Python - cours intensif pour les scientifiques

Partie 2 : Modules NumPy et SciPy

Par Rick Muller - Raphaël Seban (traducteur)

Date de publication : 2 août 2014

Pour de nombreux scientifiques, Python est LE langage de programmation par excellence, car il offre de grandes possibilités en analyse et modélisation de données scientifiques avec relativement peu de charge de travail en termes d'apprentissage, d'installation ou de temps de développement. C'est un langage que vous pouvez intégrer en un week-end, puis utiliser une vie durant.

Retrouvez la première partie de cet article :

- **Partie 1 : Présentation de Python**

Les commentaires et les suggestions d'amélioration sont les bienvenus, alors, après votre lecture, n'hésitez pas.

En complément sur Developpez.com

- Condensé Python pour les scientifiques - Partie 1
- Tutoriel Matplotlib

I - Introduction

La librairie **Numpy** contient des fonctions essentielles pour traiter les tableaux, les matrices et les opérations de type algèbre linéaire avec Python. La librairie **Scipy** contient des fonctions supplémentaires pour l'optimisation de calculs, des fonctions spéciales, etc. Les deux comprennent des modules écrits en C et en Fortran de manière à les rendre aussi rapides que possible. Utilisées ensemble, ces librairies offrent au langage Python, à peu de chose près, les mêmes capacités que **Matlab™**. Les habitués de ce logiciel trouveront sans doute un grand intérêt à lire le **guide Numpy à l'attention des utilisateurs Matlab™**.

*Cette partie est consultable également au format « notebook » en suivant le lien : **Notes: Partie 2 - Modules NumPy et SciPy** .*

II - Tableaux et matrices

L'un des points communs fondamentaux de Numpy et de Scipy est leur aptitude à gérer les tableaux et les matrices. Vous pouvez créer des tableaux à partir de listes Python (*list* ou *[]*) grâce à la fonction objet array():

In :
```
array([1,2,3,4,5,6])
```

Out :
```
array([1, 2, 3, 4, 5, 6])
```

Vous pouvez passer en second argument de la fonction array() le type numérique souhaité. Vous trouverez une liste exhaustive de ces types à **cet endroit**. Certains types ont un équivalent qui se résume en une seule lettre. Les plus courants sont 'd' (nombre à virgule flottante à double précision), 'D' (nombre complexe à double précision) et 'i' (*int32*, entier signé et codé sur 32 bits, valeurs allant de -2 147 483 648 à +2 147 483 647).

Exemples :

In :
```
array([1,2,3,4,5,6],'d')
```

Out :
```
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

In :
```
array([1,2,3,4,5,6],'D')
```

Out :
```
array([ 1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j,  5.+0.j,  6.+0.j])
```

In :
```
array([1,2,3,4,5,6],'i')
```

Out :
```
array([1, 2, 3, 4, 5, 6], dtype=int32)
```

Pour créer une matrice, vous pouvez utiliser array() avec des listes de listes Python :

In :
```
array([[0,1],[1,0]],'d')
```

Out :
```
array([[ 0.,  1.],
       [ 1.,  0.]])
```

Vous pouvez aussi générer des matrices vides (remplies de zéros) de tailles arbitraires, y compris des tableaux unidimensionnels que Numpy traitera comme des tableaux à une seule ligne grâce à la fonction objet zeros():

In :
```
zeros((3,3),'d')
```

Out :
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Le premier argument de la fonction est un tuple (m_lignes, n_colonnes) et le second argument définit le type de données à utiliser, comme pour la fonction objet array(). Ainsi, vous pouvez initialiser un tableau simple (une ligne) de cette façon :

In :
```
zeros(3,'d')
```

Out :
```
array([ 0.,  0.,  0.])
```

In :
```
zeros((1,3),'d')
```

Out :
```
array([[ 0.,  0.,  0.]])
```

ou des tableaux en colonne :

In :
```
zeros((3,1),'d')
```

Out :
```
array([[ 0.],
       [ 0.],
       [ 0.]])
```

Il y a aussi la fonction objet identity() qui produit, comme vous pourriez vous y attendre :

In :
```
identity(4,'d')
```

Out :
```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

de même que la fonction objet ones() qui fait la même chose que zeros(), sauf qu'elle initialise la matrice avec des uns (1).

III - Espace linéaire, fonctions matricielles et tracés

La fonction objet linspace(start, end) crée un tableau espace linéaire de points de valeurs comprises entre start (inclus) et end (inclus).

In :
```
linspace(0,1)
```

Out :
```
array([ 0.        ,  0.02040816,  0.04081633,  0.06122449,  0.08163265,
        0.10204082,  0.12244898,  0.14285714,  0.16326531,  0.18367347,
        0.20408163,  0.2244898 ,  0.24489796,  0.26530612,  0.28571429,
        0.30612245,  0.32653061,  0.34693878,  0.36734694,  0.3877551 ,
        0.40816327,  0.42857143,  0.44897959,  0.46938776,  0.48979592,
        0.51020408,  0.53061224,  0.55102041,  0.57142857,  0.59183673,
        0.6122449 ,  0.63265306,  0.65306122,  0.67346939,  0.69387755,
        0.71428571,  0.73469388,  0.75510204,  0.7755102 ,  0.79591837,
        0.81632653,  0.83673469,  0.85714286,  0.87755102,  0.89795918,
        0.91836735,  0.93877551,  0.95918367,  0.97959184,  1.        ])
```

Si vous fournissez un troisième argument à linspace(start, end, nb_points), celui-ci définira le nombre de points à calculer dans cet espace linéaire. Si vous omettez ce troisième argument, il prend par défaut la valeur **nb_points=50**.

In :
```
linspace(0,1,11)
```

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

La fonction objet linspace() est un moyen bien pratique de générer des valeurs de coordonnées pour un graphique *(NdT : pour un intervalle d'abscisses, par exemple)*. Les fonctions de la librairie Numpy — qui sont toutes intégrées d'office dans l'interface IPython Notebook — peuvent traiter un tableau entier (ou même une matrice) de points en une seule fois.

Exemple :

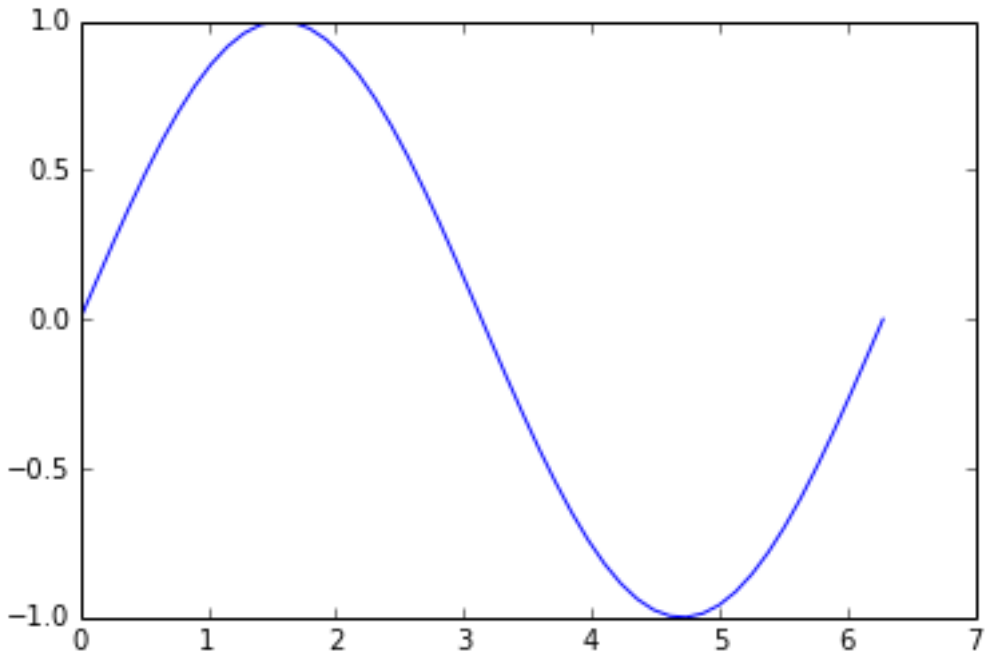
In :
```
x = linspace(0,2*pi)
sin(x)
```

