1 Tracés avec pyplot

1.1 Tracé de courbes

matplotlib.pyplot.plot(...) : fonction de tracé proprement dite, qui prend un itérable d'abscisses (liste, vecteur, etc.), un itérable d'ordonnées et éventuellement une chaîne de caractères indiquant une couleur et/ou une forme ("r" pour du rouge, "bo" pour des ronds bleus, "g+" pour des croix vertes, etc.). On peut éventuellement préciser une étiquette (label=...) pour la légende.

Il est possible de ne pas préciser d'abscisses, et de ne mettre que les ordonnées, auquel cas les abscisses seront des indices indiquant la position des valeurs dans l'itérable. Il est également possible d'effectuer plusieurs tracés dans une même instruction plot, mais si on ne précise pas les abscisses, la chaîne indiquant le format de chaque courbe est indispensable pour séparer les différentes courbes.

matplotlib.pyplot.show() : Provoque l'ouverture de la fenêtre lorsque l'on ne se trouve pas en mode interactif.

matplotlib.pyplot.xlim(chaine) : Précise les valeurs limites pour l'axe des abscisses.

matplotlib.pyplot.ylim(chaine) : Précise les valeurs limites pour l'axe des ordonnées.

matplotlib.pyplot.title(chaine) : Ajoute un titre au graphique.

matplotlib.pyplot.xlabel(chaine) : Ajoute une légende sur l'axe des abscisses.

matplotlib.pyplot.ylabel(chaine) : Ajoute une légende sur l'axe des ordonnées.

matplotlib.pyplot.legende(...) : Ajoute une légende sur le graphique. Les options sont nombreuses, mais l'essentiel est automatique si on a précisé une étiquette pour chacun des tracés.

matplotlib.pyplot.semilogx(...) : fonction de façon similaire à plot, mais donne un graphe avec une échelle logarithmique en abscisse.

matplotlib.pyplot.semilogy(...) : fonction de façon similaire à plot, mais donne un graphe avec une échelle logarithmique en ordonnée.

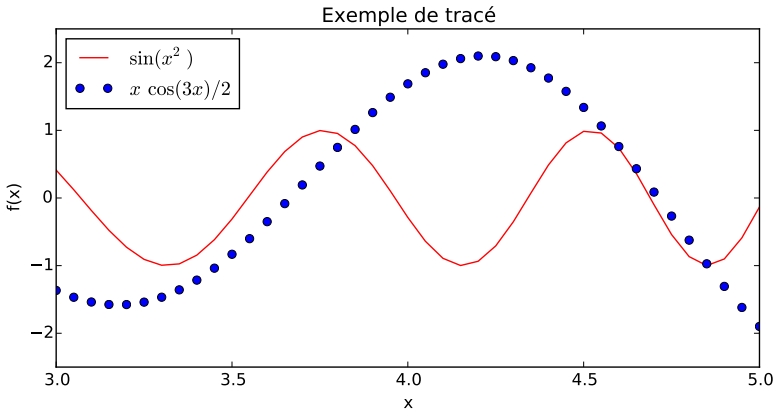
matplotlib.pyplot.loglog(...) : fonction de façon similaire à plot, mais donne un graphe avec une échelle logarithmique sur les deux axes.

matplotlib.pyplot.bar(...) : fonction de façon similaire à plot, mais donne un histogramme plutôt qu'une courbe.

Pour tous les textes, **matplotlib** accepte la syntaxe LaTeX, utile notamment pour afficher des mathématiques.

