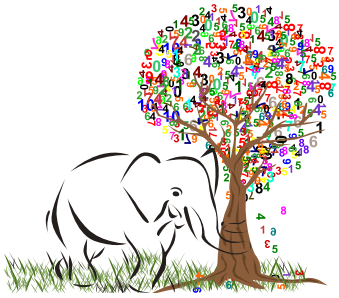

Chapitre[1] Algorithmes d'optimisation : application à des fonctions-objectifs non-linéaires Master Physique

SAMIR KENOUCHE - DÉPARTEMENT DES SCIENCES DE LA MATIÈRE - UMKB

MÉTHODES MATHÉMATIQUES ET ALGORITHMES POUR LA PHYSIQUE

Résumé

Ce chapitre débute par un bref rappel sur des notions élémentaires portant sur la dérivée, la caractérisation de la convexité et de la concavité de fonctions mono et multi-variables. Ces notions fondamentales sont indispensables afin d'appréhender le fonctionnement des algorithmes d'optimisation. Nous commencerons par la description de l'algorithme de `recherche dichotomique` et celui de `Newton`. Cette étape servira à l'assimilation du principe de fonctionnement des algorithmes d'optimisation d'un point de vue général. Ensuite, il sera question d'aborder trois algorithmes (`Newton-Gauss`, `descente de gradient` et `Levenberg-Marquardt`) très largement utilisés, avec leur variante, pour résoudre des problèmes d'optimisation. Par ailleurs, la programmation de ces algorithmes sera conduite par le biais de scripts Matlab®.

Mots clés

Algorithmes d'optimisation, fonction-objectif, descente de gradient, script Matlab®.

Galilée disait " ... Le livre de la nature est écrit en langage mathématique "

Table des matières

I. INTRODUCTION

LEs algorithmes d'optimisation sans ou sous contraintes (unconstrained and constrained problem en anglais), de fonctions mathématiques uni et multidimensionnelles, ont pour objectif de chercher un vecteur $\hat{X}$ tel que $f(\hat{X})$ soit un extremum (minimum ou maximum) de la fonction en question. L'optimisation s'apparente systématiquement

S. Kenouche est docteur en Physique de l'Université de Montpellier et docteur en Chimie de l'Université de Béjaia.
Site web : voir http://www.sites.univ-biskra.dz/kenouche
Version améliorée et actualisée le 09.11.2018.

à une minimisation car trouver le maximum d'une fonction f revient à minimiser la fonction $-f$. Dans cette topologie, la fonction à optimiser est appelée *fonction-objectif*. Ainsi, l'opération d'optimisation consiste en la minimisation de la *fonction-objectif* selon :

$$\hat{X} \in \underset{X \in \mathbb{R}^n}{argmin} f(X) \tag{1}$$

Avec $\hat{X} = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$ sont les coordonnées du point critique. On fait appel aux algorithmes d'optimisation afin de résoudre des problèmes de différente nature, comme par exemple, trouver les zéros de fonctions non-linéaires, ajustement de données expérimentales selon le critère des *moindres carrés linéaire et non-linéaire*, résolution de systèmes d'équations à une ou plusieurs variables ... etc. En général, la recherche des extremums est atteinte en procédant au calcul des dérivées premières (gradient de la fonction) et des dérivées secondes (Hessien de la fonction). Toutefois, trouver un minimum global n'est pas toujours chose facile et il n'y a pas d'algorithme d'optimisation parfait. On aura plutôt tendance donc à choisir l'algorithme le plus adapté au problème considéré.

A. *Rappel mathématique*

Soit la fonction à une variable $f : \mathbb{I} \longrightarrow \mathbb{R}$. On définit sa fonction dérivée par l'expression :

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{2}$$

Pour un point x_0 apparentant à l'intervalle de définition de f, le nombre $f^{'}(x_0)$ exprime la pente, au point $(x_0, f(x_0))$, de la droite tangente à la courbe $y = f(x)$. Le rapport :

$$f^{'}(x_0) = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \tag{3}$$

vaut la pente de la droite passant par les points $(x_0, f(x_0))$ et $(x_0 + \Delta x, f(x_0 + \Delta x))$. Il en ressort que l'équation de la droite tangente à la courbe $y = f_T(x)$ au point (x_0, y_0) est :

$$y - y_0 = f^{'}(x_0)(x - x_0) \Longrightarrow f_T(x) = y_0 + f^{'}(x_0)(x - x_0) \tag{4}$$

Cette approximation est d'autant plus vraie que $\Delta x \longrightarrow 0$. Lorsque la limite de l'Eq. (3) existe, on dira que f est dérivable en x_0. Dans le cas où la fonction f est dérivable pour tout $x \in \mathbb{I}$, on dira qu'elle est dérivable sur $\mathbb{I}$ et on aura par conséquent la fonction $f^{'}(x)$. Cette dernière peut également être dérivable et on aura $f^{''}(x)$ ... etc

B. *Convexité et concavité d'une fonction*

Nous rappelons que la dérivée traduit les variations, croissance ($f^{'}(x) > 0$) et décroissance ($f^{'}(x) < 0$), d'une fonction. Une dérivée nulle, en un point x_0, est caractérisée par une tangente horizontale et donc la fonction présente un extremum (un minimum ou un maximum) en ce point. Cette condition est nécessaire pour l'existence d'un extremum, mais elle n'est pas suffisante. A titre d'exemple, la fonction $f(x) = x^3$ n'admet pas d'extremum local en $x_0 = 0$ bien que sa dérivée $f^{'}(x) = 3\,x^2$ s'y annule.

Définition 1 *: La courbe de $f(x)$ est dite* `convexe` *si tous les points de la courbe $y = f(x)$ se trouvant au dessous de la tangente en un point quelconque de l'intervalle de définition de la fonction, autrement dit : $\forall\, x \in [a, b]\, f(x) < f_T(x)$. Cette condition est satisfaite si $f^{''}(x) < 0$.*

Définition 1' *: La courbe de $f(x)$ est dite* `concave` *si tous les points de la courbe $y = f(x)$ se trouvant au dessus de la tangente en un point quelconque de l'intervalle de définition de la fonction, autrement dit : $\forall\, x \in [a, b]\, f(x) > f_T(x)$. Cette condition est satisfaite si $f^{''}(x) > 0$.*

Ainsi, les conditions $f^{'}(x_0) = 0$ et $f^{''}(x_0) > 0$ sont suffisantes pour que la fonction $f(x)$ admette un minimum (concavité). Les conditions $f^{'}(x_0) = 0$ et $f^{''}(x_0) < 0$ prouvent l'existence d'un maximum (convexité).

Pour une fonction à plusieurs variables $f : \mathbb{R}^2 \longmapsto \mathbb{R}$. Dans ce cas, le vecteur $\hat{X} \in \mathbb{R}^2$ est un minimum local si

$$\frac{\partial^2 f(X)}{\partial x^2} > 0 \quad \text{et le déterminant} \quad \begin{vmatrix} \dfrac{\partial^2 f(X)}{\partial x_1^2} & \dfrac{\partial^2 f(X)}{\partial x_1\, \partial x_2} \\ \dfrac{\partial^2 f(X)}{\partial x_2\, \partial x_1} & \dfrac{\partial^2 f(X)}{\partial x_2^2} \end{vmatrix} > 0 \tag{5}$$

Nous rappelons que la matrice *Hessienne*, notée H d'une fonction $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, est une matrice carrée de ses dérivées partielles secondes s'écrivant selon :

$$H_{ij}(f) = \frac{\partial^2 f}{\partial x_i\, x_j} = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\, \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\, \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\, \partial x_1} & \dfrac{\partial^2 f}{\partial x_n\, x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \tag{6}$$

Bien évidement cela suppose l'existence des dérivées partielles secondes de la fonction. La matrice *Hessienne* est dite semi-définie positive si

$$\forall X \in \mathbb{R}^n : X^T\, H\, X \geq 0 \tag{7}$$

Elle dite positive si

$$\forall X \in \mathbb{R}^n, X \neq 0 : X^T\, H\, X > 0 \tag{8}$$

L'existence d'un maximum local doit satisfaire les conditions :

$$\frac{\partial^2 f(X)}{\partial x^2} < 0 \quad \text{et le déterminant} \quad \begin{vmatrix} \dfrac{\partial^2 f(X)}{\partial x_1^2} & \dfrac{\partial^2 f(X)}{\partial x_1\, \partial x_2} \\ \dfrac{\partial^2 f(X)}{\partial x_2\, \partial x_1} & \dfrac{\partial^2 f(X)}{\partial x_2^2} \end{vmatrix} > 0 \tag{9}$$

On rappellera que pour une fonction à deux variables $f(x_1, x_2)$, nous avons la possibilité de constituer quatre dérivées partielles.

$$\frac{\partial^2 f}{\partial x_1^2} = \frac{\partial}{\partial x_1}\left(\frac{\partial f}{\partial x_1}\right) \quad ; \quad \frac{\partial^2 f}{\partial x_1\, \partial x_2} = \frac{\partial}{\partial x_1}\left(\frac{\partial f}{\partial x_2}\right)$$

$$\frac{\partial^2 f}{\partial x_2^2} = \frac{\partial}{\partial x_2}\left(\frac{\partial f}{\partial x_2}\right) \quad ; \quad \frac{\partial^2 f}{\partial x_2\, \partial x_1} = \frac{\partial}{\partial x_2}\left(\frac{\partial f}{\partial x_1}\right)$$

Cependant, seulement trois de ces dérivées sont distinctes. Ainsi, on démontre que

$$\frac{\partial^2 f}{\partial x_1\, \partial x_2} = \frac{\partial^2 f}{\partial x_2\, \partial x_1}$$

On saisit ainsi que pour une dérivée partielle du second ordre mixte, l'ordre dans lequel on calcule les dérivées n'a pas d'impact sur le résultat.

II. ALGORITHMES DE MINIMISATION USUELS

Comme il a été signalé précédemment, l'opération d'optimisation s'apparente systématiquement à une minimisation, car trouver le maximum d'une fonction f revient à minimiser la fonction $-f$. Nous entamerons cette section par l'étude de deux algorithmes d'optimisation les plus simples à savoir : la recherche dichotomique et l'algorithme de Newton. Ces derniers auront pour intérêt d'illustrer le fonctionnement des algorithmes de minimisation multidimensionnelle que nous verrons dans les prochaines sections.

