td2a_cenonce_session_2A

November 26, 2021

1 2A.data - Calcul Matriciel, Optimisation

numpy arrays sont la première chose à considérer pour accélérer un algorithme. Les matrices sont présentes dans la plupart des algorithmes et *numpy* optimise les opérations qui s'y rapporte. Ce notebook est un parcours en diagonal.

[1]:
```
%matplotlib inline
```

[2]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[2]:
```
<IPython.core.display.HTML object>
```

1.0.1 Numpy arrays

La convention d'import classique de numpy est la suivante:

[3]:
```
import numpy as np
```

Creation d'un array: notion de datatype, et dimensions On part d'une liste python contenant des entiers. On peut créer un array numpy à partir de cette liste. Cet array possède des attributs indiquant le data type, le nombre de dimensions de l'array, etc...

[4]:
```
l = [1, 42, 18 ]
a = np.array(l)
print(a)
print(a.dtype)
print(a.ndim)
print(a.shape)
print(a.size)
a
```

```
[ 1 42 18]
int32
1
(3,)
3
```

[4]:
```
array([ 1, 42, 18])
```

On peut indiquer explicitement le dtype lors de la création de l'array. Sinon, Numpy sélectionne automatiquement le dtype. Numpy ajoute un grand nombre de dtype à ceux de Python. Allez jeter un oeil à la liste.

[5]:
```
b = np.array(l, dtype=float)
print(b)
print(b.dtype)
```

```
[  1.  42.  18.]
float64
```

[6]:
```
l[0] = 1.0
bb = np.array(l)
print(bb)
print(bb.dtype)
```

```
[  1.  42.  18.]
float64
```

Assigner un float dans un array de type int va caster le float en int, et ne modifie pas le dtype de l'array.

[7]:
```
a[0] = 2.5
a
```

[7]:
```
array([ 2, 42, 18])
```

On peut forcer le casting dans un autre type avec astype :

[8]:
```
aa = a.astype(float)
aa[0] = 2.5
aa
```

[8]:
```
array([  2.5,  42. ,  18. ])
```

A partir d'une liste de listes, on obtient un array bi-dimmensionnel.
On peut le transposer ou encore l'aplatir en un array 1d

[9]:
```
c = np.array([range(5), range(5,10), range(5)])
print(c)
print("ndim:{}".format(c.ndim))
print("shape:{}".format(c.shape))
print(c.transpose()) #same as c.T
print("shape transposed:{}".format(c.T.shape))
print(c.flatten())
print("ndim flattened:{}".format(c.flatten().ndim))
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]
 [0 1 2 3 4]]
ndim:2
shape:(3, 5)
[[0 5 0]
 [1 6 1]
 [2 7 2]
 [3 8 3]
 [4 9 4]]
shape transposed:(5, 3)
[0 1 2 3 4 5 6 7 8 9 0 1 2 3 4]
ndim flattened:1
```

Indexation, Slicing, Fancy indexing

[10]:
```
print(c)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]
 [0 1 2 3 4]]
```

L'indexation des array multidimensionnels fonctionne avec des tuples.
La syntaxe ':' permet d'obtenir tous les éléments de la dimension.

[11]:
```
print(c[1,3])
print(c[1,:3])
print(c[:,4])
```

```
8
[5 6 7]
[4 9 4]
```

Si on utilise pas un couple sur un array 2d on récupère un array 1d

[12]:
```
print(c[1], c[1].shape)
print(c[1][:3])
```

```
[5 6 7 8 9] (5,)
[5 6 7]
```

On peut aussi utiliser l'indexation par un array (ou une liste python) de booléens ou d'entiers (un mask). Cela s'appelle le fancy indexing. Un mask d'entiers permet de désigner les éléments que l'on souhaite extraire via la liste de leurs indices, on peut aussi répéter l'indice d'un élément pour répéter l'élement dans l'array que l'on extrait.

[13]:
```
ar = np.arange(1,10) #arange est l'equivalent de range mais retourne un numpy array
print('ar = ',ar)
idx = np.array([1, 4, 3, 2, 1, 7, 3])
print('idx = ',idx)
print("ar[idx] =", ar[idx])
print('######')
idx_bool = np.ones(ar.shape, dtype=bool)
idx_bool[idx] = False
print('idx_bool = ', idx_bool)
print('ar[idx_bool] = ', ar[idx_bool])
print('######', 'Que se passe-t-il dans chacun des cas suivants?', '######' )
try:
    print('ar[np.array([True, True, False, True])] = ', ar[np.array([True, True,␣
 ↪False, True])])
except Exception as e:
    # l'expression ar[[True, True, False, True]] déclenche une erreur depuis numpy 1.13
    print("Erreur", e)
```

```
ar =  [1 2 3 4 5 6 7 8 9]
idx =  [1 4 3 2 1 7 3]
ar[idx] = [2 5 4 3 2 8 4]
######
idx_bool =  [ True False False False False  True  True False  True]
```

```
ar[idx_bool] =  [1 6 7 9]
###### Que se passe-t-il dans chacun des cas suivants? ######
Erreur boolean index did not match indexed array along dimension 0; dimension is
9 but corresponding boolean dimension is 4
```

Pourquoi parle-t-on de fancy indexing? Essayez d'indexer des listes python de la même manière...

