

École Doctorale SMI  
Module "Calcul Scientifique et Optimisation"  
7 – 11 avril 2014

# Calcul Scientifique avec



▷ Les modules Numpy et Scipy

**Jean-Luc CHARLES**  
Arts & Métiers ParisTech, I2M



- ▶ Python ne propose de base que le type **list**, conteneur dynamique hétérogène puissant, mais pas orienté calcul numérique.
- ▶ Le module **numpy** propose un ensemble de classes, d'objets et de fonctions dédiés aux calculs numériques.
- ▶ Le module **scipy** utilise **numpy** pour proposer une "boîte" à outil de calcul scientifique très complète.
- ▶ L'utilisation des modules numériques et de tracé (**numpy**, **scipy** et **matplotlib**) avec l'IDE **spyder** fournit une solution puissante pour le calcul scientifique.
- ▶ Les docs complètes *Numpy Reference Guide*, *Numpy User Guide* et *Scipy Reference Guide* sont sur <http://docs.scipy.org/doc/>



- ▶ Le module **numpy** propose un ensemble de classes, d'objets et de fonctions dédiés aux calculs numériques :
  - ▷ classe **ndarray** (*N dimensional array*) : tableaux homogènes multi-dimensionnels ;
  - ▷ **numpy.linalg** : un module d'algèbre linéaire basique ;
  - ▷ **numpy.random** : un module pour les générateurs aléatoires ;
  - ▷ **numpy.fft** : un module basique de calculs FFT (*Fast Fourier Transform*).
- ▶ Le site web **numpy** est <http://www.numpy.org>.  
La doc en ligne de **numpy** est <http://docs.scipy.org/doc/numpy-1.7.0/reference/>.

## La classe ndarray

### ► Création d'objets ndarray :

```
>>> import numpy as np
>>> m1 = np.ndarray((2,2))    # float par défaut
>>> print m1                  #m1 pas initialisée par défaut!
[[ 0.00000000e+000  4.94944794e+173]
 [ 1.93390228e-309  4.10074486e-322]]
>>> m2 = np.ndarray((2,2), dtype=int) # dtype: float, int bool...
>>> print m2
[[140330583136152      68243040]
 [      61672336      61690368]]
```

### ► La fonction array convertit un objet list en objet ndarray :

```
>>> m1 = np.array([1, 2, 3]) ; m1
array([1, 2, 3])
>>> m2 = np.array([1, 2, 3], float) ; m2 # float, int bool...
array([ 1.,  2.,  3.])
>>> L2 = [4, 5, 6]
>>> m3 = np.array([[1, 2, 3],[4, 5, 6]]) ; m3 # liste de listes : matrice
array([[1, 2, 3],
       [4, 5, 6]])
```

## La classe ndarray

### ► Les fonctions numpy zeros et ones

```
>>> m1 = np.zeros(4) ; print m1          # float par défaut  
[ 0.,  0.,  0.,  0.]  
>>> m2 = np.ones(3, bool) ; print m2  
[ True True True]  
>>> m3 = np.ones((2,3),int) ; print m3  
[[1, 1, 1],  
 [1, 1, 1]]
```

### ► Les fonctions linspace, logspace créent des vecteurs de float :

```
>>> np.linspace(0, 10, 5)          # start, stop, number of points  
array([ 0.,  2.5,  5.,  7.5, 10.])  
>>> np.logspace(1, 2, 4)          # 4 points between 10**1 and 10**2  
array([ 10., 21.5443469, 46.4158834, 100.])  
>>> np.logspace(1, 2, 4, base=2)  # 4 points between 2**1 and 2**2  
array([ 2.,  2.5198421, 3.1748021, 4.])
```

## La classe ndarray

### ► Les fonctions numpy `eye` et `identity` :

```
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> np.eye(3,k=1)
array([[ 0.,  1.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])

>>> np.eye(3,k=-1)
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])

>>> np.identity(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

## La classe ndarray



### Copies et références d'objets ndarray

