Scipy

Organisation générale de Scipy

Le module Scipy est constitué de plusieurs sous-modules - en voici quelques-uns :

Quelques sous-modules de scipy

`scipy.fftpack`	transformation de Fourier
`scipy.integrate`	intégration et intégration d'équations différentielles
`scipy.interpolate`	interpolation
`scipy.linalg`	algèbre linéaire
`scipy.optimize`	optimisation (minimisation, recherche de racines)
`scipy.signal`	traitement du signal
`scipy.stats`	statistiques

Intégration

Il existe plusieurs fonctions pour réaliser une intégration numérique. La plus simple est la fonction `quad` qui s'utilise sous la forme (simplifiée) `quad(f,a,b)`. Elle réalise une intégration avec des subdivisions adaptives :

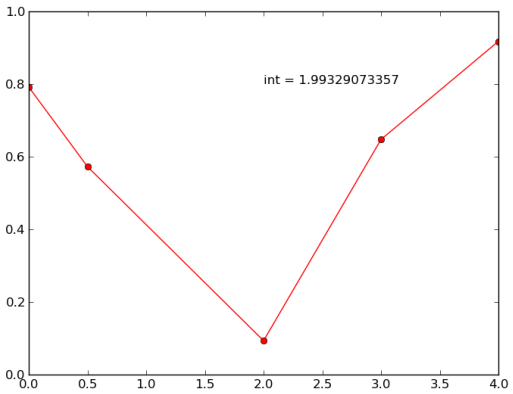
```
>>> from scipy import integrate                # on importe directement le module dans l'espace de noms
>>> def f(x):                                  # fonction à intégrer
...     return(x*x)
>>> integrate.quad(f,0,1)
(0.33333333333333337, 3.700743415417189e-15)    # couple : valeur et précision

# la fonction utilisée est scipy.integrate.quad()
```

on peut préciser quelques arguments comme `epsabs` pour la précision absolue ou `epsrel` pour la précision relative. Il y a beaucoup de paramètres (pas simples à controller).

Référence : http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html

Il existe d'autres fonctions pour des intégrations plus particulières. On dispose d'une méthode des trapèzes avec `cumtrapz(y,x=None, dx=1.0)` : méthode des trapèzes sur le tableau de valeurs y, avec éventuellement le tableau des abscisses correspondantes x (sinon le pas est dx)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate

x=np.array([0,0.5,2,3,4])   # tableau des abscisses
y=np.random.rand(5)         # valeur en ces points (au hasard)

val=integrate.cumtrapz(y,x) # val est le tableau des intégrales cumulées
val=val[len(val)-1]         # on ne conserve que la dernière

plt.plot(x,y,"r-o")
chaine="int = "+str(val)
plt.text(2,0.8,chaine)      # pour afficher une chaîne sur le graphique
plt.show()
```



Différence importante

La différence fondamentale entre ces deux fonctions se situe au niveau des objets traités : la première prend comme argument une fonction, la seconde un tableau de valeurs.

On a des fonctions similaires avec une méthode de Simpson (`integrate.simps`) et de Romberg (`integrate.romberg`) avec des options un peu plus compliquées.

Équations différentielles

On utilise de nouveau `scipy.integrate`. Il existe deux fonctions : une simple `odeint` qui appelle la version compliquée et totalement paramétrable `ode`. On se contente de la première (sans rentrer dans le détail de toutes les options : http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

On veut résoudre le problème de Cauchy :

$$\begin{cases} y' &= f(y,t) \\ y(t_0) &= y_0 \end{cases},$$

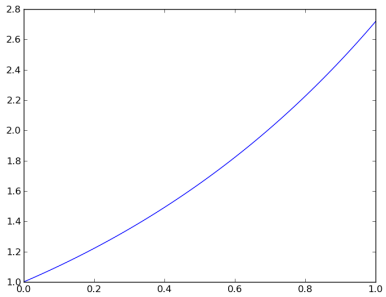
où y est une fonction à valeurs réelles, complexes ou vectorielles.

La syntaxe est `odeint(f,y0,t,options)` où

- f est la fonction apparaissant dans l'équation - **attention** : elle est sous la forme $f(y,t)$ et pas $f(t,y)$,
- y_0 est la valeur initiale,
- t est un tableau avec les valeurs de t pour lesquelles on veut calculer y (il commence à t_0 - habituellement on utilise `t=np.linspace(t0,t1,N)` où N est le nombre de points, mais ce n'est pas une obligation d'avoir une discrétisation régulière).

Exemple : exponentielle

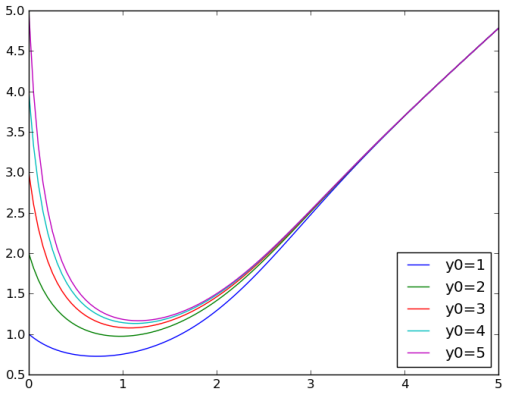
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
def f(y,t):
    return y
# Résolution de y'=y, y(0)=1 sur [0,1]
N=100
t=np.linspace(0,1,N)
y=scipy.integrate.odeint(f,1,t)
plt.plot(t,y)
plt.show()
```



