Cours POO M1: python

10 Octobre 2023

python (en général)

Fonction appliquée à un tableau

- Si on veut appliquer une fonction à un tableau numérique, il est conseillé d'utiliser *numpy* (plutôt que *math* ou les listes et tuples)

```
import numpy as np
import math
tn = np.array([1, 2, 4])
l = [1, 2, 4]; t = (1, 2, 4)
f1 = lambda x : x**2 + np.exp(-x) + 1
print(f1(tn))
print(f1(l)) ; print(f1(t))    ## erreurs
```

```
[ 2.36787944  5.13533528 17.01831564]
```

```
f2 = lambda x : x**2 + math.exp(-x) + 1
print(f2(tn)); print(f2(l)); print(f2(t))   ##
    erreurs
```

L'alternative *map*

- *map* applique une fonction à une liste ou un tuple en retournant un objet *map* qu'on peut convertir en liste ou en tuple
- Utiliser *map* sur une liste ou un tuple est une alternative à utiliser *numpy* dans notre exemple

```
map(f2, l)
<map at 0x1059245c0>
list(map(f2, l))
tuple(map(f2, t))
```

```
[2.3678794411714423, 5.135335283236612, 17.018315638888733]
(2.3678794411714423, 5.135335283236612, 17.018315638888733)
```

- On peut aussi convertir un objet *map* en tuple
- Vous devez connaître l'existence des objets *map* mais privilégiez *numpy* pour ce genre d'opérations.

zip

- *zip* parcourt plusieurs listes/tuples/tableaux à la fois.
- renvoie un objet *zip* qui peut être converti en liste ou en tuple, dont les éléments seront des tuples
- se ramène à la plus petite taille si tailles différentes

```
l1 = [1, 2, 4]
l2 = [5, 6, 7, 9, 11]
for i in zip(l1, l2):
        print(i)
```

```
(1, 5)
(2, 6)
(4, 7)
```

zip

```
l1 = [-2, -4, -3, -7]
l2 = (5, 6, 7, 9, 11)
l3 = [1, 2, 4]
for i, j, k in zip(l1, l2, l3):
        print("i = ",i, ", j =", j, ", k =", k)
        print('-------------')
```

```
i =  -2 , j = 5 , k = 1
------------
i =  -4 , j = 6 , k = 2
------------
i =  -3 , j = 7 , k = 4
------------
```

Les versions dans python, *conda* et *pip*

Connaître sa version de python

Dans l'interpréteur python,

```
import sys
print(sys.version_info)
```

Dans spyder, elle s'affiche au lancement dans la console.

conda/Anaconda

- Pour obtenir la version de conda

```
conda --version
```

(ce qui doit renvoyer 4.10.3 ou une version proche)

- La commande suivante permet d'obtenir la liste et les versions des packages de l'environnement

```
conda list
```

- La commande suivante permet d'obtenir la version d'un package donné (par ex. *numpy*)

```
conda list numpy
```

```
# packages in environment at /home/gscarella/.conda/envs/neuron
#
# Name                    Version                   Build  Char
numpy                     1.19.1           py38h30dfecb_0
numpy-base                1.19.1           py38h75fe3a5_0
```

scipy (suite)

Présentation de *scipy.fftpack*

- Transformées de Fourier discrètes
- *fft* fournit la transformée de Fourier discrète d'un tableau
- *ifft* fournit la transformée de Fourier discrète inverse d'un tableau
- *rfft* fournit la transformée de Fourier discrète d'un tableau de réels
- *irfft* fournit la transformée de Fourier discrète inverse pour des réels
- convolutions

Exemples avec *scipy.fftpack*

```
from scipy.fftpack import fft, ifft
import numpy as np
a = np.array([1.2, 6.25, 7.1, 8.11])
b = fft(a)
a2 = ifft(b)
print("b=",b)
print("a2=",a2)  # on retrouve a
```

```
b= [22.66+0.j   -5.9+1.86j -6.06+0.j   -5.9-1.86j]
a2= [1.2+0.j 6.25+0.j 7.1+0.j 8.11+0.j]
```

signal a de longueur N.
Transformation discrète $A(k) = \sum_{n=0}^{N-1} a(n) e^{-2i\pi kn/N}$