Out :
```
array([  0.00000000e+00,   1.27877162e-01,   2.53654584e-01,
         3.75267005e-01,   4.90717552e-01,   5.98110530e-01,
         6.95682551e-01,   7.81831482e-01,   8.55142763e-01,
         9.14412623e-01,   9.58667853e-01,   9.87181783e-01,
         9.99486216e-01,   9.95379113e-01,   9.74927912e-01,
         9.38468422e-01,   8.86599306e-01,   8.20172255e-01,
         7.40277997e-01,   6.48228395e-01,   5.45534901e-01,
         4.33883739e-01,   3.15108218e-01,   1.91158629e-01,
         6.40702200e-02,  -6.40702200e-02,  -1.91158629e-01,
        -3.15108218e-01,  -4.33883739e-01,  -5.45534901e-01,
        -6.48228395e-01,  -7.40277997e-01,  -8.20172255e-01,
        -8.86599306e-01,  -9.38468422e-01,  -9.74927912e-01,
        -9.95379113e-01,  -9.99486216e-01,  -9.87181783e-01,
        -9.58667853e-01,  -9.14412623e-01,  -8.55142763e-01,
        -7.81831482e-01,  -6.95682551e-01,  -5.98110530e-01,
        -4.90717552e-01,  -3.75267005e-01,  -2.53654584e-01,
        -1.27877162e-01,  -2.44929360e-16])
```

En alliant Numpy à la puissance de Matplotlib, nous obtenons de grandes capacités pour tracer des graphiques facilement :

In :
```
plot(x,sin(x))
```

[<matplotlib.lines.Line2D at 0x10f07a150>]



IV - Opérations sur les matrices

Les objets matrice font ce qu'il faut lorsqu'ils sont multipliés par des scalaires :

In :
```
0.125*identity(3,'d')
```

Out :
```
array([[ 0.125,  0.   ,  0.   ],
       [ 0.   ,  0.125,  0.   ],
       [ 0.   ,  0.   ,  0.125]])
```

de même, lorsque vous additionnez deux matrices entre elles (sous réserve toutefois qu'elles soient de mêmes dimensions) :

In :
```
identity(2,'d') + array([[1,1],[1,2]])
```

Out :
```
array([[ 2.,  1.],
       [ 1.,  3.]])
```

Une chose qui pourrait dérouter les utilisateurs de Matlab™ : l'opérateur arithmétique dénoté par l'astérisque (*) applique une multiplication élément par élément plutôt qu'une vraie multiplication entre matrices :

In :
```
identity(2)*ones((2,2))
```

Out :
```
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Pour obtenir une vraie multiplication entre matrices, utilisez la fonction dot() :

In :
```
dot(identity(2),ones((2,2)))
```

Out :
```
array([[ 1.,  1.],
       [ 1.,  1.]])
```

La fonction dot() peut aussi faire des produits scalaires :

In :
```
v = array([3,4],'d')
sqrt(dot(v,v))
```

Out :
```
5.0
```

tout comme la multiplication matrice-tableau.

Les fonctions determinant(), inverse() et transpose() produisent les résultats attendus. Une transposition peut être abrégée en plaçant .T à la fin d'un nom d'objet matrice :

In :
```
m = array([[1,2],[3,4]])
m.T
```

Out :
```
array([[1, 3],
       [2, 4]])
```

Vous avez aussi la fonction diag() qui place une liste ou un tableau le long de la diagonale d'une matrice carrée.

In :
```
diag([1,2,3,4,5])
```

Out :
```
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

Nous verrons son utilité plus tard.

V - Solveurs matriciels

Vous pouvez résoudre des systèmes d'équations linéaires avec la fonction solve() :

In :
```
A = array([[1,1,1],[0,2,5],[2,5,-1]])
b = array([6,-4,27])
solve(A,b)
```

Out :
```
array([ 5.,  3., -2.])
```

Plusieurs fonctions pour calculer des **valeurs propres** ainsi que des vecteurs propres :

- eigvals() retourne les valeurs propres d'une matrice ;
- eigvalsh() retourne les valeurs propres d'une matrice hermitienne ;
- eig() retourne les valeurs propres et les vecteurs propres d'une matrice ;
- eigh() retourne les valeurs propres et les vecteurs propres d'une matrice hermitienne.

In :
```
A = array([[13,-4],[-4,7]],'d')
eigvalsh(A)
```

Out :
```
array([  5.,  15.])
```

In :
```
eigh(A)
```

Out :
```
(array([  5.,  15.]), array([[-0.4472136 , -0.89442719],
       [-0.89442719,  0.4472136 ]]))
```

VI - Exemple : différences finies

Maintenant que nous avons tous ces outils dans notre boîte à outils, nous pouvons commencer à faire des choses intéressantes. La plupart des équations que nous cherchons à résoudre en physique implique les équations

$$y'(x) = \lim_{\substack{h \to 0 \\ h \neq 0}} \frac{y(x+h) - y(x)}{h}$$

différentielles. Nous aimerions pouvoir calculer les dérivées de fonctions en **discrétisant** la fonction $y(x)$ sur un espace de points $x_0, x_1, \ldots, x_n$ répartis uniformément et qui produiront $y_0, y_1, \ldots, y_n$. Grâce à la discrétisation des valeurs, nous pouvons approcher la dérivée $y'(x)$ par $y_i' \approx \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$.

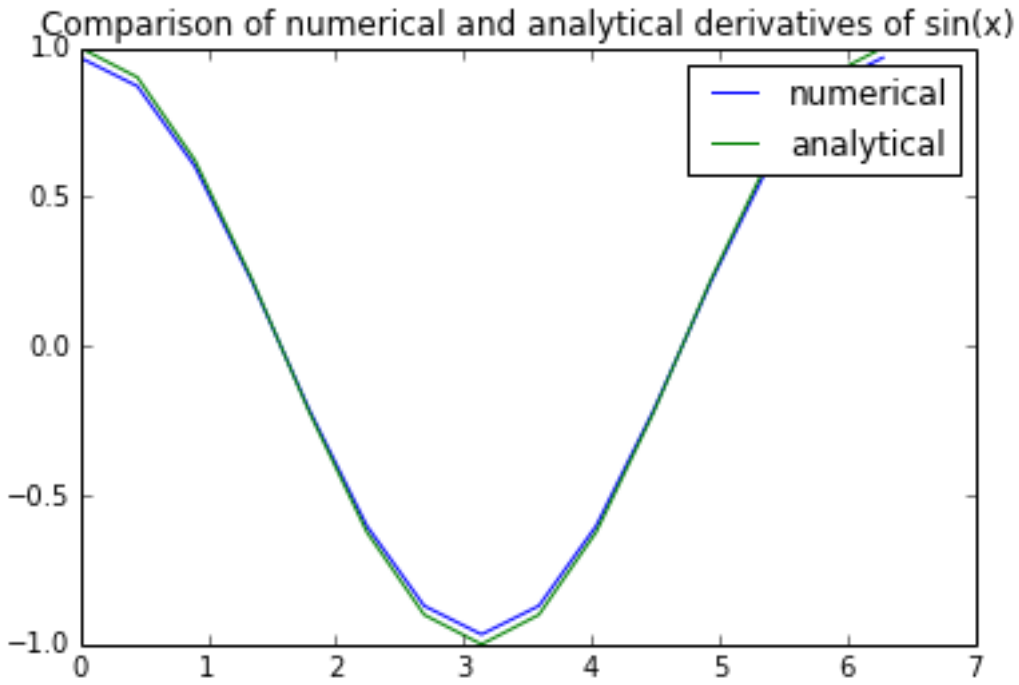
En Python, nous pourrions écrire une fonction dérivée comme ceci :

In :
```
def nderiv(y,x):
    "Différence finie, dérivée de la fonction f"
    n = len(y)
    d = zeros(n,'d') # virgule flottante à double précision (double)
    # différences de part et d'autre
    # centrées sur les points intérieurs
    for i in range(1,n-1):
        d[i] = (y[i+1]-y[i-1])/(x[i+1]-x[i-1])
    # différences sur un seul côté pour les extrémités
    d[0] = (y[1]-y[0])/(x[1]-x[0])
    d[n-1] = (y[n-1]-y[n-2])/(x[n-1]-x[n-2])
    return d
```

Voyons si cela fonctionne avec l'exemple sin(x) précédent :

In :
```
x = linspace(0,2*pi,15)
dsin = nderiv(sin(x),x)
plot(x,dsin,label='numerical')
plot(x,cos(x),label='analytical')
title("Comparison of numerical and analytical derivatives of sin(x)")
legend()
```

<matplotlib.legend.Legend at 0x7f0f5acd6910>



Très proches ! *(NdT : avec seulement 15 points pour discrétiser).*

VII - Oscillateur harmonique unidimensionnel utilisant les différences finies

Voyons si nous pouvons utiliser ce schéma aux différences finies pour calculer l'oscillateur harmonique unidimensionnel avec une bonne approximation.

