Valentin De Bortoli debortoli@cmla.ens-cachan.fr
Alain Durmus alain.durmus@cmla.ens-cachan.fr
TP 1 : optimisation numérique Première année Master Hadamard

Méthodes de descente de gradient et algorithmes de Newton

En préambule on suppose que les paquets suivants ont été chargés

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

Remarque : la plupart du temps on n'implémente pas les méthodes classiques présentées dans ce TP et on utilise des paquets du type `scipy`. Néanmoins, il convient de savoir ce que contienne ces *boîtes noires* et de se forger une intuition sur les différentes méthodes d'optimisation.

1 Méthode de gradient à pas fixe

Le but de cet exercice est d'implémenter la descente de gradient à pas fixe et d'identifier ses atouts et limites.

Banane de Rosenbrock Dans notre première exemple, on étudie la fonction $f : \mathbb{R}^2 \to \mathbb{R}$ suivante

$$f(x_1, x_2) = (x_1 - 1)^2 + 100\,(x_1^2 - x_2)^2.$$

f est de classe $\mathcal{C}^1(\mathbb{R}^2, \mathbb{R})$, positive, atteint son unique minimum en $x^* = (1, 1)$ et $f(x^*) = 0$. Voici le code python qui calcule cette fonction, son gradient et sa hessienne.

```
## Rosenbrock function and its gradient

def rosenbrock(x):
    y = np.asarray(x)
    return np.sum((y[0] - 1)**2 + 100 * (y[1] - y[0]**2)**2)


def rosenbrock_grad(x):
    y = np.asarray(x)
    grad = np.zeros_like(y)
    grad[0] = 400 * y[0] * (y[0]**2 - y[1]) + 2 * (y[0] - 1)
    grad[1] = 200 * (y[1] - y[0]**2)
    return grad
```

1. Tracer les lignes de niveau de f dans le rectangle $(x_1, x_2) \in [-1.5,\, 1.5] \times [-0.5,\, 1.5]$. On utilisera la fonction plt.contour.
2. Implémenter la méthode de gradient à pas fixe qui définit la suite $(x_k)_{k\in\mathbb{N}}$ par récurrence :
$$x_{k+1} = x_k - h\,\nabla f(x_k).$$
Proposer au moins deux critères d'arrêt pour l'algorithme et discuter de leur pertinence.
3. Tracer graphiquement les itérées de la méthode de gradient à pas fixe avec comme point initial $x_0 = (-1, 1)$ et différentes valeurs de h. Interpréter les résultats.
4. Recommencer avec le gradient normalisé
$$x_{k+1} = \begin{cases} x_k - h\,\frac{\nabla f(x_k)}{\|\nabla f(x_k)\|} & \text{if } \nabla f(x_k) \neq 0 \\ x_k & \text{otherwise .} \end{cases}$$
avec $h = 5 \times 10^{-2}$ et en affichant les 200 premières itérations. Interpréter. Cette méthode a-t-elle une chance de converger ?

Problème quadratique On introduit la fonction f définie sur $\mathbb{R}^n$ ($n \in \mathbb{N}$ avec $n \geq 2$) par $f(x) = x^{\mathrm{T}} A x$ avec A diagonale telle que $A_{1,1} = M$, $A_{2,2} = m$ avec $(M, m) \in {\mathbb{R}_+^*}^2$ et tous les autres coefficients diagonaux fixés à 1. f est une fonction strictement convexe coercive qui possède donc un unique minimum en 0. Voici le code python qui calcule cette fonction, son gradient.

```
def mk_quad(m, M, ndim=2):
  ## Quadratic function and its gradient
    def quad(x):
        y = np.copy(np.asarray(x))
        scal = np.ones(ndim)
        scal[0] = m
        scal[1] = M
        y *= scal
        return np.sum(y**2)

    def quad_grad(x):
        y = np.asarray(x)
        scal = np.ones(ndim)
        scal[0] = m
        scal[1] = M
        return 2 * scal * y

    return quad, quad_grad
```

1. Représenter les lignes de niveaux de la fonction f pour différentes valeurs de M et de m pour $n = 2$. À votre avis, quand est-ce que le problème d'optimisation est le plus facile à résoudre ?