A. *Algorithme de recherche dichotomique*

La méthode est décrite comme suit : soit $f : [a\ b] \longrightarrow R$, une fonction unimodale de classe $\mathcal{C}^1$ sur $[a\ b]$. Si $f^{'}(a) < 0 < f^{'}(b) \Longrightarrow$ il existe donc au moins un $x^* \in [a\ b]$ pour lequel la dérivée $f^{'}(x^*) = 0$. On prend $c = \dfrac{a+b}{2}$ la moitié de l'intervalle $[a\ b]$ telle que :

1) Si $f^{'}(c) = 0 \longrightarrow c$ est le minimum de la fonction $f(x)$.
2) Sinon, nous testons le signe de $f^{'}(c) < 0$ et de $f^{'}(c) > 0$.
3) Si $f^{'}(c) < 0 \longrightarrow$ le minimum se trouve dans l'autre moitié, l'intervalle $[c\ b]$ qui est la moitié de $[a\ b]$.
4) Si $f^{'}(c) > 0 \longrightarrow$ le minimum se trouve dans l'autre moitié, l'intervalle $[a\ c]$ qui est la moitié de $[a\ b]$.

Ce processus de division par deux de l'intervalle (à chaque itération on divise l'intervalle par deux) de la fonction dérivée est réitéré jusqu'à la convergence pour la tolérance considérée.

Exercice 1

Soit la fonction-objectif ci-dessous :

$$f(x) = \cos(2\,x) + \sqrt{x^2 + 1} \tag{10}$$

Elle est unimodale de classe $\mathcal{C}^2$ sur $[-3\ 0]$.

1) Écrire un script Matlab® de l'algorithme de recherche dichotomique permettant la minimisation de la fonction-objectif.

Solution 1

```
clear all ; clc ; close all ;
%%%%%%%%%%%%%%%%%%%% Algoritme de recherche dichotomique %%%%%%%%%%%%%%%%%%%%
% Samir Kenouche - Le 22/10/2018
x = -3:0.004: 0 ; it = 0 ; itmax = 20 ; lB = -3 ; uB = 0 ; errs = 10.^(-4) ;

fx = @(x) cos(2.*x) + sqrt(x.^2 + 1) ;          % FONCTION-OBJECTIF
dfx = @(x) x./sqrt((x.^2 + 1)) - 2.*sin(2.*x) ; % SA DERIVEE

while it < itmax
          center = (lB + uB)/2 ;  it = it + 1 ;

          if dfx(center) == 0 ;
                 lemin = center ;

          elseif dfx(center) < 0
                 lB = center ;
```

```
        else
                uB = center ;
        end

                sol(it+1) = center ;

       if abs(sol(it+1) - sol(it)) <= errs     % TEST D'ARRET

              lemin = fx(center) ;
              iteration_max = it ;             % NOMBRE D'ITERATION MAX
       break
       end

    plot(x, fx(x)) ; hold on ; plot(center, fx(center),'ro')  ;
   quiver(center,fx(center),dfx(center),0,0.6,'Color','k','LineWidth',1) ;
end
```

L'avantage majeur de cet algorithme est sa simplicité. C'est une procédure intuitive pour atteindre la minimisation d'une fonction-objectif au moyen d'une recherche dichotomique. L'inconvénient de cet algorithme repose sur l'imposition d'une fonction-objectif unimodale. Dans le cas contraire, il convergera vers un minimum local dans le cas où un minimum global existe. Par ailleurs, Matlab® dispose d'une boîte à outil (Symbolic Math Toolbox) dédiée au calcul symbolique. La fonction dérivée de l'exercice a été calculée suivant le script :

```
clear all ; clc ; close all ;
% Samir Kenouche - Le 22/10/2018
%%%%%%%%%%%%%%%%%%%%%%%%%% CALCUL FORMEL DES DERIVEES %%%%%%%%%%%%%%%%%%%%%%%%%%
syms x
fx = x*cos(2*x) + sqrt(x^2 + 1)

dfx = diff(fx, x, 1) ; % EXPRESSION SYMBOLIQUE DE LA 1ERE DERIVEE
pretty(dfx)            % IMRESSION DE L'EXPRESSION MATHEMATIQUE DE dfx

ddfx = diff(fx, x, 2) ;% EXPRESSION SYMBOLIQUE DE LA 2EME DERIVEE
pretty(ddfx)           % IMRESSION DE L'EXPRESSION MATHEMATIQUE DE ddfx
```

B. Algorithme de Newton

Soit $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ une fonction de classe $\mathcal{C}^2$. Le développement en série de Taylor d'ordre deux (modèle quadratique) est une fonction $P_{x_h}(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$, avec :

$$P_{x_h}(x) = f(x_h) + (x - x_h)^T \nabla f(x_h) + (x - x_h)^T \nabla^2 f(x_h) (x - x_h) \tag{11}$$

Avec, $\nabla^2 f(x_h)$ est la matrice Hessienne de f en $x = x_h$. Posons $u = x - x_h$, il vient :

$$P_{x_h}(x + u) = f(x_h) + u^T \nabla f(x_h) + u^T \nabla^2 f(x_h) u \tag{12}$$

$$\min\{P_{x_h}(x + u)\} \Longleftrightarrow \nabla P_{x_h}(x + u) = 0 \tag{13}$$

Une condition suffisante d'optimalité :

$$\nabla P_{x_h}(x + u) = \nabla f(x_h) + \nabla^2 f(x_h) u = 0 \tag{14}$$

$$\nabla f(x_h) = -u\,\nabla^2 f(x_h) \Rightarrow u = -\frac{\nabla f(x_h)}{\nabla^2 f(x_h)} \tag{15}$$

$$x = x_h - \frac{\nabla f(x_h)}{\nabla^2 f(x_h)} \quad \text{avec} \quad u = x - x_h \tag{16}$$

On calcule alors un nouveau point, x_{h+1}, qui minimise P_{x_h} soit :

$$x_{h+1} = x_h - \frac{\nabla f(x_h)}{\nabla^2 f(x_h)} \tag{17}$$

Afin d'illustrer le fonctionnement de cet algorithme, reprenant la fonction-objectif de l'exercice précédent.

Exercice 2

Soit la fonction-objectif ci-dessous :

$$f(x) = \cos(2\,x) + \sqrt{x^2 + 1} \tag{18}$$

Elle est unimodale de classe $\mathcal{C}^2$ sur $[-3\ 0]$.

1) Écrire un script Matlab® de l'algorithme de Newton permettant la minimisation de la fonction-objectif.

Solution 2

```
clear all ; clc ; close all ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Algoritme de Newton %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Samir Kenouche - LE 22/10/2018
x = -3:0.004: 0 ; fx = @(x) cos(2.*x) + sqrt(x.^2 + 1) ;

dfx = @(x) x./sqrt((x.^2 + 1)) - 2.*sin(2.*x) ;
ddfx = @(x) 1./sqrt((x.^2 + 1)) - 4.*cos(2.*x) - x.^2./(x.^2 + 1).^(3/2) ;

it = 0 ; itmax = 20 ; lB = -3 ; uB = 0 ; xinit = -2.00 ; errs = 10.^(-4) ;

while it < itmax

                xn = xinit - (dfx(xinit)./ddfx(xinit)) ;
                xinit = xn ;

                it = it + 1 ; sol(it+1) = xn ;

          if abs(sol(it+1) - sol(it)) <= errs     % TEST D'ARRET

                xmin = fx(xn) ;
                iteration_max = it ;              % NOMBRE D'ITERATION MAX
          break
          end
```

```
    plot(x, fx(x)) ; hold on ; plot(xn, fx(xn),'ro')  ;
    quiver(xn,fx(xn),dfx(xn),0,0.6,'Color','k','LineWidth',1) ;
end

xmin = xn
```

Nonobstant sa simplicité, cet algorithme souffre d'un certain nombres d'inconvénients : il peut diverger si le point initial est trop éloigné de la solution recherchée. En outre, l'algorithme ne peut être utilisé si la fonction-objectif n'est pas deux fois dérivables.

Dans les sections suivantes, nous appliquerons les algorithmes de Gauss-Newton, de descente de gradient et de Levernberg-Marquartd au cas d'une fonction-objectif répondant au critère des moindres carrés non-linéaires. Dans cette optique, la fonction-objectif prendra la forme :

$$f(\theta_0, \theta_1, ..., \theta_p; x_i) = \sum_i^n \left[\left(\sum_{k=0}^{p} \theta_k \, x_i \right) - y_i \right]^2 \tag{19}$$

Avec $f(\theta_0, \theta_1, ..., \theta_p; x_i)$ est la fonction-objectif à minimiser, $\theta_n = (\theta_0, \theta_1, ..., \theta_p)$ est le vecteur des paramètres optimaux et x_i, y_i sont des mesures expérimentales.

C. *Algorithme de descente de gradient*

Cet algorithme est à la base des processus d'optimisation. Le terme optimisation est utilisé dans le sens de la minimisation ou la maximisation d'une *fonction-objectif*. Rappelons que les deux opérations sont équivalentes, ainsi :

$$\text{Arg}\max_x f(x) \Leftrightarrow \text{Arg}\min_x(-f(x)) \tag{20}$$

Le terme *descente* vient du fait que cette méthode recherche l'extremum suivant une direction opposée à celle du gradient de la *fonction-objectif*. Nous rappellerons dans un premier temps la notion de la dérivée directionnelle. Cette étape est importante afin d'appréhender le fonctionnement d'un algorithme de descente de gradient.

1) ***Dérivée directionnelle****:* Nous souhaitons quantifier le taux de variation de la fonction $f : \mathbb{R}^2 \longmapsto \mathbb{R}$ lorsqu'elle passe du point $f(x_0, y_0)$ au point $f(x, y)$. Nous travaillerons sur le plan de projection xoy, ce taux de variation est évalué par le segment de droite $\overrightarrow{P_0P}$. Soit $\overrightarrow{d} = a\,\vec{i} + b\,\vec{j}$ un vecteur unitaire ayant la même direction que $\overrightarrow{P_0P}$.

