Livre de recettes Python

Lycée Faidherbe

7 octobre 2017

Table des matières

1 Présentation

Ce document a pour but de présenter quelques solutions à des questions portant sur l'utilisation d'outils informatiques qui peuvent se poser lors de l'étude de problèmes scientifiques.
Il ne s'agit pas d'informatique au sens de l'analyse des moyens utilisés mais juste de l'emploi d'algorithmes créés par d'autres.
Nous allons donc exposer quelques unes des fonctions qu'apporte l'immense bibliothèque associée au langage Python.

2 Comment stocker mes données ?

Le premier outil n'est pas juste une bibliothèque de fonction : le module `numpy` introduit un nouveau type de données, les tableaux ou `array`.
On charge classiquement le module par l'instruction suivante (`np` est l'abréviation conventionnelle).

```
import numpy as np
```

Les tableaux de `numpy` (`array`) sont des assemblages avec des propriétés caractéristiques.

- Leur longueur est fixée.
- Ils sont homogènes, tous les éléments sont des nombres de type fixé au départ.
- Ils peuvent avoir plusieurs dimensions.
- On peut en modifier les éléments.
- L'accès en lecture ou en écriture se fait par `t[i]`.
- Si le tableau a plusieurs dimensions on peut remplacer la notation `t[i][j]` par `t[i,j]`.

Bien qu'ils ressemblent aux listes de python, les tableaux seront très utiles quand on a besoin de données de grande taille. Les contraintes de leur construction font en effet que les calculs s'effectuent très rapidement. En particulier les extractions de sous-tableaux, qui s'écrivent de la même façon que pour les listes python [1], sont un outils très efficace.

De plus les tableaux ont un comportement adapté aux calculs mathématiques.

- L'addition de 2 tableaux de même taille s'effectue par la somme terme-à-terme (et non par la concaténation comme en python).
- La multiplication d'un tableau par un nombre est possible : elle consiste à multiplier chaque terme du tableau par le nombre.
- Plus généralement on peut appliquer une fonction à tous les termes d'un tableau `T` en écrivant simplement `f(T)`. Si `f` utilise des fonctions mathématiques elles doivent provenir de `numpy` : `np.sin` et non `math.sin`.

Dans toute la suite nous supposerons que le module `numpy` a été importé.

2.1 Comment créer des tableaux ?

Pour créer un tableau, plusieurs moyens sont possibles.

1. `np.array` transforme un assemblage classique en tableau `numpy`

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

   Le type d'un tableau est fixé, les tableaux ci-dessus sont donc des tableaux d'entiers. On ne pourra donc pas donner une valeur flottante aux composantes.

2. Pour imposer le type des données on peut ajouter le paramètre `dtype` lors de la création. Les principaux types sont `int`, `float` et `complex`.

1. `T[a:b,c:d]` par exemple.

```
>>> np.array((-2,5,-3,9,7),dtype=complex)
array([-2.+0.j,  5.+0.j, -3.+0.j,  9.+0.j,  7.+0.j])
```

`numpy` permet de préciser la représentation en mémoire.

- On peut définir des entiers non signés, c'est-à-dire compris entre 0 et $2^N - 1$ en remplacement de $[-2^{N-1}; 2^{N-1} - 1]$. Le type est `np.uint`
- On peut définir le nombre de bits des entiers : `np.int8`, `np.int16`, `np.int32`, `np.int64`, `np.uint8`, `np.uint16`, `np.uint32`, `np.uint64`, `np.float32`, `np.float64`.

```
>>> np.array((-2,5,-3,9,7),dtype=np.uint8)
array([254,   5, 253,   9,   7], dtype=uint8)
```

3. `np.arange(a,b,c)` crée un tableau de nombres de la forme $x = a + kc$ avec $a \leq x < b$ et k entier positif. Si a, b et c sont entiers on obtient le même résultat qu'avec `np.array(range(a,b,c))` avec a et c toujours optionnels.
   Le résultat est un tableau de flottants sauf si a, b et c sont entiers.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(0.8,5.3,4/3)
array([ 0.8       ,  2.13333333,  3.46666667,  4.8       ])
```

4. `np.linspace(x,y,d)` crée un tableau de d réels commençant à x et finissant à y.

```
>>> np.linspace(0,0.42,4)
array([ 0.  ,  0.14,  0.28,  0.42])
```

   C'est l'outil utilisé pour obtenir un tableau d'abscisses.

5. On peut créer des tableaux de 0 avec `np.zeros` :

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

6. `np.copy(a)` crée une copie de `a`, sans lien avec `a`. Si on change un terme de `a` le changement n'est pas reproduit dans la copie.

2.2 Comment sauvegarder des tableaux ?

`numpy` permet de sauvegarder et charger des tableaux directement sous forme d'un fichier texte lisible par d'autres applications.

- `np.savetxt('essai.npy',T)` écrit le tableau `T` sous le nom `essai.npy` dans le répertoire courant. Le suffixe `npy` ne sert qu'à indiquer la nature du fichier.
  On peut aussi donner le chemin d'accès complet vers le lieu de sauvegarde.
- `T = np.loadtxt('/home/ericd13/Documents/2015-2016/How To/essai.npy')` charge le fichier sous la forme d'un tableau sous le nom `T`. Ici le chemin complet (unix) a été indiqué.

3 Comment calculer ?

Le principal outil ici est le module `scipy` qui contient de nombreuses fonctions implémentant de manière efficace des algorithmes vus en cours.

3.1 Comment résoudre une équation ?

Une fonction généraliste est `fsolve`. Elle prend deux paramètres : une fonction et une valeur initiale, si possible proche de la solution recherchée.