```
>>> m2 = np.array([[ 1., 2., 3.],[ 4., 5., 6.]])
```

```
>>> m3 = m2
>>> id(m2), id(m3), m2 is m3
(20263504, 20263504, True)
>>> m3[1,1] = -1. ; print m3
[[ 1.  2.  3.]
 [ 3. -1.  5.]]
>>> print m2
[[ 1.  2.  3.]
 [ 3. -1.  5.]]
>>> m4 = m2.copy()
>>> id(m2), id(m4), m4 is m2
(20263504, 21073360, False)
>>> m4[1,1] = -2. ; print m4
[[ 1.  2.  3.]
 [ 3. -2.  5.]]
>>> print m2
[[ 1.  2.  3.]
 [ 3. -1.  5.]]
```

► L'affectation de m3 par l'opérateur = crée une **référence** sur m2, pas une **copie** de m2

► Il faut utiliser explicitement la méthode copie de la classe ndarray pour obtenir une copie (*deep copy*)

## La classe ndarray

### ► Les attributs de la classe ndarray :

```
>>> m3 = np.array([[1, 2, 3],[4, 5, 6]])    # list of lists gives matrix
>>> print m3
[[1 2 3]
 [4 5 6]]
>>> m.dtype           # the element type
dtype('int64')
>>> m.shape           # the length of the array along each dimension
(2, 3)                # 2 lines, 3 columns
>>> m.itemsize        # the element size in bytes element type
8                     # integer of 8 bytes (64 bits)
>>> m.size            # the number of elements
6
>>> m.ndim            # the number dimensions (axes)
2
>>> m.nbytes          # the number of bytes used by array data
48                    # 6 elements of 8 bytes each
```



## La classe ndarray

- La méthode `tolist` crée un objet `list`, copie de l'objet `ndarray` :

```
>>> m3.tolist()
[[1, 2, 3], [4, 5, 6]]
>>> L1 = m3.tolist()
>>> type(L1)
<type 'list'>
```

- La méthode `copy` permet de copier (*deep copy*) un objet `ndarray` :

```
>>> c = m.copy()
>>> print c
[[1 2 3]
 [4 5 6]]
>>> id(m), id(c), c is m
(51720800, 69723584, False)
```

## La classe ndarray

### ► Opérations élémentaires avec les ndarray :

```
>>> a, b = np.array([1, 2, 3, 4]), np.array([2, 3, 4, 5])
>>> a+b
array([3, 5, 7, 9])
>>> a-b
array([-1, -1, -1, -1])
>>> a**2
array([ 1,  4,  9, 16])
>>> 4*a, a*4
(array([ 4,  8, 12, 16]), array([ 4,  8, 12, 16]))
>>> a*b
array([ 2,  6, 12, 20])
>>> A=np.array([a,b])
>>> A
array([[1, 2, 3, 4],
       [2, 3, 4, 5]])
>>> A*A # * fait une multiplication terme à terme
array([[ 1,  4,  9, 16],
       [ 4,  9, 16, 25]])
```

## La classe ndarray

### ► Opérations élémentaires avec les ndarray :

```
>>> a < 3
array([ True,  True, False, False], dtype=bool)
>>> a <= 3
array([ True,  True,  True, False], dtype=bool)
>>> a==2
array([False,  True, False, False], dtype=bool)
>>> a <= 3
array([ True,  True,  True, False], dtype=bool)
>>> (a <= 3).all()
False
>>> (a <= 3).any()
True
>>> yes, no = np.ones_like(a), np.zeros_like(a)
>>> yes, no
(array([1, 1, 1, 1]), array([0, 0, 0, 0]))
>>> np.where(a % 3 == 0, yes, no)
array([0, 0, 1, 0])
```

## La classe ndarray

### ► Méthodes utiles de la classe ndarray :