[14]:
```
list_python = range(10)
list_python[[True, True, False, True]] # déclenche une exception
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-d185468fd957> in <module>()
      1 list_python = range(10)
----> 2 list_python[[True, True, False, True]] # déclenche une exception

TypeError: range indices must be integers or slices, not list
```

[15]:
```
list_python[[2, 3, 2, 7]]  # déclenche une exception
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-769cbcb8bc01> in <module>()
----> 1 list_python[[2, 3, 2, 7]]  # déclenche une exception

TypeError: range indices must be integers or slices, not list
```

View contre Copy Créons un array d. En plus de renvoyer directement un array, la fonction arange permet aussi d'utiliser un step flottant. (Essayer avec le range de python pour voir)

[16]:
```

```

[17]:
```
d = np.arange(1, 6, 0.5)
d
```

[17]:
```
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5])
```

Un point important est que l'on ne recopie pas un array lorsqu'on effectue une assignation ou un slicing d'un array. On travaille dans ce cas avec une View sur l'array d'origine (shallow copy). Toute modification sur la View affecte l'array d'origine.
Dans l'exemple qui suit, e est une view sur d. Lorsqu'on modifie e, d aussi est modifié. (Remarquez au passage que numpy fournit quelques constantes bien pratiques....)

[18]:
```
e = d
e[[0,2, 4]] = - np.pi
e
```

[18]:
```
array([-3.14159265,  1.5       , -3.14159265,  2.5       , -3.14159265,
        3.5       ,  4.        ,  4.5       ,  5.        ,  5.5       ])
```

[19]:
```
d
```

[19]:
```
array([-3.14159265,  1.5       , -3.14159265,  2.5       , -3.14159265,
        3.5       ,  4.        ,  4.5       ,  5.        ,  5.5       ])
```

Si on ne veut pas modifier d indirectement, il faut travailler sur une copie de d (deep copy).

[20]:
```
d = np.linspace(1,5.5,10) #Question subsidiaire: en quoi est-ce différent de np.arange␣
↪avec un step float?
f = d.copy()
f[:4] = -np.e #il s'agit du nombre d'euler, pas de l'array e ;)
print(f)
print(d)
```

```
[-2.71828183 -2.71828183 -2.71828183 -2.71828183  3.          3.5         4.
  4.5         5.          5.5       ]
[ 1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5]
```

Ce point est important car source classique d'erreurs silencieuses: les erreurs les plus vicieuses car l'output sera faux mais python ne râlera pas…
Il faut un peu de temps pour s'habituer mais on finit par savoir de manière naturelle quand on travaille sur une view, quand on a besoin de faire une copie explicitement, etc… En tout cas, vérifiez vos sorties, faites des tests de cohérence, cela ne nuit jamais.
Retenez par exemple que le slicing vous renvoie une view sur l'array, alors que le fancy indexing effectue une copie.
(Au passage, remarquez le NaN (=NotaNumber) déjà introduit lors de la séance 1 sur pandas qui est un module basé sur numpy)

[21]:
```
print('d = ',d)
slice_of_d = d[2:5]
print('\nslice_of_d = ', slice_of_d)
slice_of_d[0] = np.nan
print('\nd = ', d)
mask = np.array([2, 3, 4])
fancy_indexed_subarray = d[mask]
print('\nfancy_indexed_subarray = ', fancy_indexed_subarray)
fancy_indexed_subarray[0] = -2
print('\nd = ', d)
```

```
d =  [ 1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5]

slice_of_d =  [ 2.   2.5  3. ]

d =  [ 1.   1.5  nan  2.5  3.   3.5  4.   4.5  5.   5.5]

fancy_indexed_subarray =  [ nan  2.5  3. ]

d =  [ 1.   1.5  nan  2.5  3.   3.5  4.   4.5  5.   5.5]
```

Manipulation de shape La méthode reshape permet de changer la forme de l'array. Il existe de nombreuses manipulations possibles.
On précise à reshape la forme souhaitée: par un entier si on veut un array 1d de cette longueur, ou un couple pour un array 2d de cette forme.

