[3],'go-')
plt.grid()
plt.show()
```



On peut préciser l'étendue des axes par la commande axis, ajouter du texte à une position donnée, légender les axes. On notera que l'on peut même demander à composer du texte mathématique en LaTeX, avec une syntaxe très simple : `r'$y=\sin x$'` par exemple.

On peut créer des sous-graphiques : on précisera dans un appel à `subplot` le nombre de lignes et de colonnes de sous-graphiques, et en troisième argument le numéro du sous-graphique qu'on s'apprête à décrire.

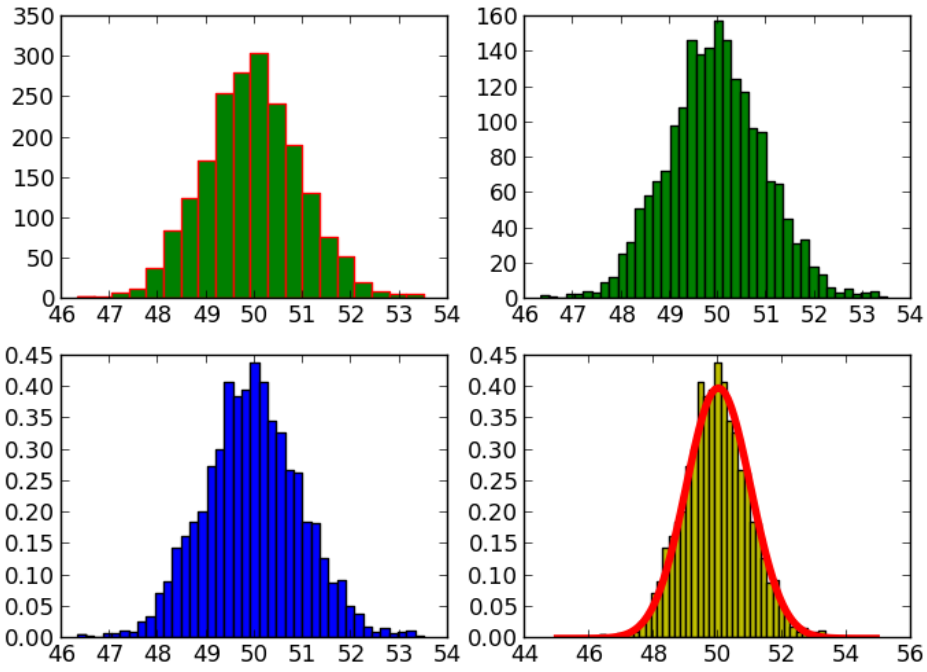
```
t = np.linspace(0.,10.,20)

plt.subplot(2,1, 1)   # 1er sous-graphe, parmi deux lignes et une colonne
plt.plot(t, [sin(x) for x in t])
plt.axis([0,10,-1.5,2])
plt.text(1,-1, r'$y=\sin(x)$', color='b')

plt.subplot(2,1, 2)   # 2e sous-graphe, parmi deux lignes et une colonne
plt.plot(t, [sin(x)+1 for x in t])
plt.text(1,1.5, 'Sinus', color='r')
plt.grid()
plt.xlabel('temps')
plt.ylabel('signal')

plt.show()
```



Histogrammes

L'exemple ci-dessous montre comment tracer des histogrammes.

```
from scipy.stats import norm

va = norm(loc=50, scal=14)   # loi normale, moyenne 50, écart-type 14
data = va.rvs(size=2000)     # tirage de 2000

plt.subplot(2,2,1)
(n,bins,patches) = plt.hist(data,20,fc='g',ec='r')
                      # fc : couleur des barres, ec : couleur des contours

plt.subplot(2,2,2)
(n,bins,patches) = plt.hist(data,40,color='g')
                      # 40 bâtons au lieu de 20

plt.subplot(2,2,3)
(n,bins,patches) = plt.hist(data,40,normed=True)
                      # normalisation (distribution de probabilité)

