Université de Montpellier **HLMA310**
2019/2020 **Alexandre Gramfort, Joseph Salmon**

TP n° 4 : Descente de gradient

Objectifs du TP : Comprendre la méthode de descente de gradient et comment interagir avec l'affichage graphique de manière dynamique.

Commencer par nommer votre fichier en suivant la même procédure qu'au précédent TP, en utilisant `filename` pour votre nom de TP comme ci-dessous :

```
# Changer ici par votre Prenom Nom:
prenom = "Joseph" # à remplacer
nom = "Salmon" # à remplacer
extension = ".ipynb"
tp = "TP4_HLMA310"
filename = "_".join([tp, prenom, nom]) + extension
filename = filename.lower()
```

Exercice 1. (Descente de gradient à pas constant)

Cette méthode remonte au moins aux travaux de Cauchy [Cau47], et sert à optimiser une fonction différentiable. Ici on va s'intéresser au cas d'une fonction de deux variables dont on cherche le minimum.

Pour cela nous considérerons la fonction f suivante :

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

1) La fonction précédente est elle convexe selon vous ? Considérez par exemple les points $(4, -4)$ et $(-4, -4)$ et leur moyenne $(0, -4)$.

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`.

2) 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
```

3) On a besoin de discrétiser (en 50 valeurs sur chaque axe) l'espace sur lequel on va observer la fonction. Dans la suite, on va s'intéresser à l'ensemble $[-5.5, 5.5] \times [-5.5, 5.5]$. On va pour cela créer les `arrays` suivants :

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

4) Lancer la commande qui suit (assurez vous bien que `matplotlib` est en mode interactif avec la commande `%matplotlib notebook`). Cliquer sur la figure produite, et en maintenant le clic enfoncé, faites pivoter 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()
```

Rem: On pourra consulter : https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html pour diverses 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 les zones de la fonction selon leur "altitude" donnée par $f(x_1, x_2)$. Pour cela on pourra lancer la commande suivante :

```
def plot(xs=None, cmap=cmap_reversed):
    levels = list(1.7 ** np.linspace(0, 10, 30) - 1.) + [300]
    plt.figure(figsize=(5, 5))
    plt.contourf(X1, X2, np.sqrt(Z), levels=np.sqrt(
        levels), cmap=cmap)
    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()
```

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

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

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

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 $x^{t_{\max}}$

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

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

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])
```

7) Compléter la fonction suivante pour qu'elle mette en œuvre l'algorithme de descente de gradient décrit dans l'Algorithme 1 et afficher les itérés obtenus, en réutilisant par exemple la fonction `plot` précédemment codée :

```
def grad_descent(x0=x0, step_size=0.01, max_iter=20):
    """Descente de gradient avec un pas constant"""
    x = x0
    xs = [x]
    for k in range(max_iter):
        x = ### XXX
        xs.append(x)
    plot(xs)
    plt.show()
```

Étudier l'impact des paramètres sur la convergence de l'algorithme : `x0, step_size, max_iter`.

8) Rajouter la commande suivante pour pouvoir manipuler des curseurs et régler la valeur du pas (`step_size`) et du nombre d'itérations (`max_iter`) de l'algorithme.

```
from ipywidgets import interact, fixed
interact(grad_descent, x_init=fixed(x0), step_size=(0., .05, 0.005)
```

Références

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