Sur la figure ci-dessus :

$$\begin{aligned} \overrightarrow{P_0P} // \overrightarrow{d} &\Rightarrow \overrightarrow{P_0P} = \lambda \overrightarrow{d} \quad \text{avec} \quad \lambda \in \mathbb{R}_+^* \\ &\Rightarrow \overrightarrow{P_0P} = \lambda\,(a\,\vec{i} + b\,\vec{j}) = \lambda\,a\,\vec{i} + \lambda\,b\,\vec{j} \end{aligned} \tag{21}$$

D'un autre côté on peut définir le vecteur $\overrightarrow{P_0P}$ par rapport à l'origine O selon :

$$\overrightarrow{P_0P} = \overrightarrow{OP} - \overrightarrow{OP_0} \tag{22}$$

$$= (x\,\vec{i} + y\,\vec{j}) - (x_0\,\vec{i} + y_0\,\vec{j}) \tag{23}$$

$$= x\,\vec{i} + y\,\vec{j} - x_0\,\vec{i} - y_0\,\vec{j} \tag{24}$$

$$= (x - x_0)\,\vec{i} + (y - y_0)\,\vec{j} \tag{25}$$

Par identification des équations (21) et (25), il vient :

$$\begin{cases} x - x_0 = \lambda\,a \\ y - y_0 = \lambda\,b \end{cases} \Rightarrow \begin{cases} x = x_0 + \lambda\,a \\ y = y_0 + \lambda\,b \end{cases} \tag{26}$$

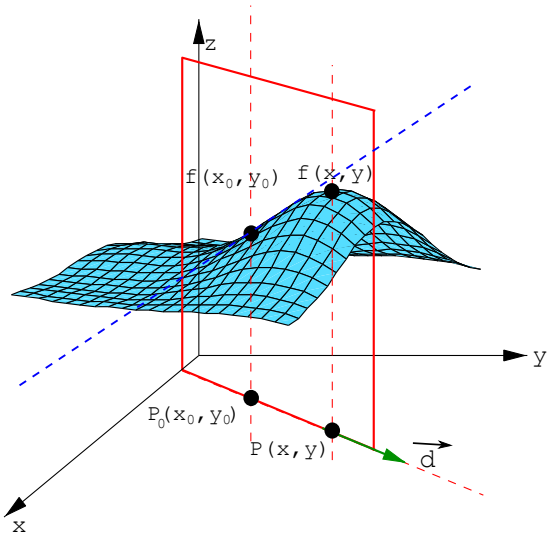

FIGURE 1: Dérivée directionnelle au point P_0 dans la direction $\vec{d}$.

Par voie de conséquence, la dérivée directionnelle de $f(x,y)$ dans la direction du vecteur unitaire $\overrightarrow{d} = a\,\vec{i} + b\,\vec{j}$ au point $f(x_0, y_0)$ est :

$$f_{\overrightarrow{d}}(x_0, y_0) = \lim_{\lambda \to 0} \frac{f(x_0 + \lambda\, a, y_0 + \lambda\, b) - f(x_0, y_0)}{\lambda} \tag{27}$$

Théorème : La dérivée directionnelle est maximale lorsque $\overrightarrow{d}$ a la même direction que $\nabla f(x_0, y_0)$ de plus le taux de variation maximal de $f(x,y)$ en (x_0, y_0) est $||\nabla f(x_0, y_0)||$.

Preuve

$$\begin{aligned} f_{\overrightarrow{d}}(x_0, y_0) &= \nabla f(x_0, y_0)\ \overrightarrow{d} \\ &= ||\nabla f(x_0, y_0)||\, ||\overrightarrow{d}||\, \cos(\theta) \\ &= ||\nabla f(x_0, y_0)||\, \cos(\theta) \end{aligned}$$

Ainsi $f_{\overrightarrow{d}}(x_0, y_0)$ est maximale si $\cos(\theta) = \pm 1$ autrement dit si la condition $\nabla f(x_0, y_0) // \overrightarrow{d}$ est satisfaite. Dans le cas où $\theta = \pi/2 \Rightarrow \nabla f(x_0, y_0) \perp \overrightarrow{d}$. Ce résultat indique que si je me déplace dans une direction perpendiculaire au ∇f, le taux de variation de la fonction $f(x,y)$ est nul.

- Si les deux vecteurs ∇f et $\overrightarrow{d}$ ont la même direction et le même sens, dans ce cas le vecteur unitaire $\overrightarrow{d}$ désigne une direction de croissance maximale de $f(x,y)$.
- Si les deux vecteurs ∇f et $\overrightarrow{d}$ ont la même direction et de sens opposé, dans ce cas le vecteur unitaire $\overrightarrow{d}$ désigne une direction de décroissance maximale de $f(x,y)$.

Preuve :

$$\begin{aligned}\nabla f(x_0,y_0)\times \overrightarrow{d} &= \nabla f(x_0,y_0)\times(-\nabla f(x_0,y_0))\\ &= -\nabla f(x_0,y_0)\times\nabla f(x_0,y_0)\\ &= \underbrace{-\underbrace{||\nabla(x_0,y_0)||^2}_{>0}}_{<0}\\ &\Rightarrow \overrightarrow{d} = -\nabla f \quad \text{est une direction de descente}\end{aligned}$$

Exercice d'application

Calculer la dérivée directionnelle de la fonction $f(x,y) = 2\,e^{(x^2\,y)}$ au point (2, 3) dans la direction formant un angle de $75°$ avec l'axe des x positif.

solution

$$\begin{aligned}f_{\overrightarrow{d}}(2,3) &= \nabla f(2,3)\ \overrightarrow{d}\\ &= \frac{\partial f}{\partial x}(2,3)\times\cos(75°) + \frac{\partial f}{\partial y}(2,3)\times\sin(75°)\\ &= 4\,x\,y\,e^{(x^2\,y)}\times\cos(75°) + 2\,x^2\,e^{(x^2\,y)}\times\sin(75°)\\ &= 24\,e^{(4\times3)}\times 0.25 + 8\,e^{(4\times3)}\times 0.96\\ &= 6\,e^{(12)} + 7.70\,e^{(12)}\\ &= 13.70\,e^{(12)}\end{aligned}$$

2) ***Schéma numérique de la descente de gradient****:* L'idée de base consiste à chercher une suite $\{x_k\}_{k\in\mathbb{N}} \in \mathbb{R}^n$ dont le successeur de x_k doit satisfaire la condition :

$$f(x_{k+1}) < f(x_k) \tag{28}$$

Afin d'établir cette suite, on exploitera la dérivée directionnelle définie précédemment. Considérant la figure ci-dessous :

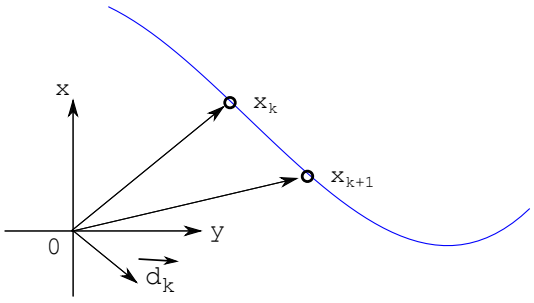

FIGURE 2: Dérivée directionnelle à une dimension dans la direction $\vec{d}$.

Compte tenu de l'Eq. (27), à une dimension on obtient :

$$f_{\overrightarrow{d}}(x_k) = \lim_{\lambda \to 0} \frac{f(x_{k+1}) - f(x_k)}{\lambda} \Rightarrow f_{\overrightarrow{d}}(x_k) = \lim_{\lambda \to 0} \frac{f(x_k + \lambda\, a) - f(x_k)}{\lambda} \tag{29}$$

Comme précédemment, nous avons

$$\overrightarrow{x_k x_{k+1}} // \overrightarrow{d_k} \Rightarrow \overrightarrow{x_k x_{k+1}} = \lambda\, \overrightarrow{d_k} \quad \text{avec} \quad \lambda \in \mathbb{R}_+^*$$

D'un autre côté, nous avons :

$$\begin{aligned} \overrightarrow{x_k x_{k+1}} &= \overrightarrow{Ox_{k+1}} - \overrightarrow{Ox_k} \Rightarrow \overrightarrow{Ox_{k+1}} \\ &= \overrightarrow{Ox_k} + \overrightarrow{x_k x_{k+1}} \Rightarrow \overrightarrow{Ox_{k+1}} \\ &= \overrightarrow{Ox_k} + \lambda\, \overrightarrow{d_k} \end{aligned}$$

D'où le schéma numérique de l'algorithme de descente de gradient :

$$x^{(k+1)} = x^{(k)} + \lambda\, d_k = x^{(k)} - \lambda \nabla f(x^{(k)}), \qquad \lambda > 0 \tag{30}$$

Ainsi,

$$f(x_{k+1}) < f(x_k) \Leftrightarrow f(x^{(k)} - \lambda \nabla f(x^{(k)}) < f(x_k) \tag{31}$$

Avec, $d_k = -\nabla f(x^{(k)})$ est la direction de plus forte descente de f au point x_k. Le paramètre λ est le pas de la descente qui peut être constant (*Méthode de gradient à pas fixe*) ou variable. La vitesse de convergence de l'algorithme est proportionnelle à la valeur de λ. Ce paramètre peut être incrémenté selon les méthodes de *Wolfe*, de *gradient à pas optimal* ($\min_{\lambda>0} f(x^{(k)} + \lambda_k \nabla f(x^{(k)}))$) et de *gradient conjugué*. Typiquement, le pas de descente est obtenu au moyen d'une recherche linéaire vérifiant : $f(x^{(k)} + \lambda_k \nabla f(x^{(k)})) < f(x^{(k)})$. Une valeur initiale pour incrémenter l'algorithme est $\lambda = 0.01$. Le gradient indique la direction de plus grande pente. Cette méthode est itérative, donc elle a besoin d'une valeur initiale pour démarrer l'incrémentation de l'algorithme.

Avant d'appliquer cet algorithme pour l'ajustement de données expérimentales, nous commencerons d'abord par chercher le minimum d'équations non-linéaires. La démarche est strictement la même, dans le premier cas on recherche un vecteur de paramètres minimisant la *fonction-objectif* et dans le second cas on recherche plutôt la variable x minimisant la fonction en question.