```
>>> m = np.linspace(1,12,12).reshape((3,4)); print m
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]]
>>> m.sum(), m.min(), m.max()           # somme, max, min de tous les éléments
(78.0, 1.0, 12.)
>>> m.sum(axis=0)                       # somme des lignes
array([ 15., 18., 21., 24.])
>>> m.min(axis=0)                       # min des lignes
array([ 1.,  2.,  3.,  4.])
>>> m.cumsum(axis=0)                   # somme cumulée des lignes
array([[ 1.,  2.,  3.,  4.],
       [ 6.,  8., 10., 12.],
       [15., 18., 21., 24.]])
>>> m.cumsum(axis=1)                   # somme cumulée des colonnes
array([[ 1.,  3.,  6., 10.],
       [ 5., 11., 18., 26.],
       [ 9., 19., 30., 42.]])
```

## Les Ufunction (Universal functions)

- ▶ Les **ufunc** sont des fonctions mathématiques courantes (sin, cos, exp...) :
  - ▷ qui s'appliquent à tous les éléments d'un objet **ndarray** ;
  - ▷ qui renvoient l'objet **ndarray** résultant.

```
>>> m = np.linspace(1,6,6).reshape((2,3)); print m
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
>>> s = np.sqrt(m); s
array([[ 1.          ,  1.41421356,  1.73205081],
       [ 2.          ,  2.23606798,  2.44948974]])
>>> np.exp(m)
array([[ 2.71828183,   7.3890561 ,  20.08553692],
       [ 54.59815003,  148.4131591 ,  403.42879349]])
>>> m = np.linspace(30,180,6).reshape((2,3)); print m
[[ 30.  60.  90.]
 [120. 150. 180.]]
>>> np.radians(m)
array([[ 0.52359878,  1.04719755,  1.57079633],
       [ 2.0943951 ,  2.61799388,  3.14159265]])
>>> np.cos(np.radians(m))
array([[ 8.66025404e-01,   5.00000000e-01,   6.12323400e-17],
       [-5.00000000e-01,  -8.66025404e-01,  -1.00000000e+00]])
```

## Algèbre linéaire avec numpy et numpy.linalg

- ▶ La doc en ligne du module `numpy.linalg` est sur <http://docs.scipy.org/doc/numpy/reference/routines.linalg.html>
- ▶ Quelques exemples :
  - ▶ `np.dot` multiplication matricielle
  - ▶ `np.inner` produit scalaire
  - ▶ `np.diag` création matrice diagonale
  - ▶ `np.transpose` matrice transposée
  - ▶ `np.linalg.det` déterminant de la matrice
  - ▶ `np.linalg.inv` inverse de la matrice
  - ▶ `np.linalg.solve` résolution du système linéaire  $A x = B$
  - ▶ `np.linalg.eig` valeurs propres, vecteurs propres
  - ▶ `np.linalg.eigvals` valeurs propres

## Algèbre linéaire avec numpy et le module numpy.linalg

▴ Produit terme à terme et produit matriciel :

$$R = \text{np.dot}(A, B) \iff R[i, j] = \sum_k A[i, k] B[k, j]$$

```
>>> A = np.array([[1,1],[0,1]]); B = np.array([[2,0],[3,4]])
>>> A*B                                     # produit terme à terme
array([[2, 0],
       [0, 4]])
>>> np.dot(A,B)                             # produit matriciel
array([[5, 4],
       [3, 4]])
>>> a, b = np.array([1,2,3]), np.array([0,1,0]) # 2 vectors
>>> np.dot(a,b)
2
```

## Algèbre linéaire avec numpy et le module numpy.linalg

### ▲ Produit interne :

$$R = \text{np.inner}(A, B) \longleftrightarrow R[i, j] = \sum_k A[i, k] B[j, k]$$

