Python en Calcul Scientifique : SciPy

Sylvain Faure

CNRS
Université Paris-Sud

Laboratoire de Mathématiques d'Orsay

6-10 décembre 2010, Autrans

Que contient *SciPy* ?

- Le module *SciPy* contient de nombreux algorithmes très utilisés par les personnes qui font du calcul scientifique : fft, méthodes directes ou itératives pour résoudre des systèmes linéaires, intégration numérique,...
- On peut voir ce module comme une extension de *Numpy* car il contient toutes les fonctions de Numpy.
- Dans toute la suite, on utilisera :

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

Numpy $\subset$ *SciPy* ? oui...

```
>>> np.sqrt(-1.)
Warning: invalid value encountered in sqrt
nan
>>>sp.sqrt(-1.)
1j
>>> np.log(-2.)
Warning: invalid value encountered in log
nan
>>> sp.log(-2.)
(0.69314718055994529+3.1415926535897931j)
>>> sp.exp(sp.log(-2.))
(-2+2.4492127076447545e-16j)
```

SciPy : des "subpackages" et des SciKits

Subpackage	Description
`cluster`	Clustering algorithms
`constants`	Physical and mathematical constants
`fftpack`	Fast Fourier Transform routines
`integrate`	Integration and ordinary differential equation solvers
`interpolate`	Interpolation and smoothing splines
`io`	Input and Output
`linalg`	Linear algebra
`maxentropy`	Maximum entropy methods
`ndimage`	N-dimensional image processing
`odr`	Orthogonal distance regression
`optimize`	Optimization and root-finding routines
`signal`	Signal processing
`sparse`	Sparse matrices and associated routines
`spatial`	Spatial data structures and algorithms
`special`	Special functions
`stats`	Statistical distributions and functions
`weave`	C/C++ integration

Source : SciPy Reference Guide `http://docs.scipy.org/doc/`

A propos des SciKits : `http://scikits.appspot.com/`

Plan

Fonctions spéciales : *scipy.special*

Ce paquet de *SciPy* contient par exemple :

- Les fonctions d'Airy, de Bessel, Gamma, Beta,...
- Les fonctions et intégrales elliptiques
- Quelques outils de statistiques
- La fonction d'erreur et l'intégrale de Fresnel
- Les polynômes de Legendre, de Chebyshev, de Jacobi, d'Hermite,...
- Les harmoniques sphériques,...

Interpolation : *scipy.interpolate*

On y trouve des fonctions :

- pour une interpolation 1D : *interp1d*.
- pour trouver le polynôme passant au plus près d'un ensemble de point : *BarycentricInterpolator*, *KroghInterpolator*, *PiecewisePolynomial* (si l'on a également les dérivées).
- pour une interpolation 2D : *interp2d* (version 0.8), *griddata* (version 0.9),...
- pour régulariser et interpoler à l'aide de splines (contient également un wrapper de *FITPACK*).

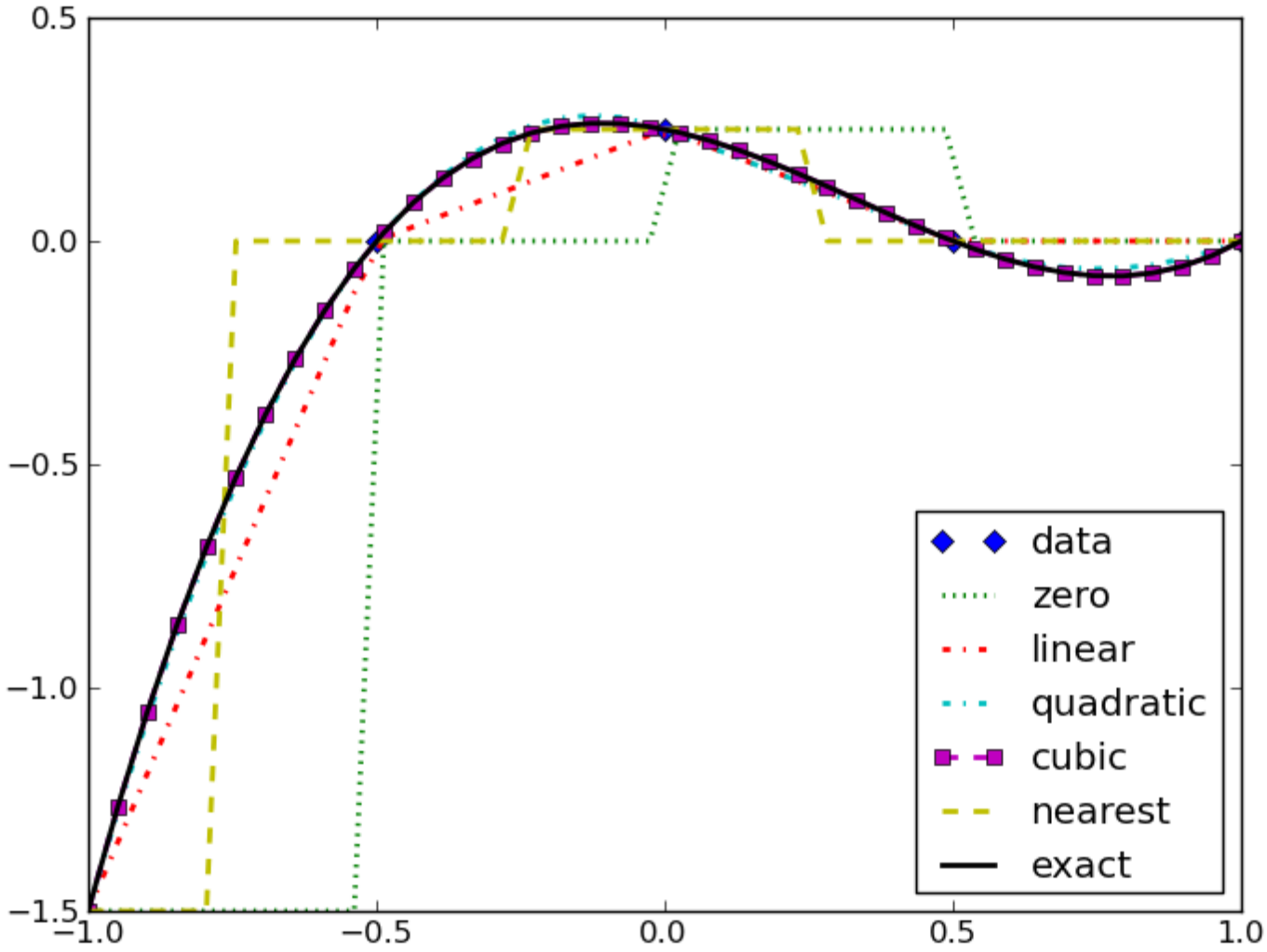
Interpolation 1D

```
interp1d(x, y, kind='linear')
```

avec kind='zero', 'linear', quadratic', 'cubic', 'nearest', 'slinear'.
L'interpolation est linéaire par défaut.

```
from scipy.interpolate import interp1d
x = sp.linspace(-1, 1, num=5)
y= (x-1.)*(x-0.5)*(x+0.5)
f0 = interp1d(x, y, kind='zero')
f1 = interp1d(x, y, kind='linear')
f2 = interp1d(x, y, kind='quadratic')
f3 = interp1d(x, y, kind='cubic')
f4 = interp1d(x, y, kind='nearest')
xnew = sp.linspace(-1, 1, num=40)
ynew= (xnew-1.)*(xnew-0.5)*(xnew+0.5)
plt.plot(x,y,'D',xnew,f0(xnew),':', xnew, f1(xnew),'-.',
     xnew, f2(xnew),'-.', xnew, f3(xnew),'s--', xnew, f4
    (xnew),'--', xnew, ynew, linewidth=2)
plt.legend(['data','zero','linear','quadratic','cubic','
    nearest','exact'],loc='best')
plt.show()
```