2. Utiliser l'algorithme du gradient à pas fixe pour minimiser f pour différentes valeurs de m, M et points initiaux x_0 pour la valeur du pas h optimal théoriquement. Illustrer par des graphiques la minimisation de f et commenter les.
3. Estimer numériquement l'ordre de convergence de l'algorithme en fonction du pas h et du nombre d'itération.
4. Vérifier expérimentalement que la vitesse de l'algorithme ne dépend pas de la taille de la matrice A.

2 Choix du pas de descente : règle d'Armijo et règle de Wolfe

A l'itération $k \in \mathbb{N}^*$ et pour une direction de descente d_k donnée, i.e $\langle d_k, \nabla f(x_k)\rangle < 0$, la *recherche linéaire exacte* consiste à estimer un pas de descente optimal h^* qui vérifie

$$h^* = \text{argmin}_{h\geq 0} f(x_k + hd_k),$$

introduisant donc un nouveau problème d'optimisation qui peut être aussi difficile que le problème de minimisation original. On introduit ici deux règles de *recherche linéaire inexacte*, la règle d'Armijo (1966) et la règle de Wolfe (1969).

Règle d'Armijo La règle d'Armijo consiste étant donné un point courant x_k, une direction de descente d_k à l'itération $k \in \mathbb{N}^*$ à trouver un pas $h_{k+1} > 0$ satisfaisant

$$f(x_k + h_{k+1}d_k) \leq f(x_k) + c\, h_{k+1} \langle \nabla f(x_k), d_k\rangle ,$$

où $c \in]0, 1/2[$ est un paramètre fixé par l'utilisateur. Proposer une méthode permettant de trouver un tel pas h_{k+1} et l'implémenter. Pour cela, on pourra considérer une suite décroissante tendant vers 0 et initialisée à la valeur $-\langle \nabla f(x_k), d_k\rangle/(L\|d_k\|^2)$, où L est un paramètre fixé par l'utilisateur.

Règle de Wolfe La règle de Wolfe consiste étant donné un point courant x_k, une direction de descente d_k à l'itération $k \in \mathbb{N}^*$ à trouver un pas $h_k > 0$ satisfaisant

$$f(x_k + h_{k+1}d_k) \leq f(x_k) + c_1\, h_{k+1} \langle \nabla f(x_k), d_k\rangle ,$$

et

$$\langle d_k, \nabla f(x_k + h_{k+1}d_k)\rangle \geq c_2 \langle d_k, \nabla f(x_k)\rangle$$

où $c_1 \in]0, 1/2[$, $c_2 \in]1/2, 1[$ sont des paramètres fixés par l'utiisateur. Proposer une méthode permettant de trouver un tel pas h_{k+1} et l'implémenter. Pour cela, on pourra par exemple rechercher ce pas dans un intervalle $[a, b]$ que l'on affinera au fur et à mesure et en prenant comme valeur initial $-\langle \nabla f(x_k), d_k\rangle/(L\|d_k\|^2)$, où L est un paramètre fixé par l'utilisateur. On pourra aussi se fixer un nombre maximal d'itérations.

Banane de Rosenbrock et problème quadratique On considère les deux problèmes d'optimisation introduits dans l'exercice précédent.

1. Implémenter la méthode de gradient avec la règle d'Armijo et règle de Wolfe.
2. Tester ces règles pour la fonction de Rosenbrock avec les valeurs suivantes pour $x_0 = (-1, 1)$, $L = 100$ et différentes valeurs des paramètres c, c_1, c_2. Comparer les entre elle et avec la méthode de descente de gradient à pas fixe. On pourra par exemple afficher l'évolution au cours des algorithmes de la distance entre les itérés et le minimiseur de f ainsi que la suite $(f(x_k) - f(x^\star))_{k\in\mathbb{N}}$
3. Tester l'algorithme sur le problème quadratique et effectuer la même étude que pour la fonction de Rosenbrock.

Remarque 1. *Il existe encore d'autres règles de recherche linéaire inexacte pour le choix d'un pas de descente comme la règle de Goldstein. Ces règles permettent d'éviter le problème de recherche linéaire exacte tout en accélérant les méthodes de gradient.*

3 Gradient conjugué

Le gradient conjugué est une méthode de type quasi-Newton qui a la propriété de converger en temps fini pour des fonctionnelles quadratiques. On présente ici une extension de cet algorithme.

Gradient conjugué On se donne une fonction $f : \mathbb{R}^n \to \mathbb{R}$ à minimiser et un point initial $x_0 \in \mathbb{R}^n$. On initialise $d_0 = 0$.

Partant d'un point courant x_k et d'une direction d_k (ayant servi à calculer x_k sauf pour $k = 0$) à l'itération $k \in \mathbb{N}$, on itère de la manière suivante :

1. *Mise à jour de la direction de descente.* On met à jour la direction de descente par
$$d_{k+1} = -\nabla f(x_k) + \beta_{k+1} d_k \ ,$$
où
$$\beta_{k+1} = \frac{\langle \nabla f(x_k) - \nabla f(x_{k-1}), \nabla f(x_k)\rangle}{\|\nabla f(x_{k-1})\|^2} \ .$$
2. *Recherche du pas.* On applique une règle de *recherche linéaire inexacte* (ici la règle de Armijo/Wolfe) pour trouver le meilleur pas de descente h_{k+1} pour la direction de descente d_{k+1}.
3. *Mise à jour des itérés.* On pose $x_{k+1} = x_k + h_{k+1} d_{k+1}$.

Remarque 2. *On a ici présenté l'algorithme de Polak-Ribière mais il existe d'autres variantes comme Fletcher-Reeves.*

Banane de Rosenbrock et problème quadratique On considère de nouveau les fonctions du premier exercice.

1. Implémenter la méthode du gradient conjugué version *Polak-Ribière*.
2. Tester l'algorithme pour la fonction de Rosenbrock avec $x_0 = (-1, 1)$. Comparer l'algorithme avec les méthodes précédentes.
3. Pour un problème quadratique, le gradient conjugué converge en au plus n itérations où n est la taille du problème. Vérifier cette affirmation pour différentes tailles de matrices dans le cadre du problème quadratique introduit précédemment.

4 Le problème de Lennard-Jones

Les problèmes de conformation atomique consistent à trouver les coordonnées spatiales de N noyaux atomiques formant une molécule d'énergie minimale. Dans le cas du problème de Lennard-Jones de taille N, la force d'interaction entre deux atomes identiques distants d'une longueur r est supposée issue d'un potentiel radial de van der Walls, s'écrivant sous forme adimensionnée

$$V(r) = \frac{1}{r^{12}} - \frac{2}{r^6}.$$

Soit une molécule u composée de N atomes $u := (X_i)_{i\in[\![1,N]\!]} = ((x_i, y_i, z_i))_{i\in[\![1,N]\!]} \in (\mathbb{R}^3)^N$, l'énergie potentielle totale du système est donnée par

$$W(u) = \sum_{\substack{(i,j)\in\{1,\dots,N\}\\ i<j}} V\left(\|X_i - X_j\|\right).$$

La configuration la plus stable de la molécule $u^\star$ correspond à un minimum global d'énergie potentielle. Lorsque $N > 4$, le problème de la recherche du minimum global de W devient complexe. Il est d'ailleurs conjecturé que le nombre de minimums locaux de W croît exponentiellement avec N. Voici le code python qui calcule la fonction à minimiser, son gradient et sa hessienne.

```
## Potential V and its gradient

def V(x):
    return x ** -12 - 2 * x ** - 6

def V2(x):
    return x ** -6 - 2 * x ** - 3

def V2der(x):
    return -6 * x ** -7 + 6 * x ** -4

## Total Lennard-Jones potential of the system
def W(u):
    N = len(u) / 3
    u_v = np.reshape(u, (N, 3))

    M = np.zeros([N, N, 3])
    M -= u_v
    M = M - np.transpose(M, (1, 0, 2))

    M = np.sum(M ** 2, 2)
    np.fill_diagonal(M, 1)
    M = V2(M)
    np.fill_diagonal(M, 0)

    return .5 * np.sum(M)
```

```
def grad_W(u):
    N = len(u) / 3
    u_v = np.reshape(u, (N, 3))

    M = np.zeros([N, N, 3])
    M -= u_v
    M = M - np.transpose(M, (1, 0, 2))

    Mnorm = np.sum(M ** 2, 2)
    np.fill_diagonal(Mnorm, 1)
    Mnorm = V2der(Mnorm)
    np.fill_diagonal(Mnorm, 0)

    grad = np.reshape(Mnorm, (N**2, 1)) * np.reshape(M, (N ** 2, 3))
    grad = np.reshape(grad, (N, N, 3))
    grad = np.sum(grad, 1)
    return 2 * np.ravel(grad)
```

Recherche du minimum global on va tenter d'appliquer les méthodes précédemment introduites pour l'étude de ce problème physique.

1. Appliquer l'algorithme de descente du gradient avec règle d'Armijo/Wolfe pour ce problème. Tester l'algorithme pour $N = 4$. on initialisera l'algorithme de la manière suivante : $X_1 = 0$ et (X_1, X_2, X_3) choisis aléatoirement de manière indépendante dans $B(0, \sqrt[3]{N})$.
2. Pour $N = 4$ le minimum de la fonction W est atteint pour tout **tétraèdre** (X_1, X_2, X_3, X_4) et vaut -6, afficher votre configuration initiale et la configuration (X_1, X_2, X_3, X_4) que vous obtenez après minimisation. Interpréter. On pourra d'aider du code suivant pour l'affichage des configurations.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(init_pos[:,0], init_pos[:,1], init_pos[:,2], '^')
ax.scatter(final_pos[:,0], final_pos[:,1], final_pos[:,2], '^')
for i in range(4):
    ax.plot([init_pos[i,0], final_pos[i,0]],
            [init_pos[i,1], final_pos[i,1]],
            [init_pos[i,2], final_pos[i,2]], 'g')
    for j in range(4):
        ax.plot([final_pos[i,0], final_pos[j,0]],
                [final_pos[i,1], final_pos[j,1]],
                [final_pos[i,2], final_pos[j,2]], 'r')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

```
print(W(np.ravel(init_pos)))
print(W(np.ravel(final_pos)))
```

3. Reprendre les questions précédentes pour $N = 13$. Dans ce cas le minimum de la fonction W est atteint pour tout **icosaèdre régulier** $(X_1, \dots, X_{13})$. Interpréter.

5 Méthode de Levenberg-Marquardt, Gauss-Newton et application à la régression non linéaire

Étant donné $(x_i, y_i)_{i\in[\![1,N]\!]} \in (\mathbb{R}^2)^N$ on introduit le problème statistique classique de la régression non linéaire, i.e on cherche à minimiser la fonction suivante

$$F(\theta) = \frac{1}{2}\sum_{i=1}^{N}(f(x_i,\theta) - y_i)^2,$$

avec $f \in \mathcal{C}^1(\mathbb{R}\times\mathbb{R}^d, \mathbb{R})$. Cela correspond à rechercher l'*estimateur du maximum a posteriori* dans le cadre où l'erreur sur les données est gaussienne. Dans notre cas on pose $f : (x, (A, \sigma, \omega)) \mapsto A\exp(-\sigma x)\sin(\omega x)$, $\theta = (A, \sigma, \omega)$. On va maintenant introduire la méthode de Levenberg-Marquardt qui est une méthode de quasi-Newton.

Méthode de Levenberg-Marquardt Dans le problème considéré, la hessienne est typiquement mal conditionnée. De plus, lorsque le nombre de donnée et la dimension du problème sont très grands, il est en général trop coûteux de calculer la hessienne de F. La méthode de Levenberg-Marquardt définie la suite $(\theta_k)_{k\in\mathbb{N}}$ partant de $\theta_0 \in \mathbb{R}^d$ par la récursion suivante :

$$\theta_{k+1} = \theta_k - hM_{LM}(\theta_k)^{-1}\sum_{i=1}^{N}\nabla_\theta f(x_i,\theta_k)(f(x_i,\theta_k) - y_i),$$

pour un pas $h > 0$ et

$$M_{LM}(\theta) = \mu\,\mathrm{I}_3 + \sum_{i=1}^{N}\nabla_\theta f(x_i,\theta)\nabla_\theta f(x_i,\theta)^{\mathrm{T}}\ ,$$

est une approximation de la hessienne de F et μ est un paramètre fixé par l'utilisateur.

1. Implémenter la méthode de Levenberg-Marquardt pour le problème considéré. Voici les codes python nécessaires.

```
## Wave function, its gradient and its hessian

def mk_wave(A, sigma, omega):

def wave_fun(x):
        return A * np.exp(-sigma * x) * np.sin(omega * x)
```

```
def wave_grad(x):
        dim = np.max(np.shape(x))
        grad = np.zeros([3, dim])
        grad[0, :] = np.sin(omega * x)
        grad[1, :] = -A * x * np.sin(omega * x)
        grad[2, :] = A * x * np.cos(omega * x)
        grad = np.exp(-sigma * x) * grad
        return grad

def wave_hessian(x):
        dim = np.max(np.shape(x))
        s = np.sin(omega * x) * x * np.exp(-sigma * x)
        c = np.cos(omega * x) * x * np.exp(-sigma * x)
        scal = A * x ** 2 * np.exp(-sigma * x)
        hessian = np.zeros([3, 3, dim])
        hessian[0, 1, :] = -x * s
        hessian[1, 0, :] = -x * s
        hessian[0, 2, :] = hessian[2, 0, :] = x * c
        hessian[1, 1, :] = scal * s
        hessian[2, 2, :] = -scal * s
        hessian[1, 2, :] = hessian[2, 1, :] = -scal * c
        return hessian
    return wave_fun, wave_grad, wave_hessian


## Non-linear regression function, its gradient and its hessian

def mk_nonlinreg(x_train, y_train):
    def nonlinreg_fun(param):
        wave_fun = mk_wave(param[0], param[1], param[2])[0]
        return np.sum((wave_fun(x_train) - y_train) ** 2)

    def nonlinreg_grad(param):
        wave_fun, wave_grad = mk_wave(param[0], param[1], param[2])[:2]
        grad = 2 * wave_grad(x_train) * (wave_fun(x_train) - y_train)
        return np.sum(grad, 1)

    def nonlinreg_hessian(param, method='newton', mu=0.1):
        wave_fun, wave_grad, wave_hessian = mk_wave(
            param[0], param[1], param[2])
        grad = wave_grad(x_train)
        if method == 'newton':
            hess1 = 2 * wave_hessian(x_train)*\
            (wave_fun(x_train) - y_train)
            hess1 = np.sum(hess1, 2)
        elif method == 'lm':
            hess1 = mu * np.identity(3)
```

```
        hess2 = 2 * np.dot(grad, grad.T)
        return hess1 + hess2

    return nonlinreg_fun, nonlinreg_grad, nonlinreg_hessian
```

2. Générer des données à l'aide de la fonction suivante.

```
## Function to generate some data
# noise quantify the amount of noise in the model
# n_outliers is the number of observations which
#                      do not come from the true model

def generate_data(x, A, sigma, omega, noise=0, n_outliers=0):
    y = A * np.exp(-sigma * x) * np.sin(omega * x)
    error = noise * rnd.randn(x.size)
    outliers = rnd.randint(0, x.size, n_outliers)
    error[outliers] *= 35
    return y + error

A = 5
sigma = 0.1
omega = 0.1 * 2 * np.pi
x_true = np.array([A, sigma, omega])

noise = 0.1
n_outliers = 4

x_min = 0
x_max = 20

x_train = np.linspace(x_min, x_max, 30)
y_train = generate_data(t_train, A, sigma, omega, noise, n_outliers)
```

   Visualiser les données générées.

3. Tester la méthode de Levenberg-Marquardt sur les données générées et la comparer à une méthode de descente de gradient à pas fixe. On initialisera les paramètres à $\theta_0 = (1, 1, 1)$.