plt.subplot(2,2,4)
(n,bins,patches) = plt.hist(data,40,normed=True,color='y')
t=np.arange(45,55,0.1)
plt.plot(t,[va.pdf(x) for x in t],'r',linewidth=4)
plt.show()
```



Contours (ou lignes de niveaux)

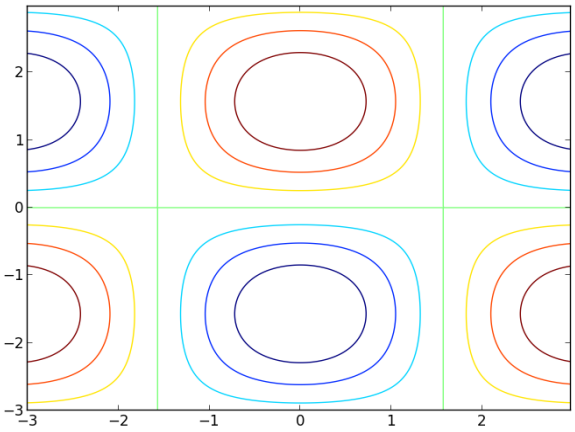
Tout d'abord, présentons la fonction `meshgrid` du module NumPy.

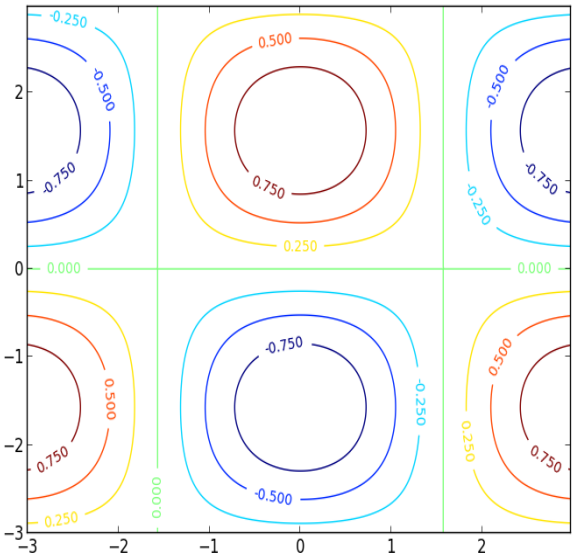
```
x = [1,2,3]
y = [4,5]
X,Y = np.meshgrid(x,y)
X                    array([[1,2,3], [1,2,3]])
Y                    array([[4,4,4], [5,5,5]])

# autre solution, presque équivalente

X = np.zeros((len(y),len(x)))
Y = np.zeros((len(y),len(x)))
for i in range(len(y)):
    for j in range(len(x)):
        X[i,j] = x[j]
        Y[i,j] = y[i]

X                    array([[1.,2.,3.], [1.,2.,3.]])
Y                    array([[4.,4.,4.], [5.,5.,5.]])
```

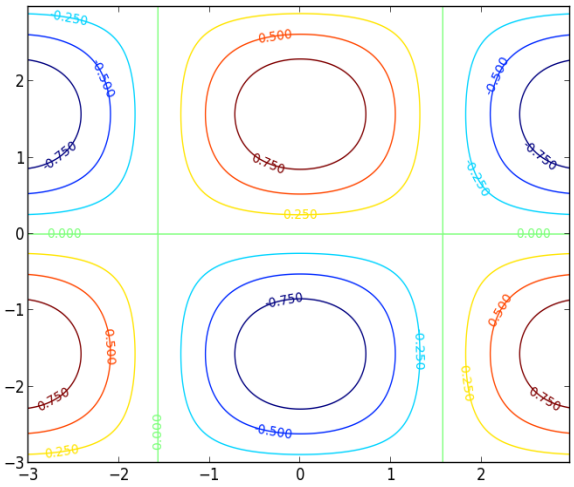
Venons en au tracé de contours.

```
from math import *
import numpy as np
import matplotlib as mp
import matplotlib.pyplot as plt

x = np.arange(-3.0,3.0,0.025)
y = np.arange(-3.0,3.0,0.025)
n = len(x)

X,Y = np.meshgrid(x,y)
Z = np.zeros((n,n))

for i in range(n):
    for j in range(n):
        Z[i,j] = sin(x[i])*cos(y[j])

fig = plt.contour(X,Y,Z)
plt.show()

fig = plt.contour(X,Y,Z)
plt.clabel(fig, inline = True, fontsize = 10)
plt.show()