Évaluation numérique

Cherchons la minimisation de la fonction-objectif :

$$f(x) = 4\, x^2 + e^x \quad \text{avec} \quad \nabla f(x) = 8\, x + e^x$$

Cette fonction est au moins de classe $\mathcal{C}^1$ sur $\mathbb{R}$. Exécutant le schéma numérique de l'algorithme de descente de gradient à pas fixe pour $x_0 = 1.50000$, $\lambda = 0.02$ et $\epsilon = 10^{-3}$, soit :

$$x^{(k+1)} = x^{(k)} + \lambda\, d_k = x^{(k)} - \lambda \nabla f(x^{(k)})$$

Solution

$$
\begin{aligned}
x^{(1)} &= 1.500000 - 0.02\,\nabla f(1.500000) = 1.170366 \\
x^{(2)} &= 1.170366 - 0.02\,\nabla f(1.170366) = 0.918644 \\
x^{(3)} &= 0.918644 - 0.02\,\nabla f(0.918644) = 0.721543 \\
x^{(4)} &= 0.721543 - 0.02\,\nabla f(0.721543) = 0.564944 \\
x^{(5)} &= 0.564944 - 0.02\,\nabla f(0.564944) = 0.439366 \\
\vdots \quad & \quad \vdots \quad \quad \vdots \quad \quad \vdots \\
\vdots \quad & \quad \vdots \quad \quad \vdots \quad \quad \vdots \\
x^{(30)} &= -0.10695 - 0.02\,\nabla f(-0.10695) = -0.10781
\end{aligned}
$$

Le test d'arrêt $|x^{(k+1)} - x^{(k)}| < \epsilon$ est positif au bout de la $30^{\text{ème}}$ itération, soit :

$$|x^{(30)} - x^{(29)}| = 8.5000 \times 10^{-4} < \epsilon$$

L'algorithme converge vers la solution $x^{(*)} = -0.10781$ correspondant au minimum de la fonction-objectif.

Exercice 3

Soit la fonction-objectif ci-dessous :

$$\begin{cases} f(x) = \cos(2\,x)\,\sqrt{x^2+1} \\ \text{Avec} \quad x_0 = -2,\ x \in [-4, 0] \end{cases} \tag{32}$$

1) Écrire un script Matlab® de l'algorithme de descente de gradient permettant la minimisation de la fonction-objectif.

Solution 3

```
clear all ; close all ; clc ;
% Le 23.10.2018 - Samir Kenouche
%%%%%%%%%%%%%%%%%%%%%%%% DSCENTE DE GRADIENT A PAS FIXE %%%%%%%%%%%%%%%%%%%%%%%%
syms x
fun = cos(2*x) + sqrt(x^2 + 1) ; dfun = diff(fun,x) ;
xi = -4:0.004: 0 ; fx = subs(fun, x, xi) ; dfx = subs(dfun, x, xi) ;

xinit = -3 ; lambda = 0.1/2 ; itmax = 100 ; it = 0 ; echelle = 0.5 ;
tol =  1e-3 ; plot(xi,fx,'LineWidth',1) ; hold on ;

 while it < itmax

    J = subs(dfun, x, xinit) ;
    xn = xinit - lambda*J ; xinit = xn ;

    it = it + 1;

    fxn = subs(fun, x, xn) ; dfxn = subs(dfun , x, xn) ;
```

```
    if abs(lambda*J) < tol
         sol = xn
         break
    end

    dk = - dfxn ; % DIRECTION DE DESCENTE
    plot(xn, fxn,'xr','MarkerSize',7) ; hold on ;
    plot(xn, fxn,'or','MarkerSize',7) ; hold on ;
   quiver(xn,fxn,dk,0, echelle,'Color', 'k','LineWidth',1) ; hold on;
 end

h = gca ;
str(1) = {'Plot of the function :'};
str(2) = {'$$y = \cos(2\,x) + \sqrt{x^2 + 1}$$'};
str(3) = {'With the minimum:'};
str(4) = {['$$x_{min} = $$', num2str(sol)]};
set(gcf,'CurrentAxes',h) ;

text('Interpreter','latex', 'String',str,'Position',[-3.9 1.5],'FontSize',12)
```

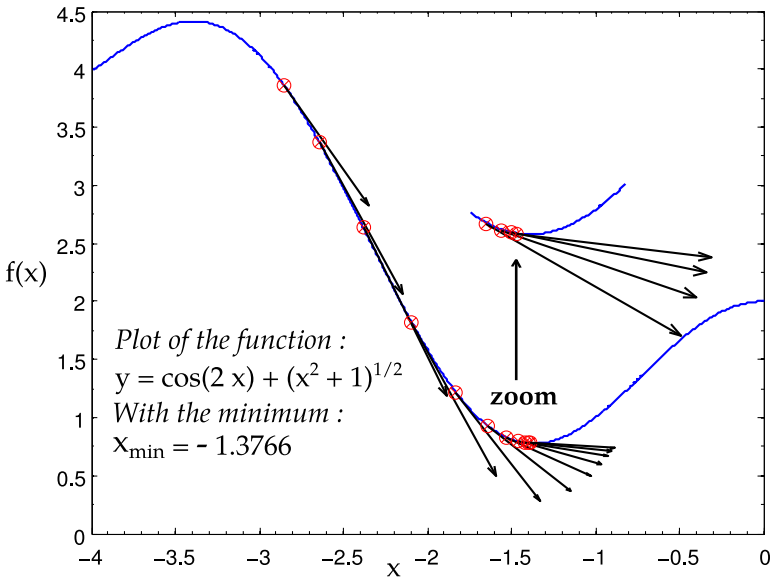

FIGURE 3: Minimisation par la méthode de la *descente de gradient à pas fixe*

Le minimum renvoyé par ce script est : `sol = -1.3668`. L'inconvénient de cette méthode est qu'elle réclame une valeur initiale très proche de la solution approchée. Dans le cas contraire, l'algorithme ne convergera pas vers le véritable minimum de la fonction. L'exemple ci-dessous illustre ce propos

Cette courbe (Fig. (4)) est générée avec le même script Matlab® appliqué à la fonction : $\cos(3\,x) + \sqrt{x^2+1}$. Dans le script Matlab® ci-dessous, nous illustrons l'influence du paramètre λ sur la convergence de l'algorithme vers la solution approchée.

D. Algorithme de descente de gradient à pas optimal

L'inconvénient d'utiliser un pas $\lambda = cst$ est que l'algorithme converge très lentement si le pas est trop petit. De plus, l'algorithme peut devenir instable dans le cas où le pas est trop grand. On comprend donc que le choix optimal

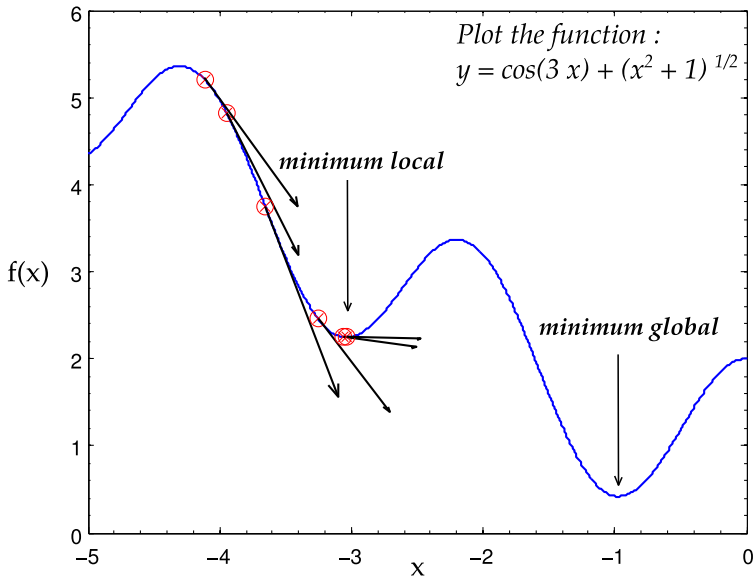

FIGURE 4: Convergence vers un minimum local de la descente de gradient à pas fixe

de la valeur de λ devient délicat dans ce cas de figure. Il faudra donc ajuster la valeur de λ à chaque itération. Cette procédure est appelée *recherche en ligne* (line search). Nous allons appliquer la méthode de *gradient à pas optimal*. L'idée de base de cette méthode est de chercher le λ_k diminuant d'avantage $f(x^{(k)})$ dans la direction $d_k = -\nabla f(x^{(k)})$, selon :

$$\varphi(\lambda_k) = \min_{\lambda>0} f(x^{(k)} - \lambda_k \nabla f(x^{(k)})) \Leftrightarrow \varphi^{'}(\lambda_k) = 0 \Leftrightarrow \varphi(\lambda_{opt}) \leq \varphi(\lambda_k) \tag{33}$$

Il convient de noter, qu'en toute rigueur le pas de descente n'est pas λ mais $\lambda\, d_k$. L'opération d'optimisation du pas de descente permet de répondre à la question : quelle distance doit-on parcourir ? Nous appliquerons cette méthode pour chercher le minimum de la fonction :

$$\begin{cases} f(x,y) = \dfrac{1}{2}\,x^2 + \dfrac{7}{2}\,y^2 \\ \text{Avec} \quad x_0 = (7.5, 2.2) \end{cases} \tag{34}$$

Nous déterminera dans un premier temps, l'expression analytique de $\varphi(\lambda_k)$. La descente de gradient (à deux dimension) impose :

$$\begin{aligned} X^{(k+1)} &= X^{(k)} - \lambda \nabla f(X^{(k)}), \quad \text{avec} \quad X^{(k)} = (x^{(k)}, y^{(k)}) \quad \text{et} \quad X^{(k+1)} = (x^{(k+1)}, y^{(k+1)}) \\ &= (x^{(k)}, y^{(k)}) - \lambda\,(x^{(k)}, 7\,y^{(k)}) \\ &= (x^{(k)}, y^{(k)}) - (\lambda\,x^{(k)}, 7\,\lambda\,y^{(k)}) \\ &= \left[(x^{(k)} - \lambda\,x^{(k)}), (y^{(k)} - 7\,\lambda\,y^{(k)})\right] \\ &= \left[x^{(k)}\,(1-\lambda), y^{(k)}\,(1-7\,\lambda)\right] \end{aligned}$$