[22]:
```
g = np.arange(12)
print(g)
g.reshape((4,3))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

[22]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Par défaut, reshape utilise l'énumération dans l'ordre du langage C (aussi appelé "row first"), on peut préciser que l'on souhaite utiliser l'ordre de Fortran ("column first"). Ceux qui connaissent Matlab et R sont habitués à l'ordre "column-first". Voir l'article wikipedia

[23]:
```
g.reshape((4,3), order='F')
```

[23]:
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

On peut utiliser -1 sur une dimension, cela sert de joker: numpy infère la dimension nécessaire ! On peut créer directement des matrices de 0 et de 1 à la dimension d'un autre array.

[24]:
```
np.zeros_like(g)
```

[24]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[25]:
```
np.ones_like(g)
```

[25]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

On peut aussi concatener ou stacker horizontalement/verticalement différents arrays.

[26]:
```
np.concatenate((g, np.zeros_like(g))) #Attention à la syntaxe: le type d'entrée est un␣
↪tuple!
```

[26]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0])
```

[27]:
```
gmat = g.reshape((1, len(g)))
np.concatenate((gmat, np.ones_like(gmat)), axis=0)
```

[27]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]])
```

[28]:
```
np.concatenate((gmat, np.ones_like(gmat)), axis=1)
```

[28]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1]])
```

[29]:
```
np.hstack((g, g))
```

[29]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  0,  1,  2,  3,  4,
        5,  6,  7,  8,  9, 10, 11])
```

[30]:
```
np.vstack((g,g))
```

[30]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]])
```

1.0.2 Exercice 1: Echiquier et Crible d'Erathosthène

- Exercice 1-A Echiquier: Créer une matrice échiquier (des 1 et des 0 alternés) de taille 8x8, de deux façons différentes
  - en vous servant de slices
  - en vous servant de la fonction tile
- Exercice 1-B Piège lors d'une extraction 2d:
  - Définir la matrice $M = \begin{pmatrix} 1 & 5 & 9 & 13 & 17 \\ 2 & 6 & 10 & 14 & 18 \\ 3 & 7 & 11 & 15 & 19 \\ 4 & 8 & 12 & 16 & 20 \end{pmatrix}$
  - En **extraire** la matrice $\begin{pmatrix} 6 & 18 & 10 \\ 7 & 19 & 11 \\ 5 & 17 & 9 \end{pmatrix}$
- Exercice 1-C Crible d'Erathosthène: On souhaite implémenter un crible d'Erathosthène pour trouver les nombres premiers inférieurs à $N = 1000$.
  - partir d'un array de booléens de taille N+1, tous égaux à True.
  - Mettre 0 et 1 à False car ils ne sont pas premiers
  - pour chaque entier k entre 2 et $\sqrt{N}$:
    - si k est premier: on passe ses multiples (entre k^2 et N) à False
  - on print la liste des entiers premiers

[31]:
```
#Exo1a-1:

#Exo1a-2:
```

[32]:
```
#Exo1B:
```

[33]:
```
#Exo1C:
```

1.0.3 Manipulation et Opérations sur les arrays

Il existe un très grand nombre de routines pour manipuler les arrays numpy: Vous trouverez sans doute utiles les pages spécifiques aux routines de stats ou de maths

Opérations élément par élément On déclare a et b sur lesquelles nous allons illustrer quelques opérations

[34]:
```
a = np.ones((3,2))
b = np.arange(6).reshape(a.shape)
print(a)
b
```

```
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
```

[34]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Les opérations arithmétiques avec les scalaires, ou entre arrays s'effectuent élément par élément. Lorsque le dtype n'est pas le même (a contient des float, b contient des int), numpy adopte le type le plus "grand" (au sens de l'inclusion).

[35]:
```
print( (a + b)**2 )
print( np.abs( 3*a - b ) )
f = lambda x: np.exp(x-1)
print( f(b) )
```

```
[[  1.   4.]
 [  9.  16.]
 [ 25.  36.]]
[[ 3.  2.]
 [ 1.  0.]
 [ 1.  2.]]
[[  0.36787944   1.        ]
 [  2.71828183   7.3890561 ]
 [ 20.08553692  54.59815003]]
```

Remarquez que la division par zéro ne provoque pas d'erreur mais introduit la valeur inf :

[36]:
```
b
```

[36]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

[37]:
```
1/b
```

```
c:\Python36_x64\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning:
divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
```

[37]:
```
array([[       inf,  1.        ],
       [ 0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ]])
```

Broadcasting Que se passe-t-il si les dimensions sont différentes?