Un exemple de tracé de $\sin(x^2)$ et $x\cos(3x)/2$ sur $[3,5]$ (subdivisé en 40) avec **matplotlib.pyplot** (sans **numpy**) :

```
import matplotlib.pyplot as plt

X = [ 3.0 + 0.05*x for x in range(41) ]
Y1 = [ sin(x**2) for x in X ]
Y2 = [ x*cos(3*x)/2 for x in X ]

plt.plot(X, Y1, 'r', label="$\\sin(x^2)$")
plt.plot(X, Y2, 'bo', label="$x \\, \\cos(3x)/2$")
plt.xlim(3.0, 5.0) # (facultatif ici)
plt.ylim(-2.5, 2.5)
plt.title("Exemple de tracé")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend(loc="upper left")
plt.show()
```



1.2 Utilisation avec numpy

Les trois instructions construisant les listes peuvent être simplifiées [1] avec **numpy** :

```
X = numpy.linspace(3.0, 5.0, 41)
Y1 = numpy.sin(X**2)
Y2 = X * numpy.cos(3*X) / 2
```

1. Seules les fonctions « vectorisées » peuvent être utilisées avec un **numpy**.array comme argument, comme sur l'exemple, il faut donc que les fonctions sin et cos viennent du module **numpy** et non du module **math** pour que cela fonctionne. On peut cependant construire une fonction « vectorisée » fv avec n'importe quelle fonction f en écrivant fv = **numpy**.vectorize(f).

1.3 Tracés 3D

Les possibilités pour obtenir des tracés en 3D sont nombreuses ! On ne présente ici que quelques exemples utiles, on se reportera à la documentation pour les détails.

Pour une courbe paramétrée $x(t), y(t), z(t)$:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

T = [ 0.01*i for i in range(629) ]
X = [ sin(2*t) for t in T ]
Y = [ cos(3*t) for t in T ]
Z = [ cos(t)*sin(2*t) for t in T ]
plt.axes(projection='3d').plot(X, Y, Z)
```

Pour une surface en 3D :

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

X = [ -1.0 + 0.05*i for i in range(41) ]
Y = [ -1.0 + 0.05*i for i in range(41) ]
X, Y = np.meshgrid(X, Y)
Z = [ x**2-y**2 for x, y in zip(X, Y) ]
plt.axes(projection='3d').plot_surface(X, Y, Z, cstride=4, rstride=4)
```

Pour un nuage de points en 3D :

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

X = [ np.random.random() for i in range(100) ]
Y = [ np.random.random() for i in range(100) ]
Z = [ sin(4*x)-cos(5*y) for x, y in zip(X, Y) ]
plt.axes(projection='3d').scatter(X, Y, Z)
```

1.4 Images

`matplotlib.pyplot.matshow(table)` : fonction prenant en argument un tableau à deux dimensions (ou plus) et traçant une grille 2D dont les couleurs sont liées aux valeurs du tableau.

`matplotlib.pyplot.imshow(table)` : même chose que `matshow`, mais avec une interpolation entre les différents points, ce qui est plus adapté aux images, ou aux fonctions continues qui ont été discrétisées sur une grille.

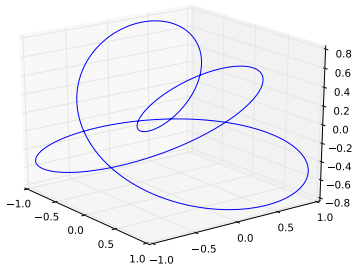
Pour afficher une carte d'intensité :

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

def foo(i, j) :
    return np.sin(0.04*i) - np.cos(0.05*j)

Z = np.fromfunction(foo, (100, 100))
plt.imshow(Z)
```

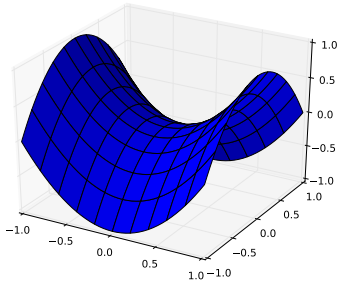
Pour afficher une image chargée en mémoire avec `imageio` par exemple :

```
import matplotlib.pyplot as plt
import imageio

image = imageio.load("image.jpg")
plt.imshow(image)
plt.show()
```

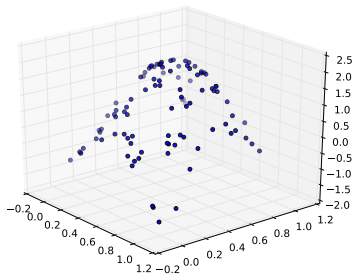
Courbe 3D paramétrique :