```
from scipy.optimize import fsolve

def f(x):
    return x*x - 2

>>> fsolve(f,1)
array([ 1.41421356])
```

On remarquera que le résultat est un tableau `numpy`. En effet on peut trouver les zéros d'une fonction de p variables à valeurs dans $\mathbb{R}^p$. `fsolve` convertira les listes et les tuples en tableaux `numpy`.

```
def f(u):
    (x,y) = u
    return x*y + 1, x+y - 2

>>> fsolve(f,(1,1))
array([-0.41421356,  2.41421356])
```

Si on veut plus de contrôle sur la racine que l'on recherche on peut utiliser une méthode de dichotomie : `bisect`.
D'après la documentation : *Basic bisection routine to find a zero of the function f between the arguments a and b. f(a) and f(b) cannot have the same signs. Slow but sure.*

```
from scipy.optimize import bisect

def f(x):
    return np.tan(x)-x

>>> bisect(f,3,4.7)
4.493409457909229
```

3.2 Comment calculer une intégrale ?

Le calcul d'intégrale se fait avec `scipy.integrate.quad`. Le résultat est un couple dont le premier élément est une valeur approchée de l'intégrale et le second un majorant de l'erreur.

```
from scipy.integrate import quad

def f(x):
    return np.exp(-x*x/2)

>>> quad(f,0,1)
(0.855624391892149, 9.499399003095619e-15)
```

On peut donner des bornes infinies

```
>>> quad(f,0,np.inf)
(1.2533141373154997, 1.2756471096158012e-08)

>>> np.sqrt(np.pi/2)
1.2533141373155001
```

3.3 Comment résoudre une équation différentielle ?

La résolution d'équations différentielles se fait avec `scipy.integrate.odeint`.
Elle prend (au moins) 3 paramètres.

- La fonction `phi` qui définit l'équation différentielle $y' = \varphi(y,t)$. On notera que la variable est placée avant le temps. `y` peut être un vecteur, dans ce cas la fonction doit renvoyer un vecteur de même taille.
- Une valeur initiale qui doit être un vecteur de même taille que `y` dans `phi`.
- Un tableau des temps qui indique à quelles valeurs on veut calculer une valeur approchée de la solution.

```
from scipy.integrate import odeint

def phi(y,t):
    return y/(1+t*t)

T = np.linspace(0,10,1000)
Y1 = odeint(phi,1,T)

def phi(y,t):
    (u,v) = y
    return (-v,u)

T = np.linspace(0,10,1000)
Y2 = odeint(phi,(1,0),T)
```

3.4 Comment déterminer un polynôme d'interpolation ?

On peut utiliser une fonction de scipy.

```
from scipy.interpolate import BarycentricInterpolator as
   Lagrange
```

La fonction attend les deux listes des valeurs $x_0, x_1, \ldots, x_n$ et $y_0, y_1, \ldots, y_n$ et retourne le polynôme P de degré n au plus tel que $P(x_i) = y_i$ sous la forme d'une fonction (les coefficients ne sont pas calculés).

```
>>> f = Lagrange([0,1,2,3],[1,0.5,1/3,0.25])

>>> f(4)
array(-0.0)
```

3.5 Comment calculer une droite de régression ?

On cherche l'équation de la droite, de la forme $y = ax + b$, qui approche le mieux un nuage de n points (x_i, y_i). L'écart est mesuré par $\sum_{i=1}^{n} (y_i - ax_i - b)^2$, c'est l'écart quadratique.

Les points sont définis par les deux listes des abscisses, `X`, et des ordonnées, `Y`.
La fonction `scipy.stats.linregress(X,Y)` renvoie 5 paramètres :

- la pente, notée a ci-dessus,
- l'abscisse à l'origine, notée b ci-dessus,
- le coefficient de corrélation,
- deux indicateurs statistiques

```
from scipy.stats import linregress

X = [1, 2, 3, 4, 5, 6]    # y varie autour de 2x+1
Y = [3.05, 4.98, 7.01, 8.96, 10.87, 13.11]

>>> linregress(X,Y)
(1.997714285714286, 1.0046666666666653, 0.99976066827094812,
    8.5912660374434614e-08, 0.021857298474381153)
```

4 Comment visualiser des fonctions scalaires ?

Quelques tracés utiles.
Ils utilisent, en plus de `numpy`, la sous-bibliothèque `pyplot` de `matplotlib`.
Une belle galerie d'exemples se trouve à `http://matplotlib.org/gallery.html`
Le module est appelé par