fig = plt.contour(X,Y,Z)
plt.clabel(fig, inline = False, fontsize = 10)
plt.show()
```

Ce script va créer successivement les trois figures ci-dessous.







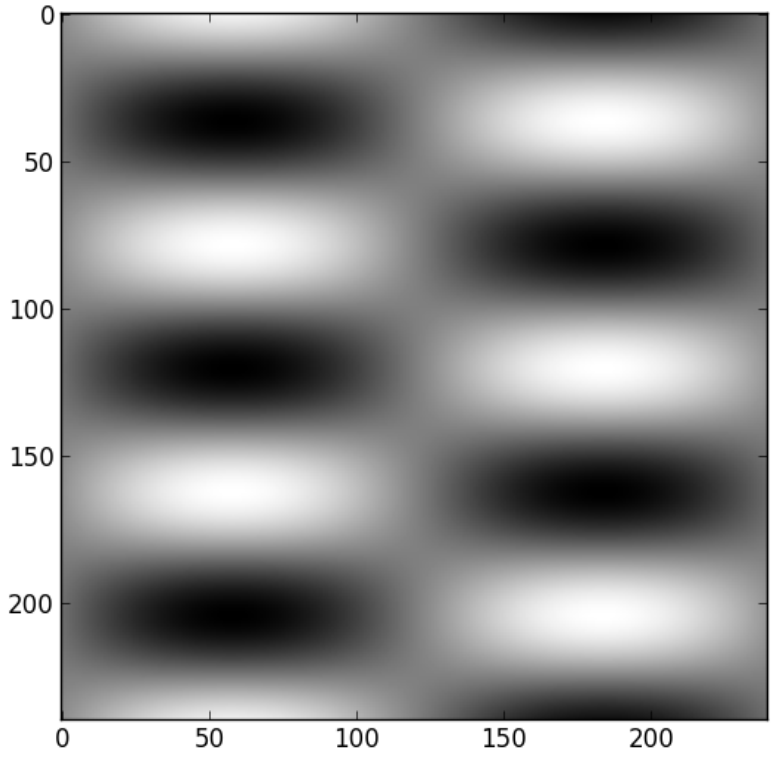
Images en niveau de gris (ou de couleurs)

On peut utiliser la fonction `imshow` de deux façons différentes : avec un argument indexé par (x,y), pour des images en niveau de gris, ou indexé par (x,y,k), pour des couleurs. Dans ce dernier cas, k=0 permet de régler le rouge, k=1 le vert et k=2 le bleu.

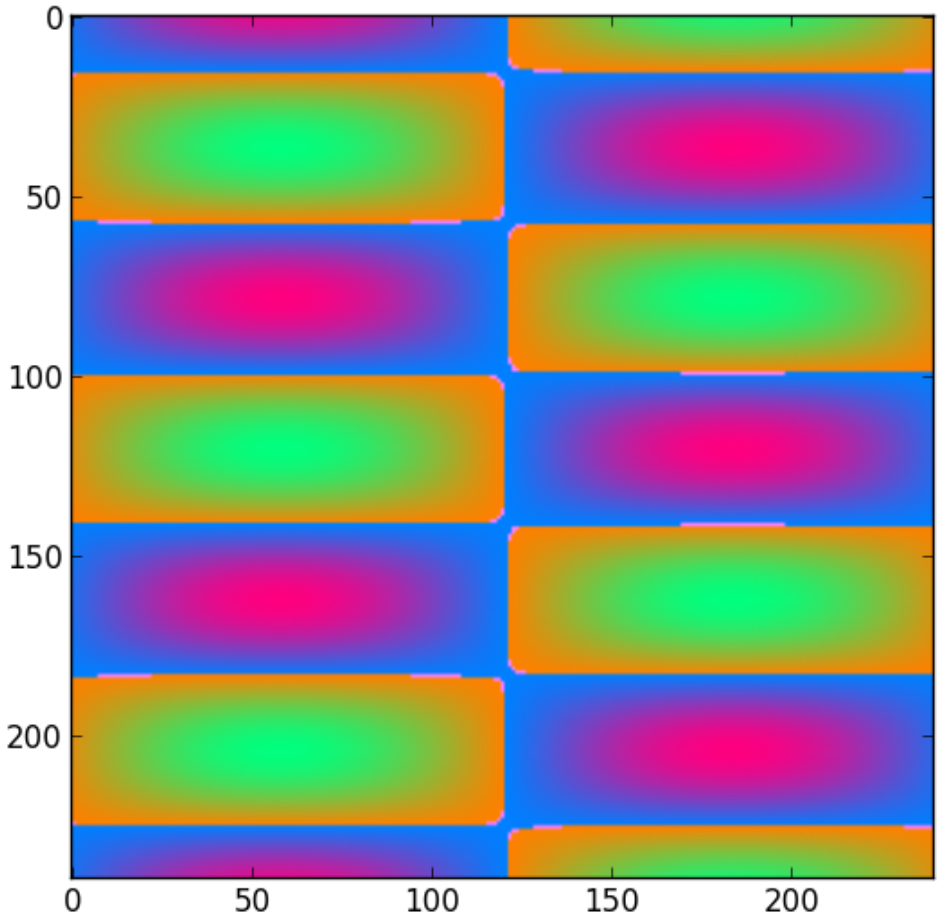
```
from math import *
import numpy as np
import matplotlib as mp
import matplotlib.pyplot as plt