[38]:
```
c = np.ones(6)
c
```

[38]:
```
array([ 1.,  1.,  1.,  1.,  1.,  1.])
```

[39]:
```
b+c   # déclenche une exception
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-39-882b3e9536b7> in <module>()
----> 1 b+c   # déclenche une exception

ValueError: operands could not be broadcast together with shapes (3,2) (6,)
```

[40]:
```
c = np.arange(3).reshape((3,1))
print(b,c, sep='\n')
b+c
```

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

[40]:
```
array([[0, 1],
       [3, 4],
       [6, 7]])
```

L'opération précédente fonctionne car numpy effectue ce qu'on appelle un broadcasting de `c` : une dimension étant commune, tout se passe comme si on dupliquait c sur la dimension non-partagée avec b. Vous trouverez une explication visuelle simple ici :

[41]:
```
a = np.zeros((3,3))
a[:,0] = -1
b = np.array(range(3))
print(a + b)
```

```
[[-1.  1.  2.]
 [-1.  1.  2.]
 [-1.  1.  2.]]
```

Par contre, il peut parfois être utile de préciser la dimension sur laquelle on souhaite broadcaster, on ajoute alors explicitement une dimension :

[42]:
```
print(b.shape)
print(b[:,np.newaxis].shape)
print(b[np.newaxis,:].shape)
```

```
(3,)
(3, 1)
(1, 3)
```

[43]:
```
print( a + b[np.newaxis,:] )
print( a + b[:,np.newaxis] )
print(b[:,np.newaxis]+b[np.newaxis,:])
print(b + b)
```

```
[[-1.  1.  2.]
 [-1.  1.  2.]
```

```
 [-1.  1.  2.]]
[[-1.  0.  0.]
 [ 0.  1.  1.]
 [ 1.  2.  2.]]
[[0 1 2]
 [1 2 3]
 [2 3 4]]
[0 2 4]
```

Réductions On parle de réductions lorsque l'opération réduit la dimension de l'array. Il en existe un grand nombre. Elles existent souvent sous forme de fonction de numpy ou de méthodes d'un array numpy. On n'en présente que quelques unes, mais le principe est le même : par défaut elles opèrent sur toutes les dimensions, mais on peut via l'argument *axis* préciser la dimension selon laquelle on souhaite effectuer la réduction.

[44]:
```
c = np.arange(10).reshape((2,-1)) #Note: -1 is a joker!
print(c)
print(c.sum())
print(c.sum(axis=0))
print(np.sum(c, axis=1))
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
45
[ 5  7  9 11 13]
[10 35]
```

[45]:
```
print(np.all(c[0] < c[1]))
print(c.min(), c.max())
print(c.min(axis=1))
```

```
True
0 9
[0 5]
```

1.0.4 Algèbre linéaire

Vous avez un éventail de fonctions pour faire de l'algèbre linéaire dans numpy ou dans scipy. Cela peut vous servir si vous cherchez à faire une décomposition matricielle particulière (LU, QR, SVD,...), si vous vous intéressez aux valeurs propres d'une matrice, etc...

Exemples simples Commençons par construire deux arrays 2d correspondant à une matrice triangulaire inférieure et une matrice diagonale :

[46]:
```
A = np.tril(np.ones((3,3)))
A
```

[46]:
```
array([[ 1.,  0.,  0.],
       [ 1.,  1.,  0.],
       [ 1.,  1.,  1.]])
```

[47]:
```
b = np.diag([1,2, 3])
b
```

[47]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

On a vu que les multiplications entre array s'effectuaient élément par élement. Si l'on souhaite faire des multiplications matricielles, il faut utiliser la fonction dot. La version 3.5 introduit un nouvel opérateur [@](https://docs.python.org/3.6/whatsnew/3.5.html#pep-465-a-dedicated-infix-operator-for-matrix-multiplication) qui désigne explicitement la multiplication matricielle.

[48]:
```
print(A.dot(b))
print(A*b)
print(A.dot(A))
```

```
[[ 1.  0.  0.]
 [ 1.  2.  0.]
 [ 1.  2.  3.]]
[[ 1.  0.  0.]
 [ 0.  2.  0.]
 [ 0.  0.  3.]]
[[ 1.  0.  0.]
 [ 2.  1.  0.]
 [ 3.  2.  1.]]
```

On peut calculer l'inverse ou le déterminant de A

[49]:
```
print(np.linalg.det(A))
inv_A = np.linalg.inv(A)
print(inv_A)
print(inv_A.dot(A))
```

```
1.0
[[ 1.  0.  0.]
 [-1.  1.  0.]
 [ 0. -1.  1.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

... résoudre des systèmes d'equations linéaires du type $Ax = b$...

[50]:
```
x = np.linalg.solve(A, np.diag(b))
print(np.diag(b))
print(x)
print(A.dot(x))
```

```
[1 2 3]
[ 1.  1.  1.]
[ 1.  2.  3.]
```

... ou encore obtenir les valeurs propres de A.

[51]:
```
np.linalg.eig(A)
```

[51]:
```
(array([ 1.,  1.,  1.]),
 array([[  0.00000000e+00,   0.00000000e+00,   4.93038066e-32],
```