Nous aimerions résoudre l'équation indépendante du temps de Schrödinger :

$$-\frac{\hbar^2}{2m}\frac{\partial^2\psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x)$$

avec $\psi(x)$ quand $V(x) = \frac{1}{2}m\omega^2x^2$ représente le potentiel de l'oscillateur harmonique. Nous utiliserons le tour de passe-passe habituel qui consiste à transformer l'équation différentielle en équation matricielle, en multipliant chaque terme de l'équation par l'expression conjuguée $\psi^*(x)$ puis en intégrant sur x. Ce qui nous donne :

$$-\frac{\hbar}{2m}\int \psi^*(x)\frac{\partial^2}{\partial x^2}\psi(x)\mathrm{d}x + \int \psi^*(x)V(x)\psi(x)\mathrm{d}x = E$$

NdlR : E étant une constante, nous pouvons la sortir de l'intégrale, reste donc pour l'expression de droite $E\int \psi^*(x)\psi(x)\mathrm{d}x$, or $\psi^*(x)\psi(x) = \| \psi(x)^2 \|$ est la densité de probabilité de présence. Celle-ci, étant intégrée sur l'ensemble du domaine, vaut 1.

Donc $\int \psi^*(x)\psi(x)\mathrm{d}x = \int \| \psi(x)^2 \| = 1$, d'où la simplification de l'expression de droite à E.

Nous allons de nouveau recourir à l'approche par différences finies. La formule de différences finies pour la dérivée seconde est :

$$y''_i = \frac{y_{i+1} - 2y_i + y_{i-1}}{x_{i+1} - x_{i-1}}$$

Nous pourrions penser au premier terme de l'équation de Schrödinger comme étant le chevauchement de la fonction d'onde $\psi(x)$ avec la dérivée seconde de la fonction d'onde $\frac{\partial^2}{\partial x^2}\psi(x)$. Avec la formule de la dérivée seconde que nous venons d'écrire, nous pouvons vérifier que si nous prenons le chevauchement des états $y_1, \ldots, y_n$ avec la dérivée seconde, nous aurons seulement trois points où le chevauchement est non nul à y_{i-1}, y_i et y_{i+1}. Sous la forme matricielle, cela nous mène à la matrice tridiagonale de Laplace qui a pour valeurs des -2 dans la diagonale principale et des 1 dans les diagonales situées au-dessus et en-dessous de cette diagonale principale.

Le second terme de l'équation de Schrödinger mène à une matrice diagonale avec pour éléments diagonaux $V(x_i)$. En assemblant toutes les pièces de ce puzzle, nous obtenons :

In :
```
def Laplacian(x):
    h = x[1]-x[0] # on suppose les points répartis uniformément
    n = len(x)
    M = -2*identity(n,'d')
    for i in range(1,n):
        M[i,i-1] = M[i-1,i] = 1
    return M/h**2
```

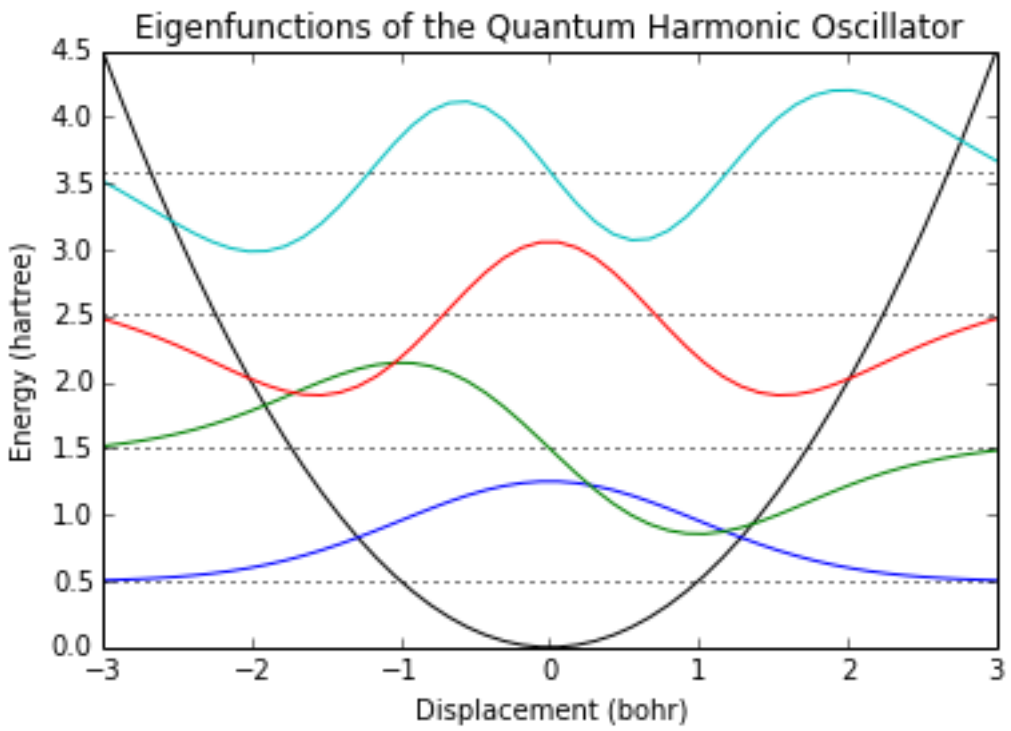
In :
```
x = linspace(-3,3)
m = 1.0
ohm = 1.0
T = (-0.5/m)*Laplacian(x)
V = 0.5*(ohm**2)*(x**2)
H =  T + diag(V)
E,U = eigh(H)
```

In :
```
h = x[1]-x[0]

# on trace le potentiel harmonique
plot(x,V,color='k')

for i in range(4):
    # Pour chacune des premières solutions, on trace le niveau d'énergie
    axhline(y=E[i],color='k',ls=":")
    # de même que les valeurs propres, décalées par le niveau d'énergie
    # de sorte qu'elles ne s'empilent pas les unes sur les autres :
    plot(x,-U[:,i]/sqrt(h)+E[i])
title("Eigenfunctions of the Quantum Harmonic Oscillator")
xlabel("Displacement (bohr)")
ylabel("Energy (hartree)")
```

<matplotlib.text.Text at 0x7f6524247f90>



Nous avons bricolé un peu pour obtenir les **orbitales** telles qu'attendues. Tout d'abord, nous avons inséré un facteur -1 devant les fonctions d'onde pour régler la phase à son état le plus bas. La phase (signe) d'une fonction d'onde quantique ne fournit aucune information, seul le carré de la fonction d'onde le fait, donc tout ceci est sans grande conséquence.

En revanche, les fonctions propres telles que nous les avons générées ne sont pas correctement normalisées. La raison en est que la différence finie n'est pas un véritable principe de base au sens de la mécanique quantique. C'est le principe de base des fonctions delta δ de Dirac à chaque point ; nous interprétons les espaces entre les points comme étant « remplis » par la fonction d'onde, mais le principe de la différence finie n'a de solution que pour les points eux-mêmes. Nous pouvons résoudre ce problème en divisant les fonctions propres de notre différence finie hamiltonienne par la racine carrée de l'espacement entre les points et ainsi, nous obtenons des fonctions correctement normalisées.

VIII - Fonctions spéciales

Les solutions de l'oscillateur harmonique sont supposées être des polynômes hermitiens. La page Wikipedia a les états HO donnés par :

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \exp\left(-\frac{m\omega x^2}{2\hbar}\right) H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right)$$

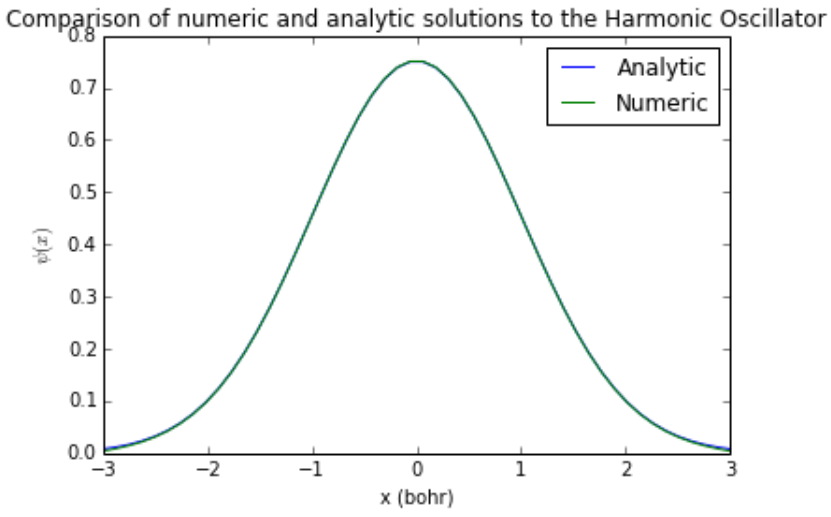
Voyons s'ils ressemblent à cela. Il existe des fonctions spéciales dans la librairie Numpy et quelques autres encore plus spéciales dans Scipy. Les polynômes hermitiens se trouvent dans Numpy :