Nous en déduisons ainsi :

$$\begin{aligned} \varphi(\lambda_k) &= f\left[x^{(k)}\,(1-\lambda_k), y^{(k)}\,(1-7\,\lambda_k)\right] \\ &= \frac{1}{2}\,(x^{(k)})^2\,(1-\lambda_k)^2 + \frac{7}{2}\,(y^{(k)})^2\,(1-7\lambda_k)^2 \end{aligned}$$

calculant désormais la dérivée :

$$\begin{aligned}
&\varphi^{'}(\lambda_k) = 0 \\
&\varphi^{'}(\lambda_k) = \frac{1}{2}\,(x^{(k)})^2\,(\lambda_k - 1) + \frac{7^2}{2}\,(y^{(k)})^2\,(7\,\lambda_k - 1) = 0 \\
&\Longrightarrow \lambda_k = \frac{(x^{(k)})^2 + 7^2\,(y^{(k)})^2}{(x^{(k)})^2 + 7^3\,(y^{(k)})^2}
\end{aligned}$$

Il en résulte que le schéma numérique de l'algorithme de descente de gradient à pas optimal prend la forme :

$$(x^{(k+1)}, y^{(k+1)}) = (x^{(k)}, y^{(k)}) - \frac{(x^{(k)})^2 + 7^2\,(y^{(k)})^2}{(x^{(k)})^2 + 7^3\ (y^{(k)})^2}\,(x^{(k)}, 7\,y^{(k)}) \tag{35}$$

Voici le script Matlab® :

```
clear all ; close all ; clc ;
% Le 08.11.2018 - Samir Kenouche
%%%%%%%%%%%%%%%%%%%%%%%% DSCENTE DE GRADIENT A PAS optimal %%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALCUL SYMBOLIQUE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
lambda = sym('lambda','positive') ;  xk = sym('xk','real') ;
yk = sym('yk','real') ;

phi = (1/2)*xk^2*(1 - lambda)^2 + (7/2)*yk^2*(1 - 7*lambda)^2 ;
lambda_opt = solve(phi==0, lambda) ;
lambda_opt = real(lambda_opt(1,1)) ;

%%%%%%%%%%%%%%%%%%%%%%%% FIN DU CALCUL SYMBOLIQUE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fxy = @(x,y) x.^2/2 + y.^2*(7/2) ; dfx = @(x) x ; dfy = @(y) 7.*y ;
it = 0 ; itmax = 40 ; echelle = 0.6 ;  tol =  1e-6 ; xinit = [7.5 2.2] ;

x = -2:0.2: 8 ; y = -6:0.2:6 ; [xgrid, ygrid] = meshgrid(x,y) ;
zgrid =  (1/2)*xgrid.^2 + (7/2)*ygrid.^2 ;

figure('color',[1 1 1]) ; contourf(xgrid,ygrid,zgrid) ; hold on ;
xlabel('x') ; ylabel('y') ;

 while it < itmax

       dk =  - [dfx(xinit(1)) dfy(xinit(2))] ;
       pas_optimal = subs(lambda_opt, {'xk','yk'}, ...
        {xinit(1) xinit(2)}) ;

       xn = xinit +  pas_optimal*dk ;
       xinit = xn ;

     %%%%%%%%%%%%%%%%%%%%%%%% TEST D'ARRET %%%%%%%%%%%%%%%%%%%%%%%%%%%%
      if abs((dfx(xinit(1)) + dfy(xinit(2)))*pas_optimal) <= tol

           xmin = xn ; it_number = it ;
```

```
        break
      end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  plot(xinit(1), xinit(2),'xw','MarkerSize',7) ; hold on ;
  plot(xinit(1), xinit(2),'ow','MarkerSize',7) ; hold on ;
  quiver(xn(1), xn(2), -dfx(xn(1)),- dfy(xn(2)), echelle,'Color', 'r', ...
     'LineWidth',1) ; hold on;
  it = it + 1 ;
 end
```

L'affichage graphique généré par ce script Matlab® est :

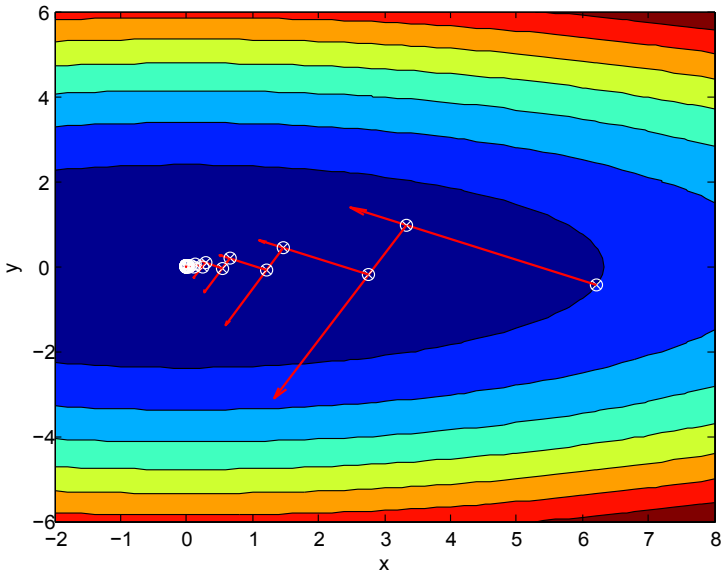

FIGURE 5: Minimisation par la méthode de la *descente de gradient à pas optimal*

Dans le direction $\overrightarrow{d_k}$, à l'itération $x^{(k+1)}$ l'algorithme calcule le minimum de $\nabla f(x^{(k+1)})$, soit

$$\varphi(\lambda) = d_k \, \nabla f(x^{(k+1)}) \tag{36}$$

La recherche du minimum impose de calculer la dérivée :

$$\varphi^{'}(\lambda) = 0 \Leftrightarrow < d_k | \nabla f(x^{(k+1)}) >= 0 \Leftrightarrow - < \nabla f(x^{(k)}) | \nabla f(x^{(k+1)}) >= 0 \tag{37}$$

Ainsi, le produit scalaire des deux gradients est nul, par conséquent les directions de descente successives calculé par l'algorithme sont orthogonales. Ceci explique pourquoi la convergence suit une trajectoire en zigzag à angles droits. Par ailleurs, l'expression symbolique générée par ce script Matlab® est :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALCUL SYMBOLIQUE %%%%%%%%%%%%%%%%%%%%%%%%%%%%
lambda = sym('lambda','positive') ;  xk = sym('xk','real') ;
yk = sym('yk','real') ;

phi = (1/2)*xk^2*(1 - lambda)^2 + (7/2)*yk^2*(1 - 7*lambda)^2 ;
lambda_opt = solve(phi==0, lambda) ;
```

```
lambda_opt = real(lambda_opt(1,1)) ;
pretty(lambda_opt)

>>
          2         2
        xk  + 49 yk
       -------------
         2          2
       xk  + 343 yk
```

L'avantage d'optimiser le pas de descente de gradient rend l'algorithme moins sensible, par rapport au pas fixe, au choix de la valeur initiale. La vitesse de convergence est améliorée également. Néanmoins, cette méthode peut se révéler moins efficace dans le cas où la fonction est caractérisée par des pentes peu marquées. L'autre inconvénient tient au fait qu'il est difficile voir impossible de trouver une expression analytique $\lambda_k = f(x_k)$ pour des fonctions plus complexes. Chaque méthode est adaptée pour un problème spécifique.

On s'intéressera désormais à l'optimisation d'une *fonction-objectif* à plusieurs variables. Nous appliquerons l'algorithme de *descente de gradient* pour l'ajustement, au sens des moindres carrés non-linéaires, de données expérimentales par un modèle théorique. Dans ce type d'optimisation, la non-linéarité est associée aux paramètres à estimer et non pas à la variable indépendante. L'algorithme prend la forme :

$$\begin{cases} -\ \delta_0 \quad \text{valeurs initiales} \\ -\ \delta_n = \delta_0 - \lambda\, F^t\, J \\ -\ \delta_0 = \delta_n \quad \text{mise à jour} \\ -\ \text{critère d'arrêt} \end{cases} \tag{38}$$

Avec, F^t est le vecteur transposé des résidus.

Exercice ❶ ☞ ®

1) À partir des données expérimentales (x_i, y_i) ci-dessous, déterminer avec la méthode de *descente de gradient à pas fixe* les paramètres optimaux du modèle suivant :

$$f(x) = \frac{1}{\theta_1\, x + \theta_2}$$

2) Afficher, sur la même figure, les données expérimentales et la courbe théorique.

Voici le script Matlab® :

```
clear all ; clc ; close all ;
% Le 24.10.2018 - Samir Kenouche
%%%%%%%%%%%%%%%%%%%%%%%%% DONNEES EXPERIMENTALES %%%%%%%%%%%%%%%%%%%%%%%
xi = 0:0.18 :11 ;
yi = [1.22 9.63e-01 8.31e-01 5.73e-01 4.46e-01 3.96e-01 3.83e-01 ...
 4.92e-01  1.81e-01 2.70e-01 3.13e-01 3.48e-01 1.50e-01 2.23e-01 ...
 2.32e-01 2.14e-01 2.34e-01 1.77e-01 1.09e-01 1.32e-01 1.65e-01 ...
 2.93e-01 1.92e-01 1.51e-01 1.69e-01 1.821e-01 3.08e-01 1.52e-01 ...
 2.73e-01 1.49e-01 1.95e-01 1.69e-01 2.84e-01 1.14e-02 1.07e-02 ...
 1.37e-01 1.88e-01 1.10e-01 7.23e-02 2.39e-01 2.61e-02 1.67e-01 ...
```