Exemple : avec t

On s'intéresse à $y' = ty - y^2$ avec différentes conditions initiales à $t = 0$

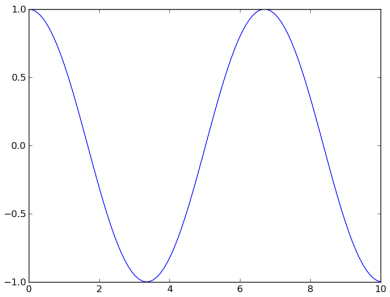
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

def f(y,t):
    return t*y-y**2

N=100
t=np.linspace(0,5,N)
for a in range(1,6):
    y=scipy.integrate.odeint(f,a,t)
    plt.plot(t,y,label="y0="+str(a))
plt.legend(loc='lower right')
plt.show()
```



Exemple : pendule

On transforme l'équation $y'' = -\sin(y)$ en $\begin{cases} y' &= z \\ z' &= -\sin(y) \end{cases}$. Cela donne l'équation $Y' = f(Y,t)$ avec $Y = (y,z)$ et $f(Y,t) = (z, -\sin(y))$. On résout avec $y(0) = 1$ et $y'(0) = z(0) = 0$:

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

def f(y,t):
    return np.array([y[1],-np.sin(y[0])])    # y[0] est la première coordonnée et y[1] la seconde

N=100
t=np.linspace(0,10,N)
y=scipy.integrate.odeint(f,[1,0],t)          # condition initiale (y0,z0)=(1,0)
plt.plot(t,y[:,0])                           # on obtient un tableau de taille Nx2, la première colonne
    donne y
plt.show()
```



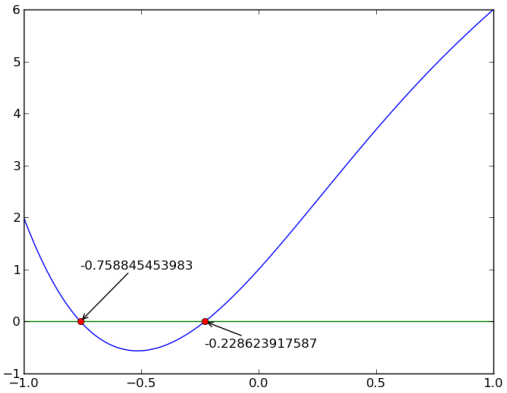
Résolution d'équations

Cas général

On utilise `scipy.optimize.fsolve` (on peut aussi utiliser `scipy.optimize.root` avec un peu plus de réglagles). La syntaxe simplifiée est `fsolve(f,x0,fprime)` où

- `f` est une fonction (une ou plusieurs variables, à valeurs réelles ou vectorielles),
- `x0` la valeur initiale à partir de laquelle l'algorithme s'applique. On peut aussi transmettre un tableau de valeurs initiales - l'algorithme s'exécute pour chaque valeur et renvoie un tableau de solutions.
- `fprime` est une fonction qui renvoie le jacobien de f. Si elle n'est pas donnée, elle est approchée numériquement.