In :
```
from numpy.polynomial.hermite import Hermite
from math import factorial
def ho_evec(x,n,m,ohm):
    vec = [0]*9
    vec[n] = 1
    Hn = Hermite(vec)
    return (1/sqrt(2**n*factorial(n)))*pow(m*ohm/pi,0.25)*exp(-0.5*m*ohm*x**2)*Hn(x*sqrt(m*ohm))
```

Comparons notre première fonction à notre solution :

In :
```
plot(x,ho_evec(x,0,1,1),label="Analytic")
plot(x,-U[:,0]/sqrt(h),label="Numeric")
xlabel('x (bohr)')
ylabel(r'$\psi(x)$')
title("Comparison of numeric and analytic solutions to the Harmonic Oscillator")
legend()
```

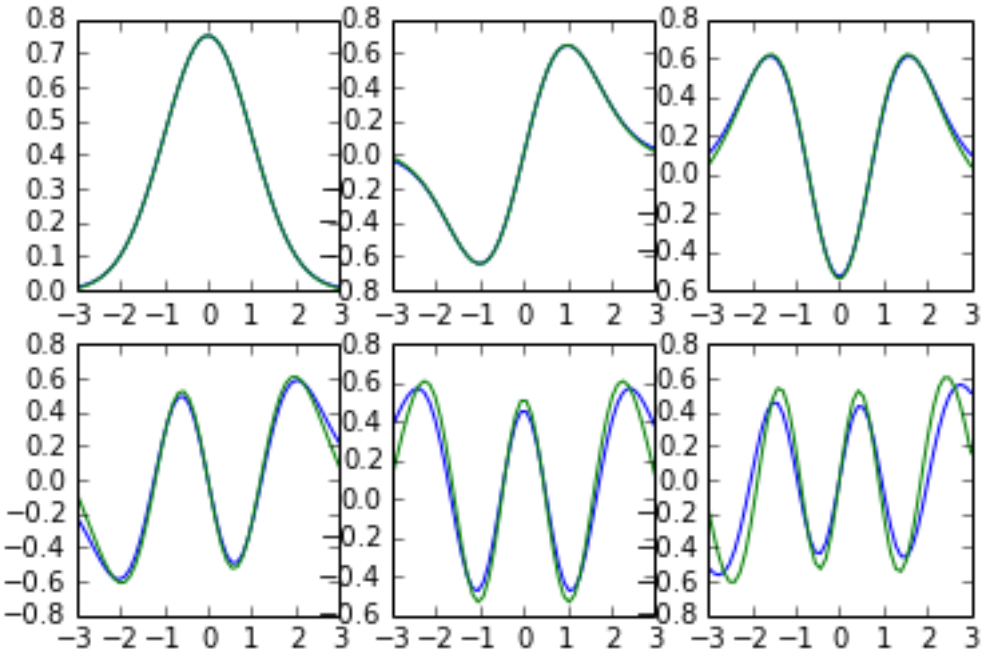
<matplotlib.legend.Legend at 0x7f0f5aacea90>



L'accord est presque parfait.

Nous pouvons utiliser la fonction matplotlib subplot(rows, cols, index) pour afficher en grille plusieurs tracés dans un même graphique :

In :
```
phase_correction = [-1,1,1,-1,-1,1]
for i in range(6):
    subplot(2,3,i+1)
    plot(x,ho_evec(x,i,1,1),label="Analytic")
    plot(x,phase_correction[i]*U[:,i]/sqrt(h),label="Numeric")
```



Hormis les erreurs de phase que j'ai corrigées par un petit bidouillage — vous voyez lequel ? — la concordance entre les courbes est plutôt pas mal, bien que cela empire au fur et à mesure que le niveau d'énergie augmente, en partie parce que nous avons utilisé une résolution de 50 points seulement.

Le module Scipy a encore plus de fonctions spéciales :

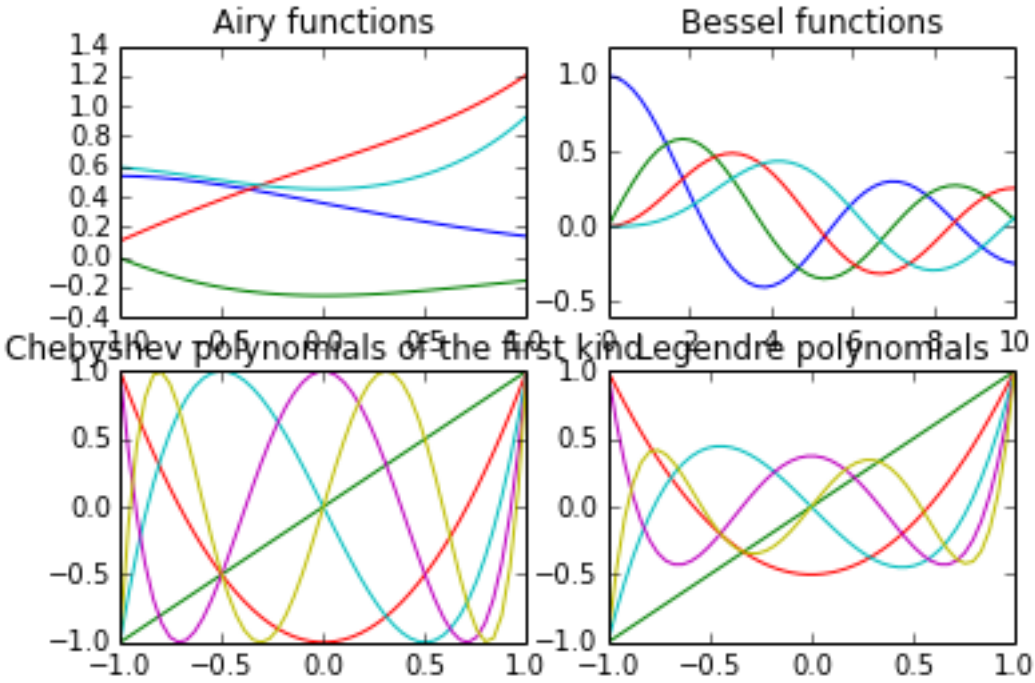
In :
```
from scipy.special import airy,jn,eval_chebyt,eval_legendre
subplot(2,2,1)
x = linspace(-1,1)
Ai,Aip,Bi,Bip = airy(x)
plot(x,Ai)
plot(x,Aip)
plot(x,Bi)
plot(x,Bip)
title("Airy functions")

subplot(2,2,2)
x = linspace(0,10)
for i in range(4):
    plot(x,jn(i,x))
title("Bessel functions")

subplot(2,2,3)
x = linspace(-1,1)
for i in range(6):
    plot(x,eval_chebyt(i,x))
title("Chebyshev polynomials of the first kind")

subplot(2,2,4)
x = linspace(-1,1)
for i in range(6):
    plot(x,eval_legendre(i,x))
title("Legendre polynomials")
```

<matplotlib.text.Text at 0x7f0f53792a10>



Par exemple les polynômes de Jacobi, Laguerre, Hermite, les fonctions hypergéométriques et plein d'autres. Vous trouverez une liste complète sur la page **fonctions spéciales Scipy**.

IX - Résolution par les moindres carrés

Très souvent, nous traitons des données que nous voulons cadrer avec un comportement attendu. Supposons que nous ayons ceci :

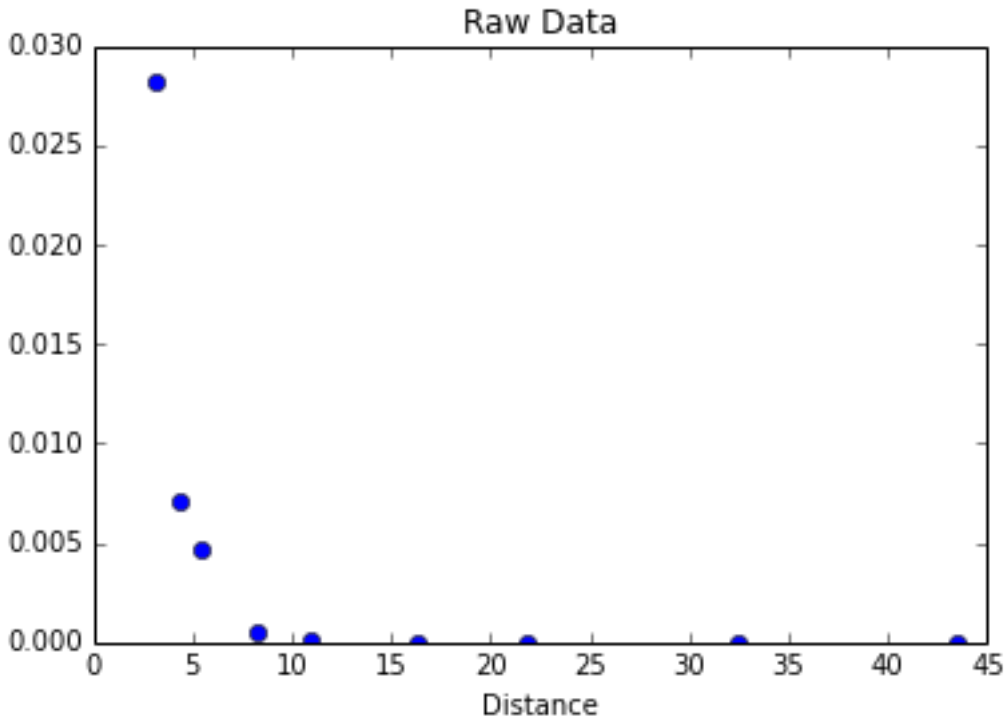
In :
```
raw_data = """\
3.1905781584582433,0.028208609537968457
4.346895074946466,0.007160804747670053
5.374732334047101,0.0046962988461934805
8.201284796573875,0.0004614473299618756
10.899357601713055,0.00005038370219939726
16.295503211991434,4.377451812785309e-7
21.82012847965739,3.0799922117601088e-9
32.48394004282656,1.524776208284536e-13
43.53319057815846,5.5012073588707224e-18"""
```