```
>>> A = np.array([[1,1],[0,1]]); B = np.array([[2,0],[3,4]])
>>> np.inner(A, B)
array([[2, 7],
       [0, 4]])
>>> a, b = np.array([1,2,3]), np.array([0,1,0])
>>> np.inner(a, b)
2
```



# Algèbre linéaire avec numpy et le module numpy.linalg

## ▴ Matrices diagonales :

```
>>> np.diag(np.arange(3))  
array([[0, 0, 0],  
       [0, 1, 0],  
       [0, 0, 2]])  
  
>>> np.diag(np.arange(1,3),1)  
array([[0, 1, 0],  
       [0, 0, 2],  
       [0, 0, 0]])  
  
>>> np.diag(np.arange(1,3),-1)  
array([[0, 0, 0],  
       [1, 0, 0],  
       [0, 2, 0]])
```

# Algèbre linéaire avec numpy et le module numpy.linalg

▲ importation de tout `numpy` dans l'espace de nom courant :

```
>>> from numpy import *
>>> from numpy.linalg import *
>>> m = random.random(4).reshape((2,2)); m
array([[ 0.57401918,  0.64990241],
       [ 0.02683847,  0.01328225]])
>>> print det(m)
-0.00981811548021
>>> m.transpose() # ne modifie pas m
array([[ 0.57401918,  0.02683847],
       [ 0.64990241,  0.01328225]])
>>> mi = inv(m); print mi
[[ -1.35283131  66.19421103]
 [  2.73356598 -58.46531103]]
>>> residu = dot(m, mi) - identity(2); print residu
[[ 0.00000000e+00 -7.10542736e-15]
 [ 0.00000000e+00 -1.11022302e-16]]
>>> norm(residu)
7.1062946664058918e-15
```

# Algèbre linéaire avec numpy et le module numpy.linalg

▲ importation de `numpy` et `numpy.linalg` avec des alias :

```
>>> import numpy as np
>>> import numpy.linalg as npl
>>> m = np.random.random(4).reshape((2,2)); m
array([[ 0.29983518,  0.24003911],
       [ 0.63791418,  0.97157407]])
>>> print npl.det(m)
0.138187733609
>>> m.transpose()
array([[ 0.29983518,  0.63791418],
       [ 0.24003911,  0.97157407]])
>>> mi = npl.inv(m); print mi
[[ 7.03082713 -1.73705074]
 [-4.6162866  2.16976696]]
>>> residu = np.dot(m, mi) - np.identity(2); print residu
[[ 4.44089210e-16  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00]]
>>> npl.norm(residu)
4.4408920985006262e-16
```

# Algèbre linéaire avec numpy et le module numpy.linalg

## ▲ Algèbre linéaire avec numpy :

```
>>> import numpy as np
>>> import numpy.linalg as npl
>>> m = np.random.random(9).reshape((3,3)); m
array([[ 0.73377208,  0.26787043,  0.63561671],
       [ 0.9495004 ,  0.73684802,  0.01673223],
       [ 0.08534191,  0.77328228,  0.700254  ]])
>>> y = np.random.random(3); y
array([ 0.92367858,  0.32060105,  0.64198517])
>>> x = npl.solve(m, y); x
array([ 0.42757753, -0.13899785,  1.01817263])
>>> np.dot(m, x)
array([ 0.92367858,  0.32060105,  0.64198517])
>>> np.dot(m, x) - y
array([ 1.11022302e-16,  0.00000000e+00,  0.00000000e+00])
```

# Algèbre linéaire avec numpy et le module numpy.linalg

## ▲ Valeurs propres, vecteurs propres :