Transformation inverse $a(n) = \frac{1}{N} \sum_{k=0}^{N-1} A(k) e^{2i\pi kn/N}$

Exemples avec *scipy.fftpack*

```
from scipy.fftpack import rfft, irfft
import numpy as np
a = np.array([1.2, 6.25, 7.1, 8.11])
b = rfft(a)
a2 = irfft(b)
print("a2=",a2)  # on retrouve a
```

```
a2= [1.2 6.25 7.1 8.11]
```

Plus efficace sur les signaux réels, en exploitant la redondance de termes conjugués lorsque $a(n) \in \mathbb{R}$,

$$a(n)e^{-2i\pi n(N-k)/N} = \overline{a(n)e^{-2i\pi nk/N}} = a(n)e^{2i\pi nk/N}$$

Renvoie

- * $A(0), Re(A(1)), Im(A(1)), ..., Re(A(n/2))]$ si n est pair
- * $[A(0), Re(A(1)), Im(A(1)), ..., Re(A((n-1)/2)), Im(A((n-1)/2))]$ si n est impair.

Présentation de *scipy.optimize*

- Optimisation mathématique
- *scipy.optimize.minimize* calcule le minimum d'une fonction scalaire en 1D ou nD
- *scipy.optimize.fsolve* calcule le 0 d'une fonction (cas 1D)
- *scipy.optimize.root* calcule le 0 d'une fonction (cas nD)
- *scipy.optimize.newton* méthode de Newton

Exemple avec *scipy.optimize.fsolve*

On cherche:

$$x \text{ tel que } x^3 - 2x - 5 = 0$$

```
from scipy.optimize import fsolve
f = lambda x: x**3-2*x-5
x0 = fsolve(f, 2)
for xsol in x0:
  print([xsol,f(xsol)])  # ok
x0 = fsolve(f, -10)
for xsol in x0:
  print([xsol,f(xsol)]) # warning, pas ok
```

Avec 10 comme point de départ, l'algorithme ne converge pas. *fsolve* (algorithme de Powell) n'est généralement pas très performante, on lui préférera par exemple la fonction *newton*.

Présentation de *scipy.sparse*

scipy.sparse contient un ensemble de fonctions pour manipuler les matrices creuses ($card\{a_{i,j} \neq 0,\ 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant n\} << mn$).

- Les méthodes numériques classiques conduisent à manipuler des matrices creuses
- Les matrices creuses contiennent beaucoup de zéros $\rightarrow$ on peut optimiser leur gestion en mémoire et ne nombre d'opérations lors des produits matriciels
- *scipy.sparse* contient *scipy.sparse.linalg* qui contient des méthodes d'algèbre linéaire dédiées aux matrices creuses

scipy.sparse et *scipy.sparse.linalg*

- *scipy.sparse* contient les fonctions *csc_matrix*, *csr_matrix*
- *csc_matrix(...).toarray()* → convertit en matrice pleine
- Plusieurs formats de stockage possibles dans *scipy.sparse*

Stockage de matrices creuses (CSC)

Considérons la matrice creuse suivante à 3 lignes, 4 colonnes

$$A = \begin{pmatrix} 1 & 0 & -2 & 0 \\ 0 & 4 & 6 & 0 \\ 0 & -1 & 3 & 5 \end{pmatrix}$$

Pour les méthodes de stockage creux, on utilise des tableaux 1D au lieu d'une matrice 2D.