Vous trouverez plus tard une rubrique qui traite de l'analyse de données au format CSV (*Comma-Separated Values*). Nous récupérerons l'analyseur qui s'y trouve. Pour une explication détaillée, passez la rubrique ici présente et lisez plus loin. Sinon, considérez simplement qu'il s'agit d'un moyen d'extraire un tableau Numpy de la chaîne de caractères raw_data, tableau que nous pourrons tracer en graphique et soumettre à diverses analyses par la suite.

In :
```
data = []
for line in raw_data.splitlines():
    words = line.split(',')
    data.append(map(float,words))
data = array(data)
```

In :
```
title("Raw Data")
xlabel("Distance")
plot(data[:,0],data[:,1],'bo')
```

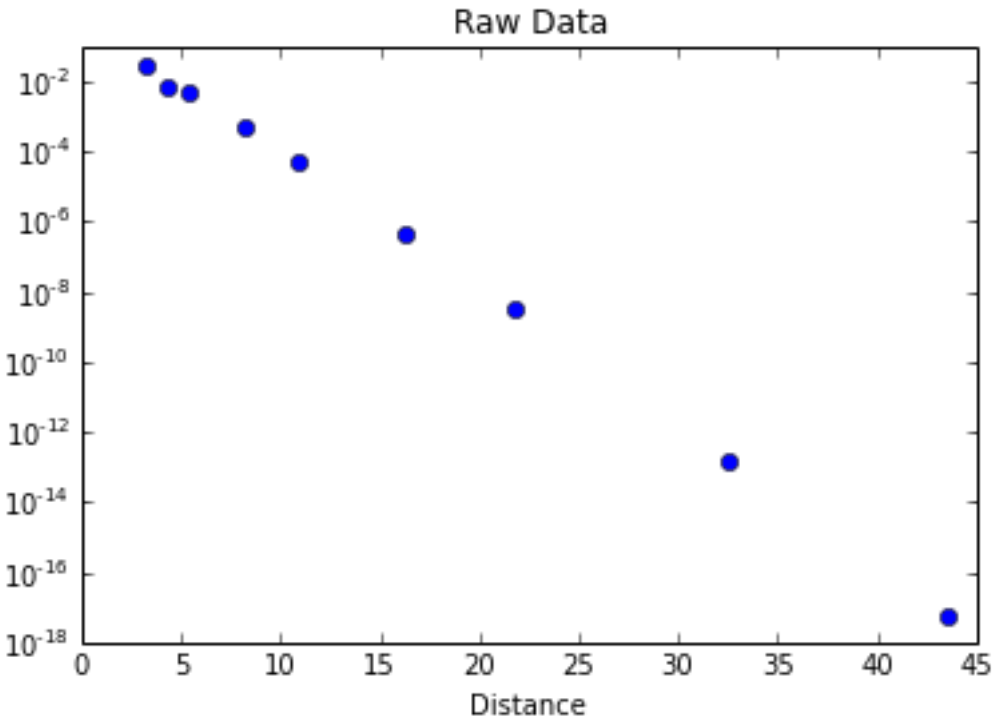
[<matplotlib.lines.Line2D at 0x7f0f5a6ea8d0>]



Comme nous nous attendons à ce que nos données décroissent exponentiellement, nous pouvons utiliser une échelle semi-logarithmique :

In :
```
title("Raw Data")
xlabel("Distance")
semilogy(data[:,0],data[:,1],'bo')
```

[<matplotlib.lines.Line2D at 0x7f0f5a746a10>]



Pour une décroissance exponentielle pure comme celle-ci, nous pouvons assimiler les points représentés à une droite. Le tracé ci-dessus suggère que nous avons là une bonne approximation, étant donné que la fonction $y = Ae^{-ax} \Leftrightarrow \log(y) = \log(Ae^{-ax})$, $\log(y) = \log(A) - ax$ est du type affine. Ainsi, en ajustant l'échelle par rapport aux abscisses x, nous devrions obtenir une droite de coefficient directeur $-a$ et d'ordonnée à l'origine $\log(A)$.

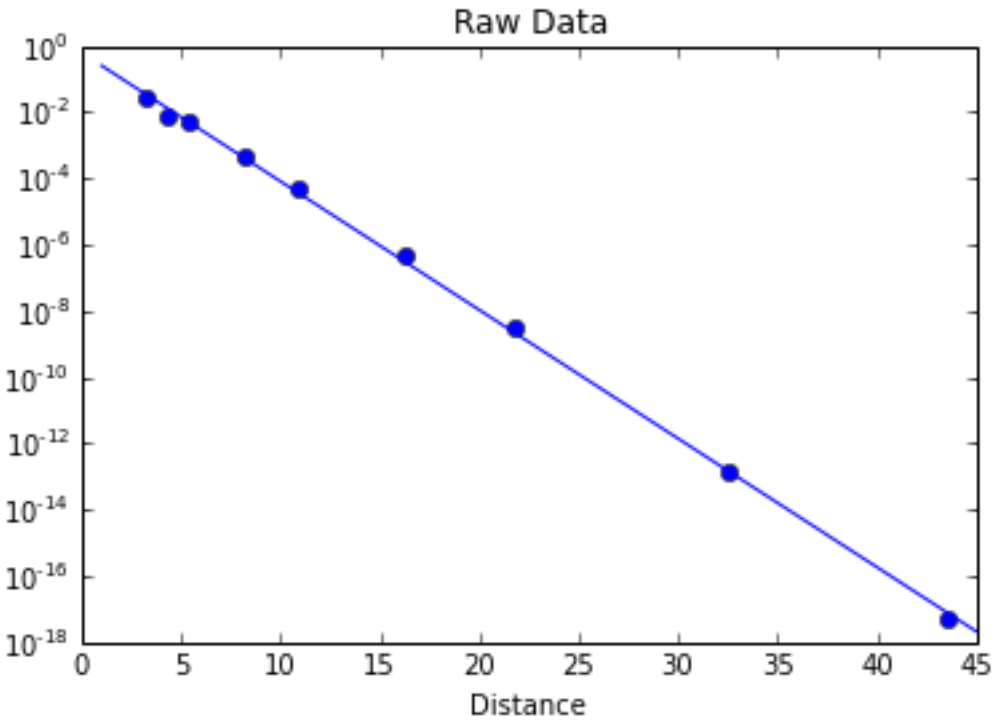
Il existe une fonction Numpy nommée polyfit() qui ajuste les données pour cadrer avec une forme polynomiale. Utilisons-la pour obtenir une droite (polynôme d'ordre 1) :

In :
```
params = polyfit(data[:,0],log(data[:,1]),1)
a = params[0]
A = exp(params[1])
```

Voyons à présent si cette courbe s'ajuste aux données que nous avons :

In :
```
x = linspace(1,45)
title("Raw Data")
xlabel("Distance")
semilogy(data[:,0],data[:,1],'bo')
semilogy(x,A*exp(a*x),'b-')
```

```
[<matplotlib.lines.Line2D at 0x7f0f5a9fc890>]
```