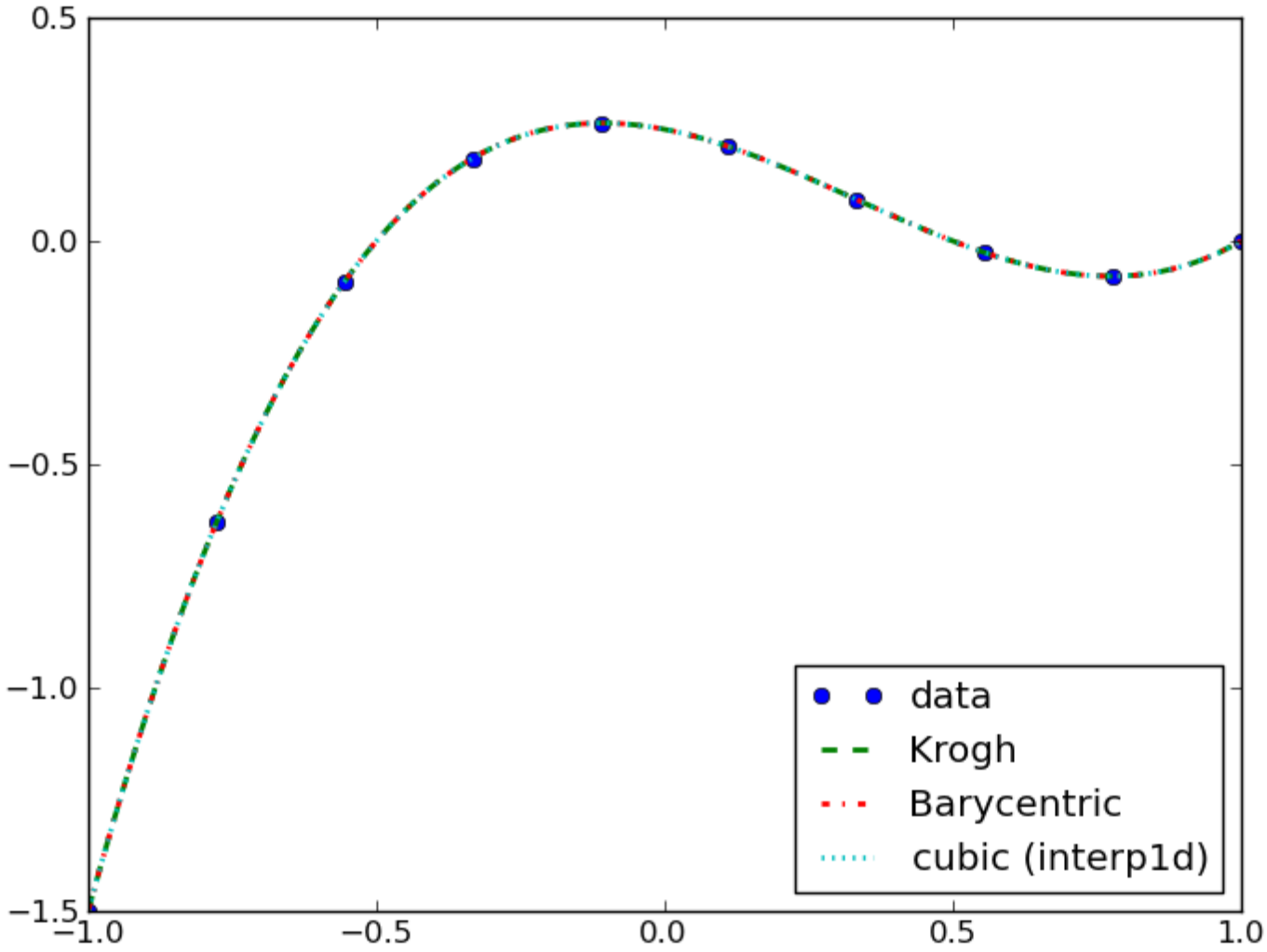
Interpolation 1D



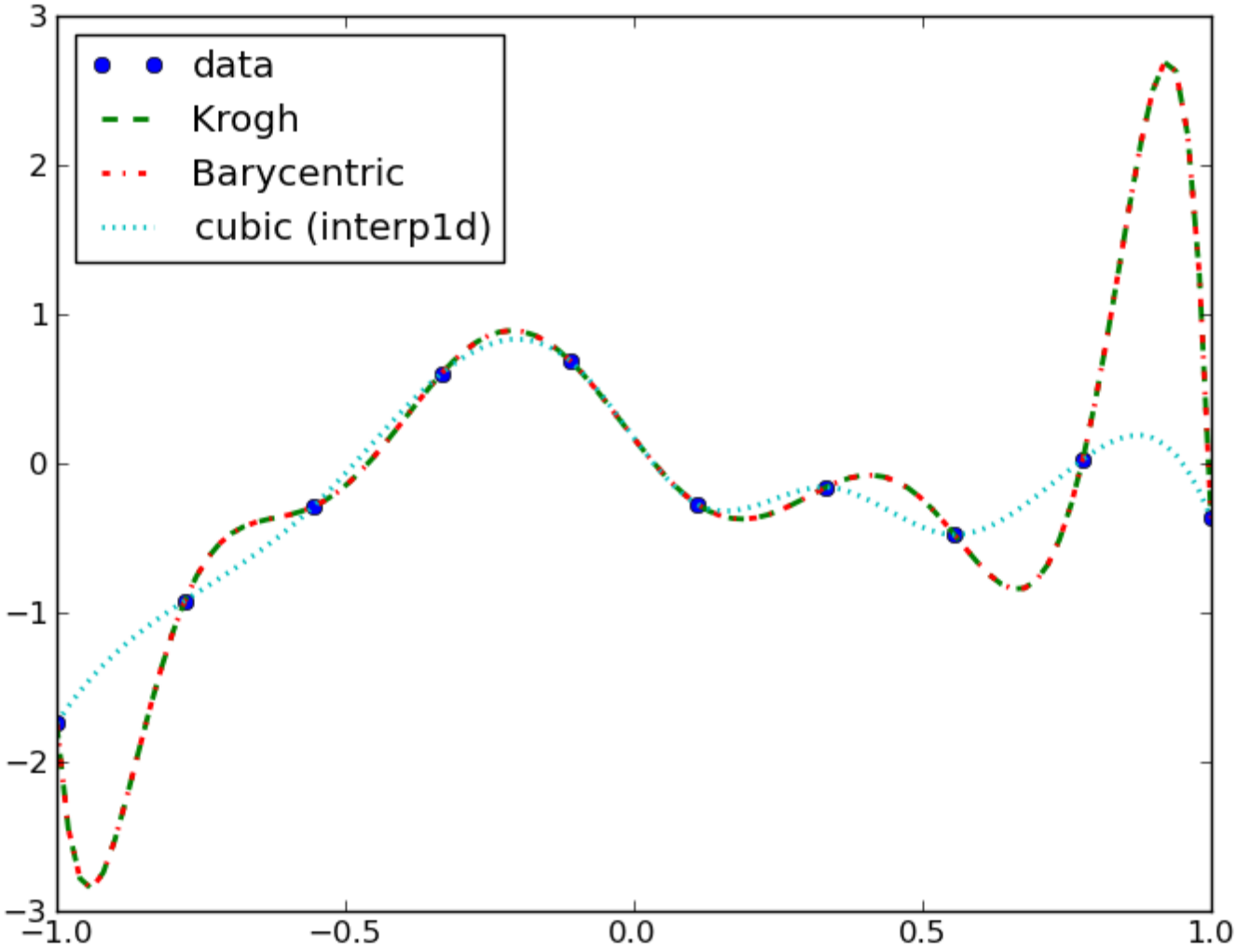
Interpolation 1D

```
a=sp.random.rand(10)-0.5
x = sp.linspace(-1, 1, num=10)
y = (x-1.)*(x-0.5)*(x+0.5)
yn= (x-1.)*(x-0.5)*(x+0.5)+a
K=sp.interpolate.KroghInterpolator(x,y)
Kn=sp.interpolate.KroghInterpolator(x,yn)
B=sp.interpolate.BarycentricInterpolator(x,y)
Bn=sp.interpolate.BarycentricInterpolator(x,yn)
f3 = interp1d(x, y, kind='cubic')
f3n = interp1d(x, yn, kind='cubic')
xnew = sp.linspace(-1, 1, num=100)
plt.figure(1)
plt.plot(x,y,'o',xnew,K(xnew),'--',xnew,B(xnew),'-.',
    xnew,f3(xnew),':',linewidth=2)
plt.legend(['data','Krogh','Barycentric','cubic␣(
    interp1d)'],loc='best')
plt.figure(2)
plt.plot(x,yn,'o',xnew,Kn(xnew),'--',xnew,Bn(xnew),'-.',
    xnew,f3n(xnew),':',linewidth=2)