```
>>> npl.eigvals(m)
array([ 1.63893416+0.j , 0.26596997+0.5535386j, 0.26596997-0.5535386j])
>>> m = np.random.random(9).reshape((3,3)); m
array([[ 0.83998644,  0.24385328,  0.5285008 ],
       [ 0.79324559,  0.42156439,  0.67233731],
       [ 0.06364909,  0.97863566,  0.80485252]])
>>> vvp = eig(m)
>>> vvp[0]
array([-0.03068944,  0.30647658,  1.7906162 ])
>>> vvp[1]
array([[ -0.27711595, -0.5789238 ,  0.5012003 ],
       [ -0.61339946, -0.33980812,  0.59770115],
       [  0.73955923,  0.74120016,  0.62574084]])
>>> v0, v1, v2 = vvp[1][:,0], vvp[1][:,1], vvp[1][:,2]
>>> dot(m, v0) - vvp[0][0]*v0
array([ -2.08166817e-17,  4.51028104e-17, -4.16333634e-17])
>>> dot(m, v1) - vvp[0][1]*v1
array([  1.94289029e-16,  8.32667268e-17, -1.66533454e-16])
>>> dot(m, v2) - vvp[0][2]*v2
array([ -6.66133815e-16, -2.22044605e-16,  0.00000000e+00])
```

## Générateurs aléatoires avec numpy.random

- ▶ La doc en ligne du module `numpy.random` est disponible sur <http://docs.scipy.org/doc/numpy/reference/routines.random.html>
- ▶ Exemples de générateurs aléatoires uniformes :
  - ▶ `np.random.rand` tirage uniforme continu dans  $[0,1[$
  - ▶ `np.random.random` tirage uniforme continu dans  $[0,1[$
  - ▶ `np.random.randint` tirage uniforme discret dans  $[a, b[$
  - ▶ `np.random.random_integers` tirage uniforme discret dans  $[a, b]$
  - ▶ `np.random.randn` tirage gaussien dans  $[0,1[$
- ▶ Générateurs aléatoires suivant des distributions classiques
  - ▶ vaste catalogue (beta, binomial, chisquare, dirichlet, gamma...)

## Générateurs aléatoires avec numpy.random

### ▲ Générateurs aléatoires (loi uniforme) :

```
>>> import numpy.random as npr
>>> npr.rand(3,2)      # matrice 3X2, tirages uniformes dans [0,1[
array([[ 0.14022471,  0.96360618],
       [ 0.37601032,  0.25528411],
       [ 0.49313049,  0.94909878]])
>>> np.random.randint(5,10, size=10)
array([6, 6, 8, 6, 6, 6, 5, 9, 7, 5])
>>> np.random.randint(5, size=(2, 4)) # tuple pour donner la dimension
array([[4, 0, 2, 4],
       [3, 3, 0, 3]])
>>> np.random.random_integers(1,5,10) # 10 values in [1,5]
array([2, 2, 3, 4, 4, 1, 2, 2, 3, 3])
>>> npr.random()
0.47108547995356098
>>> npr.random((5,))      # tuple pour donner la dimension (shape)
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
>>> 5 * npr.random((3, 2)) - 5      # 3 X 2 array of random reals
array([[ -3.99149989, -0.52338984],
       [ -2.99091858, -0.79479508],
       [ -1.23204345, -1.75224494]])
```

## Générateurs aléatoires avec numpy.random

### ▲ Générateurs aléatoires (loi normale) :

```
>>> np.random.rand(3,2) # matrice 3X2, tirages uniformes dans [0,1]
array([[ 0.14022471,  0.96360618],
       [ 0.37601032,  0.25528411],
       [ 0.49313049,  0.94909878]])

>>> np.random.random_sample()
0.47108547995356098

>>> type(np.random.random_sample())
<type 'float'>

>>> np.random.random_sample((5,)) # tuple pour donner la dimension (shape)
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])