```
        [  0.00000000e+00,   2.22044605e-16,  -2.22044605e-16],
        [  1.00000000e+00,  -1.00000000e+00,   1.00000000e+00]]))
```

[52]:
```
np.linalg.eigvals(A)
```

[52]:
```
array([ 1.,  1.,  1.])
```

Numpy Matrix Matrix est une sous classe spécialisée pour le calcul matriciel. Il s'agit d'un array numpy 2d qui conserve sa dimension 2d à travers les opérations. Pensez aux différences que cela implique… On peut les construire classiquement depuis les array ou les objets pythons, ou via une string à la Matlab (où les points virgules indiquent les lignes).

[53]:
```
m = np.matrix(' 1 2 3; 4 5 6; 7 8 9')
a = np.arange(1,10).reshape((3,3))
print(m)
print(a)
print(m[0], a[0])
print(m[0].shape, a[0].shape)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 2 3]] [1 2 3]
(1, 3) (3,)
```

Matrix surcharge par ailleurs les opérateurs * et ** pour remplacer les opérations élément par élément par les opérations matricielles. Enfin, une Matrix possède des attributs supplémentaires. Notamment, Matrix.I qui désigne l'inverse, Matrix.A l'array de base.
Il est probable que cela évolue puisque Python 3.5 a introduit le symbol `@` pour la multiplication matricielle.

[54]:
```
m * m
```

[54]:
```
matrix([[ 30,  36,  42],
        [ 66,  81,  96],
        [102, 126, 150]])
```

[55]:
```
a * a
```

[55]:
```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

[56]:
```
m * a # La priorité des matrix est plus importantes que celles des arrays
```

[56]:
```
matrix([[ 30,  36,  42],
        [ 66,  81,  96],
        [102, 126, 150]])
```

[57]:
```
print(m**2)
print(a**2)
```

```
[[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]
[[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

La syntaxe est plus légère pour effectuer du calcul matriciel

[58]:
```
m[0,0]= -1
print("det", np.linalg.det(m), "rank",np.linalg.matrix_rank(m))
print(m.I*m)
a[0,0] = -1
print("det", np.linalg.det(a), "rank",np.linalg.matrix_rank(a))
print(a.dot(np.linalg.inv(a)))
```

```
det 6.0 rank 3
[[  1.00000000e+00   3.99680289e-15   4.49640325e-15]
 [  0.00000000e+00   1.00000000e+00   0.00000000e+00]
 [ -8.88178420e-16   0.00000000e+00   1.00000000e+00]]
det 6.0 rank 3
[[  1.00000000e+00   8.88178420e-16  -1.77635684e-15]
 [ -6.66133815e-16   1.00000000e+00   0.00000000e+00]
 [ -3.33066907e-16   2.66453526e-15   1.00000000e+00]]
```

1.0.5 Génération de nombres aléatoires et statistiques

Le module numpy.random apporte à python la possibilité de générer un échantillon de taille n directement, alors que le module natif de python ne produit des tirages que un par un. Le module numpy.random est donc bien plus efficace si on veut tirer des échantillon conséquents. Par ailleurs, scipy.stats fournit des méthodes pour un très grand nombre de distributions et quelques fonctions classiques de statistiques.
Par exemple, on peut obtenir un array 4x3 de tirages gaussiens standard (soit en utilisant *randn* ou *normal*):

[59]:
```
np.random.randn(4,3)
```

[59]:
```
array([[-0.53862576,  0.7316812 , -0.43393759],
       [-0.39077735, -1.48022294,  0.61423791],
       [ 1.29123337, -2.92158205, -2.33375479],
       [-0.63012998,  0.37943656,  0.33758665]])
```

Pour se convaincre que numpy.random est plus efficace que le module *random* de base de python. On effectue un grand nombre de tirages gaussiens standard, en python pur et via numpy.

[60]:
```
N = int(1e7)
from random import normalvariate
%timeit [normalvariate(0,1) for _ in range(N)]
```