plt.legend(['data','Krogh','Barycentric','cubic␣(
    interp1d)'],loc='best')
plt.show()
```

Interpolation 1D



Interpolation 1D

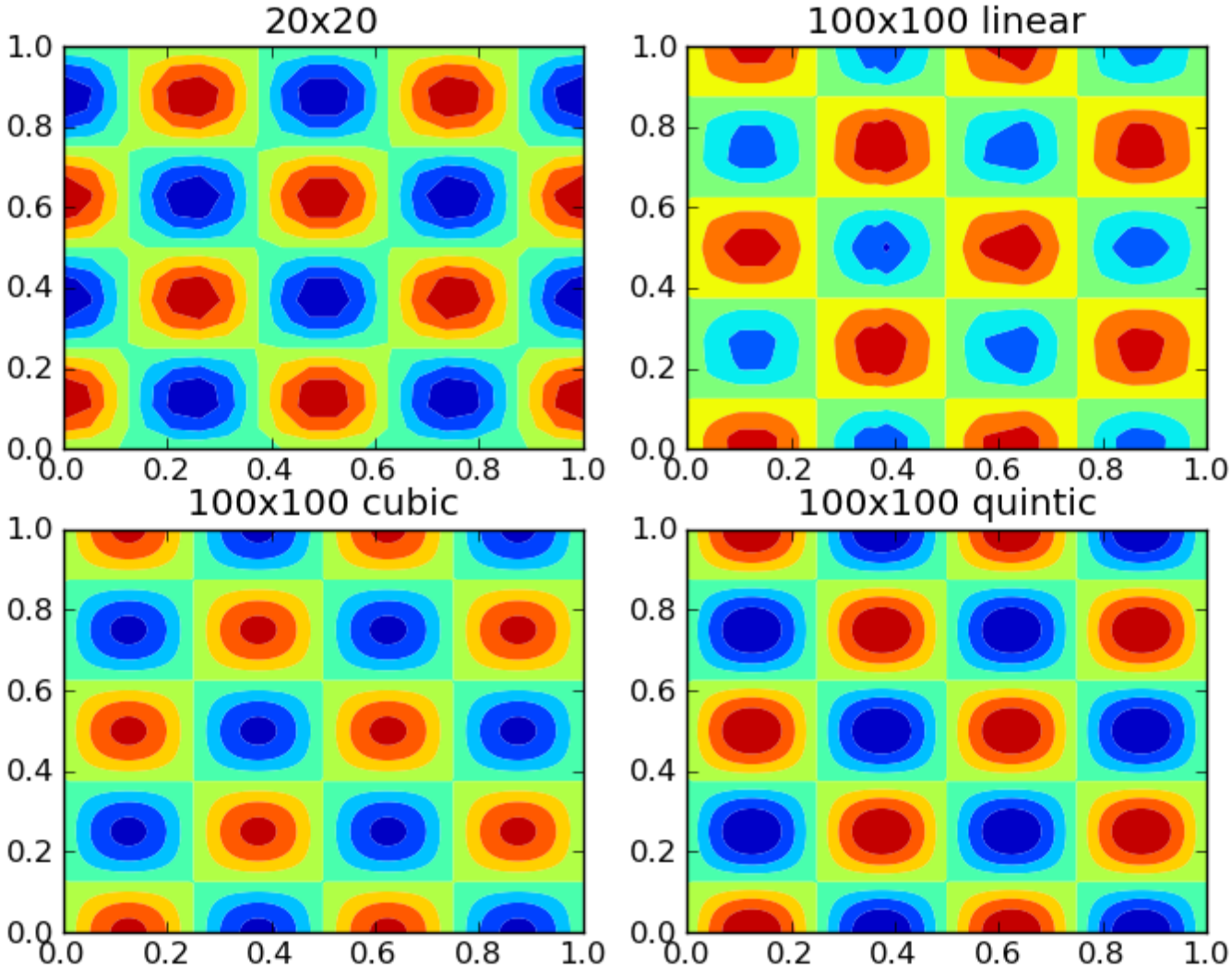


Interpolation 2D

Jusqu'à la version 0.8, on ne dispose que de la fonction *interp2d* :

```
x,y=sp.mgrid[0:1:20j,0:1:20j]
z=sp.cos(4*sp.pi*x)*sp.sin(4*sp.pi*y)
T1=interp2d(x,y,z,kind='linear')
T2=interp2d(x,y,z,kind='cubic')
T3=interp2d(x,y,z,kind='quintic')
X,Y=sp.mgrid[0:1:100j,0:1:100j]
plt.figure(1)
plt.subplot(221)
plt.contourf(x,y,z)
plt.title('20x20')
plt.subplot(222)
plt.contourf(X,Y,T1(X[:,0],Y[0,:]))
plt.title('100x100 linear')
plt.subplot(223)
plt.contourf(X,Y,T2(X[:,0],Y[0,:]))
plt.title('100x100 cubic')
plt.subplot(224)
plt.contourf(X,Y,T3(X[:,0],Y[0,:]))
plt.title('100x100 quintic')
plt.show()
```