```
import matplotlib.pyplot as plt
```

Entre deux tracés on efface la fenêtre graphique avec `plt.clf()`.

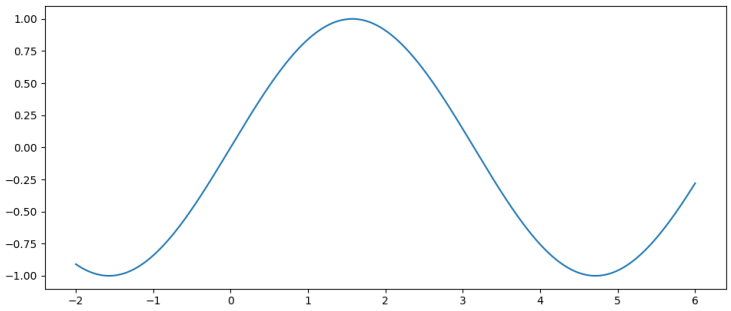
Il existe deux modes de tracé.

- En mode interactif les tracés sont immédiatement effectués.
- En mode non interactif les tracés ne sont réalisés qu'après l'appel de la fonction `plt.show()`.
- On passe en mode interactif avec `plt.ion()` on en sort par `plt.ioff()`.

4.1 Comment tracer le graphe d'une fonction réelle ?

1. On définit la fonction (`def f(x)`) en utilisant les fonctions `numpy`.
2. On définit le tableau des abscisses à l'aide des fonctions `arange` si on définit le pas entre deux points ou `linspace` si on définit le nombre de points.
3. On définit la tableau des ordonnées, `Y = f(X)`.
4. On dessine : `plt.plot(X,Y)`.

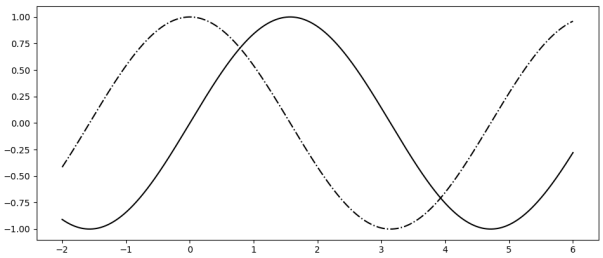
```
X=np.linspace
    (-2,6,1000)
Y=np.sin(X)
plt.clf()
plt.plot(X,Y)
plt.show()
```



On peut changer le style et la couleur des lignes en ajoutant des paramètres optionnels.

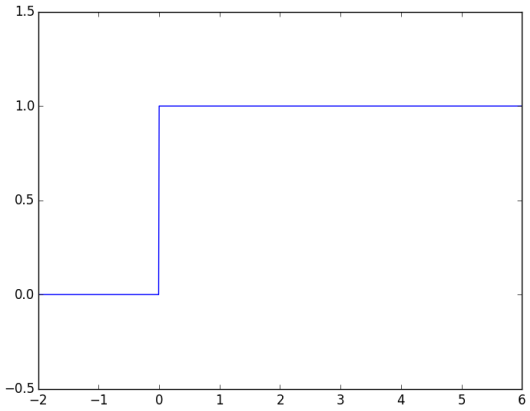
- `color = 'red'` pour une ligne rouge, `'blue'` pour bleue, `'green'` pour verte, `'black'` pour noire, `'cyan'` pour cyan, `'magenta'` pour magenta, `'yellow'` pour jaune
- `linestyle = 'solid'` pour une ligne classique (valeur par défaut), `'dashed'` pour des tirets, `'dashdot'` pour des successions de tirets et points, `'dotted'` pour des pointillés

```
X=np.linspace(-2,6,1000)
Y=np.sin(X)
Y1 = np.cos(X)
plt.clf()
plt.plot(X,Y,color='black')
plt.plot(X,Y1,color='black',
    linestyle='dashdot')
plt.show()
```



Quand la fonction comporte des tests elle n'est pas utilisable directement : il faut la rendre vectorielle à l'aide de la fonction `vectorize`.

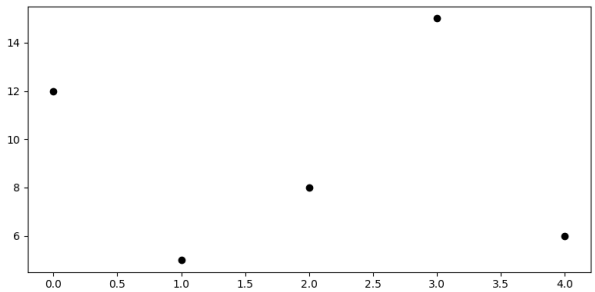
```
def f(x):
    if x < 0:
        return 0
    else:
        return 1

heaviside = np.vectorize(f)