>>> 5 * np.random.random_sample((3, 2)) - 5 # 3 X 2 array of random reals
array([[ -3.99149989,  -0.52338984],
       [ -2.99091858,  -0.79479508],
       [ -1.23204345,  -1.75224494]])
#in [-5, 0]
```



## Générateurs aléatoires avec numpy.random

▲ Vaste catalogue de générateurs aléatoires disponible sur  
[docs.scipy.org/doc/numpy-1.7.0/reference/routines.random.html#distributions](https://docs.scipy.org/doc/numpy-1.7.0/reference/routines.random.html#distributions)

### Distributions

<code>beta(a, b[, size])</code>	The Beta distribution over <code>[0, 1]</code> .
<code>binomial(n, p[, size])</code>	Draw samples from a binomial distribution.
<code>chisquare(df[, size])</code>	Draw samples from a chi-square distribution.
<code>dirichlet(alpha[, size])</code>	Draw samples from the Dirichlet distribution.
<code>exponential([scale, size])</code>	Exponential distribution.
<code>f(dfnum, dfden[, size])</code>	Draw samples from a F distribution.
<code>gamma(shape[, scale, size])</code>	Draw samples from a Gamma distribution.
<code>geometric(p[, size])</code>	Draw samples from the geometric distribution.
<code>gumbel([loc, scale, size])</code>	Gumbel distribution.
<code>hypergeometric(ngood, nbad, nsample[, size])</code>	Draw samples from a Hypergeometric distribution.
<code>laplace([loc, scale, size])</code>	Draw samples from the Laplace or double exponential distribution with
<code>logistic([loc, scale, size])</code>	Draw samples from a Logistic distribution.
<code>lognormal([mean, sigma, size])</code>	Return samples drawn from a log-normal distribution.
<code>logseries(p[, size])</code>	Draw samples from a Logarithmic Series distribution.
<code>multinomial(n, pvals[, size])</code>	Draw samples from a multinomial distribution.
<code>multivariate_normal(mean, cov[, size])</code>	Draw random samples from a multivariate normal distribution.
<code>negative_binomial(n, p[, size])</code>	Draw samples from a negative_binomial distribution.
<code>noncentral_chisquare(df, nonc[, size])</code>	Draw samples from a noncentral chi-square distribution.
<code>noncentral_f(dfnum, dfden, nonc[, size])</code>	Draw samples from the noncentral F distribution.
<code>normal([loc, scale, size])</code>	Draw random samples from a normal (Gaussian) distribution.
<code>pareto(a[, size])</code>	Draw samples from a Pareto II or Lomax distribution with specified shape.
<code>poisson([lam, size])</code>	Draw samples from a Poisson distribution.

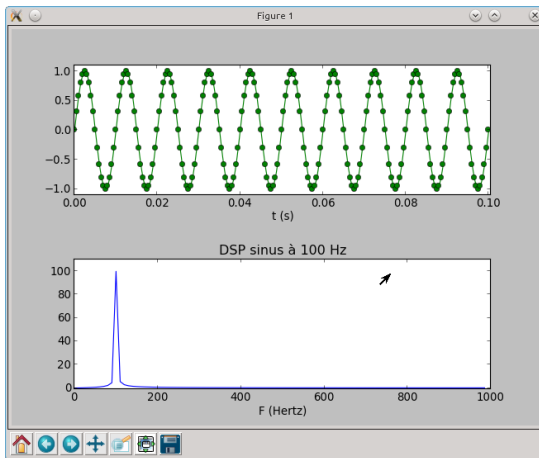
## Calculs FFT avec numpy.fft

- ▶ La doc en ligne est disponible sur  
<http://docs.scipy.org/doc/numpy/reference/routines.fft.html>
- ▶ Calculs de FFT :
  - ▶ **fft** Transformée de Fourier discrète 1D
  - ▶ **ifft** Transformée de Fourier discrète inverse 1D
  - ▶ **fft2** Transformée de Fourier discrète 2D
  - ▶ **ifft2** Transformée de Fourier discrète inverse 2D
  - ▶ **fftn** Transformée de Fourier discrète ND
  - ▶ **ifftn** Transformée de Fourier discrète inverse ND

## Calculs de FFT avec numpy.fft [fft\_sin100.py]

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
# 512 points, 100. Hertz
N, F0 = 201, 100
# Te, Fe : période, fréquence d'échantillonnage :
Fe = 20.*F0
Te, deltaF = 1./Fe, Fe/N
# vecteur temps, fréquence:
vT, vF = np.arange(N)*Te, np.arange(N)*deltaF
# échantillonnage du sinus :
Se = np.sin(2*np.pi*F0*vT)
TF_Se = np.fft.fft(Se)
plt.subplots_adjust(hspace=0.5); plt.subplot(211)
plt.plot(vT, Se, 'o-g')
plt.xlabel('t (s)') ; plt.axis([0, N*Te, -1.1, 1.1])
plt.subplot(212)
plt.plot(vF[:N//2], abs(TF_Se[:N//2]))
plt.ylim(-1, 1.1*max(abs(TF_Se[:N//2])))
plt.xlabel('F (Hertz)') ; plt.title(u'DSP sinus à 100 Hz')
plt.show()
```

