Bien démarrer en Numpy/Scipy/Matplotlib

Version alpha

Olivier Guibé

04 November 2010

Laboratoire de Mathématiques Raphaël Salem
UMR 6085 CNRS – Université de Rouen
Avenue de l'Universitée, BP.12
76801 Saint-Étienne du Rouvray, France
(olivier.guibe@univ-rouen.fr)

Table des matières

1 Introduction **2**
1.1 Installation 2
1.2 Importer les modules Numpy/Scipy/Matplotlib 2

2 Vecteurs, matrices et tableaux **2**
2.1 Les bases 2
2.2 Construire 5
2.3 Matrices particulières 8
2.4 Opérations 9
2.5 Booléens 10
2.6 Autres commandes 12

3 Calcul numérique matriciel **13**

4 Tracer les courbes **15**
4.1 Comportement en mode interactif 16
4.2 Des détails 17
4.3 Quelques exemples 18

Contents :

1 Introduction

Blabla sur Numpy/Scipy.

1.1 Installation

Sous Linux c'est facile, sous Windows je ne sais pas.

1.2 Importer les modules Numpy/Scipy/Matplotlib

Comme pour tous les modules Python une première méthode consiste à taper

```
>>> import numpy
```

Comme il devient très vite lassant de taper `numpy.X` il est courant d'importer Numpy sous une abréviation, comme par exemple

```
>>> import numpy as np
```

Ainsi toute commande spécificique de Numpy devient `np.X`. Il est **vivement déconseillé** de charger entièrement Numpy par la commande

```
>>> from numpy import *
```

Cela évite les conflits entre deux commandes de même nom mais issues de deux modules différents (voire avec une commande de Python). Dans la suite nous supposerons que `import numpy as np` a été utilisé.

Pour Scipy et Matplotlib l'appel suivant est courant

```
>>> import scipy as sp
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

Le module Scipy concerne le calcul scientifique en général (interpolation, fft, optimisation, algèbre linéaire). Certaines fonctions non présentes dans Numpy le sont dans Scipy. Dans ce cas `sp.triu` sera un appel à la fonction `triu` de Scipy tandis la signification de `mpl.plot` est laissée au lecteur.

Une autre solution consiste à utiliser le module `pylab` (quelles différences, à détailler).

2 Vecteurs, matrices et tableaux

2.1 Les bases

Numpy ajoute 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 voire de booléens.

Une première méthode consiste à convertir une liste en un tableau via la commande `array`. Le deuxième argument est optionnel et spécifie le type des éléments du tableau.

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1., 4., 5., 8.])
>>> type(a)
<type 'numpy.ndarray'>
```

Un tableau peut être multidimensionnel ; ici dimension 3

```
>>> a=np.array([[[1,2],[1,2]],[[1,2],[1,2]]])
>>> a
array([[[1, 2],
        [1, 2]],

       [[1, 2],
        [1, 2]]])
>>> type(a)
<type 'numpy.ndarray'>
>>> a[1,1,1]
2
```

Nous limiterons notre étude aux tableaux {uni,bi}-dimensionnels. En Python modifier une donnée d'une extraction d'un tableau entraîne aussi une modification du tableau initial ! Si nécessaire la fonction `np.copy(a)` ou `a.copy()` permet de faire une copie d'un tableau a.

```
>>> a=np.array([1, 2, 3, 4, 5])
>>> c=np.array([1, 2, 3, 4, 5])
>>> b=a[1:3]
>>> b
array([2, 3])
>>> b[1]=0
>>> a
array([1, 2, 0, 4, 5])   # a  modifié
>>> b=c[1:3].copy()      # première méthode
>>> b[1]=0
>>> bb=np.copy(c[1:3])   # seconde méthode
>>> bb[1]=0
>>> c
array([1, 2, 3, 4, 5])   # c non modifié
```

Comme pour les listes le *slicing* extrait les tableaux

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [début:fin:pas]
array([2, 5, 8])
>>> a[2:8:3] # le dernier élément n'est pas inclus
```

```
array([2, 5])
>>> a[:5]    # le dernier élément n'est pas inclus
array([0, 1, 2, 3, 4])
```

Dans l'instruction `[début:fin:pas]` deux des arguments peuvent être omis : par défaut l'indice de début vaut 0 (le 1er élément), l'indice de fin est celui du dernier élément et le pas vaut 1. Il faut tout de même noter une différence entre `[2:9]` et `[2:]`

```
>>> a[2:9]
array([2, 3, 4, 5, 6, 7, 8])
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])
```

Un pas négatif inversera l'ordre du tableau et les tableaux étant considérés cycliques la commande `a[-2::1]` extrait l'avant dernier et dernier élément. Pour un tableau bi-dimensionnel on peut bien sûr jouer avec les deux indices.

```
>>> a=np.eye(5,5)
>>> print a
[[ 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.]]
>>> a[:3,:4]=4
>>> a[::3,::4]=5
>>> a[:,3]=6
>>> print a
[[ 5.  4.  4.  6.  5.]
 [ 4.  4.  4.  6.  0.]
 [ 4.  4.  4.  6.  0.]
 [ 5.  0.  0.  6.  5.]
 [ 0.  0.  0.  6.  1.]]
```

Une autre méthode d'extraction consiste à écrire `a[b]` où `b` est un tableau d'entiers qui correspondra aux indices à extraire et `a` un vecteur. Pour les matrices c'est `a[b,c]` et on prend les indices de ligne dans `b`, les indices de colonnes dans `c`, successivement.