Interpolation 2D



Interpolation : futurs développements

Dans la "roadmap" de la version 0.9 de SciPy (actuellement en cours de développement), on peut lire :
"Support for scattered data interpolation is now significantly improved. This version includes a *scipy.interpolate.griddata* function that can perform linear and nearest-neighbour interpolation for N-dimensional scattered data, in addition to cubic spline (C1-smooth) interpolation in 2D and 1D. An object-oriented interface to each interpolator type is also available."
`http://projects.scipy.org/scipy/roadmap`

Transformées de Fourier : *scipy.fftpack*

Contenu :

- Algorithmes de Transformées de Fourier rapides (FFT) pour le calcul de transformées de Fourier discrètes en dimension 1, 2 et n (une exponentielle complexe pour noyau et des coefficients complexes) : *fft*, *ifft* (inverse), *rfft* (pour un vecteur de réels), *irfft*, *fft2* (dimension 2), *ifft2*, *fftn* (dimension n), *ifftn*.
- Transformées en cosinus discrètes de types I, II et III (un cosinus pour noyau et des coefficients réels) : *dct*
- Produit de convolution : *convolve*

```
>>> from scipy.fftpack import *
>>> x=sp.arange(5)
>>> sp.all(abs(x-fft(ifft(x)))<1.e-15)
True
```

Algèbre linéaire : *scipy.linalg*

Contenu : des outils d'algèbre linéaire (matrices pleines ou bandes)

Ce paquet a des choses en commun avec *Numpy* :

- Algèbre linéaire de base : *norm*, *inv*, *solve*, *det*, *lstsq*, *pinv*, *matrix_power* sont dans *Numpy*. On trouve en plus dans *SciPy* la résolution de systèmes linéaires à matrices bandes *solve_banded* et une autre méthode de calcul pour la pseudo-inverse utilisant la décomposition *svd* au lieu de *lstsq*.
- Valeurs propres : dans *Numpy*, on dispose de *eig*(*h*), *eigvals*(*h*) (h pour les matrices hermitiennes). Dans *SciPy* on trouve en plus ces mêmes méthodes pour des matrices bandes.

Algèbre linéaire

- Décompositions : les méthodes *qr*, *svd*, *cholesky* sont communes avec *Numpy* et les méthodes *lu*, *lu_solve*, *orth*, *schur*, *hessenberg* (plus quelques variantes) ont été ajoutées dans *SciPy*.
- Tenseurs : les méthodes *tensorsolve*, *tensorinv* sont présentes dans *Numpy* et absentes dans *SciPy*.

Parmi ce qui existe uniquement dans *scipy.linalg*, on trouve :

- Des fonctions de matrices : *expm*, *sinm*, *sinhm*,... (à ne pas confondre avec le calcul de ces fonctions pour chaque coefficient de la matrice...). Calcul de la matrice signe, *signm*, de la racine carrée d'une matrice *sqrtm*.
- Matrices par blocs diagonales, triangulaires inférieures ou supérieures, circulantes, Toeplitz, compagnon, d'Hadamard, d'Hankel.

Algèbre linéaire : $Ax = b$

```
>>> import scipy.linalg as spl
>>> b=sp.ones(5)

>>> A=sp.array([
[ 1., 3., 0., 0., 0.],
[ 2., 1.,-4 , 0., 0.],
[ 6., 1., 2 ,-3., 0.],
[ 0., 1., 4.,-2.,-3.],
[ 0., 0., 6.,-3., 2.]])
>>> print "x=",spl.solve(A,b,sym_pos=False) # LAPACK (
    gesv ou posv si matrice symetrique)
x= [-0.24074074  0.41358025 -0.26697531 -0.85493827
    0.01851852]

>>> AB=sp.array([
[ 0., 3.,-4.,-3.,-3.],
[ 1., 1., 2.,-2., 2.],
[ 2., 1., 4.,-3., 0.],
[ 6., 1., 6., 0., 0.]])
>>> print "x=",spl.solve_banded((2,1),AB,b) # LAPACK (
    gbsv)
x= [-0.24074074  0.41358025 -0.26697531 -0.85493827
    0.01851852]
```

Algèbre linéaire : $A = P\,L\,U$

```
>>> P,L,U=spl.lu(A) # written for scipy
>>> print "P=",P
P= [[ 0.  1.  0.  0.  0.]
    [ 0.  0.  0.  1.  0.]
    [ 1.  0.  0.  0.  0.]
    [ 0.  0.  0.  0.  1.]
    [ 0.  0.  1.  0.  0.]]
>>> print "L=",L
L= [[ 1.  0.  0.  0.  0.]
    [ 0.16666667  1.  0.  0.  0.]
    [ 0.  0.  1.  0.  0.]
    [ 0.33333333  0.23529412 -0.76470588  1.  0.]
    [ 0.  0.35294118  0.68627451  0.08333333  1.]]
>>> print "U=",U
U= [[ 6.  1.  2. -3.  0.]
    [ 0.  2.83333333 -0.33333333  0.5  0.]
    [ 0.  0.  6. -3.  2.]
    [ 0.  0.  0. -1.41176471  1.52941176]
    [ 0.  0.  0.  0. -4.5]]
```

Algèbre linéaire : $A = P\,L\,U$

```
>>> LU,Piv=spl.lu_factor(A) # LAPACK (getrf)
LU= [[ 3.   0.   0.33333333   0.   0.]
     [ 1.  -4.  -0.41666667   0.   0.]
     [ 1.   2.   6.5  -0.  -0.46153846]
     [ 1.   4.   1.33333333 -3.   0.46153846]
     [ 0.   6.   2.5   2.   -2.76923077]]
 Pivot= [1 2 2 4 4]
>>> print "x=",spl.lu_solve((LU,Piv),b) # LAPACK (getrs)
x= [-0.24074074  0.41358025 -0.26697531 -0.85493827
    0.01851852]
```

Rmq : pour toutes ces méthodes on dispose de l'option "overwrite" (mise à "False" par défaut).