## Calculs de fft avec numpy.fft





- ▶ Le module **scipy** utilise **numpy** pour proposer une "boîte" à outil de calcul scientifique
- ▶ Le module **scipy** contient un grand nombre de sous-modules, classes, objets, fonctions... dédiés aux calculs numériques
- ▶ La prise en main du module **scipy** fait l'objet du tutoriel  
<http://docs.scipy.org/doc/scipy-0.12.0/reference/tutorial/index.html>
- ▶ Le site web **scipy** est <http://www.scipy.org>  
La doc en ligne de **scipy** est ici : <http://docs.scipy.org/doc/scipy-0.12.0/reference/>

# Le module Scipy

## ► Tutoriel de prise en main

<http://docs.scipy.org/doc/scipy-0.12.0/reference/tutorial/index.html>

Numpy and Scipy Documentation » SciPy v0.12 Reference Guide (DRAFT) »



Previous topic

SciPy

Next topic

Introduction

Quick search

Go

Enter search terms or a module, class or function name.

## SciPy Tutorial

- Introduction
- Basic functions
- Special functions (`scipy.special`)
- Integration (`scipy.integrate`)
- Optimization (`scipy.optimize`)
- Interpolation (`scipy.interpolate`)
- Fourier Transforms (`scipy.fftpack`)
- Signal Processing (`scipy.signal`)
- Linear Algebra (`scipy.linalg`)
- Sparse Eigenvalue Problems with ARPACK
- Compressed Sparse Graph Routines `scipy.sparse.csgraph`
- Spatial data structures and algorithms (`scipy.spatial`)
- Statistics (`scipy.stats`)
- Multi-dimensional image processing (`scipy.ndimage`)
- File IO (`scipy.io`)
- Weave (`scipy.weave`)

Numpy and Scipy Documentation » SciPy v0.12 Reference Guide (DRAFT) »

## Références bibliographiques

<http://docs.python.org/2/index.html>

[fr.openclassrooms.com/informatique/python/cours](http://fr.openclassrooms.com/informatique/python/cours) [fr.openclassrooms.com/informatique-a-programmer-en-python](http://fr.openclassrooms.com/informatique-a-programmer-en-python)



*Apprenez à programmer en Python*

Vincent Le Goff

Simple IT éd. (Le livre du zéro)

ISBN 979-10-90085-03-9

≈25 €



*Python Essential Reference*

David M. Beazley

Addison Wesley

ISBN 0-672-32978-4



*Matplotlib for Python Developers*

Sandro Tosi

PACTK publishing

ISBN 1847197906



*Apprendre à programmer avec Python*

Gérard Swinnen

Télécharger le PDF



*Programmation en Python  
pour les mathématicques*

A. Casamayou-Boucau

Dunod

ISBN 978-2-10-057422-3



*Python Scripting for  
Computational Science*

T.J. Barth

Springer

ISBN 978-3-540-73915-9



---

✉ [jean-luc.Charles@ensam.eu](mailto:jean-luc.Charles@ensam.eu)  
✉ [eric.Ducasse@ensam.eu](mailto:eric.Ducasse@ensam.eu)