```
>>> a=np.array([2,4,6,8],float)
>>> b=np.array([0,0,1,3,2,1],int)
>>> a[b]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
>>> a=a.reshape(2,2)
>>> b=np.array([0,0,1,1,0],int)
>>> c=np.array([0,1,1,1,1],int)
>>> a[b,c]
array([ 2.,  4.,  8.,  8.,  4.])
>>>
```

Numpy/Scipy proposent aussi le type `mat` comme matrice, exclusivement un tableau bi-dimensionnel. Comme les fonctions telles que `ones`, `eye` retournent un objet de type `array`

nous n'utiliserons pas dans la suite le type `mat`. Il permet cependant de saisir les matrices à-la-Matlab et de faire le produit matriciel par le simple symbole *

```
>>> a=np.mat('[1 2 4 ; 3 4 5.]')
>>> b=np.mat('[2. ; 4 ; 6]')
>>> print a
[[ 1.  2.  4.]
 [ 3.  4.  5.]]
>>> print b
[[ 2.]
 [ 4.]
 [ 6.]]
>>> print a*b
[[ 34.]
 [ 52.]]
```

Tout objet `array` est convertible en type `mat` et réciproquement (sous la condition que le tableau (`array`) soit {uni,bi}-dimensionnel)

```
>>> a=np.array([1, 2, 4])
>>> np.mat(a)
matrix([[1, 2, 4]])
```

2.2 Construire

Nous avons déjà vu comment accéder aux éléments d'un tableau soit par `a[i,j]` soit par le *slicing*. Pour définir certaines matrices (au sens `array` {uni,bi}-dimensionnel) il est possible d'utiliser les boucles, des matrices prédéfinies de Numpy (Scipy) et les opérations élémentaires (multiplication par un scalaire, produit matriciel, transposition, produit de Kronecker, jeu avec les booléens, etc).

Numpy propose `arange` équivalent de `range` mais de type `array`, ce qui évite une conversion.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=np.float)
array([ 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([ 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
```

Comme il y quelques subtilités dans la fonction `range` et `arange` quant au dernier élément, Pour éviter tout désagrément la fonction `linspace(premier,dernier,n)` renvoie un `array` commençant par `premier`, se terminant par `dernier` avec `n` éléments régulièrement espacés.

```
>>> np.linspace(1., 4., 6)
array([ 1. ,  1.6,  2.2,  2.8,  3.4,  4. ])
```

Numpy propose le redimensionnement d'un tableau avec la fonction `reshape`. Il faut tout de même respecter une condition : le nombre d'éléments doit être le même !

```
>>> a=np.arange(16)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> a.reshape(4,4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a.reshape(2,8)
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
```

Il est possible aussi de transformer un `array` bi-dimensionnel en un `array` uni-dimensionnel avec `reshape` ou `flatten`. L'instruction `b.flatten()` renvoie une copie de `b`, ce qui n'est pas le cas de `reshape`. Ici un exemple avec la syntaxe `np.reshape(array,dimension)`

```
>>> b=np.ones((4,4),float)+np.eye(4,4)
>>> print b
[[ 2.  1.  1.  1.]
 [ 1.  2.  1.  1.]
 [ 1.  1.  2.  1.]
 [ 1.  1.  1.  2.]]
>>> np.reshape(b,16)
array([ 2.,  1.,  1.,  1.,  1.,  2.,  1.,  1.,  1.,  1.,  2.,  1.,  1.,
        1.,  1.,  2.])
>>> np.reshape(b,(2,8))
array([[ 2.,  1.,  1.,  1.,  1.,  2.,  1.,  1.],
       [ 1.,  1.,  2.,  1.,  1.,  1.,  1.,  2.]])
>>> b.flatten()
array([ 2.,  1.,  1.,  1.,  1.,  2.,  1.,  1.,  1.,  1.,  2.,  1.,  1.,
        1.,  1.,  2.])
```

Numpy permet d'assembler les vecteurs et les matrices, de les concaténer. Par défaut l'assemblage se fait selon la 1ère dimension (les lignes, donc assemblage vertical). L'option `axis=1` assemble "horizontalement".

```
>>> a=np.arange(4).reshape(2,2)
>>> b=4+np.arange(4).reshape(2,2)
>>> b
array([[4, 5],
       [6, 7]])
>>> c=(np.arange(4)+6)[newaxis,:]
>>> np.concatenate((a,b))
```

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
>>> np.concatenate((a,b),axis=0)
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
>>> np.concatenate((a,b),axis=1)
array([[0, 1, 4, 5],
       [2, 3, 6, 7]])
>>> np.concatenate((c,c,np.concatenate((a,b),axis=1)))
array([[6, 7, 8, 9],
       [6, 7, 8, 9],
       [0, 1, 4, 5],
       [2, 3, 6, 7]])
```

Pour concaténer un vecteur et une matrice il est nécessaire de convertir le vecteur en matrice à 1 ligne (ou 1 colonne). Par rapport à Matlab/Scilab c'est bien dommage. Au passage la commande `np.newaxis` permet d'ajouter une dimension à un tableau (si `b` est un tableau unidimensionnel `b[np.newaxis,:]` retournera une matrice à 1 ligne tandis que `b[:,np.newaxis]` retournera une matrice à 1 colonne).

```
>>> c=(np.arange(4)+6)[np.newaxis,:]
>>> d=(np.arange(4)+6)
>>> np.concatenate((c,d))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: arrays must have same number of dimensions
>>> np.concatenate((c,d),axis=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: arrays must have same number of dimensions
>>> np.concatenate((c,d[np.newaxis,:]),axis=0)
array([[6, 7, 8, 9],
       [6, 7, 8, 9]])
>>> np.concatenate((c.transpose(),d[:,np.newaxis]),axis=0)
array([[6],
       [7],
       [8],
       [9],
       [6],
       [7],
       [8],
       [9]])
```

Le “slicing” permet d'extraire des éléments d'un vecteur ou matrice. Il existe aussi la commande `np.take` qui extrait lignes/colonnes d'un `array`. L'option `axis` détermine une extraction selon les lignes/colonnes. La syntaxe se comprend aisément.

```
>>> a=np.arange(16).reshape(4,4)
>>> a.take([0,3],axis=1)
array([[ 0,  3],
       [ 4,  7],
       [ 8, 11],
       [12, 15]])
>>> a.take([0,3])
array([0, 3])
>>> a.take([0,3],axis=0)
array([[ 0,  1,  2,  3],
       [12, 13, 14, 15]])
>>> a.take([0,3,2,2,2],axis=0)
array([[ 0,  1,  2,  3],
       [12, 13, 14, 15],
       [ 8,  9, 10, 11],
       [ 8,  9, 10, 11],
       [ 8,  9, 10, 11]])
```

La réciproque de `np.take` est `np.put`