```
9.04 s ± 149 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[61]:
```
%timeit np.random.randn(N)
```

```
301 ms ± 7.46 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

1.0.6 Exercice 2 : marches aléatoires

Simulez (**en une seule fois!**) 10000 marches aléatoires de taille 1000, partant de 0 et de pas +1 ou -1 équiprobables

- Faites un graphe représentant la racine de la moyenne des carrés des positions (=cumul des pas à un instant donné) en fonction du temps
- Quels sont les amplitudes maximales et minimales atteintes parmi l'ensemble des marches aléatoires?
- Combien de marches s'éloigne de plus de 50 de l'origine?
- Parmi celles qui le font, quelle est la moyenne des temps de passage (i.e. le premier moment où ces marches dépassent +/-50)?

Vous aurez peut-être besoin des fonctions suivantes: np.abs, np.mean, np.max, np.where, np.argmax, np.any, np.cumsum, np.random.randint.

[62]:

1.0.7 Exercice 3 : retrouver la série aléatoire à partir des marches aléatoires

L'exercice précédent montre comment générer une marche aléatoire à partir d'une série temporelle aléatoire. Comment retrouver la série initiale à partir de la marche aléatoire ?

[63]:

1.0.8 Optimisation avec scipy

Le module scipy.optimize fournit un panel de méthodes d'optimisation. En fonction du problème que vous souhaitez résoudre, il vous faut choisir la méthode adéquate. Je vous conseille vivement la lecture de ce tutoriel sur l'optimisation numérique, écrit par Gaël Varoquaux.
Récemment, l'ensemble des solvers ont été regroupés sous deux interfaces, même si on peut toujours faire appel à chaque solver directement, ce qui n'est pas conseillé car les entrées sorties ne sont pas normalisées (par contre vous devrez sans doute aller voir l'aide de chaque méthode pour vous en servir):

- Pour minimiser une fonction scalaire d'une ou plusieurs variables:scipy.optimize.minimize
- Pour minimiser une fonction scalaire d'une variable uniquement:scipy.optimize.minimize_scalar

Vous obtiendrez en sortie un objet de type scipy.optimize.OptimizeResult.
Dans la suite, je développe un petit exemple inspiré du tutoriel de la toolbox d'optimisation de Matlab. Par ailleurs, la documentation de cette toolbox est plutôt claire et peut toujours vous servir lorsque que vous avez besoin de vous rafraichir la mémoire sur l'optimisation numérique.
On commence par définir la fonction *bowl_peak*

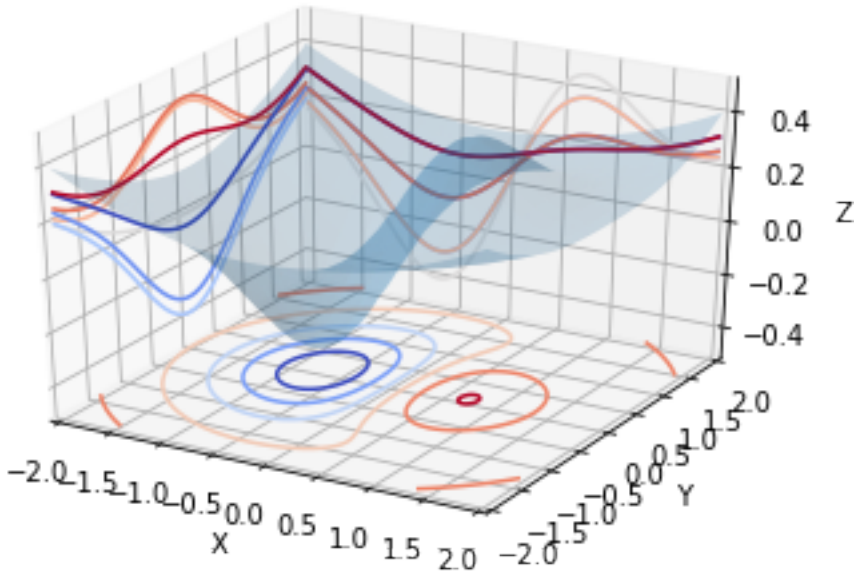
[64]:
```
def bowl_peak(x,y):
    return x*np.exp(-x**2-y**2)+(x**2+y**2)/20
```

On va ensuite chercher un exemple dans la gallerie matplotlib pour la représenter: contour3d_demo3. On modifie légèrement le code pour l'utiliser avec *bowl_peak*

[65]:
```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm #colormaps

min_val = -2
max_val = 2

fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
x_axis = np.linspace(min_val,max_val,100)
y_axis = np.linspace(min_val,max_val,100)
X, Y = np.meshgrid(x_axis, y_axis, copy=False, indexing='xy')
Z = bowl_peak(X,Y)
#X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=5, cstride=5, alpha=0.2)
cset = ax.contour(X, Y, Z, zdir='z', offset=-0.5, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=min_val, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=max_val, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(min_val, max_val)
ax.set_ylabel('Y')
ax.set_ylim(min_val, max_val)
ax.set_zlabel('Z')
ax.set_zlim(-0.5, 0.5)
```

[65]: (-0.5, 0.5)



On voit que le minimum se trouve près de $[-\frac{1}{2}, 0]$. On va utiliser ce point pour initialiser l'optimisation. On va tester différentes méthodes et comparer les sorties obtenues.

[66]:
```
from scipy import optimize
x0 = np.array([-0.5, 0])
fun = lambda x: bowl_peak(x[0],x[1])
methods = [ 'Nelder-Mead', 'CG', 'BFGS', 'Powell', 'COBYLA', 'L-BFGS-B' ]
for m in methods:
    optim_res = optimize.minimize(fun, x0, method=m)
    print("---\nMethod:{}\n".format(m),optim_res, "\n")
```