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

def f(x):
    return x**4-3*x**3+2*x**2+5*x+1

x=np.linspace(-1,1,100)
plt.plot(x,f(x),"b")

racines=scipy.optimize.fsolve(f,[-1,0])   # racines en partant de -1 et de 0

plt.plot([-1,1],[0,0],"g-")               # une ligne y=0
plt.plot(racines,[0,0],"r o")             # les deux racines trouvées

# un peu de décoration
plt.annotate(str(racines[0]),xy=(racines[0],0), xytext=(racines[0],1), arrowprops=dict(arrowstyle="->"))
plt.annotate(str(racines[1]),xy=(racines[1],0), xytext=(racines[1],-0.5), arrowprops=dict(arrowstyle="->"))

plt.show()
```



Cas des polynômes

Il existe un type polynôme dans numpy : `poly1d` (et aussi un type pour les polynômes à plusieurs indéterminées) avec des opérations usuelles.

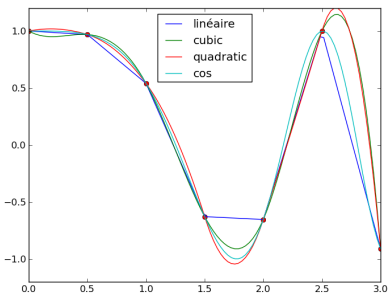
```
>>> poly=np.poly1d([1,3,-2,2])
>>> print(poly)
   3     2
1 x + 3 x - 2 x + 2
>>> poly2=poly**3-1
>>> print(poly2)
   9     8      7     6     5      4      3      2
1 x + 9 x + 21 x - 3 x - 6 x + 66 x - 68 x + 60 x - 24 x + 7
>>> np.roots(poly)
array([-3.68909532+0.j        ,  0.34454766+0.65071134j,
        0.34454766-0.65071134j])
>>> np.roots(poly2)
array([-3.71982912+0.05037998j, -3.71982912-0.05037998j,
       -3.62736508+0.j        ,  0.49664865+0.72640528j,
        0.49664865-0.72640528j,  0.22318047+0.77678525j,
        0.22318047-0.77678525j,  0.31368254+0.42105281j,
        0.31368254-0.42105281j])
```

Interpolation

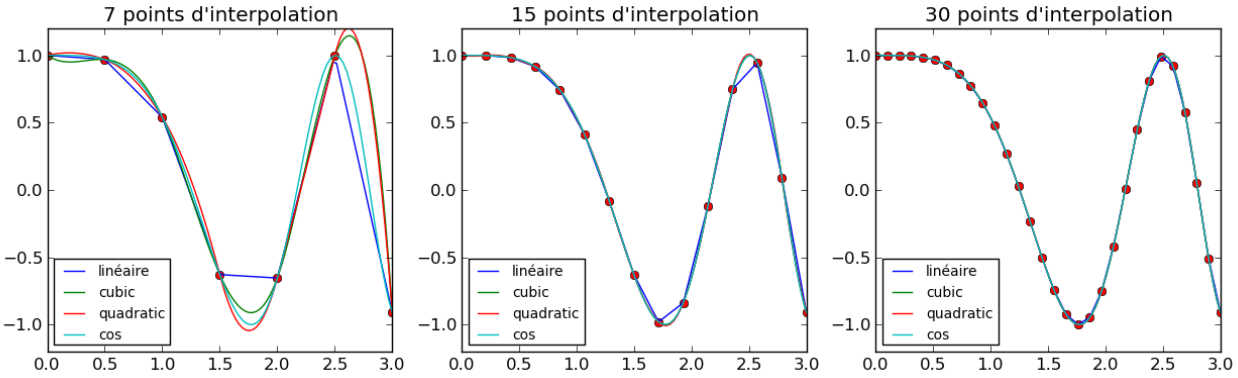
La résolution d'une équation différentielle avec `odeint` ne renvoie pas une fonction mais seulement un tableau de valeurs. Cela ne permet donc pas d'obtenir la solution partout. On peut alors créer une fonction par interpolation avec la fonction `interp1d` avec la syntaxe `interp1d(x,y,kind='type')` où x et y sont les tableaux des abscisses et des ordonnées et *type*, le type d'interpolation utilisée : 'linear', 'nearest', 'zero', 'slinear', 'quadratic, 'cubic'.

Un exemple avec $x \mapsto \cos(x^2)$ sur $[0,3]$:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.interpolate
# on discrétise y=cos(x*x) en 7 points sur [0,3]
x=np.linspace(0,3,7)
y=np.cos(x**2)

f1=scipy.interpolate.interp1d(x,y,kind='linear')
f2=scipy.interpolate.interp1d(x,y,kind='cubic')
f3=scipy.interpolate.interp1d(x,y,kind='quadratic')

X=np.linspace(0,3,100)                # tracé avec 100 points
plt.plot(x,y,"r o")                   # points d'interpolation
plt.plot(X,f1(X),label="linéaire")    # trois méthodes
plt.plot(X,f2(X),label="cubic")
plt.plot(X,f3(X),label="quadratic")

plt.plot(X,np.cos(X**2),label="cos")
plt.legend(loc="upper center")
plt.axis([0,3,-1.2,1.2])
plt.show()
```