```
>>> a=np.array([0, 1, 2, 3, 4, 5],float)
>>> b=np.array([9,8,7],float)
>>> a.put([0,3],b)
>>> a
array([ 9.,  1.,  2.,  8.,  4.,  5.])
```

Comme deux positions seulement ont été données, seules la valeurs 0 et 3 ont été remplacées.

2.3 Matrices particulières

Nous avons déjà utilisé certaines commandes

Commande	Description (rapide)
`np.zeros(n) ((n,p))`	vecteur nul de taille `n`, matrice nulle de taille `n,p`
`np.eye(n) ((n,p))`	matrice de taille `n` (`n,p`) avec des 1 sur la diagonale et des zéros ailleurs
`np.ones(n) ((n,p))`	vecteur de taille `n`, matrice de taille `n,p` remplie de 1
`np.diag(v)`	matrice diagonale dont la diagonale est le vecteur `v`
`np.diag(v,k)`	matrice dont la ‘diagonale’ décalée de `k` est le vecteur `v` (`k` est un entier relatif)
`np.random.rand(n) ((n,p))`	vecteur (taille `n`), matrice (taille `n,p`) à coefficients aléatoires uniformes sur [0,1]

```
>>> v=np.ones(5)
>>> v
array([ 1.,  1.,  1.,  1.,  1.])
>>> np.diag(v,2)
array([[ 0.,  0.,  1.,  0.,  0.,  0.,  0.],
```

```
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> np.diag(v,-1)
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.]])
```

2.4 Opérations

L'opérateur + additionne terme à terme deux tableaux de même dimension. La commande `2.+a` renvoie le vecteur/matrice dont tous les éléments sont ceux de a plus 2. Multplier un vecteur/matrice par un scalaire se fait selon le même principe : la commande `2*a` renvoie le vecteur/matrice de même dimension dont tous les éléments ont été multipliés par 2.

L'opérateur * ne fait que multiplier terme à terme deux tableaux de même dimension :

```
>>> 4*np.ones((4,4))*np.diag([2, 3, 4., 5])
array([[  8.,   0.,   0.,   0.],
       [  0.,  12.,   0.,   0.],
       [  0.,   0.,  16.,   0.],
       [  0.,   0.,   0.,  20.]])
```

Dans le même ordre `a**3` renvoie le vecteur/matrice de même dimension dont tous les éléments sont ceux de a élevés à la puissance 3. Devinez ce que fait `1./a` ?

Pour le produit matriciel (le vrai) la commande `np.dot` est là

```
>>> np.dot(4*np.ones((4,4)),np.diag([2, 3, 4., 5]))
array([[  8.,  12.,  16.,  20.],
       [  8.,  12.,  16.,  20.],
       [  8.,  12.,  16.,  20.],
       [  8.,  12.,  16.,  20.]])
```

Si `v` est un vecteur (`array` uni-dimensionnel) et `A` une matrice (`array` bi-dimensionnel) alors `np.dot(A,v)` renvoie le produit Av tandis que `np.dot(v,A)` renvoie $v^t A$. Si le produit n'est pas possible, Python vous le signale.

```
>>> a=np.arange(4).reshape(2,2)
>>> v=np.array([-3,2])
>>> np.dot(a,v)
array([2, 0])
>>> np.dot(v,a)
array([4, 3])
```

```
>>> w=np.concatenate((v,v))
>>> w
array([-3,  2, -3,  2])
>>> dot(a,w)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
```

`np.vdot(v,w)` : produit scalaire des vecteurs `v` et `w`.

Il y a enfin le produit de Kronecker, `np.kron`, assez pratique pour créer des matrices/vecteurs particuliers. Si A et B sont deux matrices non nécessairement de même taille, le produit de Kronecker de A B est la matrice bloc (B de taille $n \times p$)

$$\begin{pmatrix} b_{11}A & b_{12}A & \cdots & b_{1p}A \\ \vdots & & & \vdots \\ b_{n1}A & b_{n2}A & \cdots & b_{np}A \end{pmatrix}$$

```
>>> a=np.arange(4,dtype=float).reshape(2,2)
>>> b=np.arange(5,dtype=float)
>>> print a,b
[[ 0.  1.]
 [ 2.  3.]] [ 0.  1.  2.  3.  4.]
>>> np.kron(a,b)
array([[  0.,   0.,   0.,   0.,   0.,   0.,   1.,   2.,   3.,   4.],
       [  0.,   2.,   4.,   6.,   8.,   0.,   3.,   6.,   9.,  12.]])
>>> np.kron(b,a)
array([[  0.,   0.,   0.,   1.,   0.,   2.,   0.,   3.,   0.,   4.],
       [  0.,   0.,   2.,   3.,   4.,   6.,   6.,   9.,   8.,  12.]])
```

Un autre exemple qui peut être utile pour certains exercices.

```
>>> a=np.arange(5,dtype=float)
>>> b=np.ones((1,5))
>>> np.kron(a,b.transpose())
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 0.,  1.,  2.,  3.,  4.],
       [ 0.,  1.,  2.,  3.,  4.],
       [ 0.,  1.,  2.,  3.,  4.],
       [ 0.,  1.,  2.,  3.,  4.]])
```

2.5 Booléens

On rappelle les opérateurs logiques en Python.

Symbole	Signification	Symbole	Signification
`a and b`	a et b	`a>b`	a>b
`a or b`	a ou b	`a>=b`	a≥b
`a==b`	égalité de a et b	`a != b`	a différent de b
`a<b`	a<b	`a<=b`	a≤b

Comme dans Matlab/Octave/Scilab, Numpy étend ==, <, >, <=, >= et != aux vecteurs/matrices (type `array`) et ajoute le "ou exclusif" avec `np.logical_xor(a,b)`. Les opérateurs de comparaisons ==, <, >, <=, >= et != existent aussi respectivement sous les noms de `equal(a,b)`, `np.less(a,b)`, `np.greater(a,b)`, `np.less_equal(a,b)`, `np.greater_equal(a,b)` et `np.not_equal(a,b)`.

Ce qui est formidable avec Numpy (si si) : `a` et `b` étant deux `array` de tailles identiques, la commande `a<b` renvoie un `array` de taille égale à celle de a dont les éléments sont des booléens et correspondent au test < terme à terme. Voici une illustration plus compréhensible

```
>>> b=6*np.eye(4,4)-np.diag(4+np.arange(3),1)+2*np.ones((4,4))
>>> a=np.arange(16).reshape(4,4)
>>> (a<b)
array([[ True, False, False, False],
       [False,  True, False, False],
       [False, False, False, False],
       [False, False, False, False]], dtype=bool)
>>> (a==b)
array([[False, False,  True, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]], dtype=bool)
```

Si les deux arguments sont de tailles différentes, Python vous insulte. Mais si `a` ou `b` est de type `float` ou `int`, la commande `a<2` renvoie le tableau de taille égale à celle de `a` correspondant à comparaison des éléments de a avec 2.