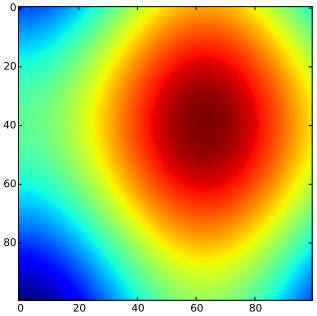
Surface 3D :



Nuage de points 3D :



Carte d'intensité :



2 Calcul numérique avec scipy

2.1 Calcul intégral

`scipy.integrate.quad(f, a, b)` : retourne $\int_a^b f(u)\,\mathrm{d}u$.

Le résultat est en fait un tuple, le second élément indiquant une estimation de l'erreur commise (qu'il est recommandé de contrôler).

```
In []: import numpy as np

In []: import scipy.integrate as integr

In []: def f(x) :
  ...:     return (np.sin(x)/x)**2

In []: integr.quad(f, 0.0, 1.0)
Out[]: (0.8973395585291236, 9.962470387860063e-15)
```

On peut désigner $-\infty$ et $+\infty$ avec `-np.inf` et `np.inf` :

```
In []: def g(x) :
  ...:     return np.exp(-2.0*x)

In []: integr.quad(f, 0.0, 1.0)
Out[]: (0.5, 7.735031684717554e-11)
```

Si les données se trouvent sous la forme d'une liste ou d'un tableau de valeurs (ordonnées), on peut utiliser la fonction `scipy.integrate.simps` (attention à l'ordre des paramètres !) qui calcule l'intégrale par la méthode de Simpson :

```
In []: X = [ 0.01*i for i in range(101) ]

In []: Y = [ 1.0 if x==0.0 else (np.sin(x)/x)**2 for x in X ]

In []: integr.simps(Y, X)
Out[]: 0.89733955856968928
```

La fonction `scipy.integrate.trapz` fonctionne exactement comme la précédente, mais avec la méthode des trapèzes.

```
In []: integr.trapz(Y, X)
Out[]: 0.89733533477390148
```

2.2 Résolution d'équations différentielles

`scipy.integrate.odeint(f, y0, t)` : retourne, pour chaque instant t_k dans l'itérable `t`, un ensemble de y_k, approximant la fonction $y(t)$ solution de l'équation $y' = f(y,t)$ pour la condition initiale $y(t_0) = y_0$.

Par exemple, pour $y'(t) = \sin(x) \times t$ sur l'intervale $[0,5]$ en partant de $y(0) = 1$:

```
In []: import numpy as np

In []: import scipy.integrate as integr

In []: def f(y, t) :
  ...:     return y*sin(t)

In []: T = np.linspace(0.0, 5.0, 50)

In []: Y = integr.odeint(f, 1.0, T)
```

La même fonction `odeint` permet de résoudre des systèmes d'équation différentielles, si `f` prend pour premier élément un itérable et retourne un itérable de même longueur, et que $y0$ est également un itérable.

Par exemple, le système de Lorenz est défini par $\begin{cases} y_0'(t) = 10(y_1(t) - y_0(t)) \\ y_1'(t) = 28y_0(t) - y_1(t) - y_0(t)y_2(t) \\ y_2'(t) = y_0(t)y_1(t) - 8y_2(t)/3 \end{cases}$

Pour le résoudre pour $t \in [0,50]$ en partant de $y_0(0) = 0.5$, $y_1(0) = 0.2$ et $y_2(0) = 0$:

```
In []: def f(Y, t) :
  ...:     return 10*(Y[1]-Y[0]), 28*Y[0]-Y[1]-Y[0]*Y[2],
                          Y[0]*Y[1]-8*Y[2]/3

In []: T = numpy.linspace(0.0, 50.0)

In []: res = integr.odeint(f, [0.5, 0.2, 0.0], T)
```

$y_0(t)$ est obtenu en prenant la première colonne du résultat `res[:,0]`. De la même façon, `res[:,1]` et `res[:,2]` donnent $y_1(t)$ et $y_2(t)$.