```
---
Method:Nelder-Mead
  final_simplex: (array([[ -6.69025421e-01,  -1.44567490e-04],
       [ -6.69110179e-01,  -1.81386054e-04],
       [ -6.68989849e-01,  -2.01126337e-04]]), array([-0.40523686, -0.40523685,
-0.40523684]))
           fun: -0.40523685823917283
       message: 'Optimization terminated successfully.'
          nfev: 38
           nit: 20
        status: 0
       success: True
             x: array([ -6.69025421e-01,  -1.44567490e-04])

---
Method:CG
      fun: -0.4052368583334503
     jac: array([ -2.12926418e-04,   3.72529030e-09])
 message: 'Desired error not necessarily achieved due to precision loss.'
    nfev: 24
     nit: 1
    njev: 3
  status: 2
 success: False
       x: array([ -6.69183901e-01,  -3.71395638e-09])

---
Method:BFGS
       fun: -0.40523687025688715
 hess_inv: array([[ 0.52865446,  0.        ],
       [ 0.        ,  1.        ]])
      jac: array([ -6.08339906e-06,   0.00000000e+00])
  message: 'Optimization terminated successfully.'
     nfev: 28
      nit: 6
     njev: 7
   status: 0
  success: True
        x: array([ -6.69075034e-01,  -7.45058060e-09])

---
Method:Powell
    direc: array([[  0.00000000e+00,   1.00000000e+00],
       [ -6.85432298e-04,  -4.67045589e-11]])
     fun: -0.40523687026669025
 message: 'Optimization terminated successfully.'
    nfev: 62
     nit: 2
  status: 0
 success: True
       x: array([ -6.69071822e-01,  -1.15386055e-08])

---
Method:COBYLA
```

```
      fun: -0.4052368678399868
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 32
  status: 1
 success: True
       x: array([ -6.69108584e-01,  -4.89154557e-05])

---
Method:L-BFGS-B
       fun: -0.40523687026621352
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
      jac: array([  1.35447209e-06,   0.00000000e+00])
  message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 15
      nit: 3
   status: 0
  success: True
        x: array([ -6.69071114e-01,  -8.35621530e-09])
```

On trouve un minimum à -0.4052 en $[-0.669, 0.000]$ pour toutes les méthodes qui convergent. Notez le message de sortie de 'CG' qui signifie que le gradient ne varie plus assez. Personnellement, je ne trouve pas ce message de sortie très clair. Le point trouvé est bien l'optimum cherché pourtant. Notez aussi le nombre d'évaluations de la fonction (*nfev*) pour chaque méthode, et le nombre d'évaluation de gradient (*njev*) pour les méthodes qui reposent sur un calcul de gradient.
Remarquez aussi que si on relance *Anneal* plusieurs fois, on n'est pas assuré d'obtenir la même solution, puisqu'il s'agit d'une métaheuristique.

[67]:
```
for i in range(4):
    optim_res = optimize.minimize(fun, x0, method='BFGS')
    print("---\nMethod:{} - Test:{}\n".format(m,i),optim_res, "\n")
```

```
---
Method:L-BFGS-B - Test:0
       fun: -0.40523687025688715
 hess_inv: array([[ 0.52865446,  0.        ],
       [ 0.        ,  1.        ]])
      jac: array([ -6.08339906e-06,   0.00000000e+00])
  message: 'Optimization terminated successfully.'
     nfev: 28
      nit: 6
     njev: 7
   status: 0
  success: True
        x: array([ -6.69075034e-01,  -7.45058060e-09])

---
Method:L-BFGS-B - Test:1
       fun: -0.40523687025688715
 hess_inv: array([[ 0.52865446,  0.        ],
       [ 0.        ,  1.        ]])
      jac: array([ -6.08339906e-06,   0.00000000e+00])
  message: 'Optimization terminated successfully.'
```

```
     nfev: 28
      nit: 6
     njev: 7
   status: 0
  success: True
        x: array([ -6.69075034e-01,  -7.45058060e-09])

---
Method:L-BFGS-B - Test:2
       fun: -0.40523687025688715
 hess_inv: array([[ 0.52865446,  0.        ],
       [ 0.        ,  1.        ]])
      jac: array([ -6.08339906e-06,   0.00000000e+00])
  message: 'Optimization terminated successfully.'
     nfev: 28
      nit: 6
     njev: 7
   status: 0
  success: True
        x: array([ -6.69075034e-01,  -7.45058060e-09])