Stockage CSC de matrices creuses

1ère méthode : CSC (*Compressed sparse column*)
A est décrite par trois tableaux:

- val: tableau des valeurs non nulles de la matrice
- row: numéros de lignes des valeurs non nulles
- iptr: tableau donnant, pour chaque colonne, la position dans val du 1er élément non nul de la colonne. Si la colonne i est pleine de zéros, `ipr[i]=iptr[i+1]`. Se termine par `len(val)`.

$$\begin{aligned} val &= (1, \ 4, \ -1, \ -2, \ 6, \ 3, \ 5) \\ row &= (0, \ 1, \ 2, \ 0, \ 1, \ 2, \ 2) \\ iptr &= (0, \ 1, \ 3, \ 6, \ 7) \end{aligned}$$

Construction d'une matrice creuse avec *csc_matrix*

```
import scipy as sp
import scipy.sparse as spp
val = np.array([1, 4, -1, -2, 6, 3, 5])
row = np.array([0, 1, 2, 0, 1, 2, 2])
iptr = np.array([0, 1, 3, 6, 7])
M = spp.csc_matrix((val, row, iptr))
print(M)
print(M.toarray())
```

```
In [7]: print(M)
  (0, 0) 1      (1, 2) 6
  (1, 1) 4      (2, 2) 3
  (2, 1) -1     (2, 3) 5
  (0, 2) -2
```

```
In [8]: print(M.toarray())
[[ 1  0 -2  0]
 [ 0  4  6  0]
 [ 0 -1  3  5]]
```

Stockage CSR de matrices creuses

Méthode CSR (*Compressed sparse row*) : comme CSC mais en suivant les lignes

$$\begin{array}{lccccccc} val = & (1, & -2, & 4, & 6, & -1, & 3, & 5) \\ col = & (0, & 2, & 1, & 2, & 1, & 2, & 3) \\ iptr = & (0, & 2, & 4, & 7) & & & \end{array}$$

Additions, produits termes à termes et matriciels entre matrices creuses CSC ou CSR plus efficaces.
CSC (resp. CSR) plus utilisé sur les matrices $M_{m,n}$ avec $m > n$ (resp $m < n$).
CSC (resp. CSR) plus efficace si l'on souhaite fréquemment accéder aux colonnes (resp. lignes).

Construction d'une matrice creuse avec *csr_matrix*

```
import scipy as sp
import scipy.sparse as spp
val = np.array([1, -2, 4, 6, -1, 3, 5])
col = np.array([0, 2, 1, 2, 1, 2, 3])
iptr = np.array([0, 2, 4, 7])
M = spp.csr_matrix((val, col, iptr))
print(M)
print(M.toarray())
```

```
In [7]: print(M)
  (0, 0) 1        (2, 1) -1
  (0, 2) -2       (2, 2) 3
  (1, 1) 4        (2, 3) 5
  (1, 2) 6
```

```
In [8]: print(M.toarray())
[[ 1  0 -2  0]
 [ 0  4  6  0]
 [ 0 -1  3  5]]
```

Stockage CSR et CSC par coordonnées

On peut aussi utiliser la syntaxe `csc_matrix((data, (row, col)))` (idem avec `csc_matrix`) avec :

- val : tableau des valeurs non nulles de la matrice
- row : numéros de lignes des valeurs non nulles
- col : numéros de colonne des valeurs non nulles

$$\begin{aligned} data = &\quad (1, \quad 4, \quad -1, \quad -2, \quad 6, \quad 3, \quad 5) \\ row = &\quad (0, \quad 1, \quad 2, \quad 0, \quad 1, \quad 2, \quad 2) \\ col = &\quad (0, \quad 1, \quad 1, \quad 2, \quad 2, \quad 2, \quad 3) \end{aligned}$$

NB : matrice de taille $(\max(row), max(col))$ par défaut, sinon utiliser l'option `shape=(m, n)`.

Matrice creuse avec *csc_matrix* (coordonnées)

```
import scipy.sparse as spp
data = np.array([1, 4, -1, -2, 6, 3, 5])
row = np.array([0, 1, 2, 0, 1, 2,2])
col = np.array([0, 1, 1, 2, 2, 2, 3])
AS=spp.csc_matrix((data, (row, col))) ; print(AS)
    ; print(AS.toarray())
```

```
  (0, 0) 1
  (1, 1) 4
  (2, 1) -1
  (0, 2) -2
  (1, 2) 6
  (2, 2) 3
  (2, 3) 5
```

```
[[ 1  0 -2  0]
 [ 0  4  6  0]
 [ 0 -1  3  5]]
```