```
 2.77e-01 1.22e-01 -5.31e-03 3.22e-01 -6.50e-03 1.040e-01 1.56e-01 ...
 1.95e-01 6.53e-02 1.34e-01 -1.07e-01 7.38e-02 -9.27e-03 -1.03e-01 ...
 1.74e-01 2.09e-01 8.60e-02 -4.83e-02 3.01e-01 1.9277162e-02];

err = 0.2.*randn(1,numel(xi)) ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
syms x a b
modelFun = 1/(a*x + b) ; dfuna = diff(modelFun, a) ;
dfunb = diff(modelFun, b) ; idelta(1,1) = 1.45; idelta(1,2) = 0.89;
da =  subs(dfuna, x, xi) ;  db =  subs(dfunb, x, xi) ;
it = 0 ; itmax = 100 ; lambda = 0.005 ; tol = 0.1 ;

while it < itmax
Jacob(1:numel(xi), 1) = subs(da,{'a','b'},{idelta(1) idelta(2)}) ;
Jacob(1:numel(xi), 2) = subs(db,{'a','b'},{idelta(1) idelta(2)}) ;

    F = yi - (1./ (idelta(1).*xi + idelta(2))) ;
    ndelta = idelta - lambda*F*Jacob ; idelta = ndelta ;

     if mean(abs(F)) < tol
        sol = ndelta ;
        break
     end

    it = it + 1;
end

xn = xi(1) : 0.01 : xi(end) ; model = (1./ (sol(1).*xn + sol(2))) ;

%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%%%
figure('color',[1 1 1]) ;
plot(xn, model,'r','LineWidth',1) ; hold on
errorbar(xi,yi,err,'o') ; hold on ;
xlabel('xdata'); ylabel('ydata');
legend({'Ajustement', 'Data'},'location','SouthEast');

h = gca ;
str(1) = {'Plot of the model :'};
str(2) = {'$$y = 1/(\hat{\theta}_1 \, x + \hat{\theta}_2)$$'};
str(3) = {'With the values:'};
str(4) = {['$$\hat{\theta}_1 = $$', num2str(sol(1))]};
str(5) = {['$$\hat{\theta}_2 = $$', num2str(sol(2))]};
set(gcf,'CurrentAxes',h) ;

text('Interpreter','latex', 'String',str,'Position',[4 0.85],'FontSize',12)
```

Les paramètres optimaux sont :

```
>> sol(1) = 1.5803 % 1er PARAMETRE
>> sol(2) = 0.8893 % 2em PARAMETRE
```

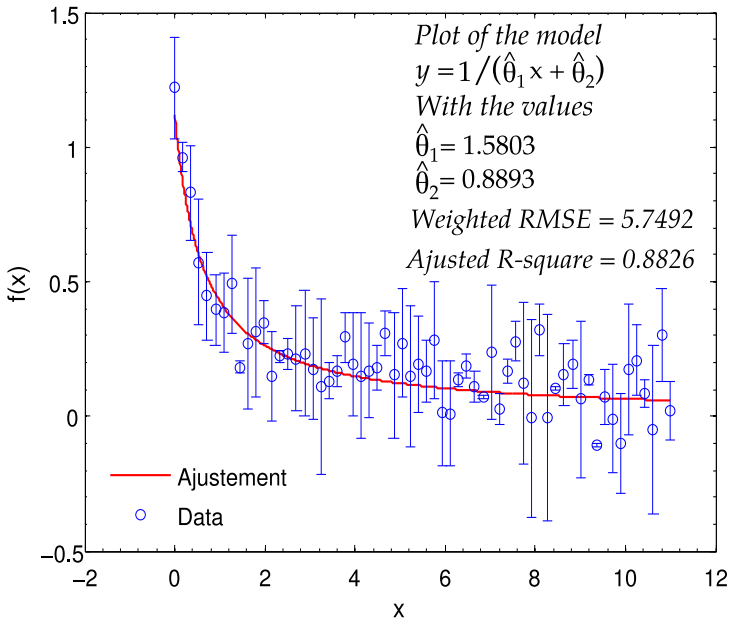

FIGURE 6: Ajustement par la méthode de la *descente de gradient à pas constant*

Par ailleurs, Matlab® compte une multitude de fonctions prédéfinies dédiées au calcul statistique, en voici quelques unes : `corrcoef` (coefficient de corrélation), `cov` (matrice covariance), `mean` (moyenne), `median` (médiane), `std` (écart-type), `var` (variance). Les incertitudes sur les paramètres estimés, sont accessibles via l'instruction `sqrt(diag(cov(X)))`, avec X est la matrice covariance. La liste complète des fonctions statistiques est disponible dans la boite à outils `Statistics Toolbox`.

Exercice ❸ ☞ Ⓢ

1) À partir des données expérimentales suivantes :

1/2	3	4	6	7	9	10	13	14
0.20	1/5	1.10	1.50	2.40	3.80	5.0	6.20	9.1

déterminer avec la méthode de *descente de gradient à pas fixe* les paramètres du modèle :

$$f(x) = \theta_1 \exp(\theta_2 x) + \theta_3$$

2) Afficher, sur la même figure, les données expérimentales et la courbe théorique.

E. Algorithme de Gauss-Newton

L'algorithme de *Gauss-Newton* est basé sur la minimisation du carré de la *fonction-objectif*. Comme il a été mis en évidence au début du chapitre, la forme générale d'un modèle de régression ajustant le mieux les mesures expérimentales est donné par : $f(x_i;\theta_i)$. La méthode est itérative, donc elle réclame un vecteur de paramètres initiaux $(\theta_i^{(0)} = \theta_0^{(0)}, \theta_1^{(0)}, ...)$ pour s'exécuter. Écrivons le développement limité de cette fonction au voisinage du vecteur des paramètres initiaux :

$$f = f(x, \theta_0^{(0)}, \theta_1^{(0)}, ...) + \frac{\delta f(x, \theta_0^{(0)}, \theta_1^{(0)}, ...)}{\delta \theta_0} (\theta_0 - \theta_0^{(0)}) + \frac{\delta f(x, \theta_0^{(0)}, \theta_1^{(0)}, ...)}{\delta \theta_1} (\theta_1 - \theta_1^{(0)})$$

$$+\frac{\delta f(x,\theta_0^{(0)},\theta_1^{(0)},...)}{\delta\theta_2}\,(\theta_2-\theta_2^{(0)})+... \tag{39}$$

$$f-f^0=f'_{\theta_0}\,(\theta_0-\theta_0^{(0)})+f'_{\theta_1}\,(\theta_1-\theta_1^{(0)})+f'_{\theta_2}\,(\theta_2-\theta_2^{(0)})+... \tag{40}$$

Cette équation peut se mettre sous la forme matricielle suivante :

$$F=J\,\theta_n \tag{41}$$

Avec,

$$F=\begin{bmatrix} f(x_1)-f^0(x_1)\\ f(x_2)-f^0(x_2)\\ \vdots\\ f(x_n)-f^0(x_n)\end{bmatrix},\ J=\begin{bmatrix} \frac{\delta f(x_1;\theta_0^{(0)},...,\theta_p^{(0)})}{\delta\theta_0} & ... & \frac{\delta f(x_1;\theta_0^{(0)},...,\theta_p^{(0)})}{\delta\theta_p}\\ \vdots & \vdots & \vdots\\ \frac{\delta f(x_n;\theta_0^{(0)},...,\theta_p^{(0)})}{\delta\theta_0} & ... & \frac{\delta f(x_n;\theta_0^{(0)},...,\theta_p^{(0)})}{\delta\theta_p}\end{bmatrix}$$

$$\theta_n=\begin{bmatrix}\theta_0-\theta_0^{(0)}\\ \theta_1-\theta_1^{(0)}\\ \vdots\\ \theta_n-\theta_n^{(0)}\end{bmatrix}$$

Avec, θ_n est le vecteur des paramètres, qu'on peut exprimer aussi sous la forme : $\theta_n=\delta_n-\delta_0$. Avec δ_n est le vecteur des paramètres (solutions) $\hat{\theta}_0,\hat{\theta}_1,...,\hat{\theta}_p$ et δ_0 est le vecteur des paramètres initiaux, $\theta_0^{(0)},\theta_1^{(0)},...,\theta_p^{(0)}$. En multipliant et divisant l'équation (41) par la transposée de la matrice *Jacobienne*, l'algorithme de la méthode de *Gauss-Newton* s'écrit :

$$\begin{cases} -\ \delta_0 \quad \text{valeurs initiales}\\ -\ \delta_n=(J\,J^t)^{-1}\times(J^t\,F)+\delta_0\\ -\ \delta_0=\delta_n \quad \text{mise à jour}\\ -\ \text{critère d'arrêt}\end{cases} \tag{42}$$

Exercice ❷ ☞ ®

1) À partir des données expérimentales (x_i,y_i) ci-dessous, déterminer avec la méthode de *Gauss-Newton* les paramètres optimaux du modèle suivant :

$$f(x)=\theta_3\left(\frac{x}{\theta_1}\right)^{(\theta_2-1)}e^{\left(-\frac{x}{\theta_1}\right)^{\theta_2}} \tag{43}$$

2) Afficher, sur la même figure, les données expérimentales et le profil théorique.

Voici le script Matlab® :

```
clear all ; close all ; clc ;
% Samir Kenouche - Le 23/10/2018
%%%%%%%%%%%%%%%%%%%%%%%%%% Gauss-Newton %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
idelta(1,1) = 9 ; idelta(2,1) = 3/2 ; idelta(3,1) = 4 ;

xi = [0.1   0.1   0.3   0.3   1.3   1.7   2.1   2.6   3.9   3.9 ...
      5.1   5.6   6.2   6.4   7.7   8.1   8.2   8.9   9.0   9.5 ...
      9.6  10.2  10.3  10.8  11.2  11.2  11.2  11.7  12.1  12.3 ...
      12.3  13.1  13.2  13.4  13.7  14.0  14.3  15.4  16.1  16.1 ...
      16.4  16.4  16.7  16.7  17.5  17.6  18.1  18.5  19.3  19.7];
yi = [0.01  0.08  0.13  0.16  0.55  0.90  1.11  1.62  1.79  1.59 ...
      1.83  1.68  2.09  2.17  2.66  2.08  2.26  1.65  1.70  2.39 ...
      2.08  2.02  1.65  1.96  1.91  1.30  1.62  1.57  1.32  1.56 ...
      1.36  1.05  1.29  1.32  1.20  1.10  0.88  0.63  0.69  0.69 ...
      0.49  0.53  0.42  0.48  0.41  0.27  0.36  0.33  0.17  0.20];

err = 0.3.*randn(1,numel(yi)) ; it = 0 ;
%%%%%%%%%%%%%%%%%%%% CALCUL FORMEL DES DERIVEE %%%%%%%%%%%%%%%%%%%%%%
syms p1 p2 p3 x
modelFun = p3.*(x./p1).^(p2-1).*exp(-(x./p1).^p2);

dy_d1 = diff(modelFun,p1) ; dy_d2 = diff(modelFun,p2) ;
dy_d3 = diff(modelFun,p3) ; dy_d11 = subs(dy_d1,x,xi) ;
dy_d22 = subs(dy_d2,x,xi) ; dy_d33 = subs(dy_d3,x,xi);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
itmax = 100 ; tol = 1e-10 ;
while it < itmax

it = it + 1 ;
dy_da= subs(dy_d11,{'p1','p2','p3'},{idelta(1) idelta(2) idelta(3)});
dy_db= subs(dy_d22,{'p1','p2','p3'},{idelta(1) idelta(2) idelta(3)});
dy_dc= subs(dy_d33,{'p1','p2','p3'},{idelta(1) idelta(2) idelta(3)});