```
>>> (a<2.)
array([[ True,  True, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]], dtype=bool)
>>> (2<b)
array([[ True, False, False, False],
       [False,  True, False, False],
       [False, False,  True, False],
       [False, False, False,  True]], dtype=bool)
```

Encore plus formidable, si `d` est un tableau de booléens de taille égale à celle de `a`, `a[d]` extrait les éléments de a qui correspondent à la valeur `True` dans `d`. La commande `b[b<4]` renvoie les éléments de `b` strictement inférieurs à 4.

```
>>> b=6*np.eye(4,4)-np.diag(4+np.arange(3),1)+2*np.ones((4,4))
>>> b
array([[ 8., -2.,  2.,  2.],
```

```
       [ 2.,  8., -3.,  2.],
       [ 2.,  2.,  8., -4.],
       [ 2.,  2.,  2.,  8.]])
>>> b[b<4]
array([-2.,  2.,  2.,  2., -3.,  2.,  2.,  2., -4.,  2.,  2.,  2.])
```

(à faire : `np.all()` `np.any()`, *np.where*‘)

2.6 Autres commandes

Numpy permet de “vectoriser”, i.e. appliquer une fonction à un vecteur/matrice et éviter les boucles. Comme nous avons choisi d’utiliser Numpy à travers `import numpy as np`, il faut choisir les fonctions usuelles définies dans Numpy

```
>>> from math import cos
>>> a=np.arange(4, dtype=float)
>>> a
array([ 0.,  1.,  2.,  3.])
>>> cos(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
>>> np.cos(a)
array([ 1.        ,  0.54030231, -0.41614684, -0.9899925 ])
```

Minimum/maximum global ou selon une direction : `np.amin` et `np.amax`

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)           # élément minimum
0
>>> np.amin(a, axis=0)         # minimum selon ligne
array([0, 1])
>>> np.amin(a, axis=1)         # minimum selon colonne
array([0, 2])
```

Les commandes `np.argmin` et `np.argmax` retournent l’indice ou le tableau d’indices des éléments réalisant les extrema. Si plusieurs éléments réalisent l’extremum seul le premier indice est retourné.

```
>>> b=6*np.eye(4,4)-np.diag(4+np.arange(3),1)+2*np.ones((4,4))
>>> b
array([[ 8., -2.,  2.,  2.],
       [ 2.,  8., -3.,  2.],
       [ 2.,  2.,  8., -4.],
       [ 2.,  2.,  2.,  8.]])
>>> np.argmin(b)
```

```
11
>>> np.argmax(b)
0
>>> np.argmin(b,axis=1)
array([1, 2, 3, 0])
>>> np.argmin(b,axis=0)
array([1, 0, 1, 2])
```

Les sommes, produits de tous les éléments ou selon les lignes/colonnes se font aisément avec `np.sum` et `np.prod`. À vous de vérifier que les résultats sont justes.

```
>>> b=6*np.eye(4,4)-np.diag(4+np.arange(3),1)+2*np.ones((4,4))
>>> np.sum(b)
41.0
>>> np.prod(b)
-50331648.0
>>> np.sum(b,axis=0)
array([ 14.,  10.,   9.,   8.])
>>> np.sum(b,axis=1)
array([ 10.,   9.,   8.,  14.])
>>> np.prod(b,axis=1)
array([ -64.,  -96., -128.,   64.])
```

Pour faire les moyennes de tous les éléments ou selon les lignes/colonnes, `np.mean` a le fonctionnement attendu.

3 Calcul numérique matriciel

Très rapidement, listons les commandes. Certaines commandes étant spécialisées (tout le monde n'a pas besoin d'inverser les matrices) il faut ajouter `linalg` : `np.linalg.X`

- rang d'une matrice : `np.rank(a)`

```
>>> a=np.arange(15).reshape(3,5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> np.rank(a)
2
>>> np.rank(a.transpose())
2
```

- inverse d'une matrice : `np.linalg.inv(a)` inverse d'une matrice carrée

```
>>> a=np.array([2,4,6,8],float).reshape(2,2)
>>> np.linalg.inv(a)
array([[-1.  ,  0.5 ],
       [ 0.75, -0.25]])
```

Comme d'habitude avec les logiciels de calcul scientifique, il faut d'abord savoir si la matrice est inversible pour l'inverser, ou encore rester critique vis à vis du résultat retourné. L'exemple suivant est caractéristique.

```
>>> a=np.arange(16).reshape(4,4)
>>> a
array([[ 0,  1,  2,  3],
     [ 4,  5,  6,  7],
     [ 8,  9, 10, 11],
     [12, 13, 14, 15]])
>>> np.rank(a)  # la matrice n'est pas inversible
2
>>> np.linalg.inv(a)
array([[  9.00719925e+14,  -4.50359963e+14,  -1.80143985e+15,
          1.35107989e+15],
       [ -2.40191980e+15,   2.70215978e+15,   1.80143985e+15,
         -2.10167983e+15],
       [  2.10167983e+15,  -4.05323966e+15,   1.80143985e+15,
          1.50119988e+14],
       [ -6.00479950e+14,   1.80143985e+15,  -1.80143985e+15,
          6.00479950e+14]])
```

Les valeurs très grandes laissent tout de même planer un certain soupçon.