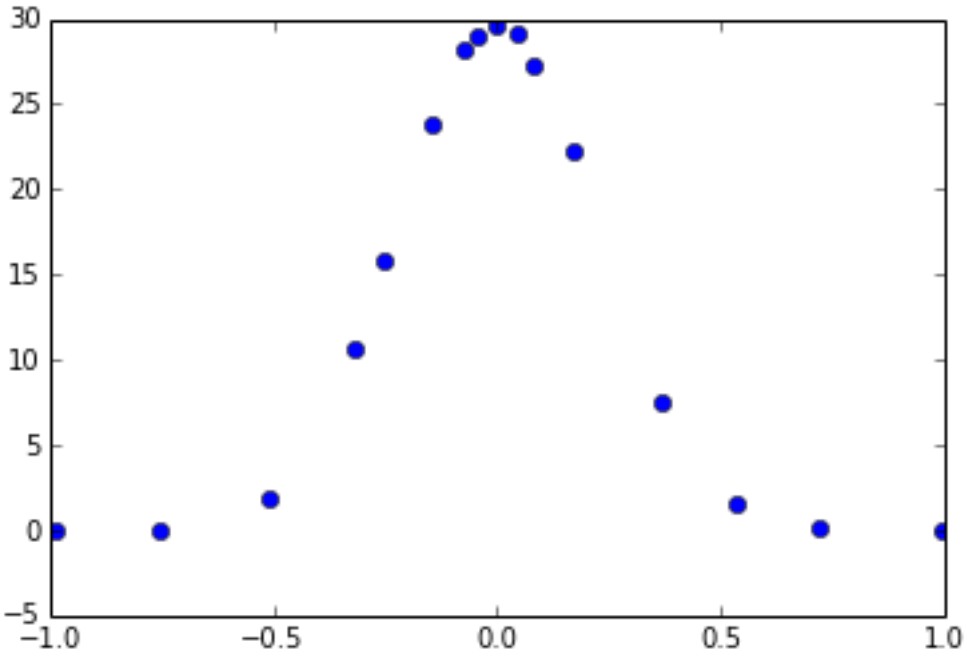
Avec des fonctions plus compliquées, il est possible que l'on n'arrive pas à s'ajuster sur un simple polynôme. Par exemple, avec les données suivantes :

In :
```
gauss_data = """\
-0.9902286902286903,1.4065274110372852e-19
-0.7566104566104566,2.2504438576596563e-18
-0.5117810117810118,1.9459459459459454
-0.31887271887271884,10.621621621621626
-0.250997150997151,15.891891891891893
-0.1463309463309464,23.756756756756754
-0.07267267267267263,28.135135135135133
-0.04426734426734419,29.02702702702703
-0.0015939015939017698,29.675675675675677
0.04689304689304685,29.10810810810811
0.0840994840994842,27.324324324324326
0.1700546700546699,22.216216216216214
0.370878570878571,7.540540540540545
0.5338338338338338,1.621621621621618
0.722014322014322,0.08108108108108068
0.9926849926849926,-0.08108108108108646"""

data = []
for line in gauss_data.splitlines():
    words = line.split(',')
    data.append(map(float,words))
data = array(data)

plot(data[:,0],data[:,1],'bo')
```

[<matplotlib.lines.Line2D at 0x7f0f5a6d0e10>]



Ces données ressemblent plus à une **courbe de Gauss** qu'à une courbe exponentielle. Si nous le voulions, nous pourrions utiliser la fonction polyfit() pour ce cas aussi, mais utilisons plutôt la fonction curve_fit() du module Scipy qui peut s'ajuster à toute sorte de fonctions arbitraires. Pour en savoir plus, tapez help(curve_fit) dans une cellule de code (ou en console interactive IPython).

Tout d'abord, définissons une fonction gaussienne générique à ajuster :

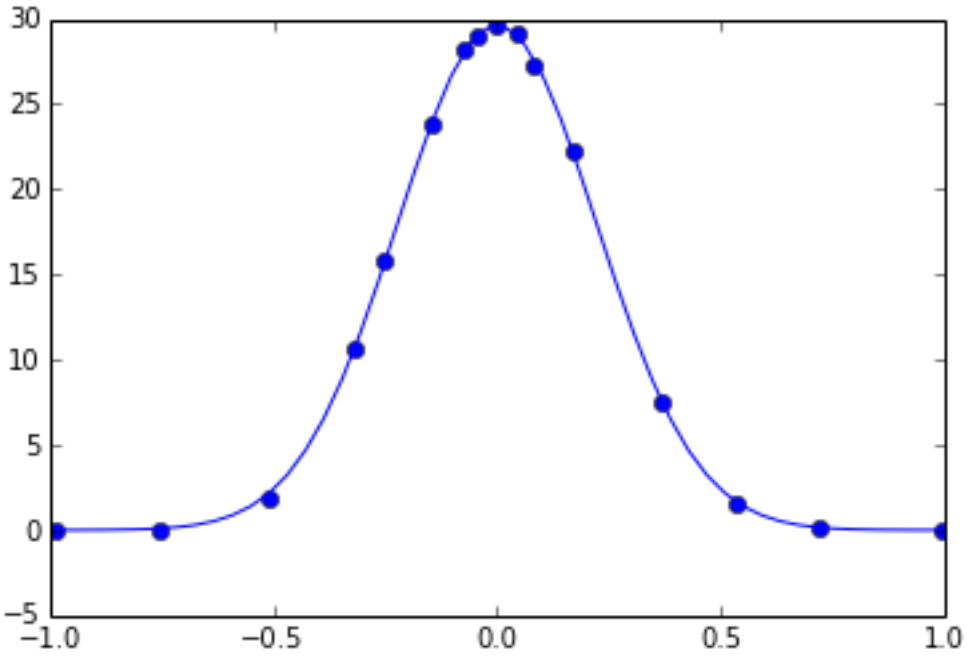
In :
```
def gauss(x,A,a): return A*exp(a*x**2)
```

À présent, ajustons en utilisant curve_fit() :

In :
```
from scipy.optimize import curve_fit

params,conv = curve_fit(gauss,data[:,0],data[:,1])
x = linspace(-1,1)
plot(data[:,0],data[:,1],'bo')
A,a = params
plot(x,gauss(x,A,a),'b-')
```

[<matplotlib.lines.Line2D at 0x7f0f5043a190>]



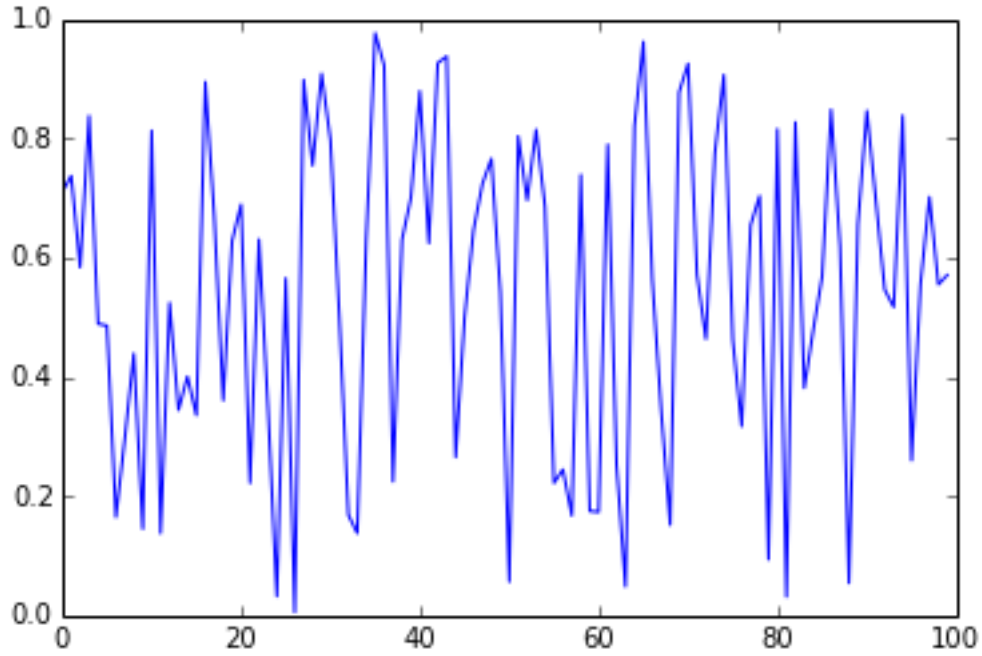
La fonction curve_fit() que nous venons d'utiliser est basée sur les excellents algorithmes d'**optimisation** (**minimisation**) de Scipy. Pour en savoir plus, consultez la **documentation Scipy**.

X - Monte Carlo, nombres aléatoires et calcul de Pi

Plusieurs techniques de calcul scientifique reposent sur la **méthode de Monte-Carlo** où une séquence de nombres (pseudo) aléatoires est utilisée pour approcher l'intégrale d'une fonction. Python a un bon générateur de nombres aléatoires dans sa librairie standard. La fonction random() génère des nombres **pseudo-aléatoires** uniformément répartis entre 0 et 1.