Matrices creuses

Le stockage des matrices creuses peut être effectué aux formats suivants :

- *csc_matrix* : Compressed Sparse Column format
- *csr_matrix* : Compressed Sparse Row format
- *bsr_matrix* : Block Sparse Row format
- *lil_matrix* : List of Lists format
- *dok_matrix* : Dictionary of Keys format
- *coo_matrix* : COOrdinate format
- *dia_matrix* : DIAgonal format

Matrices creuses : *csc_matrix*

```
>>> import scipy.sparse as spsp

>>> row = sp.array([0,2,2,0,1,2])
>>> col = sp.array([0,0,1,2,2,2])
>>> data = sp.array([1,2,3,4,5,6])
>>> Mcsc1=spsp.csc_matrix((data,(row,col)),shape=(3,3))
>>> Mcsc1.todense()
matrix([[1, 0, 4],
        [0, 0, 5],
        [2, 3, 6]])

>>> indptr = sp.array([0,2,3,6])
>>> indices = sp.array([0,2,2,0,1,2])
>>> data = sp.array([1,2,3,4,5,6])
>>> Mcsc2=spsp.csc_matrix((data,indices,indptr),shape
    =(3,3))
>>> Mcsc2.todense()
matrix([[1, 0, 4],
        [0, 0, 5],
        [2, 3, 6]])
```

Matrices creuses : *csc_matrix*

```
>>> print Mcsc2
  (0, 0)    1
  (2, 0)    2
  (2, 1)    3
  (0, 2)    4
  (1, 2)    5
  (2, 2)    6
>>> Mcsc2[1,1]
 0
>>> Mcsc2[1,2]
 5
>>> print "Mcsc2[1:3,2]=",Mcsc2[1:3,2]
Mcs2[1:3,2]=
  (0, 0)    5
  (1, 0)    6
>>> print "Mcsc2[2,1:3]=",Mcsc2[2,1:3]
Mcs2[2,1:3]=
  (0, 0)    3
  (0, 1)    6
```

Matrices creuses : *csc_matrix*

Avantages :

- Opérations arithmétiques efficaces : $csc + csc$, $csc * csc$,...
- "Slicing" efficace selon les colonnes (renvoie une vue de tableau pas une copie)
- Produit matrice vecteur efficace

Inconvénients :

- "Slicing" selon les lignes moins efficaces qu'avec une matrice de type *csr*
- Conversion coûteuse à d'autres formats de matrices creuses (par rapport aux formats *lil* et *dok*)

Matrices creuses : *csr_matrix*

```
>>> row = sp.array([0,0,1,2,2,2])
>>> col = sp.array([0,2,2,0,1,2])
>>> data = sp.array([1,2,3,4,5,6])
>>> Mcsr1=spsp.csr_matrix((data,(row,col)),shape=(3,3))
>>> Mcsr1.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])

>>> indptr = sp.array([0,2,3,6])
>>> indices = sp.array([0,2,2,0,1,2])
>>> data = sp.array([1,2,3,4,5,6])
>>> Mcsr2=spsp.csr_matrix((data,indices,indptr),shape
    =(3,3))
>>> Mcsr2.todense()
matrix([[1, 0, 2],
        [0, 0, 3],
        [4, 5, 6]])
```

Matrices creuses : *csr_matrix*

Avantages :

- Opérations arithmétiques efficaces : $csc + csc$, $csc * csc$,...
- "Slicing" efficace selon les lignes (renvoie une vue de tableau pas une copie)
- Produit matrice vecteur efficace

Inconvénients :

- "Slicing" selon les colonnes moins efficaces qu'avec une matrice de type *csc*
- Conversion coûteuse à d'autres formats de matrices creuses (par rapport aux formats *lil* et *dok*)

Matrices creuses : *bsr_matrix*

```
>>> indptr  = sp.array([0,2,3,6])
>>> indices = sp.array([0,2,2,0,1,2])
>>> data = sp.array([1,2,3,4,5,6]).repeat(4)
>>> print data
[1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6]
>>> data = data.reshape(6,2,2)  # 6 blocs de taille 2x2
>>> Mbsr=spsp.bsr_matrix((data,indices,indptr),shape
    =(6,6))
>>> Mbsr.todense()
matrix([[1, 1, 0, 0, 2, 2],
        [1, 1, 0, 0, 2, 2],
        [0, 0, 0, 0, 3, 3],
        [0, 0, 0, 0, 3, 3],
        [4, 4, 5, 5, 6, 6],
        [4, 4, 5, 5, 6, 6]])
```

Format approprié pour des matrices creuses à blocs denses. Très proche du format *csr*. Peut permettre une accélération des opérations arithmétiques et des produits matrices vecteurs.

Matrices creuses : *lil_matrix*

```
>>> A=spsp.lil_matrix((3,3))
>>> A[2,0]=-10
>>> A[1,2]=10
>>> A[1,1]=1
>>> print A
  (1, 1)    1.0
  (1, 2)    10.0
  (2, 0)    -10.0
>>> print A.rows
[[] [1, 2] [0]]
>>> print A.data
[[] [1.0, 10.0] [-10.0]]
>>> A
<3x3 sparse matrix of type '<type 'numpy.float64'>'
    with 3 stored elements in LInked List format>
>>> B=A.tocsc()
>>> B
<3x3 sparse matrix of type '<type 'numpy.float64'>'
    with 3 stored elements in Compressed Sparse Column
        format>
```

Matrices creuses : *lil_matrix*

Avantages :

- Flexibilité pour le "Slicing", changement de structure permis.
- Conversion à d'autres formats de matrices creuses performante

Inconvénients :

- *lil* + *lil* est lent
- Produit matrice vecteur lent
- "Slicing" suivant les colonnes lent

C'est un format agréable pour construire des matrices creuses mais pas pour calculer ensuite... A comparer au format *coo_matrix*.

Matrices creuses : *dok_matrix*

C'est un autre format permettant de créer une matrice creuse de façon incrémentale :

```
>>> Mdok = spsp.dok_matrix((3,3), dtype=float)
>>> for i in range(3):
>>>     for j in range(3):
>>>         Mdok[i,j] = i+j
>>> print Mdok
  (0, 1)    1.0
  (1, 2)    3.0
  (0, 0)    0.0
  (2, 1)    3.0
  (1, 1)    2.0
  (2, 0)    2.0
  (2, 2)    4.0
  (1, 0)    1.0
  (0, 2)    2.0
```

Accès en $O(1)$ à un élément. Conversion efficace à d'autres format.

Matrices creuses : *coo_matrix*

```
>>> row  = sp.array([0,0,1,3,1,0,0])
>>> col  = sp.array([0,2,1,3,1,0,0])
>>> data = sp.array([1,1,1,1,1,1,1])
>>> Mcoo = spsp.coo_matrix((data,(row,col)),shape=(4,4))
>>> print Mcoo
  (0, 0)    1
  (0, 2)    1
  (1, 1)    1
  (3, 3)    1
  (1, 1)    1
  (0, 0)    1
  (0, 0)    1
>>> print Mcoo.todense()
[[3 0 1 0]
 [0 2 0 0]
 [0 0 0 0]
 [0 0 0 1]]
>>> print Mcoo.tocsr()
  (0, 0)    3
  (0, 2)    1
  (1, 1)    2
  (3, 3)    1
```

Matrices creuses : *coo_matrix*

Avantages :

- Conversion à d'autres formats de matrices creuses performante (très rapide vers les formats *csc*/*csr*)
- Permet la duplication des entrées (automatiquement sommées lors d'une conversion vers un autre format)

Inconvénients :

- Les opérations arithmétiques du type de *coo* + *coo* sous-entendent une conversion vers un autre format :

```
>>> Mcoo
<4x4 sparse matrix of type '<type␣'numpy.int64'>'
    with 7 stored elements in COOrdinate format>
>>> Mcoo+Mcoo
<4x4 sparse matrix of type '<type␣'numpy.int64'>'
  with 4 stored elements in Compressed Sparse Row
      format>
```

- Impossible de renvoyer la valeur d'un élément sans conversion préliminaire.