- résolution d'un système linéaire : `np.linalg.solve(a,b)` où a est une matrice carrée et b un vecteur ou une matrice (avec condition de compatibilité)

```
>>> a=np.array([2,4,6,8],float).reshape(2,2)
>>> b=np.array([1, 4],float)
>>> np.linalg.solve(a,b)
array([ 1.  , -0.25])
>>> b=np.array([[1, 1],[4, -4]],float)
>>> np.linalg.solve(a,b)
array([[ 1.  , -3.  ],
     [-0.25,  1.75]])
```

Il en est de même que pour l'inversion de matrice : il faut savoir rester critique.

```
>>> a=np.arange(16,dtype=float).reshape(4,4)
>>> b=np.arange(4,dtype=float)
>>> b[1]=5
>>> x=np.linalg.solve(a,b)
>>> np.dot(a,x)
array([ -2.,   2.,   6.,  10.])
>>> b
array([ 0.,  5.,  2.,  3.])
```

- calcul du déterminant : `np.linalg.det(a)` bien sûr !
- calcul des valeurs propres et vecteurs propres : `np.linalg.eig(a)`

```
>>> a=np.array([2,4,6,8],float).reshape(2,2)
>>> np.linalg.eig(a)
(array([ -0.74456265,  10.74456265]), array([[-0.82456484, -0.41597356],
     [ 0.56576746, -0.90937671]]))
```

Le premier argument représente les valeurs propres, tandis que le deuxième retourne la matrice des vecteurs propres en colonne (une colonne est un vecteur propre).

- calcul de norme vectorielle/matricielle : `np.linalg.norm(x[, ord])` où `ord` est pour une matrice
  - `Fro` , norme de Frobenius
  - `inf` : `max(sum(abs(x), axis=1))`
  - `-inf` : `min(sum(abs(x), axis=1))`
  - `1` : `max(sum(abs(x), axis=0))`
  - `-1` : `min(sum(abs(x), axis=0))`
  - `2` : norme 2 matricielle
  - `-2` : plus petite valeur singulière

  pour un vecteur
  - `Fro` , norme de Frobenius
  - `inf` : `max(sum(abs(x), axis=1))`
  - `-inf` : `min(sum(abs(x), axis=1))`
  - les autres donnent les mêmes résultats (à tester)

4 Tracer les courbes

Le module Matplotlib est chargé de tracer les courbes :

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

D'une manière générale les fonctions `plt.plot` attendent des vecteur/matrice, bref des tableaux de points du plan. Selon les options, ces points du plan sont reliés entre eux de façon ordonnée par des segments : le résultat est une courbe.

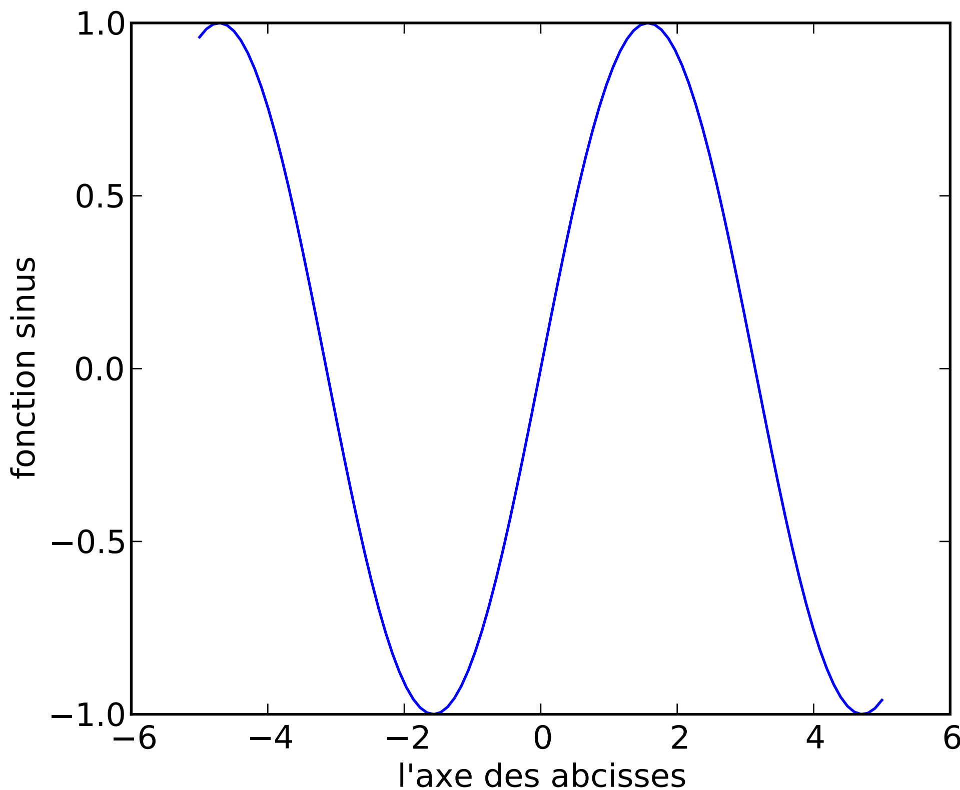
Commençons par la fonction sinus.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
plt.plot(x,np.sin(x))  # on utilise la fonction sinus de Numpy
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 menus dans le bas de cette fenêtre : zoomer, déplacer la figure, etc et surtout sauvegarder dans un format PNG, PDF, EPS, etc.

`plt.clf()` efface la fenêtre graphique

`plt.savefig()` sauvegarde le graphique. Par exemple `plt.savefig("mongraphe.png")` sauve sous le nom "mongraphe.png" le graphique. Par défaut le format est PNG. Il est possible d'augmenter la résolution, la couleur de fond, l'orientation, la taille (a0, a1, lettertype, etc) et aussi 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”.

4.1 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("ce que vous voulez")` modifiera l'étiquette sous l'axe des abcisses. Si vous fermez la fenêtre alors la commande `plt.xlabel("ce que vous voulez")` se contentera de faire afficher une fenêtre graphique avec axe des abcisses, des ordonnées allant de 0 à 1 et une étiquette “ce que vous voulez” sous l'axe des abcisses. 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.