X = np.linspace(-2,6,1000)
Y = heaviside(X)
plt.clf()
plt.plot(X,Y)
plt.ylim([-0.5,1.5])
# Pour agrandir verticalement
plt.show()
```



4.2 Comment représenter un ensemble de valeurs ?

Si `X` est un tableau de nombre, `plt.plot` trace les points de coordonnées `(i, X[i])` en les reliant par des segment de droite.
On peut indiquer les points avec le paramètre optionnel `marker = 'o'` qui dessine un point, d'autres dessins possibles sont `'D'`, `'+'`, `'.'`, `'s'`, `'*'`, `'x'` ...
`linestyle = ''` supprime le tracé des lignes.

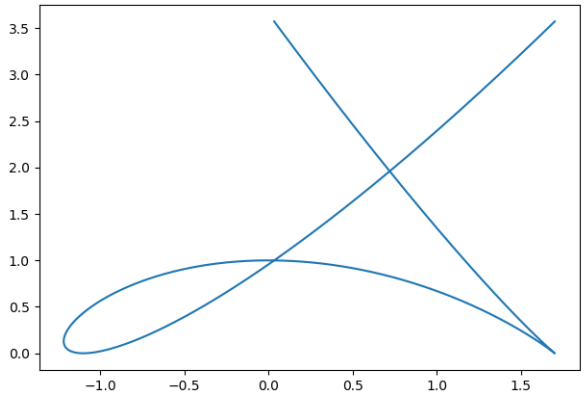
```
X=[12, 5, 8, 15, 6]
plt.clf()
plt.plot(X,marker='o',linestyle
    ='',color='black')
plt.show()
```



4.3 Comment tracer un arc paramétré ?

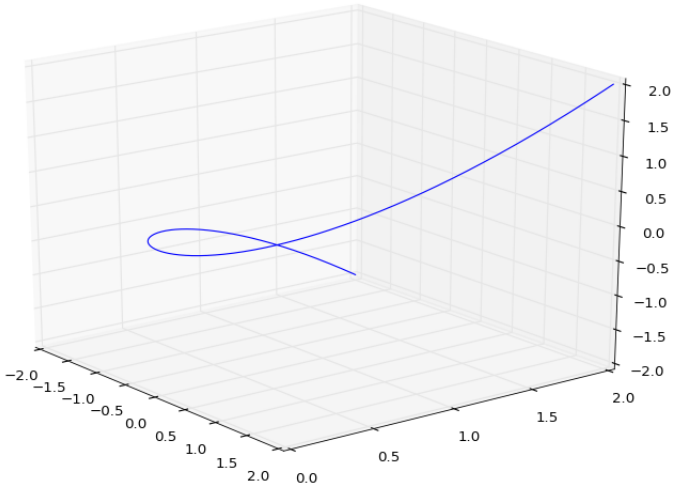
- On définit les fonctions `def x(t)` et `def y(t)` en utilisant les fonctions `numpy`.
- On définit le tableau des temps, `T`, où l'on veut tracer l'arc : `arange` ou `linspace`.
- On définit la tableau des abscisses, `X = x(T)`.
- On définit la tableau des ordonnées, `Y = y(T)`.
- On dessine : `plt.plot(X,Y)`.

```
def x(t):
    return t*(-2.4+t*(0.3+t))

def y(t):
    return (t**2-1)**2

T = np.linspace(-1.7,1.7,1000)
X = x(T)
Y = y(T)
plt.clf()
plt.plot(X,Y)
plt.show()
```



Un arc de $\mathbb{R}^3$ peut aussi être représenté.
On a besoin de la fonction `Axes3D` de la bibliothèque `mpl_toolkits.mplot3d`.
On définit un cadre de dessin 3D avec `Axes3D(plt.figure())` dans lequel on dessine.

```
from mpl_toolkits.mplot3d
    import Axes3D

dessin = Axes3D(plt.figure())