Conversion de array vers matrice sparse

```
A=np.array([[1.,0.,-2.,0.], [0.,4.,6.,0.],
    [0.,-1.,3.,5.]])
AS=spp.csc_matrix(A) # Crée une matrice sparse à
    partir d'un array
print(AS)
```

```
  (0, 0) 1.0
  (1, 1) 4.0
  (2, 1) -1.0
  (0, 2) -2.0
  (1, 2) 6.0
  (2, 2) 3.0
  (2, 3) 5.0
```

De même, `AR=spp.csr_matrix(A)` renvoie la matrice en stockage CRS.

scipy.sparse.linalg

- *scipy.sparse.linalg* contient un ensemble de méthodes permettant la résolution numérique de systèmes linéaires formés de matrices creuses.
- Méthodes directes de résolution
- Méthodes itératives
- Principales fonctions de *scipy.sparse.linalg*
  - *spsolve*
  - *isolve.bicgstab*
  - *isolve.bicg*
  - *isolve.gmres*

Exemple avec *spsolve* (voir Jupyter notebook)

```
import numpy as np
import scipy.linalg as sl
import scipy.sparse as s
import scipy.sparse.linalg as ssl
M=2*np.eye(8000)+3*np.eye(8000,k=314)-2*np.eye
   (8000,k=-123)
M=spp.csr_matrix(M)
b = 1 + np.arange(8000)  # second membre
x = ssl.spsolve(M,b)  # resolution
```

sympy

sympy

- *sympy* permet de faire du calcul symbolique (comme Maple, Mathematica ou sagemath)
- *sympy* est utilisable dans CoCalc

Calculs de trigonométrie avec sympy

```
import sympy as sy
theta =  sy.symbols('theta')

print(sy.sin(theta+sy.pi/2)) #On peut utiliser
    display au lieu de print pour un meilleur
    rendu
```

cos(theta)

```
sy.cos(sy.pi/2-theta)
```

sin(theta)

Affichage de fonctions avec sympy

```
x = sy.symbols('x')
def f(x):
    return sy.exp(-x**2)*sy.sin(3*x)-4*x*sy.cos(x
        **2) +2
sy.plot(f(x))
```