In :
```
from random import random
rands = []
for i in range(100):
    rands.append(random())
plot(rands)
```

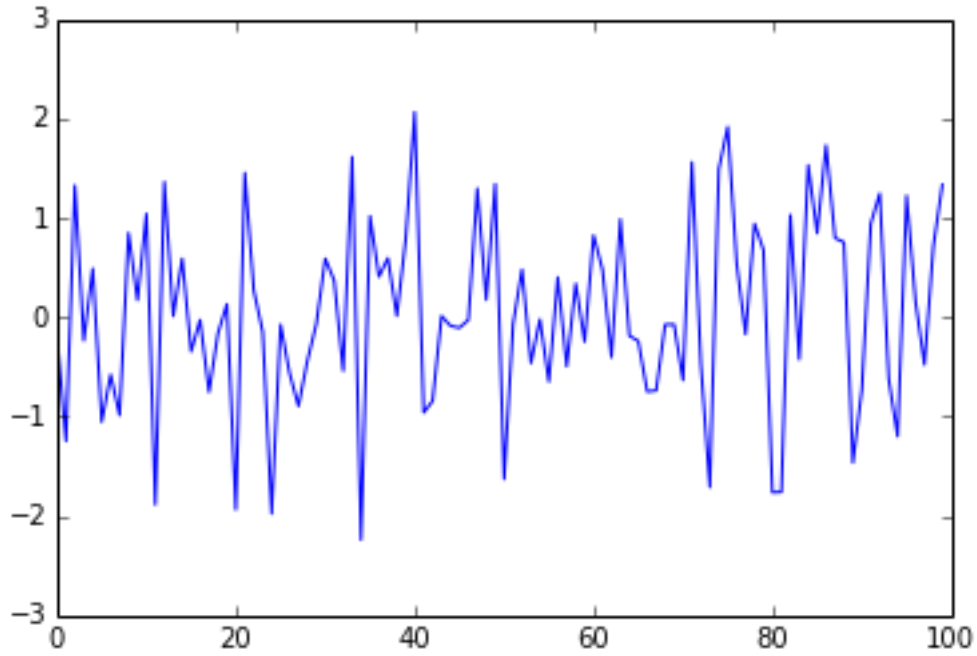
[<matplotlib.lines.Line2D at 0x7f0f50375890>]



La fonction random() se base sur l'algorithme de **Mersenne Twister** (**page officielle**) qui est un générateur de nombres pseudo-aléatoires particulièrement apprécié. Le module Python standard random contient aussi des fonctions pour générer des entiers aléatoires (**random.randint(a, b)**), des listes d'entiers aléatoires (**random.randrange(n)**), pour mélanger une liste (**random.shuffle(mutable)**) ainsi que des fonctions pour extraire des nombres aléatoires d'une distribution particulière telle que la **distribution normale**, par exemple :

In :
```
from random import gauss
grands = []
for i in range(100):
    grands.append(gauss(0,1))
plot(grands)
```

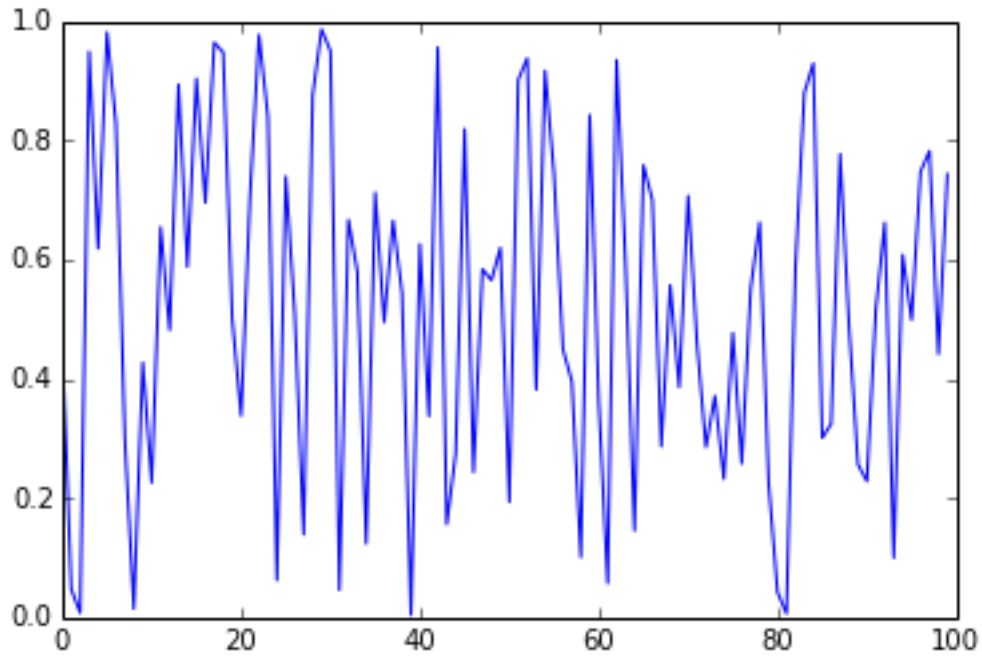
[<matplotlib.lines.Line2D at 0x7f0f50318a10>]



Il est généralement plus efficace de générer une liste complète de nombres aléatoires en une seule opération, surtout si vous extrayez des valeurs d'une loi non uniforme. La librairie Numpy possède des fonctions pour générer des tableaux et des matrices de plusieurs lois de probabilité.

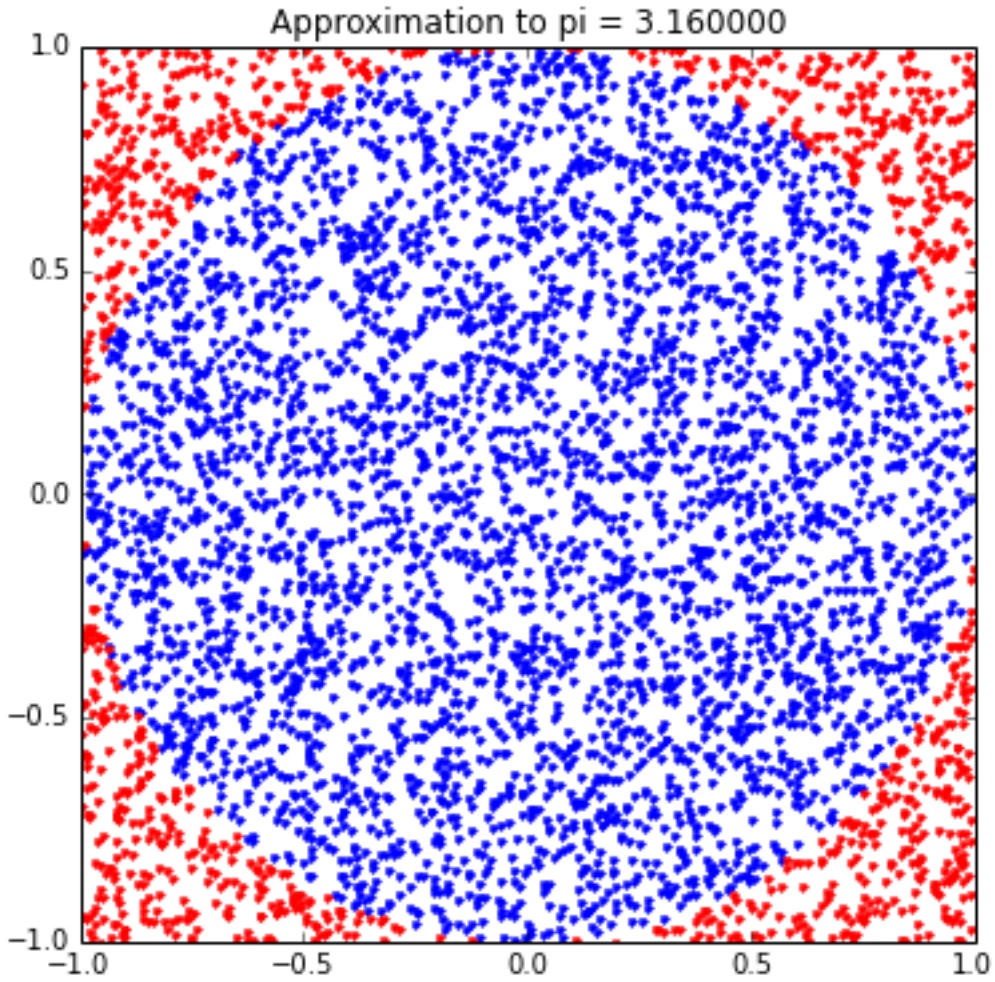
In :
```
plot(rand(100))
```

[<matplotlib.lines.Line2D at 0x7f0f5023f7d0>]



L'un des tous premiers programmes que j'ai écrit devait servir à calculer le nombre π en prenant des nombres aléatoires pour les coordonnées de points (x, y) et en comptant les occurrences situées à l'intérieur du cercle unitaire. Par exemple :

In :
```
npts = 5000
xs = 2*rand(npts)-1
ys = 2*rand(npts)-1
r = xs**2+ys**2
ninside = (r<1).sum()
figsize(6,6) # donne une forme carrée au graphique
title("Approximation to pi = %f" % (4*ninside/float(npts)))
plot(xs[r<1],ys[r<1],'b.')
plot(xs[r>1],ys[r>1],'r.')
figsize(8,6) # change les dimensions pour le reste du document
```



L'idée qui sous-tend ce programme est que le ratio entre l'aire du cercle unitaire et l'aire du carré circonscrit est de l'ordre de $\frac{\pi}{4}$. Alors, en comptant la portion de points aléatoires qui se trouvent à l'intérieur du cercle unitaire par rapport à l'ensemble du carré, nous obtenons une assez bonne estimation de la valeur de π.