T = np.linspace(-2,2,500)
X = T
Y= T*T/2
Z = T*T*T/4
dessin.plot(X,Y,Z)
plt.show()
```



Le cadre est manipulable à la souris.

4.4 Comment voir les variations d'un paramètre ?

On a parfois besoin de visualiser les graphes d'une fonction avec un paramètre variable.

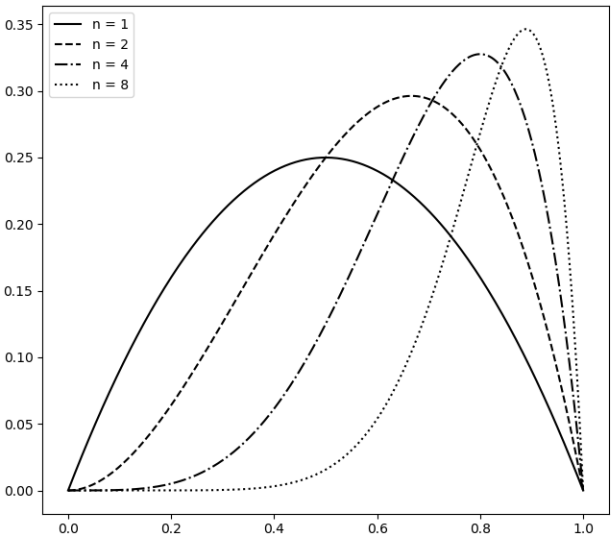
1. On peut dessiner plusieurs graphe dans le même repère. Il est alors utile de donner une légende à chacun.

   Le texte de la légende d'un graphe est créé avec le graphe en ajoutant la variable nommée `label`.

   Les légendes sont ajoutées par la commande `legend` ; on peut les placer sur le haut ou le bas (`upper` ou `lower`) et à gauche, au centre ou à droite (`left`, `center` ou `right`) avec la variable nommée `pos`.

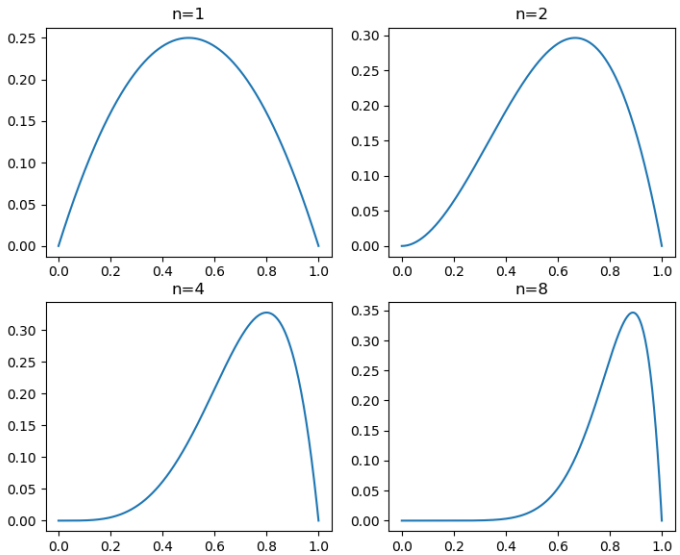
   ```
   def f(x,n):
       return n*x**n*(1-x)

   X = np.linspace(0,1,300)
   motif = ['solid', 'dashed', 'dashdot', 'dotted']
   N = [1, 2, 4, 8]

   plt.clf()
   for i in range(3):
   ```

```
    n = N[i]
    plt.plot(X,f(X,N), color="black", linestyle=motif[i],
        label="n = {}".format(n))
plt.legend(pos="upper right")
```



2. On peut dessiner plusieurs graphes juxtaposés. Ils seront représentés simultanément sous formes de sous-graphes avec l'instruction `subplot`. On divise le cadre en p lignes et q colonnes ce qui permet de dessiner $p.q$ graphes. Ils seront numérotés de 1 à $p.q$ [2].
   Chaque tracé doit être précédé de la définition du sous-graphe : `plt.subplot(p,q,i)`.

```
def f(x,n):
    return n*x**n*(1-x)

X = np.linspace(0,1,300)

plt.clf()
plt.subplot(2,2,1)
plt.plot(X,f(X,1)
plt.subplot(2,2,2)
plt.plot(X,f(X,1)
plt.subplot(2,2,3)
plt.plot(X,f(X,4)
plt.subplot(2,2,4)
plt.plot(X,f(X,8)
```



3. Il existe une fonction, indépendante de l'interface graphique, qui permet de faire varier interactivement un paramètre. Cela se fait sous la forme d'un curseur. Il y a de nombreuses étapes car on modifie la structure interne de la représentation.

2. Et non de 0 a $p.q - 1$ selon l'usage classique de Python.

