Algorithme du gradient

On se propose d'étudier un algorithme permettant de déterminer un minimum local d'une fonction $f : \mathbb{R}^d \longrightarrow \mathbb{R}$. Si la fonction f est strictement convexe, ce minimum est unique et est le minimum global de la fonction f, sinon il peut y avoir plusieurs minimums locaux. La méthode la plus simple pour déterminer un tel minimum est l'algorithme du gradient. On détermine itérativement une suite $(x_n)_{n\in\mathbb{N}}$ d'éléments de $\mathbb{R}^d$ définie de la manière suivante, où ∇f est le gradient de f. Intuitivement, cet algorithme choisit la direction où f décroit le plus et déplace le candidat x dans cette direction, opposée à celle du gradient.

- x_0 est un élément quelconque de $\mathbb{R}^d$
- $x_{n+1} = x_n - \tau\nabla f(x_n)$ où $\tau > 0$ est un paramètre appelé pas.

On utilisera la bibliothèque numpy et son type numpy.array pour représenter les vecteurs $x \in \mathbb{R}^d$.

```
import numpy
```

Voici une implémentation de l'algorithme du gradient en python:

```
def algo_gradient(gradient_f, tau, x_0, n_iterations):
    x = [x_0]
    for i in range(n_iterations):
        new_x = x[-1] - tau * gradient_f(x[-1])
        x.append(new_x)
    return x
```

Que contient la variable x renvoyée par la fonction `algo_gradient` ?

Un exemple simple en 2D

On se propose d'étudier la convergence de l'algorithme du gradient dans le cas d'une fonction simple en 2D, donnée par

$$f(x_1, x_2) = \frac{1}{2}(x_1^2 + \eta x_2^2).$$

On a:

$$\nabla f(x_1, x_2) = (x_1, \eta x_2).$$

On fixe $\eta = 10$, et $\tau = 0.02$. On fixe aussi le nombre d'itérations `n_iterations` à 50.

```
eta = 10.0
tau = 0.02
n_iterations = 50
```

En quel point f atteint-elle son minimum ?

Définir f et ∇f en python. `grad_f` doit renvoyer un vecteur de type `numpy.array`

En utilisant la bibliothèque matplotlib, représenter graphiquement f sur le domaine $[-1, 1] \times [-1, 1]$. On pourra utiliser la fonction `numpy.linspace` pour construire un maillage régulier de $[-1, 1]$. On utilisera la fonction `plt.pcolor` pour représenter une fonction de deux variables.

Sur ce même graphique, représenter le vecteur $\nabla f(x)$ pour $x = (0.5, 0.5)$. On pourra utiliser la fonction `plt.quiver` de `matplotlib`.

Tester l'algorithme du gradient sur f pour différentes valeurs de $x_0 \in [-1, 1] \times [-1, 1]$. Sur un même graphique, représenter $f(x_n)$ en fonction de n, pour plusieurs valeurs de x_0. Dans quel cas l'algorithme du gradient converge-t-il rapidement ? Dans quel cas converge-t-il plus lentement ?

On fixe $x_0 = (0.5, 0.5)$. Sur un même graphique, représenter $f(x_n)$ en fonction de n, pour plusieurs valeurs de τ. Que se passe-t-il si τ est trop grand ? S'il est trop petit ? On pourra par exemple utiliser les valeurs suivantes de τ :

$$\tau = 0.01, 0.05, 0.2, 0.201.$$

Application : débruitage TV

On considère le problème de débruitage d'une série temporelle. L'inconnue est un vecteur $X_0 = (g(t_k), k \in \{0, \dots, n-1\}) \in \mathbb{R}^n$, où $g : [0,1] \to \mathbb{R}$ et $t_k = \frac{k}{n}, k \in \{0, \dots, n-1\}$.

Ce signal inconnu est déterioré par du bruit, le vecteur des observations étant donné par:

$$Y = X_0 + W,$$

où W est un terme de buit. Notre objectif est de récupérer le vecteur X_0 à partir des observations Y. Cela est bien sûr impossible dans le cas le plus général, on va donc commencer par essayer de le faire dans un cas particulier: le cas où la fonction g est constante par morceaux et le terme de bruit W est un vecteur gaussien dont les entrées sont indépendentes. On va utiliser pour cela un algorithme de minimisation.

On commence, pour pouvoir tester l'efficacité de cet algorithme, par générer des données pour lequelles on connaît le vecteur sans bruit.

Le code python suivant génère de telles données :

```
import matplotlib.pyplot as plt
import numpy
import numpy.random as rd
rd.seed(5000) # On fixe la graine du générateur pseudo-aléatoire (comme ça, je sais quel X vous avez)
n_jumps = 5 # Nombre de sauts
jumps = rd.rand(n_jumps) # Localisation des sauts au hasard
jumps = numpy.sort(jumps)
jumps = numpy.insert(jumps, 0, 0.0)
jumps = numpy.insert(jumps, n_jumps + 1, 1.0)
n = 100 # Nombre de points
t = numpy.linspace(0, 1-1.0/n, n) # Localisations des points
signal_no_noise = numpy.zeros(n) # Le vecteur X
for i in range(n_jumps + 1):
    signal_no_noise[(t >= jumps[i]) & (t < jumps[i + 1])] = 0.5* (1 + rd.rand()) # Valeur de x en les poin
noise_level = 0.01
signal_noisy = signal_no_noise + noise_level * rd.randn(n) # Signal bruité

## Affichage graphique
plt.figure() # Créer un graphique
plt.plot(t, signal_no_noise, 'b', label="Signal sans bruit") # En bleu, la valeur de x
plt.plot(t, signal_noisy, 'g', label=r"Signal avec bruit") # En vert, la valeur de y
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, # Ajouter une légende au graphique
           ncol=2, mode="expand", borderaxespad=0.)
plt.show() # Afficher le graphique
```