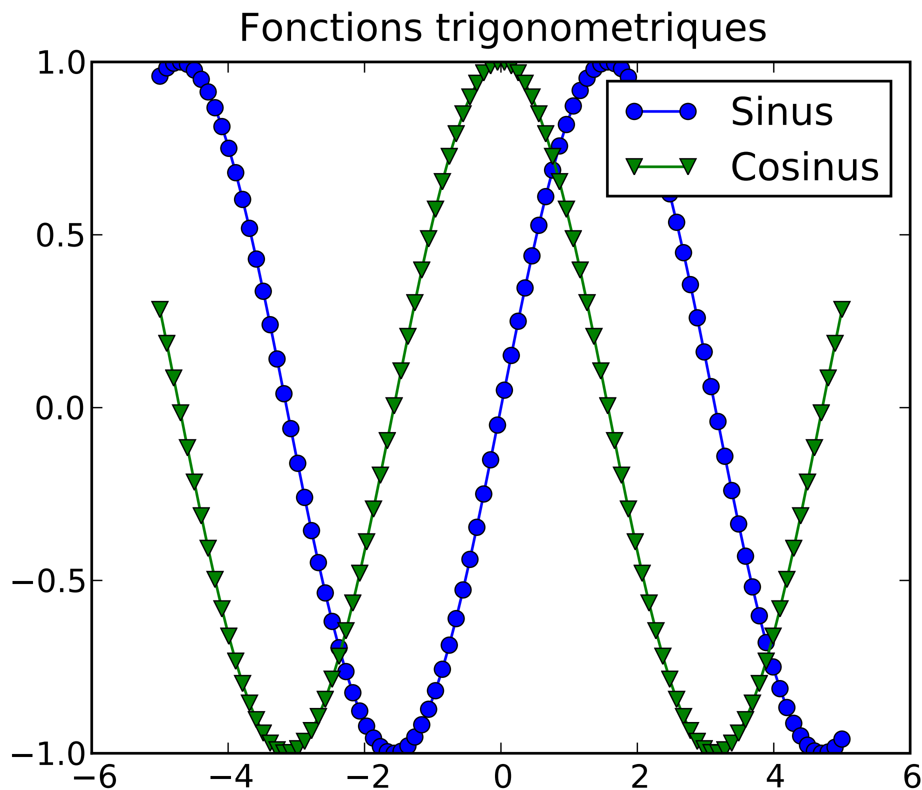
4.2 Des détails

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

- bornes : 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])`
- 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 `color='(un flottant entre 0 et 1)'`. Enfin pour avoir encore plus de couleurs, comme en HTML la séquence `color='#eeefff'` donnera la couleur attendu et les amateurs de RGB sont servis par `color=( R, G, B)` avec trois paramètres compris entre 0 et 1 (RGBA est possible aussi).
- symboles : 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' | ' ' | '' ]`.
- 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”. Plutôt que `linestyle`, `ls` (plus court) fait le même travail.
- épaisseur du trait : `linewidth=flottant` (comme `linewidth=2`) donne un trait, pointillé (tout ce qui est défini par style du trait) d'épaisseur “flottant” en `points`. Il est possible d'utiliser `lw` en lieu et place de `linewidth`.
- 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. Remarquez que si la couleur n'est pas spécifiée pour chaque nouvel appel la couleur des “markers” change de façon cyclique.
- étiquettes sur l'axe 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` est un vecteur de flottants ordonnés de façon croissante.
- ajouter un titre : `plt.title("Mon titre")`
- légendes : c'est un peu plus compliqué. D'après ce que j'ai compris il faut assigner à des variables le tracé, via `g1=plt.plot()`, etc. Enfin `plt.legend((g1, g2), ("ligne 2","ligne 1"))` fait le boulot. Par exemple

```python
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
p1=plt.plot(x,np.sin(x),marker='o')
p2=plt.plot(x,np.cos(x),marker='v')
plt.title("Fonctions trigonometriques")  # Problemes avec accents (plot_directive) !
```

```
plt.legend([p1, p2], ["Sinus", "Cosinus"])
plt.show()
```