```
1  from matplotlib.widgets import Slider
2
3  n0 = 3
4  Y = f(X,n0)
5  plt.clf()
6  gr = plt.plot(X,Y)[0]
7  plt.ylim([0,0.5])
8
9  plt.subplots_adjust(top=0.8)
10 placeCurseur = plt.axes((0.15, 0.88, 0.7, 0.04))
11 curseur = Slider(placeCurseur, 'n', 1, 20, valinit=n0)
12
13 def modifier(n):
14     Y = f(X,n)
15     gr.set_ydata(Y)
16
17 curseur.on_changed(modifier)
```

(a) On importe la fonction qui définit le curseur (ligne 1).

(b) On commence par créer un graphe avec une valeur du paramètre par défaut en affectant une variable avec le graphe.

La ligne 3 efface l'éventuel graphe qui existerait.

Le `[0]` à la ligne 6 sert à extraire la première composante du résultat de `plot`.

La ligne 7 ajuste les limites verticales.

(c) On réserve une place pour le curseur dans le graphe. Pour cela on diminue la proportion du graphe dans la fenêtre à la ligne 9 : `top=0.8` signifie que le haut du graphe est à 80% du cadre, par défaut il est à 90%. On peut aussi réserver de la place en bas, (`bottom=0.2`).

(d) On définit le rectangle du curseur à la ligne 10.

L'ensemble de 4 paramètres (noter qu'ils ne forment qu'une variable pour la fonction d'où les doubles parenthèses) signifient ici

le rectangle part à 15% de la gauche et à 88% du bas du cadre

sa largeur est de 70% de celle du cadre et sa hauteur de 4% de la hauteur du cadre.

Tout reste proportionné si on change la taille de la fenêtre.

(e) On place le curseur dans le rectangle à la ligne 11.

La fonction `Slider` admet 4 paramètres plus un paramètre nommé.

i. Le premier est le rectangle défini auparavant

ii. `'n'` est l'intitulé qui sera écrit à gauche du curseur

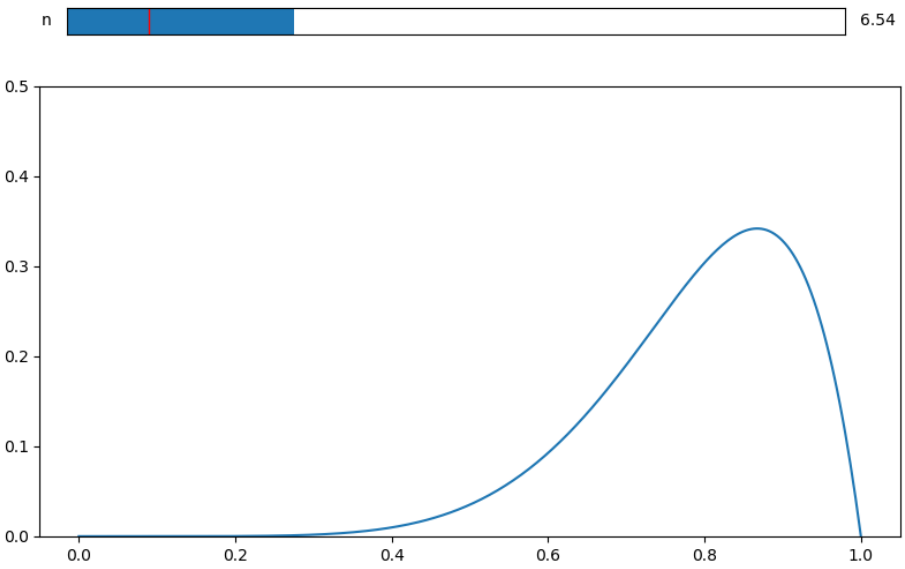
iii. `1` et `20` sont les bornes entre lesquelles varie le curseur

iv. `valinit=n0` permet de donner une valeur initiale, il est utile de donner celle du graphe initial.

(f) Il faut maintenant mettre à jour le graphe quand la valeur du curseur déterminée à la souris change.

On commence par définir une fonction de changement qui reçoit la valeur renvoyée par le curseur (lignes 13 à 15). `gr.set_ydata` modifie l'ensemble des ordonnées du graphe (`gr` ici).

Cette fonction est appelée à chaque changement de valeur du paramètre en l'associant au curseur par la méthode `on_changed` (ligne 17).



5 Comment visualiser des fonctions de deux variables ?

Quand l'ensemble des valeurs dans lesquelles on calcule une fonction est un rectangle on a besoin de tableaux à 2 dimensions pour chacune des deux coordonnées. La fonction `meshgrid` de `numpy` calcule ces tableaux.
Elle prend deux listes pour arguments, la première décrit les valeurs de x (de gauche à droite), la seconde décrit les valeurs de y **du haut vers le bas**.

```
>>> X,Y = np.meshgrid([-1,0,1,2],[-3,-2,-1,0])
>>> X
array([[-1,  0,  1,  2],
       [-1,  0,  1,  2],
       [-1,  0,  1,  2],
       [-1,  0,  1,  2]])
>>> Y
array([[-3, -3, -3, -3],
       [-2, -2, -2, -2],
       [-1, -1, -1, -1],
       [ 0,  0,  0,  0]])
```

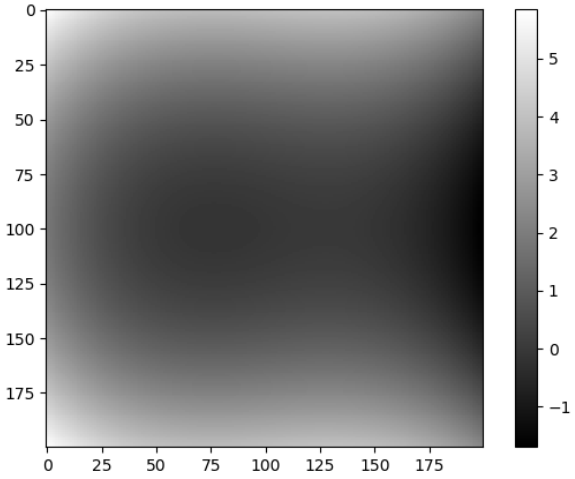
Les valeurs d'une fonction $f(x, y)$ sont données par `Z = f(X,Y)`.
Nous utiliserons les valeurs suivantes