  Jacob(1:numel(xi), 1) = dy_da ; Jacob(1:numel(xi), 2) = dy_db ;
  Jacob(1:numel(xi), 3) = dy_dc ;

  F = yi - subs(modelFun,{'x','p1','p2','p3'},...
      {xi idelta(1) idelta(2) idelta(3)});

  ndelta = inv(Jacob'*Jacob)*(Jacob'*F') + idelta ; idelta = ndelta ;
  fx = subs(modelFun,{'x','p1','p2','p3'},...
      {xi idelta(1) idelta(2) idelta(3)}) ;

  Fn(it) = sum(F) ;
  rmse(it+1) = sqrt(Fn(it).^2)/(length(yi) - length(idelta)) ;

  if rmse(it + 1) - rmse(it) < tol
        sol = idelta ;
     break
  end
```

```
end
%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%%%
n = 500 ; step = (xi(end) - xi(1))/n ; xn = xi(1) :step: xi(end) ;
fun = subs(modelFun,{'x','p1','p2','p3'}, {xn sol(1) sol(2) sol(3)});

figure('color',[1 1 1]) ; errorbar(xi,yi,err,'s') ; hold on ;
plot(xn, fun,'r','LineWidth',2) ; xlim([-2 22]) ; format long
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
h = gca ;
str = {'$${\it p_3}\times\left( {\frac {x}{{\it p_1}}} \right) ^{{\it p_2
  }-1}\,{e^{-\left( {\frac {x}{{\it p_1}}}\right)^{{\it p_2}}}}$$'} ;
set(gcf,'CurrentAxes',h) ;
text('Interpreter','latex', 'String',str,'Position',[2 0.2],'FontSize',14)
```

Les paramètres de minimisation générés par ce code sont :

```
>> sol(1) =  10.041754552364827 % parametre theta1
>> sol(2) =   2.002656719575507 % parametre theta2
>> sol(3) =   5.115914274259778 % parametre theta3
```

L'ajustement obtenu est représenté dans la figure ci-dessous :

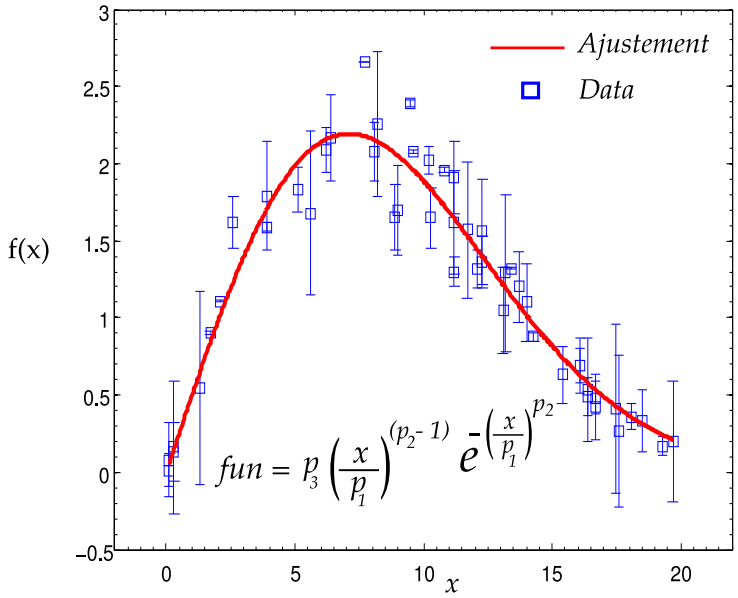

FIGURE 7: Ajustement avec la méthode de *Gauss-Newton*

L'algorithme de *Gauss-Newton* est robuste dans la mesure où la matrice $J^t(\theta_i)J(\theta_i)$ est presque constamment définie positive et ainsi la direction calculée est une direction de descente. Néanmoins, la convergence de l'algorithme vers la solution approchée est très tributaire du choix de la valeur initiale des paramètres à optimiser.

F. Algorithme de Levenberg-Marquardt

L'algorithme de *Levenberg-Marquardt* utilise comme direction de descente, une combinaison entre les directions des algorithmes de *descente de gradient* et de *Gauss-Newton*. Dans sa version la plus récente, l'algorithme de

Levenberg-Marquardt prend la forme :

$$\begin{cases} -\ \delta_0 \quad \text{valeurs initiales} \\ -\ \delta_n = -(J^t\,J + \lambda\, diag(H))^{-1} \times (J^t\,F^t) \\ -\ \delta_0 = \delta_0 + \delta_n \quad \text{mise à jour} \\ -\ \text{critère d'arrêt} \end{cases} \tag{44}$$

Avec H est la matrice *Hessienne* qu'on approxime suivant $H(\theta_i) \approx J^t(\theta_i)\,J(\theta_i)$. À partir de cette expression, il en ressort que pour un λ nul, la direction utilisée est celle de l'algorithme de *Gauss-Newton*. Inversement, pour un λ tendant vers les grandes valeurs, on retrouve la direction de l'algorithme de *descente de gradient*. Le paramètre λ est modifié à chaque itération en prenant comme critère l'inégalité $f(x_i;\theta_{i+1}) > f(x_i;\theta_i)$. Si cette inégalité est vérifiée cela signifie qu'on est dans une zone où le gradient Δf n'est pas très linéaire, on augmente alors la valeur de λ. Dans le cas où elle n'est pas vérifié, on diminuera ce paramètre, en prenant le dixième par exemple.

Exercice ❸ ☞ ®

1) À partir des données expérimentales (x_i, y_i) ci-dessous, déterminer avec la méthode de *Levenberg-Marquardt* les paramètres des modèles suivants

$$\begin{cases} f_1(x) = \theta_1\ \log(x) + \theta_2, \text{Avec les valeurs initiales} \quad [0, 0.2] \\ f_2(x) = \theta_1\ \exp(\theta_2\, x), \text{Avec les valeurs initiales} \quad [5, -0.2] \end{cases} \tag{45}$$

2) Déterminer le `RMSE pondéré` pour les deux ajustements.

3) Afficher, sur la même figure, les données expérimentales et le profil théorique.

Voici le script Matlab® :

```
close all ; clc ; clear all ;
% Le 25.10.2018 - Samir Kenouche
%%%%%%%%%%%%%%%%%%%%%%%% METHODE DE LEVENBERG-MARQUARDT %%%%%%%%%%%%%%%%%%%%%%%%
idelta(1,1) = 3.4; idelta(2,1) = 0.001;

xi = 1:60 ;
yi = [1.1 2.58 4.11 4.59 5.49 4.89 5.72 6.42 7.86 7.33 ...
      9.55 8.56 7.01 8.52 8.56 9.44 9.34 9.47 9.23 9.99 8.91 ...
      10.9 9.30 1.00e+01 1.06e+01 1.11e+01 1.09e+01 1.26e+01 ...
      1.26e+01 1.19e+01 1.15e+01 1.08e+01 1.12e+01 1.041e+01 ...
      1.14e+01 1.14e+01 1.35e+01 1.27e+01 1.22e+01 1.190e+01 ...
      1.23e+01 1.30e+01 1.44e+01 1.28e+01 1.33e+01 1.321e+01 ...
      1.32e+01 1.37e+01 1.40e+01 1.29e+01 1.27e+01 1.321e+01 ...
      1.40e+01 1.42e+01 1.27e+01 1.41e+01 1.32e+01 1.317e+01 ...
      1.39e+01 1.43e+01];

err = 2.*randn(1, numel(xi)) ;
poids = 1./err.^2 ; tol =  1e-06 ; lambda = 0.01 ;
itMax = 100 ; it = 0 ; fx = @(xi, a, b) a.*log(xi) + b ;

while it < itMax

    it = it + 1 ; dy_da = log(xi) ; dy_db = 1 ;
```

```
    Jacob(1:length(xi),1) = dy_da ; Jacob(1:length(xi),2) = dy_db ;
    F = yi - (idelta(1).*log(xi) + idelta(2)) ;