Le bout de code précédent utilise des notations Numpy de haut niveau pour calculer le rayon de chaque point de coordonnées (x, y) en une seule ligne de code, puis pour compter combien de rayons sont inférieurs à 1 — toujours en une seule ligne de code — et enfin pour filtrer les points (x, y) par rapport à leurs rayons respectifs. Pour être honnête, j'écris rarement du code sous cette forme : je trouve que ces techniques Numpy sont par trop pointues pour que l'on puisse s'en souvenir et je préfère personnellement utiliser les techniques de **liste en compréhension** (voir plus loin) pour extraire les points que je veux, ce qui est plus facile à mémoriser.

À comparer des méthodes actuelles de calcul de π, celle-ci est parmi les pires. Une bien meilleure méthode consiste à utiliser la **série alternée de Leibniz** (développement de arctan(1)) :

$$\frac{\pi}{4} = \sum_k \frac{(-1)^k}{2 \times k + 1}$$

In :
```
n = 100
total = 0
for k in range(n):
    total += pow(-1,k)/(2*k+1.0)
print 4*total
```

Out :
```
3.13159290356
```

Si vous recherchez une méthode remarquable, consultez la **méthode de Srinivasa Ramanujan**. Elle converge si vite que vous aurez réellement besoin d'une précision mathématique arbitraire pour afficher les décimales. Vous pouvez faire cela avec le module Python standard decimal, si cela vous intéresse.

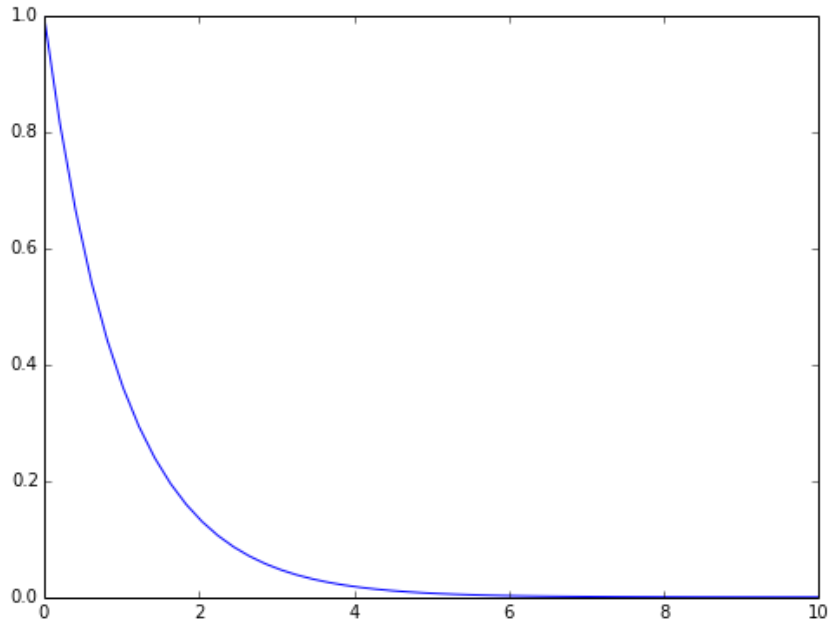
XI - Intégrales

Le **calcul numérique d'une intégrale** peut s'avérer épineux et parfois, il est plus facile d'évaluer le résultat par le biais d'une approximation. Par exemple, supposons que nous voulions voir ce que donne l'intégrale :

$$\int_0^\infty \exp(-x)\mathrm{d}x = 1$$

In :
```
def f(x): return exp(-x)
x = linspace(0,10)
plot(x,exp(-x))
```

[<matplotlib.lines.Line2D at 0x7f0f5012f590>]



La librairie Scipy possède une fonction de calcul numérique d'intégrale nommée `quad()` (on appelle parfois **quadrature** ce genre de calculs), que nous pouvons utiliser ici :

In :
```
from scipy.integrate import quad
quad(f,0,inf)
```

Out :
```
(1.0000000000000002, 5.842606742906004e-11)
```

Vous avez aussi des intégrateurs 2D et 3D dans Scipy. Pour en savoir plus, consultez **la documentation**.

XII - Transformée de Fourier rapide et traitement du signal

Nous avons très souvent recours aux techniques FFT (*Fast Fourier Transform* - **transformée de Fourier rapide**) pour nous aider à extraire un signal à partir de données « bruitées » :

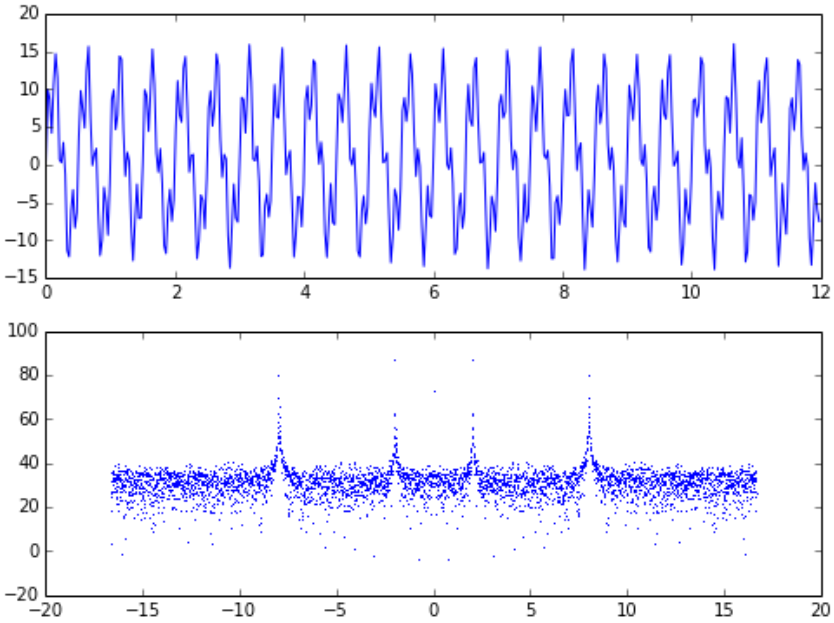
In :
```
from scipy.fftpack import fft,fftfreq

npts = 4000
nplot = npts/10
t = linspace(0,120,npts)
def acc(t): return 10*sin(2*pi*2.0*t) + 5*sin(2*pi*8.0*t) + 2*rand(npts)

signal = acc(t)

FFT = abs(fft(signal))
freqs = fftfreq(npts, t[1]-t[0])

subplot(211)
plot(t[:nplot], signal[:nplot])
subplot(212)
plot(freqs,20*log10(FFT),',')
show()
```



Dans Scipy, il existe des fonctions additionnelles qui gèrent le **traitement du signal**.

À suivre (en cours de traduction) :

- *Partie 3 - Python avancé ;*
- *Partie 4 - Optimiser le code.*

XIII - Remerciements

Un grand merci à Alex et Tess pour tout !

Remerciements chaleureux à Barbara Muller et Tom Tarman pour leurs précieuses suggestions.

Sandia est un laboratoire pluridisciplinaire géré par la Sandia Corporation®, une filiale de la Lockheed Martin Company®, pour le compte des États-Unis d'Amérique, département de l'Énergie, administration de la Sûreté Nucléaire, sous contrat n° DE-AC04-94AL85000.

XIV - Remerciements Developpez

Nous remercions Rick Muller qui nous a aimablement autorisé à traduire son cours "**A Crash Course in Python for Scientists**".

Nos remerciements à Raphaël SEBAN (**tarball69**) pour la traduction et à Fabien (**f-leb**) pour la mise au gabarit. Merci aussi à Sébastien (**-Nikopol-**) pour son renfort sur les points mathématiques délicats dans cet article.

Nous remercions également Malick SECK (**milkoseck**) pour sa relecture orthographique.