---
Method:L-BFGS-B - Test:3
       fun: -0.40523687025688715
 hess_inv: array([[ 0.52865446,  0.        ],
       [ 0.        ,  1.        ]])
      jac: array([ -6.08339906e-06,   0.00000000e+00])
  message: 'Optimization terminated successfully.'
     nfev: 28
      nit: 6
     njev: 7
   status: 0
  success: True
        x: array([ -6.69075034e-01,  -7.45058060e-09])
```

On va évaluer le temps de calcul nécessaire à chaque méthode.

[68]:
```
for m in methods:
    print("Method:{}:".format(m))
    %timeit optim_res = optimize.minimize(fun, x0, method=m)
    print('#############')
```

```
Method:Nelder-Mead:
894 µs ± 40.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
#############
Method:CG:
788 µs ± 49 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
#############
Method:BFGS:
592 µs ± 4.57 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
#############
Method:Powell:
1.01 ms ± 26.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
############
Method:COBYLA:
193 µs ± 12.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
############
Method:L-BFGS-B:
222 µs ± 18.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
############
```

On peut aussi fournir des arguments supplémentaires à la fonction qu'on optimise. Par exemple, les données lorsque vous maximisez une log-vraissemblance. En voici un exemple: on considère une version rescaled de la fonction *bowl_peak*. Vous pourriez aussi utiliser une lambda fonction.

[69]:
```
def shifted_scaled_bowlpeak(x,a,b,c):
    return (x[0]-a)*np.exp(-((x[0]-a)**2+(x[1]-b)**2))+((x[0]-a)**2+(x[0]-b)**2)/c
a = 2
b = 3
c = 10
optim_res = optimize.minimize(shifted_scaled_bowlpeak, x0, args=(a,b,c), method='BFGS')
print(optim_res)
print('#######')
optim_res = optimize.minimize(lambda x:shifted_scaled_bowlpeak(x,a,b,c), x0,␣
↪method='BFGS')
print(optim_res)
```

```
      fun: 0.05000000675226609
 hess_inv: array([[  1.40782352e+00,  -1.59338758e+02],
       [ -1.59338758e+02,   7.19318682e+05]])
      jac: array([ -9.78726894e-06,   5.63450158e-08])
  message: 'Optimization terminated successfully.'
     nfev: 96
      nit: 23
     njev: 24
   status: 0
  success: True
        x: array([ 2.49997551, -1.22943768])
#######
      fun: 0.05000000675226609
 hess_inv: array([[  1.40782352e+00,  -1.59338758e+02],
       [ -1.59338758e+02,   7.19318682e+05]])
      jac: array([ -9.78726894e-06,   5.63450158e-08])
  message: 'Optimization terminated successfully.'
     nfev: 96
      nit: 23
     njev: 24
   status: 0
  success: True
        x: array([ 2.49997551, -1.22943768])
```

Vous pouvez continuer ce petit benchmark en ajoutant le gradient et la hessienne... les calculs seront plus précis et plus rapides.

1.0.9 Exercice 4: simulation, régression, estimation par maximisation de la vraisemblance

- On commence par simuler la variable $Y = 3X_1 - 2X_2 + 2 + \epsilon$ où $X_1, X_2, \epsilon \sim \mathcal{N}(0, 1)$

- On souhaite ensuite retrouver les coefficients dans la régression linéaire de Y sur X_1 et X_2 dans un modèle avec constante, par la méthode des Moindres Carrés Ordinaires. On rappelle que la forme matricielle de l'estimateur des MCO est $\hat{\beta} = (X'X)^{-1}X'Y$
- Enfin, Y étant normale, on souhaite estimer ses paramètres par maximisation de vraisemblance:
  - La densité s'écrit: $f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
  - La log-vraisemblance: $\ln \mathcal{L}(\mu, \sigma^2) = \sum_{i=1}^{n} \ln f(x_i; \mu, \sigma^2) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2$.
  - L'écriture des conditions au premier ordre donne une formule fermée pour les estimateurs du maximum de vraisemblance: $\hat{\mu} = \overline{x} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i$, $\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$.
  - Vérifiez en les implémentant directement que vous trouvez bien la même solution que le minimum obtenu en utilisant scipy.optimize.minimize pour minimiser l'opposé de la log-vraissemblance.

[70]:

1.0.10 Exercice 5 : Optimisation quadratique (sous contraintes) avec cvxopt

Voir l'exercice 1 ici

1.0.11 Références

- 100 numpy exercises
- Un tutoriel bien fait et très complet sur numpy. L'un des auteurs n'est autre que Gaël Varoquaux qui sera présent pour la séance 3. Voir aussi l'ensemble du tutoriel et notamment la partie optimisation

[71]: