
TRAVAUX PRATIQUES N° 3 : Descente de gradient et variantes

Préambule : Pour ce TP, on pourra utiliser les fichiers suivants¹ pour certaines questions : `dico_math_functions.py`, `widget_convergence.py`, `widget_level_set.py` et les télécharger dans un même dossier. Les questions notées **Pour aller plus loin**, peuvent être omises en première lecture.

Certains packages peuvent nécessiter une installation préalable. Il sera donc sûrement nécessaire de lancer les commandes suivantes dans votre fenêtre interactive lors de la première utilisation des packages suivant :

```
pip install matplotlib==3.4.1
pip install ipynb
pip install numba
```

Dans ce TP, nous allons proposer plusieurs algorithmes pour trouver un minimiseur d'une fonction $f : \mathbb{R}^d \rightarrow \mathbb{R}$ pour $d \geq 2$.

1 Algorithme de la descente de gradient

Dans cette partie on suppose f , et on va introduire la méthode de descente de gradient (🇬🇧 : *gradient descent*, *GD*) pour une telle fonction.

Algorithme 1 : Algorithme de descente de gradient (à pas fixe)

```
input  :  $\nabla f, x_0, \gamma, n_{\text{iter}}$ 
param :  $\epsilon$ 
init   :  $x \leftarrow x_0$ 
for  $i = 1, \dots, n_{\text{iter}}$  do
     $g \leftarrow \nabla f(x)$ 
    if  $\|g\|^2 \leq \epsilon^2$  then                                // Sortie de boucle si le gradient est petit
        Break
     $x \leftarrow x - \gamma g$                                 // mise à jour dans la direction opposée au gradient
return  $x$ 
```

QUESTION 1. (Algorithme de descente de gradient) Adapter et coder en Python cet algorithme pour avoir en sortie la liste de tous les itérés de la suite. On ajoutera aussi un critère d'arrêt qui stoppe l'algorithme si la norme euclidienne du gradient est plus petite qu'un seuil ϵ .

Vérifier que votre méthode obtient bien le minimum global pour la fonction :

$$f_{\text{test}} : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto (x_1 - 1)^2 + 3 \cdot (x_2 + 1)^2 .$$


1. disponibles ici : <http://josephsalmon.eu/HAX606X.html>

QUESTION 2. (Application au cas quadratique) On rappelle que toute matrice M symétrique définie positive de taille 2×2 peut s'écrire :


$$M(\theta, \sigma_1, \sigma_2) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}^\top \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} . \quad (1)$$

On s'intéressera dans la suite aux fonctions quadratiques de $\mathbb{R}^2 \rightarrow \mathbb{R}$ suivantes :

$$f_{\theta, \sigma_1, \sigma_2}(x) = x^\top M(\theta, \sigma_1, \sigma_2)x . \quad (2)$$

Écrire une fonction qui prenne en entrée : $\theta, \sigma_1, \sigma_2$ (et comme paramètre optionnel $n_{\text{iter}}, \epsilon$) et qui renvoie le graphique de l'évolution de la valeur de l'objectif $f_{\theta, \sigma_1, \sigma_2}(x_k)$, au cours des itérations de l'algorithme de descente de gradient. Comparer avec la vitesse de convergence théorique vue en cours pour plusieurs choix de triplets $\sigma_1, \sigma_2, \theta$. Pour visualiser l'impact des paramètres sur la fonction, on pourra utiliser les visualisations de la surface et ses lignes de niveaux ( : *level set*) disponibles dans le fichier `widget_level_set.py`.

2 Algorithmes de descente par coordonnée.

Dans cette partie on va noter e_1, \dots, e_d la base canonique de \mathbb{R}^d . Les méthodes de descente par coordonnée ( : *coordinate descent, CD*) consiste à résoudre le problème d'optimisation en résolvant itérativement des problèmes (potentiellement beaucoup) plus petits.

QUESTION 3. (Algorithme de descente par coordonnée exacte)

Algorithme 2 : Algorithme de relaxation (ou de descente par coordonnée exacte)


```

input  :  $x_0, n_{\text{iter}}$ 
param :  $\epsilon$ 
init   :  $x \leftarrow x_0$ 
for  $i = 1, \dots, n_{\text{iter}}$  do
    for  $j = 1, \dots, d$  do
         $\tau_j \in \arg \min_{t \in \mathbb{R}} f(x + t \cdot e_j)$            // résolution problème 1D
         $x_j \leftarrow x_j + \tau_j$                        // mise à jour
    end for
return  $x$ 

```

Coder l'algorithme précédent en utilisant la fonction `optimize` de `scipy` pour résoudre le problème d'optimisation 1D. On ajoutera de nouveau un critère d'arrêt après chaque "époque" (*i.e.*, à la fin de chaque boucle interne).

QUESTION 4. (Algorithme de descente de gradient par coordonnée) Résoudre le problème d'optimisation 1D peut être très coûteux pour certaines fonctions. On préfère donc souvent utiliser des méthodes de descente à pas fixe (ou variable). Comparer la descente de gradient classique à la descente de gradient par coordonnées pour la fonction f_{test} . À l'aide de l'interface interactive disponible dans le fichier `widget_convergence.py`, comparer la convergence de la descente de gradient classique à la descente de gradient par coordonnées dans le cas quadratique. On fera notamment varier $\sigma_1, \sigma_2, \theta$ et la taille du pas pour chaque méthode.

QUESTION 5. (Pour aller plus loin avec la comparaison) Ajouter la descente par coordonnée exacte dans la visualisation. Pour cela, il faudra créer une entrée dans `dic_optim_algos` qui associe au nom de l'algorithme la fonction renvoyant les itérés, une couleur, un type de marqueur. Pour simplifier, on pourra utiliser un curseur ( : *slider*) comme pour les autres méthodes, mais

Algorithme 3 : Algorithme descente par gradient par coordonnée à pas fixe

```
input  :  $\nabla f, x_0, \gamma, n_{\text{iter}}$ 
param :  $\epsilon$ 
init   :  $x \leftarrow x_0$ 
for  $i = 1, \dots, n_{\text{iter}}$  do
  for  $j = 1, \dots, d$  do
     $x_j \leftarrow x_j - \gamma \frac{\partial f}{\partial x_j}(x)$  // Itération de descente de gradient sur la coordonnée  $j$ 
  end for
return  $x$ 
```

qui n'aura aucun impact sur l'algorithme en question. En entrée, la fonction devra prendre les mêmes arguments que `quad_coordinate` (ou `quad_grad_descent`, on utilisera les mêmes valeurs par défaut).