```
X1 = np.linspace(-1,1.6,200)
Y1 = np.linspace(-2,2,200)
X,Y = np.meshgrid(X1,Y1)
Z = X*X+Y*Y-X*X*X-4/27
```

5.1 Comment représenter les valeurs de $f(x, y)$ selon une echelle ?

La matrice des valeurs est représentable comme une image monochrome. On peut donc la dessiner (`plt.imshow`) en choisissant l'échelle des couleurs (`cmap = 'gray'`). Les échelles possibles sont décrites sur la page `http://matplotlib.org/examples/color/colormaps_reference.html`

```
plt.imshow(Z,cmap='gray')
plt.colorbar()
plt.show()
```

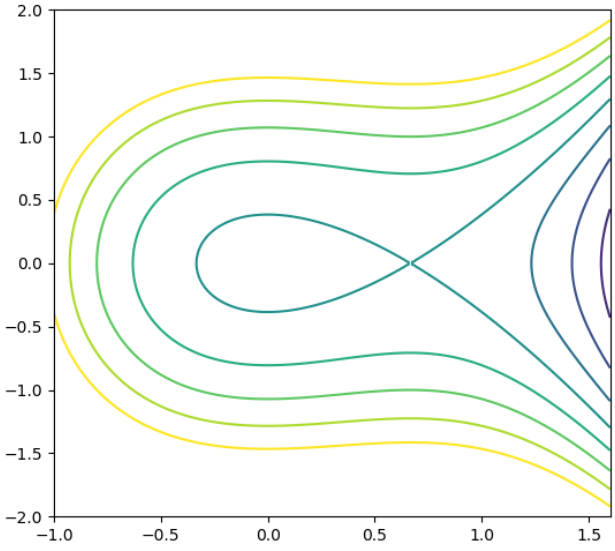


5.2 Comment tracer une ligne de niveau ?

Il peut être plus expressif de tracer les courbes de niveau.

La fonction `contour` de `matplotlib` attend les 3 matrices de coordonnées et la liste des valeurs de niveaux pour paramètres.

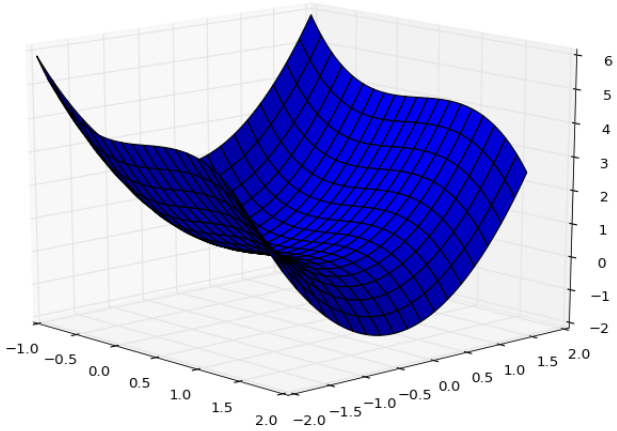
```
niveaux = np.linspace(-2,2,9)
plt.contour(X,Y,Z,niveaux,color
    ='black')
plt.show()
```



5.3 Comment tracer une surface ?

On peut tracer une surface dans un cadre 3D.

```
from mpl_toolkits.mplot3d
    import Axes3D

dessin = Axes3D(plt.figure())
dessin.plot_surface(X,Y,Z)
plt.show()
```



6 Comment faire des calculs linéaires ?

On a vu que les tableaux `numpy` permettaient le calcul dans un espace vectoriel.
Les tableaux à deux dimensions sont aussi utilisables en tant que matrice.

1. Le produit de deux éléments qui ont des dimensions cohérentes se fait avec `np.dot`.

```
U = np.array([1,2])
A = np.array([[1,1] , [-1,2]])
B = np.array([[1,-1] , [2,0] , [-2,3]])

>>> np.dot(A,U)
array([3, 3])

>>> np.dot(U,A)
array([-1,  5])

>>> np.dot(U,U)
5

>>> np.dot(B,A)
array([[ 2, -1],
       [ 2,  2],
       [-5,  4]])

>>> np.dot(A,B)
ValueError: shapes (2,2) and (3,2) not aligned: 2 (dim 1) != 3
    (dim 0)
```

   On voit que le traitement des vecteurs est assez souple : ils sont considérés comme de taille $1 \times n$ ou $n \times 1$ selon les besoins.

2. Le produit vectoriel est défini par `cross`

```
>>> U1 = (1,2,1)
>>> U2 = np.array([-1,1,3])
>>> np.cross(U1,U2)
array([ 5, -4,  3])
```

   Python convertit en tableaux `numpy` les assemblages sans qu'on ait besoin de lui dire.

3. Quelques fonctions de création, noter que le paramètre de taille doit être un tuple.

```
>>> np.zeros((2,4))
```

```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones((3,2))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.diag([1, 2, 1])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 1]])
```

4. Quelques fonctions simples.