C'est là encore un format pour construire des matrices creuses...

Matrices creuses : *dia_matrix*

Matrice creuse avec un stockage diagonal.

```
>>> data = sp.array([[1,2,3,4]])
>>> print data.repeat(3)
[1 1 1 2 2 2 3 3 3 4 4 4]
>>> print data.repeat(3,axis=0)
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
>>> data=data.repeat(3,axis=0)
>>> offsets = sp.array([0,-1,2])
>>> Mdia=spsp.dia_matrix((data,offsets),shape=(4,4))
>>> Mdia.todense()
matrix([[1, 0, 3, 0],
        [1, 2, 0, 4],
        [0, 2, 3, 0],
        [0, 0, 3, 4]])
```

Matrices creuses : fonctions *identity*, *eye*

```
>>> Id=spsp.identity(3)
>>> Id
<3x3 sparse matrix of type '<type␣'numpy.float64'>'
    with 3 stored elements in Compressed Sparse Row
        format>
>>> Id.todense()
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> Idmn=spsp.eye(3,4)
>>> Idmn
<3x4 sparse matrix of type '<type␣'numpy.float64'>'
    with 3 stored elements (1 diagonals) in DIAgonal
        format>
>>> Idmn.todense()
matrix([[ 1.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.]])
```

Matrices creuses : fonctions *spdiags*, *find*, *triu/l*, *isspmatrix_**

```
>>> data=sp.array
    ([[10,20,30,40],[1,2,3,4],[100,200,300,400]])
>>> diags=sp.array([-1,0,2])
>>> M=spsp.spdiags(data, diags, 4, 4)
>>> M.todense()
matrix([[  1,   0, 300,   0],
        [ 10,   2,   0, 400],
        [  0,  20,   3,   0],
        [  0,   0,  30,   4]])
>>> print spsp.isspmatrix_csc(M),spsp.isspmatrix_dia(M)
False True
>>> spsp.triu(M).todense()
matrix([[  1,   0, 300,   0],
        [  0,   2,   0, 400],
        [  0,   0,   3,   0],
        [  0,   0,   0,   4]])
>>> r,c,d=spsp.find(M)
>>> print r,c,d
[0 0 1 1 1 2 2 3 3] [0 2 0 1 3 1 2 2 3]
[1 300 10 2 400 20 3 30 4]
```

Matrices creuses : *scipy.sparse.linalg*

```
import scipy.sparse.linalg as spspl
```

Contenu :

- *speigen*, *speigen_symmetric*, *lobpcg*, pour le calcul de valeurs et vecteurs propres (ARPACK).
- *svd* pour une décomposition en valeurs singulières (ARPACK).
- Méthodes directes (UMFPACK si présent ou SUPERLU) ou itératives (`http://www.netlib.org/templates/`, Fortran) pour la résolution de $Ax = b$. Le format *csc* ou *csr* est conseillé.
  - *spsolve* pour les non initiés
  - *dsolve* ou *isolve* pour les moyennement initiés
  - Pour les initiés : méthodes directes *splu* et *spilu* ; méthodes itératives *cg*, *cgs*, *bicg*, *bicgstab*, *gmres*, *lgmres* et *qmr*.
- Algorithmes de minimisation : *lsqr* et *minres*

Matrices creuses : *LinearOperator*

Pour des méthodes itératives telles que *cg*, *gmres*, il n'est pas nécessaire de connaître la matrice du système, le produit matrice vecteur est suffisant. La classe *LinearOperator* permet d'utiliser ces méthodes sans leur donner la matrice du système mais en leur donnant l'opérateur permettant de faire le produit matrice vecteur.

```
>>> def mv(v):
...     return sp.array([ 2*v[0], 3*v[1]])
>>> A=spspl.LinearOperator((2,2),matvec=mv,dtype=float)
>>> A
<2x2 LinearOperator with dtype=float64>
>>> A.matvec(sp.ones(2))
array([ 2.,  3.])
>>> A*sp.ones(2)
array([ 2.,  3.])
>>> A.matmat(sp.array([[1.,-2.],[3.,6.]]))
array([[  2.,  -4.],
       [  9.,  18.]])
```

Matrices creuses : *lu*

```
>>> N=50
>>> un=sp.ones(N)
>>> w=sp.rand(N+1)
>>> A=spsp.spdiags([w[1:],-2*un,w[:-1]],[-1,0,1],N,N)
>>> A=A.tocsc()
>>> b = un
>>> op=spspl.splu(A)
>>> print op
<factored_lu object at 0x102a810b0>
>>> x=op.solve(b)
>>> spl.norm(A*x-b)
1.2312968984861581e-15
```

Matrices creuses : *cg*

Sans préconditionneur :

```
>>> global k
>>> k=0
>>> def f(xk):
...     global k
...     print "iteration ",k," residu=",spl.norm(A*xk-b)
...     k=k+1
>>> x,info=spspl.cg(A, b, x0=sp.zeros(N), tol=1.0e-12,
    maxiter=N, M=None, callback=f)
iteration   0   residu= 2.29978879649
iteration   1   residu= 0.770759215089
...
iteration   21   residu= 2.89531962012e-11
iteration   22   residu= 7.46966041627e-12
iteration   23   residu= 2.41065087099e-12
>>> info
0
```

Matrices creuses : *cg*

Avec préconditionneur :

```
>>> preconditionneur=spspl.spilu(A, drop_tol=1e-1)
>>> xp=preconditionneur.solve(b)
>>> spl.norm(A*xp-b)
0.342171607482
>>> def mv(v):
...     return preconditionneur.solve(v)
>>> lo = spspl.LinearOperator( (N,N), matvec=mv )
>>> k=0
>>> x,info=spspl.cg(A, b, x0=sp.zeros(N), tol=1.0e-12,
    maxiter=N, M=lo, callback=f)
iteration   0   residu= 0.300312169262
iteration   1   residu= 0.0118565322142
...
iteration   6   residu= 8.03964580847e-10
iteration   7   residu= 3.43027715198e-11
iteration   8   residu= 1.06044011533e-12
>>> info
0
```

Intégration : *scipy.integrate*

Contenu :

- Intégration numérique de fonctions : *quad*, *dblquad*, *tplquad*,... Les intégrales doivent être définies. La librairie *Fortran* utilisée est *QUADPACK*.

```
>>> import scipy.integrate as spi
>>> x2 = lambda x: x**2
>>> x2(4)
16
>>> spi.quad(x2,0.,4.)
(21.333333333333336, 2.3684757858670008e-13)
>>> 4.**3/3
21.333333333333332
```

- Intégration numérique de données discrètes : *trapz* (dans *Numpy*), *simps*,...

Intégration : *odeint*

Résolution d'équations aux dérivées ordinaires. Utilise *lsoda* de la librairie *Fortran ODEPACK*.