x = np.arange(-3.0,3.0,0.025)
y = np.arange(-3.0,3.0,0.025)
n = len(x)

X,Y = np.meshgrid(x,y)
def f(x,y):
                        return(np.sin(x)*np.cos(3*y))

Z = f(X,Y)
plt.imshow(Z,cmap=mp.cm.gray)
```

Remarquons que l'utilisation des fonctions sinus et cosinus de Numpy ont permis de *vectoriser l'appel* f(X,Y) : il n'a pas été nécessaire d'écrire les deux boucles emboîtées

```
for i in range(n): for j in range(n): Z[i,j] = sin(x[i])*cos(y[j])
```



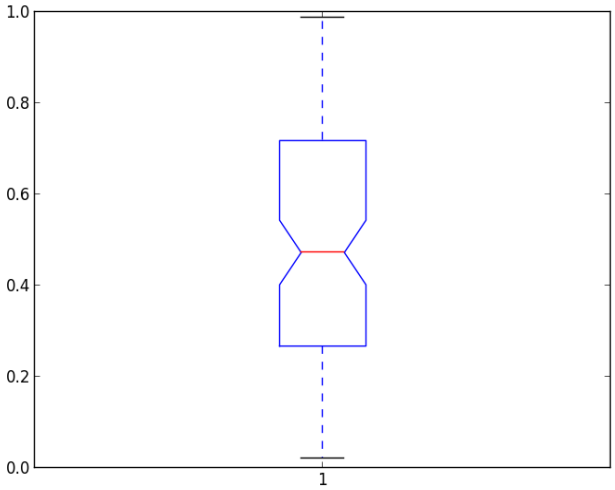
```
W = np.zeros([n,n,3])
W[:,:,0] = Z
W[:,:,1] = (1-Z)/2.0
W[:,:,2] = 0.5+W[:,:,1]
plt.imshow(W)
```

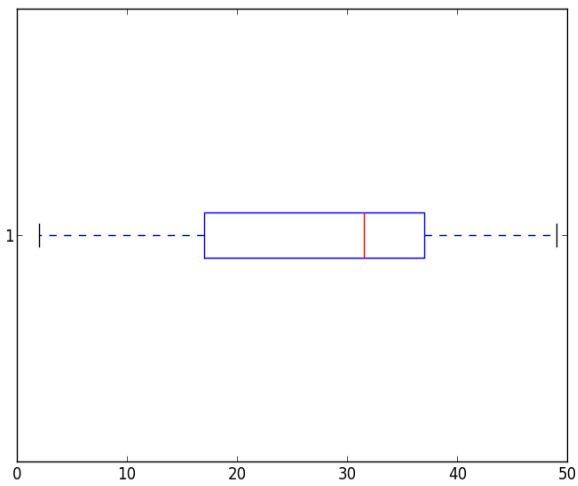


Boîtes à moustaches

On utilise la fonction boxplot de Matplotlib. Les valeurs par défaut sont : `notch=False` pour le « pincement », `vert=True` pour la disposition verticale/horizontale et `widths=0.5` pour la largeur de la boîte.

```
import numpy as np
import numpy.random as alea
import matplotlib as mp
import matplotlib.pyplot as plt

x = alea.rand(100)
plt.boxplot(x, notch = True)
plt.show()

y = alea.random_integers(0,50,size=40)
plt.boxplot(y, vert = False, widths=0.1)
plt.show()
```





Mises à jour

Ce document est mis à jour régulièrement. La dernière version est disponible à cette adresse, où on trouvera également d'autres documents utiles : http://goo.gl/AVV5t

Auteur

Laurent Chéno, laurent.cheno@education.gouv.fr

Licence d'utilisation de ce document