```
>>> np.shape(B)
(3, 2)
>>> np.trace(A)
3
>>> np.transpose(B)
array([[ 1,  2, -2],
       [-1,  0,  3]])
```

5. On peut réarranger la taille d'une matrice, en particulier pour l'aplatir.

```
>>> np.reshape(B,(2,3))
array([[ 1, -1,  2],
       [ 0, -2,  3]])
>>> np.reshape(A,4)
array([ 1,  1, -1,  2])
```

6. `numpy` contient une sous-bibliothèque, `linalg`, qui réalise les opérations de l'algèbre linéaire.

```
>>> import numpy.linalg as al
```

`help(al)` donne toutes les fonctions du module.

```
>>> al.det(A)
3.0000000000000004
>>> al.matrix_power(A,3)
array([[-3,  6],
       [-6,  3]])
>>> al.inv(A)
array([[ 0.66666667, -0.33333333],
       [ 0.33333333,  0.33333333]])
```

En particulier la résolution de systèmes s'écrit `al.solve`. Il ne donne la solution que dans le cas d'un système de Cramer.

```
>>> C = (3,3)
>>> al.solve(A,C)
array([ 1.,  2.])
```

On remarquera enore que `C` a été converti silencieusement en format `array`.

7 Comment faire des calculs polynomiaux ?

`numpy` contient une sous-bibliothèque, `polynomial`, qui définit des objets représentant les polynômes.

```
>>> from numpy.polynomial import Polynomial
```

1. On définit un polynôme à l'aide de la liste de ses coefficients par ordre de degré croissant.
   On représente $P = 2X^2 - 3X + 1$ et $Q = X^3 - X$.

   ```
   >>> P = Polynomial((1,-3,2))
   >>> Q = Polynomial([0,-1,0,1])
   ```

   L'assemblage entré comme paramètre peut être une liste, un tuple, un tableau `numpy`, ...
2. L'affichage montre une structure composée

   ```
   >>> P
   Polynomial([ 1., -3.,  2.], [-1,  1], [-1,  1])
   ```

   Pour obtenir les coefficients on utilise la méthode `coef`

   ```
   >>> Q.coef
   array([ 0., -1.,  0.,  1.])
   ```

3. L'évaluation est directe

   ```
   >>> P(2)
   3.0

   >>> Q(3)
   24.0
   ```

4. Les calculs algébriques sont fait par les opérations usuelles.
   Pour la division euclidienne le quotient est calculé par `//`, le reste par `%`.

   ```
   >>> A = P//Q
   >>> B=P%Q
   >>> B.coef
   array([ 1., -3.,  2.])
   >>> P == Q*A+B
   True
   ```

5. Les caractéristiques d'un polynôme sont accessibles par des méthodes.

   ```
   >>> P.degree()         # Degré
   2
   >>> Q.roots()           # Liste des racines
   array([-1.,  0.,  1.])
   >>> P.deriv().coef      # Dérivée
   array([-3.,  4.])
   >>> Q.deriv(2).coef     # On dérive 2 fois
   array([ 0.,  6.])
   ```

8 Comment simuler le hasard ?

`numpy` contient une bibliothèque, `numpy.random`, qui reproduit les fonctions le la bibliothèque standard `random` en ajoutant la possibilité de construire des tableaux de nombres aléatoires directement.

```
>>> import numpy.random as rd
```

On crée des tableaux de réels compris entre 0 et par `rand(n)` `rand((p,q))` et des tableaux d'entiers dans l'intervalle [3] $[a; b-1]$ par `randint(a,b,n)` ou `randint(a,b,(p,q))`.
Si le paramètre de taille n'est pas donné la fonction renvoie une valeur unique.

```
>>> rd.rand(5)
array([ 0.60545539,  0.30610473,  0.55519555,  0.50989005,
   0.4193768 ])
>>> rd.rand(3,3)
array([[ 0.14495267,  0.26517115,  0.59222602],
       [ 0.7433966 ,  0.67095325,  0.23064075],
       [ 0.47085792,  0.35824555,  0.1192762 ]])
>>> rd.randint(1,7,10)
array([5, 1, 3, 1, 1, 1, 3, 6, 3, 2])
>>> rd.randint(1,4,(3,3))
array([[3, 1, 1],
       [1, 2, 2],
       [1, 1, 3]])
```

On peut aussi simuler les lois usuelles, `taille` est le nombre de tirages.

1. La loi binômiale $\mathcal{B}(n,p)$ est simulée par `binomial(n,p,taille)`.
2. La loi de Bernoulli $\mathcal{B}(p)$ est simulée par `binomial(1,p,taille)`.
3. La loi de Poisson $\mathcal{P}(\lambda)$ est simulée par `poisson(lambda,taille)`.
4. La loi géométrique $\mathcal{G}(p)$ est simulée par `geometric(p,taille)`.

3. Les bornes sont différentes que dans la fonction `random.randint`.