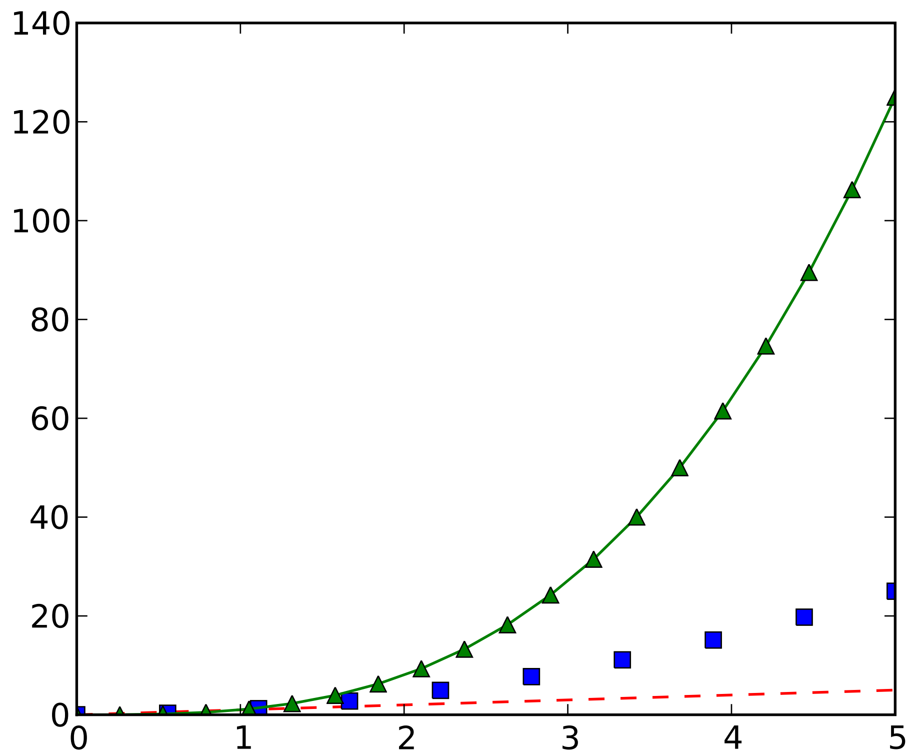
4.3 Quelques exemples

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()
```

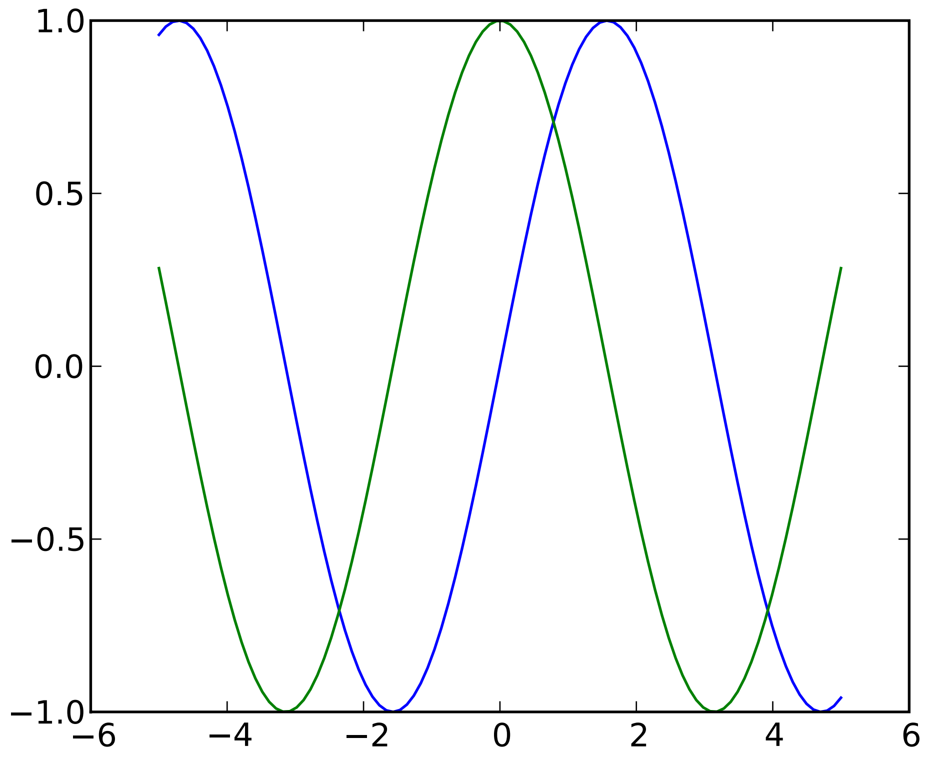
Si un seul vecteur (ou une seule matrice) est donné, on trace le graphe avec comme abscisse l'indice des éléments

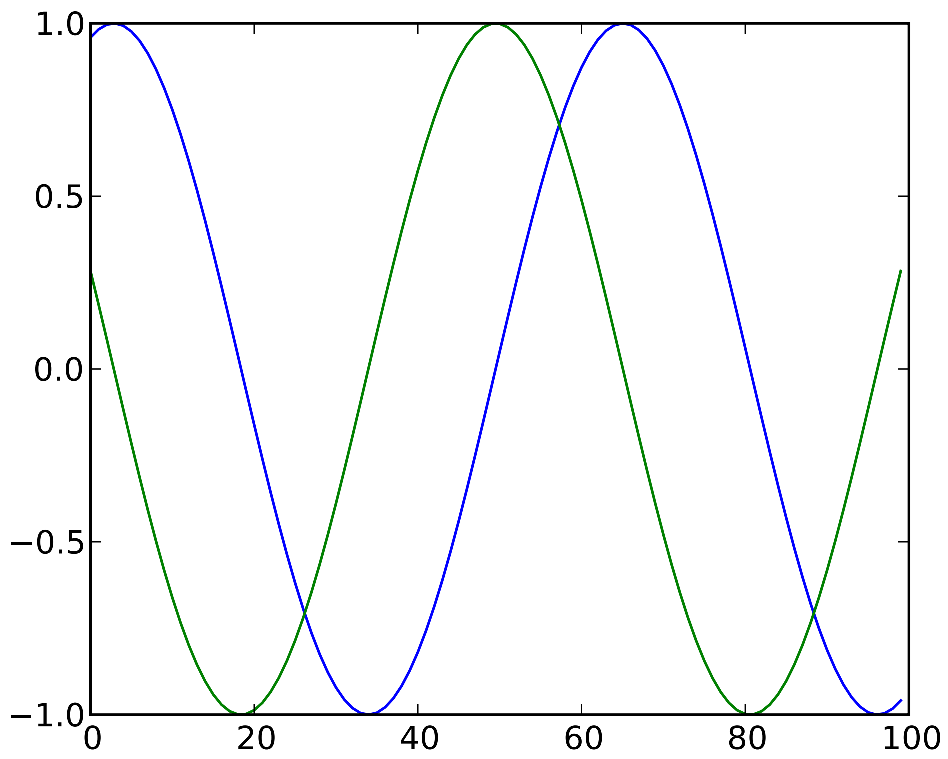
```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
y=np.zeros((100,2))
y[:,0]=np.sin(x)
y[:,1]=np.cos(x)
plt.plot(y)
plt.show()
```

Enfin un système de *sous-figures* permet de juxtaposer différents graphiques

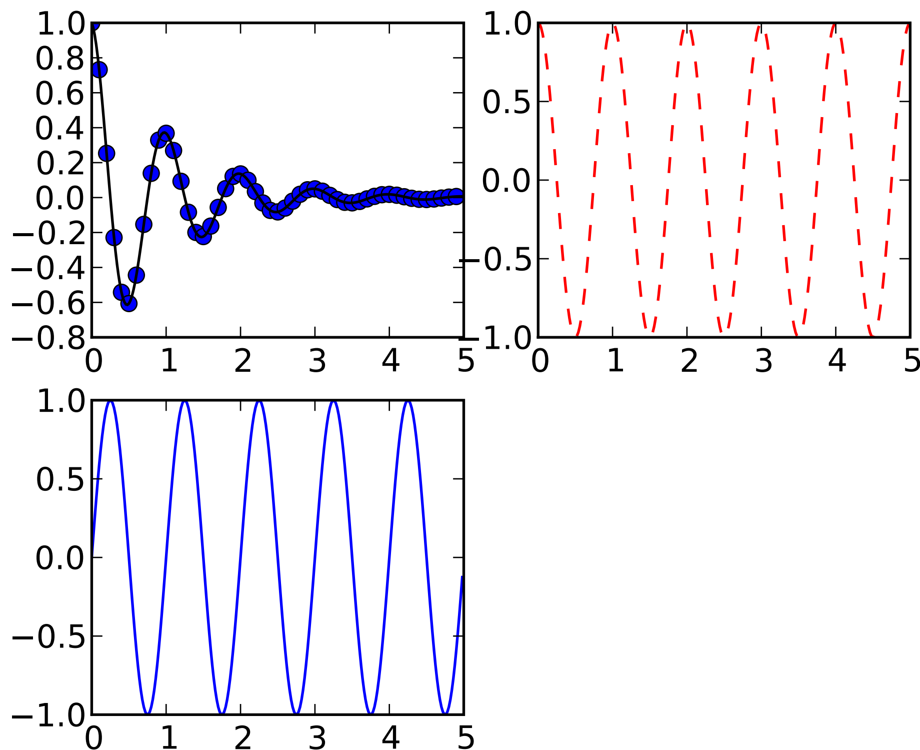
```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    import numpy as np  # non necessaire
    import matplotlib.pyplot as plt  # non necessaire
    return np.exp(-t) * np.cos(2*np.pi*t)
```


1.0
0.5
0.0
−0.5
−1.0
−6
−4
−2
0
2
4
6


1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100