Avec en bonus :

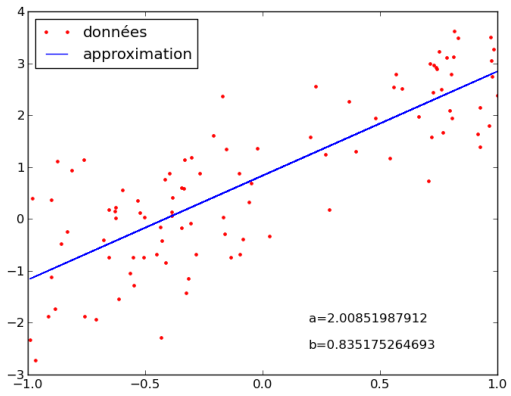


Bien évidemment, le calcul d'une valeur de la fonction obtenue par interpolation en dehors de l'intervalle d'interpolation renvoie une erreur.

Optimisation par moindres carrés

Approximation linéaire

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

# fonction type : variable x et 2 paramètres a et b
def f(x, a, b):
    return a*x+b

# on tire des abscisses dans [-1,1]
x_pert=-1+2*np.random.rand(100)
# on crée des valeurs à partir de y=2x+1
y_pert=f(x_pert,2,1)+0.8*np.random.randn(100)
plt.plot(x_pert,y_pert,"r .",label="données")
# calcul de la meilleure approximation
popt, pcov = scipy.optimize.curve_fit(f, x_pert, y_pert)
# popt est un tableau correspondant à [a,b] calculé
y_fit=f(x_pert,*popt)
# *popt transforme le tableau en une suite d'arguments
plt.plot(x_pert,y_fit,"b",label="approximation")
plt.legend(loc="upper left")
plt.text(0.2,-2,"a="+str(popt[0]))
plt.text(0.2,-2.5,"b="+str(popt[1]))
plt.show()
```



Approximation avec des fonctions modèles

Ici on a un modèle du type $x \mapsto ae^{-b(x-c)^2}$. On crée un nuage de points à partir d'une telle fonction et on optimise.

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

# fonction type : variable x et 3 paramètres a,b et c
def f(x, a, b, c):
    return a*np.exp(-b*(x-c)**2)

x=np.linspace(-1,1,100)
y_pert=f(x,2,1,0.5)+0.2*np.random.randn(100)
plt.plot(x,y_pert,"r .",label="données")

popt, pcov = scipy.optimize.curve_fit(f, x, y_pert)
# popt est un tableau de 3 éléments correspondant aux valeurs de a,b et c
y_fit=f(x,*popt)
# *popt permet de transformer le tableau en une séquence d'arguments
plt.plot(x,y_fit,"b",label="approximation")

plt.legend(loc="upper left")

plt.show()
```