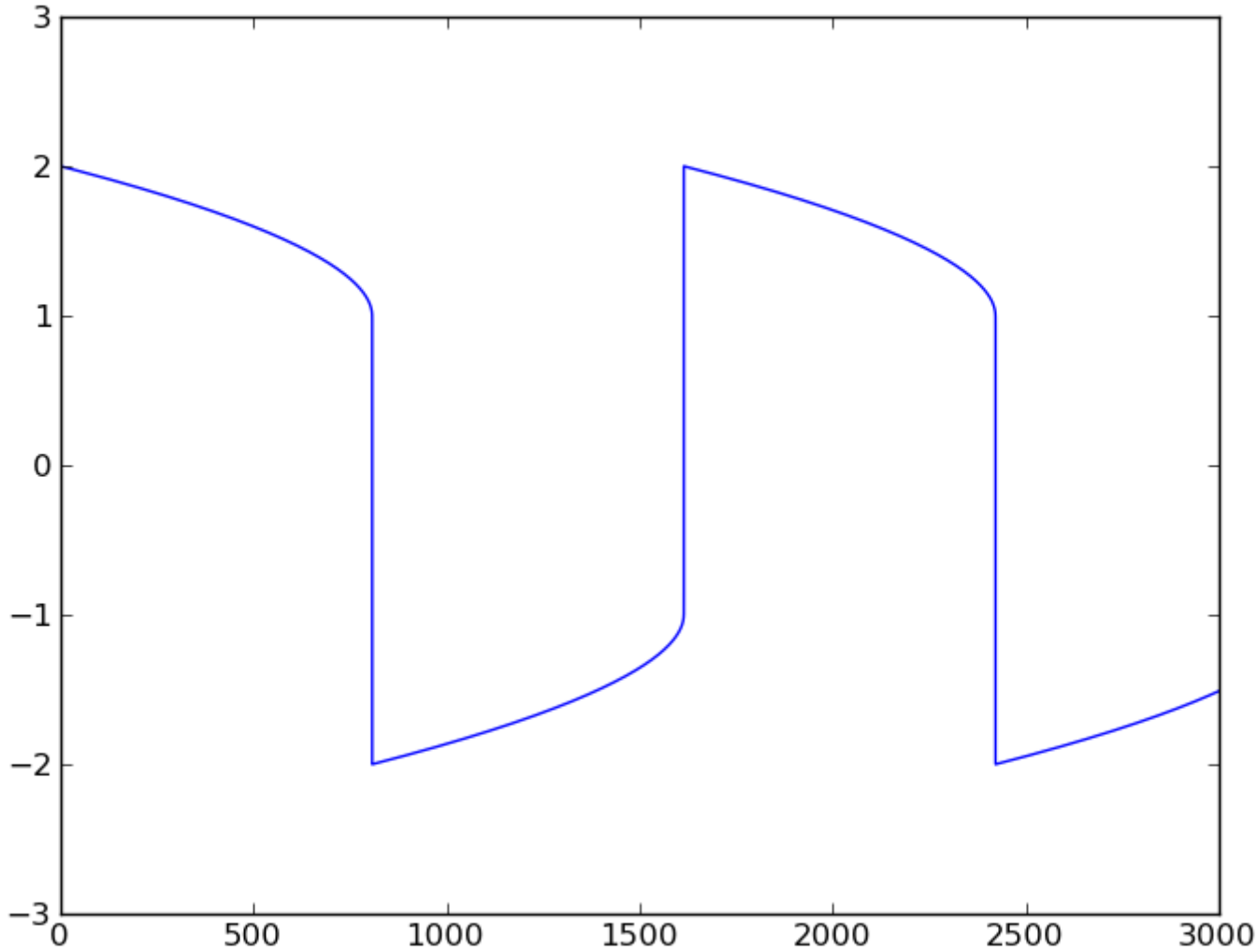
Exemple : l'oscillateur de van der Pol

$$\begin{aligned} y_1'(t) &= y_2(t), \\ y_2'(t) &= 1000(1 - y_1^2(t))y_2(t) - y_1(t), \end{aligned}$$

avec $y_1(0) = 2$ et $y_2(0) = 0$.

```
def vdp1000(y,t):
    dy = sp.zeros(2)
    dy[0] = y[1]
    dy[1] = 1000.*(1. - y[0]**2)*y[1] - y[0]
    return dy
t0=0.
tf=3000.
N=300000
dt=(tf-t0)/N
tgrid=sp.linspace(t0, tf, num=N)
y=spi.odeint(vdp1000, [2.,0.], tgrid)
print y
plt.plot(tgrid,y[:,0])
plt.show()
```

Intégration : *odeint*



Optimisation et $F(x) = 0$: *scipy.optimize*

Contenu :

- Outils "classiques" en optimisation : *fmin*, *fmin_powell*, *fmin_cg*, *fmin_bfgs*, *fmin_ncg* et *leastsq*
- Optimisation sous contraintes : *fmin_l_bfgs_b*, *fmin_tnc* et *fmin_cobyla*
- Optimisation globale : *anneal* et *brute*.
- Résolution de $F(x) = 0$: *fsolve* (*MINPACK*, $N - D$), *bisect*, *newton* (méthode de Newton-Raphson ou de la sécante, $1 - D$), *fixed_point*, *broyden*1/2/3,...

```
>>> import scipy.optimize as spo
```

Optimisation

Exemple de minimisation avec *fmin* : on cherche le minimum de

$$f(x) = \sum_{i=1}^{N-1} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$$

```
>>> def rosen(x):
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x
    [:-1])**2.0)
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = spo.fmin(rosen, x0, xtol=1e-8)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

$$F(x) = 0$$

La fonction *fsolve* est un "wrapper" d'algorithmes issus de MINPACK.

Cas $1 - D$:

```
>>> def f(x):
...     out = [x[0]*sp.cos(x[1]) - 4]
...     out.append(x[1]*x[0] - x[1] - 5)
...     return out
>>> x0 = spo.fsolve(f, [1,1])
>>> print x0, f(x0)
[ 6.50409711  0.90841421] [3.732125719579926 2e-12,
    1.61701763090604 8e-11]
```

$$F(x) = 0$$

Cas $N - D$: résolution de

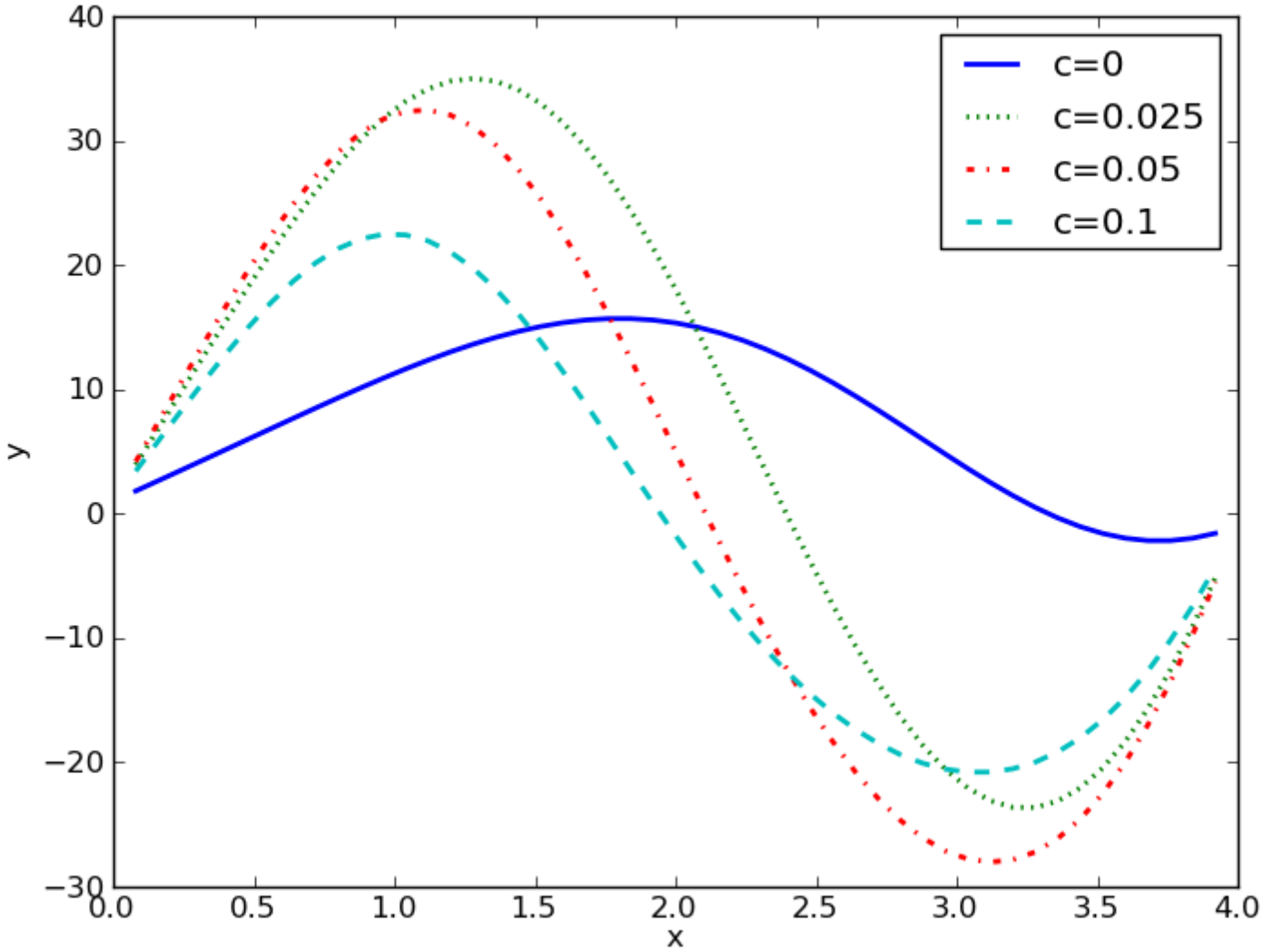
$$y''(x) + xy(x) + cy(x)^2 = 6x \text{ pour } x \in (0,4),$$
$$y(0) = 1 \text{ et } y(4) = -1.$$

```
x0=0.
xN=4.
N=50
dx=(xN-x0)/N
xgrid=sp.arange(0,N+1,1)*dx
un=sp.ones(N-1)
## c=0
A=spsp.spdiags([un, -2*un+xgrid[1:-1]*dx**2, un], [-1,
    0, 1], N-1, N-1)
bc=sp.zeros(N-1)
bc[0]=1
bc[-1]=-1
sm=6*xgrid[1:-1]*dx**2-bc
y0,info=spspl.cg(A, sm, x0=sp.zeros(N-1), tol=1.0e-12,
    maxiter=N-1)
```

$F(x) = 0$

```
## c>0
def f(y,N,dx,xgrid,A,bc,c):
    return A*y+(c*y**2-6*xgrid[1:-1])*dx**2+bc
c=0.025
y1=spo.fsolve(f,y0,args=(N,dx,xgrid,A,bc,c),xtol=1.0e
    -12)
c=0.05
y2=spo.fsolve(f,y1,args=(N,dx,xgrid,A,bc,c),xtol=1.0e
    -12)
c=0.1
def df(y,N,dx,xgrid,A,bc,c):
    return (A+2*c*dx**2*spsp.identity(N-1)).todense()
y3=spo.fsolve(f,y2,args=(N,dx,xgrid,A,bc,c),fprime=df,
    xtol=1.0e-12)
plt.figure()
plt.plot(xgrid[1:-1],y0,xgrid[1:-1],y1,':',xgrid[1:-1],
    y2,'-.',xgrid[1:-1],y3,'--',linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['c=0','c=0.025','c=0.05','c=0.1'],loc='best'
    )
plt.show()
```

$$F(x) = 0$$



Ce qui n'a pas été abordé...

- Les sous-paquets *cluster*, *constants*, *io*, *maxentropy*, *misc*, *ndimage*, *odr*, *signal*, *spatial*, *stats*, *weave*.
- Les *SciKits* `www.scipy.org/scipy/scikits`.

TP équation de la chaleur

On considère l'équation de la chaleur permettant de décrire le phénomène physique de conduction thermique (en l'absence de source thermique dans le domaine) :

$$\frac{\partial u(x,y,t)}{\partial t} - D\,\Delta\, u(x,y,t) = 0,$$

où $u = u(x,y,t)$ est la température (en Kelvin K) et D le coefficient de diffusivité thermique (autour de $0.02\,m^2/s$ pour Fer).
Afin que le problème soit bien posé, on spécifie une condition initiale $u(x,y,0) = u_0(x,y)$ et des conditions aux limites sur le bord du domaine de type Dirichlet ($u(x,y,t)$ connue sur le bord) ou Neumann ($\frac{\partial u(x,y,t)}{\partial n}$ connue sur le bord, n étant la normale extérieure).

TP équation de la chaleur

Discrétisation spatiale : on utilise des différences finies d'ordre 2 et un maillage composé de $Nx * Ny$ rectangles de taille $\Delta x \times \Delta y$.
Discrétisation temporelle : schéma d'Euler implicite de pas de temps Δt
Cela donne, en notant $u_{ij}^n \simeq u(x_i, y_j, t_n)$:

$$\frac{u_{ij}^{n+1} - u_{ij}^n}{\Delta t} - D\left(\frac{u_{i-1j}^{n+1} - 2u_{ij}^{n+1} + u_{i+1j}^{n+1}}{\Delta x^2} - \frac{u_{ij-1}^{n+1} - 2u_{ij}^{n+1} + u_{ij+1}^{n+1}}{\Delta y^2}\right) = 0.$$

- Construire la matrice creuse associée à cette discrétisation en tenant compte des conditions aux bords.
- Etudier et comparer les différentes possibilités de stockages pour cette matrice.
- A chaque itération temporelle, on a besoin de résoudre un système linéaire. Pour cela, utiliser l'algorithme du Gradient Conjugué. On pourra également utiliser un préconditionneur.

TP équation de la chaleur

- Tracer l'évolution de la température dans une plaque métallique chauffée à deux de ses extrémités.

```
fig = plt.figure()
c = np.linspace(u.min(), u.max(), 60)
plt.contourf(x, y, u, c)
plt.colorbar()
plt.show()
```