Pour les équations différentielles d'ordre 2 (ou plus), on les réécrit comme des systèmes du premier ordre. Ainsi, pour résoudre l'équation différentielle du second ordre à une dimension $y'' = f(y', y, t)$, on résoud l'équation différentielle du premier ordre à deux dimensions

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} y' \\ f(y', y, t) \end{pmatrix}$$

La condition initiale devenant ici, naturellement, le vecteur $\begin{pmatrix} y(t_0) \\ y'(t_0) \end{pmatrix}$.

2.3 Recherche de racines

Pour les fonctions à une variable, les possibilités sont nombreuses :

`scipy.optimize.fsolve(f, x0, fprime=Nonextol=1.49012e-08, ...)` : recherche et retourne une solution à l'équation $f(x) = 0$ en partant de l'estimation $x0$. Il est possible de préciser une dérivée avec le paramètre `fprime` et une précision via le paramètre `xtol`.

`scipy.optimize.root(f, x0, jac=None, tol=None, ...)` : l'utilisation est la même que `fsolve`, seule la méthode change (plusieurs méthodes peuvent en fait être utilisées), et le résultat qui est présenté sous la forme d'un dictionnaire. Le paramètre contenant la dérivée se nomme ici `jac`.

`scipy.optimize.newton(f, x0, fprime=None, tol=1.48e-8, ...)` : recherche d'une solution à l'équation $f(x) = 0$ par la méthode de Newton (si `fprime` est précisé) ou par la méthode de la sécante.

`scipy.optimize.bissect(f, a, b, xtol=1e-12, ...)` : recherche d'une solution à l'équation $f(x) = 0$ dans $[a, b]$ par la méthode de la bissection.

Par exemple, pour trouver la racine strictement positive de $\sin(x) = x/2$:

```
In [1]: import scipy.optimize as opt

In [2]: def f(x) :
   ...:     return sin(x)-0.5*x
```

```
In [3]: opt.fsolve(f, 1.0)
Out[3]: array([ 1.89549427])
```

```
In [4]: sol = opt.root(f, 1.0)

In [5]: sol.success
Out[5]: True

In [6]: sol.x
Out[6]: array([ 1.89549427])
```

```
In [7]: opt.newton(f, 1.0)
Out[7]: 1.895494267033981
```

```
In [8]: opt.bisect(f, 0.5, 2.0)
Out[8]: 1.8954942670345645
```

Les fonctions `fsolve` et `root` peuvent résoudre des systèmes d'équations si la fonction f accepte un itérable et retourne un itérable de même longueur.

Ainsi, on peut obtenir la solution du système $\begin{cases} x^2 - y^2 = 1 \\ x + 2y = 3 \end{cases}$

```
In [9]: import scipy.optimize as opt

In [10]: def f(v) :
   ...:     return v[0]**2 - v[1]**2 - 1, v[0] + 2*v[1] - 3
```

```
In [11]: opt.fsolve(f, [ 0.0, 0.0 ])
Out[11]: array([ 1.30940108,  0.84529946])
```

```
In [12]: sol = opt.root(f, [ 0.0, 0.0 ])

In [13]: sol.success
Out[13]: True

In [14]: sol.x
Out[14]: array([ 1.30940108,  0.84529946])
```

2.4 Dérivation numérique

`scipy.misc.derivative(f, x0, dx=1.0)` : retourne une estimation de la dérivée de `f` en `x0`, en calculant le taux d'accroissement sur un intervale de largeur `dx`.

Bien que le paramètre `eps` soit facultatif, il est indispensable pour avoir une bonne estimation de la dérivée (la valeur par défaut, 1.0, étant souvent trop grande).

Exemple de calcul de $f'(0.5)$ pour $f(x) = x^2 - 3x + 4$:

```
In [15]: import scipy.misc as misc

In [16]: def f(x) :
   ...:     return x**2 - 3*x + 4

In [17]: misc.derivative(f, 0.5, 1e-5)
Out[17]: -1.9999999999908977
```