On définit la variation totale (TV) lissée d'un vecteur x de $\mathbb{R}^n$ comme ceci:

$$\|x\|_{\mathrm{TV},\delta} = \sum_{i=0}^{n-1} \sqrt{\delta + |x_{i+1} - x_i|^2}$$

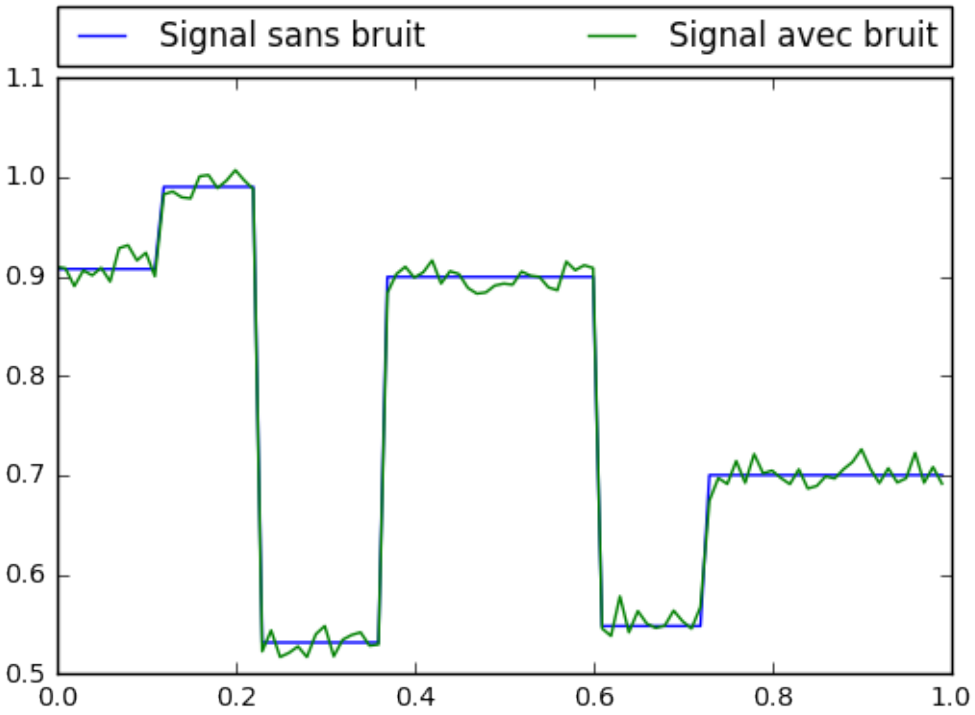
Ses dérivées partielles sont données par:

$$\frac{\partial}{\partial x_i}\|x\|_{\mathrm{TV},\delta} = \frac{(x_i - x_{i+1})}{\sqrt{\delta + |x_{i+1} - x_i|^2}} + \frac{(x_i - x_{i-1})}{\sqrt{\delta + |x_{i-1} - x_i|^2}},$$

si $i \in \{1, \dots, n-1\}$ et

$$\frac{\partial}{\partial x_0}\|x\|_{\mathrm{TV},\delta} = \frac{(x_0 - x_1)}{\sqrt{\delta + |x_0 - x_1|^2}},$$

$$\frac{\partial}{\partial x_n}\|x\|_{\mathrm{TV},\delta} = \frac{(x_n - x_{n-1})}{\sqrt{\delta + |x_n - x_{n-1}|^2}}.$$



On rappelle que le gradient est le vecteur composé des dérivées partielles.

On fixe $\delta = 10^{-5}$.

Implémenter la norme TV lissée ainsi que son gradient en python:

Comparer les valeurs des normes TV pour les vecteurs bruités et non bruités définis plus haut.

On propose d'approcher X_0 par le vecteur $\hat{X}$ qui minimise la fonction suivante :

$$\hat{X} = \text{argmin}_X J(X)$$

où

$$J(X) = \frac{1}{2}\|X - Y\|^2 + \lambda \|X\|_{\text{TV},\delta},$$

où $\lambda \geq 0$ est un paramètre. On rappelle que le gradient de la fonction $h : x \mapsto \frac{1}{2}\|x-y\|^2$ est donné par $\nabla h(x) = x - y$.

Implémenter la fonction J et son gradient en python

Tester l'algorithme du gradient pour reconstruire `signal_no_noise`. On pourra initialiser l'algorithme avec `signal_noisy`. Sur un même graphique, représenter le vecteur reconstruit $\hat{X}$ pour différents nombre d'itérations.

Choisir λ proche de 0, à quoi ressemble le signal reconstruit ?

Choisir λ très grand (par exemple $\lambda = 5.0$), à quoi ressemble le signal reconstruit ?

Changer le niveau de bruit. Comment faut-il ajuster λ en fonction du niveau de bruit ? Jusqu'à quel niveau de bruit arrivez-vous à obtenir une reconstruction correcte ?

Générer d'autres signaux et étudier l'efficacité de l'algorithme de minimisation TV pour leur débruitage.

Un exemple de minimisation globale

En utilisant l'algorithme du gradient, déterminer le minimum sur $[-512, 512]^2$ de la fonction suivante :

$$f(x,y) = -(y+47)\sin(\sqrt{|\frac{x}{2} + (y+47)|}) - x\sin(\sqrt{|x-(y+47)|})$$