```
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-')
```



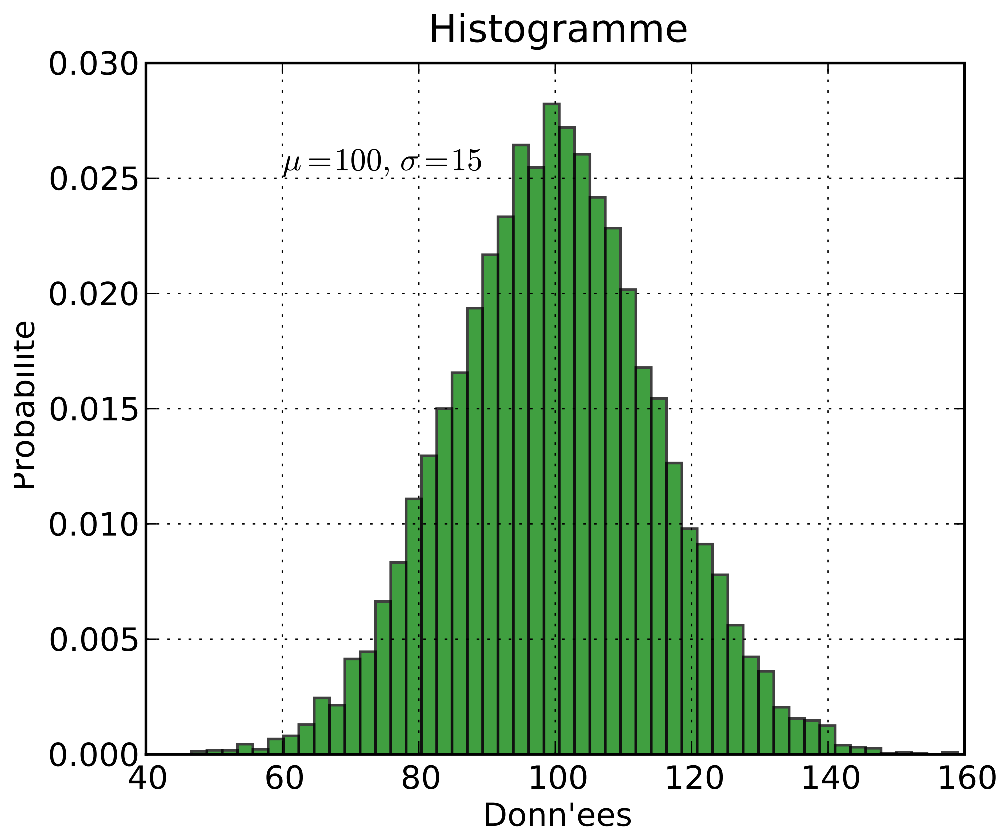
Dans la commande `plt.subplot` l'argument est `nbre de lignes, nbre de colonnes, numéro de la figure`. Il y a une condition à respecter : le nombre de lignes multiplié par le nombre de colonnes est supérieur ou égal au nombre de figure. Ensuite Matplotlib place les figures au fur et à mesure dans le sens des lignes.

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

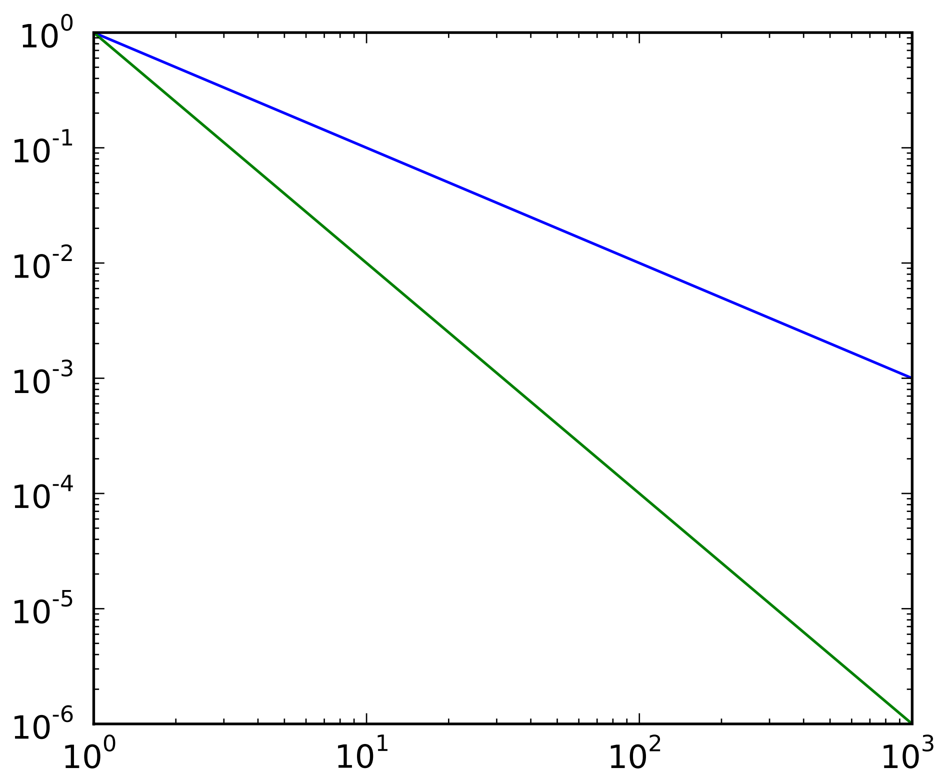
```
import numpy as np
import matplotlib.pyplot as plt
```

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
#  histogramme des donn\'es
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 (évolution de l'erreur dans une approximation numérique) 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.

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(1,1000,50)
plt.loglog()
plt.plot(x,1./x)
plt.plot(x,1./x**2)
plt.show()
```


10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^0
10^1
10^2
10^3