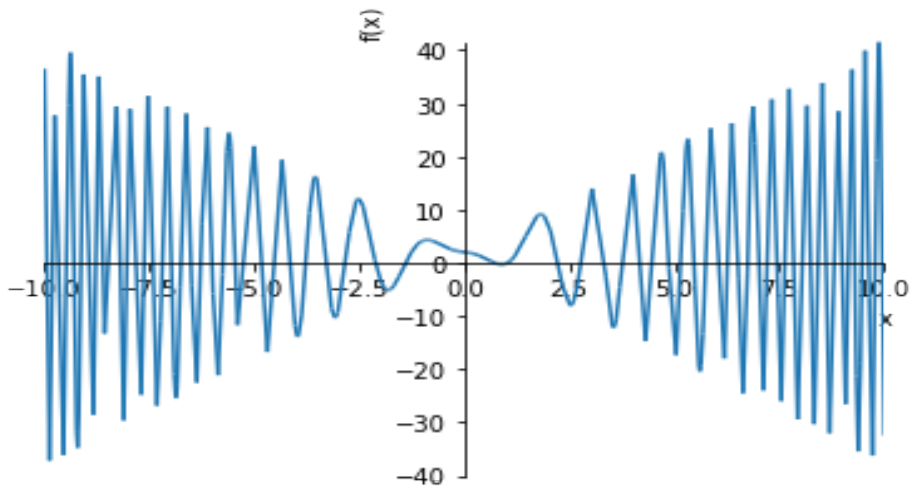
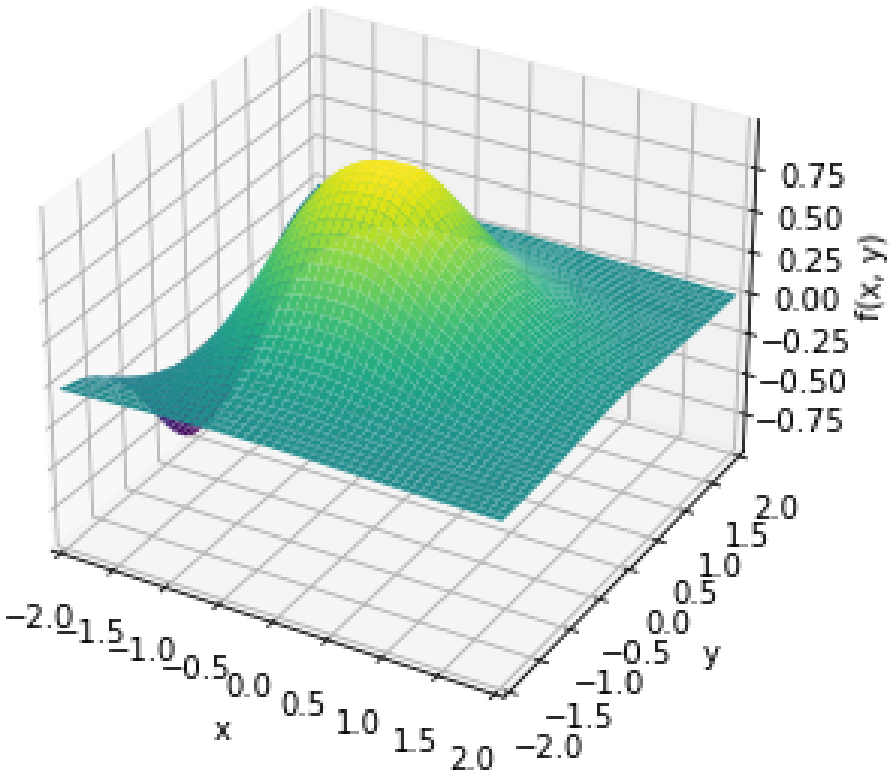


Figure: Plot avec sympy

Affichage de fonctions $\mathbb{R}^2 \to \mathbb{R}$ avec sympy

```
x,y=sp.symbols('x y')
sy.plotting.plot3d((sy.exp(-(x**1.5+y**2))),(x
    ,-2,2),(y,-2,2))
```



Calcul de primitives, d'intégrales, de dérivées

- Primitive

```
x = sy.symbols('x')
sy.integrate(x**2 + 7*x -2, x)
```

```
x**3/3 + 7*x**2/2 - 2*x
```

- Calcul d'intégrale

```
sy.integrate(4/(1+x**2),(x,0,sy.oo))
```

```
2*pi
```

- Dérivation

```
sy.diff(x**2 + 7*x -2, x)
```

```
2*x + 7
```

- Dérivation d'ordre n

```
sy.diff(x**2 + 7*x -2, x,2) #ici pour n=2
```

```
2
```

Résolution de systèmes linéaires

```
x,y=sy.symbols('x,y')
#Système à deux équations :
e1= 2*x+y+1
e2=-4*x-5*y+4

solution=sy.solve((e1,e2),x,y)
print(solution)
```

```
{x: -3/2, y: 2}
```

On peut aussi faire plus simplement

```
solution=sp.solve((2*x+y+1,-4*x-5*y+4),x,y)
```

Racines de polynômes

```
x=sy.symbols('x')
solution=sp.solve(x**3-2*x**2+x+4)
for i in range(len(solution)):
    display(solution[i] )
```

```
-1
3/2 - sqrt(7)*i/2
3/2 + sqrt(7)*i/2
```

On peut aussi faire plus simplement

```
solution=sp.solve((2*x+y+1,-4*x-5*y+4),x,y)
```

Conclusion sympy

Et bien d'autres possibilités

- Résolution d'EDO (exemple dans le notebook)
- Algèbre linéaire (décomposition, élément propres,...)
- systèmes d'équations non linéaires
- simplification
- développement limités (exemple dans le notebook)
- calcul de limites (exemple dans le notebook)
- ...