Université de Montpellier **HLMA310**
2018/2019 **Alexandre Gramfort, Joseph Salmon**

TP n° 4 : Descente de gradient

Objectifs du TP : comprendre la méthode de descente de gradient et interaction avec l'affichage graphique dynamique.

Commencer par nommer votre fichier en suivant la même procédure, et en utilisant `filename` pour votre nom de TP :

```
# Changer ici par votre Prenom Nom:
prenom = "Joseph" # à remplacer
nom = "Salmon" # à remplacer
extension = ".ipynb"
tp = "TP4_HMLA310"
filename = "_".join([tp, prenom, nom]) + extension
filename = filename.lower()
```

Exercice 1. Descente de gradient à pas constant

Cette méthode remonte aux moins aux travaux de Cauchy [Cau47], et sert à optimiser une fonction différentiable. Ici on va faire le cas de fonction de deux variables dont on cherche le minimium.

Pour cela commençons par nous intéresser à la fonction f suivante :

$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

On peut définir cette fonction dans `numpy` de la manière suivante

```
import numpy as np


def f(x):
    x1, x2 = x
    return (x1**2 + x2 - 11)**2 + (x1 + x2**2 - 7)**2
```

Comme la fonction est de deux variables on peut la représenter de plusieurs façons.

La première manière consiste à visualiser directement l'altitude de la fonction avec la fonction `plot_surface` de `matplotlib`. Pour cela lancer la commande suivante pour initialiser les packages et les colormaps dont on a besoin :

```
%matplotlib notebook
import matplotlib.pyplot as plt
import matplotlib
cmap_reversed = matplotlib.cm.get_cmap('RdBu_r')
from mpl_toolkits import mplot3d
```

Puis pour discrétiser l'espace sur lequel on va observer la fonction (ici sur la zone $[-5.5, 5.5] \times [-5.5, 5.5]$) on va créer les valeurs suivantes :

```
X1, X2 = np.meshgrid(np.linspace(-5.5, 5.5, 50),
                     np.linspace(-5.5, 5.5, 50))
Z = f([X1, X2]) # Altitude
```

1) lancer la commande qui suit (assurez vous bien que matplotlib est en mode interactif avec la commande `%matplotlib notebook`). Cliquer en maintenant le clic enfoncé et visualiser la surface qui représente la fonction f.

```
fig = plt.figure(figsize=(6, 6))
ax = plt.axes(projection='3d')
ax.plot_surface(X1, X2, Z, rstride=1, cstride=1,
                cmap=cmap_reversed, edgecolor='none')
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)
ax.set_zlim(0, 500)
plt.show()
```

On pourra consulter : https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html pour diverse représentions similaires.

La seconde manière de visualiser la fonction consiste à afficher ses lignes de niveaux (à la manière d'une carte IGN) : on colorie de manières différentes les zones de la fonction selon leur "altitude" donnée par $f(x_1, x_2)$. Pour cela on pourra langer la commande suivante :

```
%matplotlib inline


def plot(xs=None):
    levels = list(1.7 ** np.linspace(0, 10, 30) - 1.) + [300]
    plt.figure(figsize=(8, 8))
    plt.contourf(X1, X2, np.sqrt(Z), levels=np.sqrt(
        levels), cmap=cmap_reversed)
    plt.colorbar(extend='both')
    if xs is not None:
        x1, x2 = np.array(xs).T
        plt.plot(x1, x2, 'k')
        plt.plot(x1, x2, 'o', color='purple')
    plt.show()


plot()
```

2) Proposer une inversion des couleurs de la colormap (altitude basse rouge, et altitude haute en bleu).

Dans la suite on rappelle maintenant la méthode de descente de gradient. Pour rappelle le gradient de f en $x = (x_1, x_2)^\top$ noté $\nabla f(x)$ est le vecteur :

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x_1, x_2) \\ \frac{\partial f}{\partial x_1}(x_1, x_2) \end{pmatrix}.$$

Algorithme 1 : DESCENTE DE GRADIENT

input : Initialisation $x^0 = (x_1^0, x_2^0)$, max. itérations $t_{\max}$, pas α
for $0 \leqslant t \leqslant t_{\max} - 1$ **do**
  $x^{t+1} \leftarrow x^t - \alpha \nabla f(x^t)$
return $x^{t_{\max}}$

3) Vérifier que les dérivées partielles de la fonction f sont données par

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x_1, x_2) \\ \frac{\partial f}{\partial x_1}(x_1, x_2) \end{pmatrix} = \begin{pmatrix} 2(-7 + x_1 + x_2^2 + 2x_1(-11 + x_1^2 + x_2)) \\ 2(-11 + x_1^2 + x_2 + 2x_2(-7 + x_1 + x_2^2)) \end{pmatrix}.$$

Créer alors la fonction qui calcule le gradient :

```
def f_grad(x):
    x1, x2 = x
    df_x1 = 2 * (-7 + x1 + x2**2 + 2 * x1 * (-11 + x1**2 + x2))
    df_x2 = 2 * (-11 + x1**2 + x2 + 2 * x2 * (-7 + x1 + x2**2))
    return np.array([df_x1, df_x2])
```

4) Compléter la fonction suivante pour qu'elle mette en œuvre l'algorithme de descente de gradient, et afficher les itérés en réutilisant la fonction `plot` précédemment codée :

```
def grad_descent(step_size=0.01, max_iter=0):
    """Gradient descent with constant step size"""
    x = x0
    xs = [x]
    for k in range(max_iter):
        d_k = - f_grad(x) # direction of descent
        x += ### TODO
        xs.append(x)
    plot(xs)
    plt.show()
```

5) Rajouter la commande suivante pour pouvoir jouer avec des curseurs et régler la valeur du pas et du nombre d'itérations de l'algorithme.

```
from ipywidgets import interact

interact(grad_descent, step_size=(0., .05, 0.005), max_iter=(0, 50, 1));
```

Références

[Cau47] A. Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847) :536–538, 1847. 1