    Hes = Jacob'*Jacob ; diagI = eye(numel(idelta),numel(idelta)) ;
    diagI(diagI == 1) = diag(Hes) ;

    ndelta = - inv(Hes + lambda*diagI)*(Jacob'*F') ;
    idelta = idelta + ndelta ;

    for ii = 1:numel(xi)

      if   fx(xi(ii),ndelta(1),ndelta(2)) < fx(xi(ii),idelta(1),idelta(2))

           lambda = lambda*10;
      else

           lambda = lambda/10;
      end

    end

    Fn(it) = sum(yi - (idelta(1).*log(xi) + idelta(2))) ;
    rmse(it+1) = sqrt(poids(it)*Fn(it).^2)/(length(yi) - length(idelta)) ;
    % RMSE PONDERE
       if rmse(it + 1) - rmse(it) < tol
          soln = idelta ;
         break
       end
end
%%%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%%%%%%%
xn = xi(1):0.01:xi(end) ; model = soln(1).*log(xn) + soln(2) ;
figure('color',[1 1 1]) ;
errorbar(xi,yi,err,'o','MarkerSize',7,'LineWidth',1) ; hold on ;
plot(xn, model,'r','LineWidth',1.5) ;
legend('data','Ajustement selon Levenberg-Marquardt') ;

h = gca ;
str(1) = {'Plot of the model :'};
str(2) = {'$$y = \hat{\theta}_1 \log(x) + \hat{\theta}_2$$'};
str(3) = {'With the values:'};
str(3) = {['$$\hat{\theta}_1 = $$', num2str(soln(1))]};
str(4) = {['$$\hat{\theta}_2 = $$', num2str(soln(2))]};
str(5) = {['Root Mean Square Error = ', num2str(rmse(end))]};
set(gcf,'CurrentAxes',h) ;
text('Interpreter','latex', 'String',str,'Position',[4.8 4],'FontSize',12)
```

L'algorithme de *Levenberg-Marquardt* est particulièrement robuste, dans le sens où il converge avec beaucoup moins d'itérations. Cette robustesse est liée aussi au fait que le terme $(J^t J + \lambda\, diag(H))$ est systématiquement positif, contrairement à la méthode de *Newton-Gauss*. Le paramètre λ permet de contrôler la vitesse de convergence et de tester à la fois, les algorithmes de *Newton-Gauss* (λ petit) et de *descente de gradient* (λ grand). La

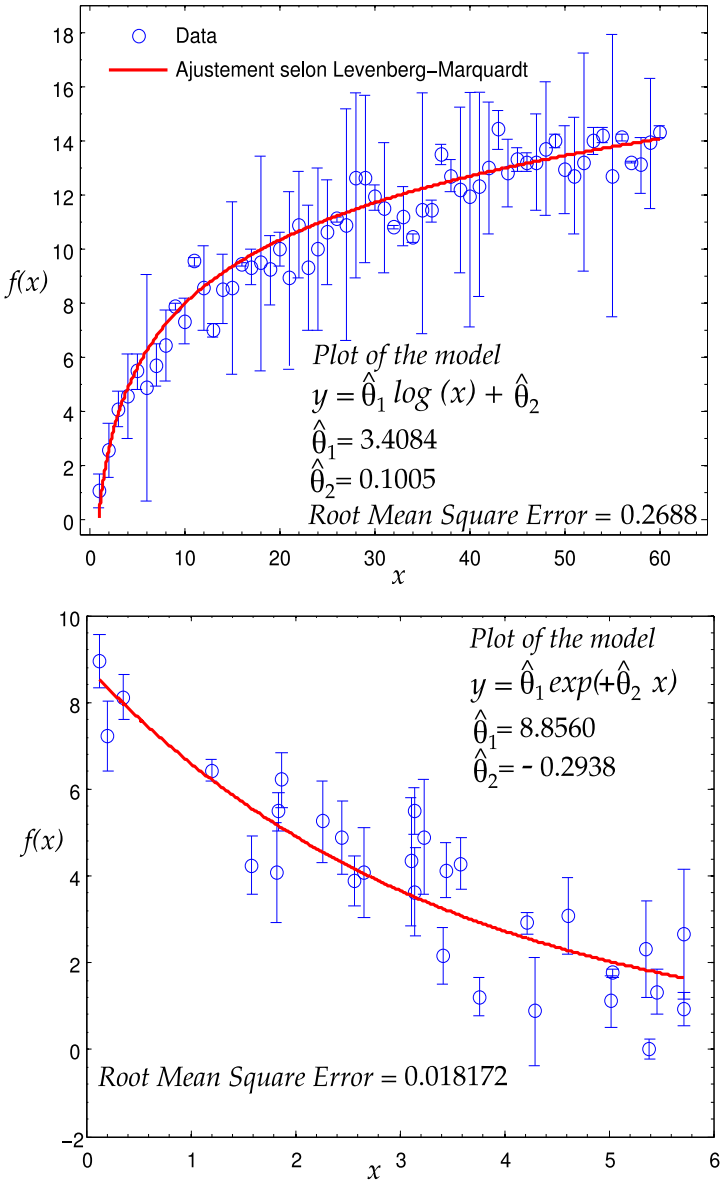

FIGURE 8: Ajustement avec la méthode de *Levenberg-Marquardt*

méthode de *Levenberg-Marquardt* constitue l'algorithme d'optimisation de référence de nombreux logiciels du calcul scientifique. Notons également que l'approximation de la matrice des dérivées secondes (*Hessien*) par le produit des gradients peut se révéler obsolète dans le cas où les valeurs du vecteur des résidus sont trop grandes.

III. SUPPLÉMENT : AVEC DES COMMANDES MATLAB

A. *Optimisation sans contraintes*

La formulation générale d'un problème d'optimisation sans contraintes s'écrit selon :

$$\min_{x \in X} f(x) \tag{46}$$

Avec, X est un sous-ensemble de $\mathbb{R}^n$. Les variables $x = (x_1, x_2, ..., x_n)$ sont appelées *variables d'optimisation* ou *variables de décision*. La fonction f, à valeurs réelles, définie par $f : X \subset \mathbb{R}^n \to \mathbb{R}$ est la *fonction-objectif*. Dans cette section, on abordera les commandes numériques, dédiées à la résolution de problèmes d'optimisation

sans contraintes. Toutes ces commandes sont disponibles dans la boite à outil `Optimization Toolbox` de Matlab®.

Commandes `fminbnd` et `fminunc`

Les commandes Matlab® destinées à la minimisation de fonctions monovariables sont `fminbnd` et `fminunc`. Notons que cette dernière peut également être utilisée pour les fonctions à plusieurs variables. Ces commandes présentent une syntaxe très similaire.

```
options = optimset('param 1', value 1, 'param 2', value 2, ...)
[x, fval, exitflag, output, grad, hessian]= fminunc(fun, x0, options)
% pour fminunc
[x, fval, exitflag, output] = fminbnd(fun ,lB, uB, options)
% pour fminbnd
```

La commande `fminunc` accepte comme arguments en entrée, la fonction à minimiser `fun`, les valeurs initiales `x0` pour initialiser la recherche des minimums et en dernier lieu l'argument `options` spécifiant les différents champs d'optimisation. Ces derniers sont modifiés en appelant la commande `optimset`, dont ses différents paramètres seront décrits dans l'exercice ci-dessous. La fonction `fun` peut être définie en tant que *objet inline*, *fonction anonyme* ou bien une *fonction M-file*. Les sorties renvoyées sont : `x` représentant le minimum trouvé après convergence et `fval` est le nombre d'évaluation de la fonction `fun`. Une valeur de la sortie `exitflag = 1` signifie que l'algorithme a bel et bien convergé vers la solution approchée. Plus généralement, une valeur de `exitflag > 1` signifie que l'algorithme a convergé vers la solution. Une valeur de `exitflag < 1` signifie que l'algorithme n'a pas convergé. Dans le cas où `exitflag = 0`, cela veut dire que le nombre d'itération ou le nombre d'évaluation de la fonction est atteint. La sortie `output` renvoie des champs relatifs au type d'algorithme utilisé, le nombre d'itération conduisant à la solution approchée, un message sur l'état de l'optimisation ... etc. Les sorties `grad` et `hessian` renvoient respectivement le *Jacobien* (première dérivée) et le *Hessien* (second dérivée) de la fonction `fun`.

La différence entre les commandes `fminunc` et `fminbnd` se situe au niveau de l'initialisation de la recherche de la solution approchée. En effet, la commande `fminunc` démarre la recherche à partir d'une valeur initiale apportée par `x0`. La commande `fminbnd` effectue sa recherche à partir d'un intervalle dont les bornes inférieure et supérieure sont indiquées respectivement par les entées `lB` et `uB`.

Exercice 1

1) Minimiser la fonction définie par :

$$\begin{cases} f(x) = (x-1) \times \exp(-x^2 + 2\,x + 1) \\ \text{Avec,} \quad x \in [-4, 4] \end{cases} \tag{47}$$

en utilisant les commandes `fminbnd` et `fminunc`

Voici le script Matlab®

```
clear all; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
lB = -4 ; uB = 4 ; xinit = 1 ;
fx = @(x) (x-1).*exp(-x.^2 + 2.*x + 1) ;

opts = optimset('Display','iter','FunValCheck','on','TolX',1e-8) ;
```

```
% parametres d'optimisation
[xMin1, funEval1, exitTest1, output1, grad1, hessian1] = fminunc(fx,  xinit,
   opts) ;
% premiere possibilite
[xMin2, funEval2, exitTest2, output2] = fminbnd(fx, lB, uB, opts) ;
% Deuxieme posibilite
```

Les arguments de sortie renvoyés par `fminunc` sont :

```
%%%%%%%%%%%%%%%%%%%% AFFICHAGE PAR DEFAUT 1ER CAS %%%%%%%%%%%%%%%%%%%%
 Iteration  Func-count       f(x)
     0           2               0
     1           4         -2.71828
     2           6         -3.16544
     3           8         -3.16849
     4          10         -3.16903
     5          12         -3.16903
     6          14         -3.16903
Optimization terminated: relative infinity-norm of gradient less than options.
   TolFun.
%%%%%%%%%%%%%%%%%%%% AFFICHAGE PAR DEFAUT 2EME CAS %%%%%%%%%%%%%%%%%%%%
 Func-count     x          f(x)
    1      -0.944272    -0.327815
    2       0.944272    -0.410501
    3        2.11146      2.38771
    4      0.0274024     -2.79063
    5     0.00805821     -2.74001
    6       0.252738     -3.15901
    7        0.51688     -2.82668
    8       0.278372     -3.16771
    9        0.29087     -3.16901
   10       0.293071     -3.16903
   11         0.2929     -3.16903
   12       0.292893     -3.16903
   13       0.292893     -3.16903
   14       0.292893     -3.16903
   15       0.292893     -3.16903

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of
   1.000000e-08
```

Les autres sorties sont affichées comme suit

```
>> xMin1 =
            0.2929  % minimum trouve

>> funEval1 =
           -3.1690 % valeur de la fonction a la derniere iterat.
```

```
>> exitTest1 =
              1       % teste de convergence positif

>> grad1 =

        -5.9605e-08   % valeur du gradient de la fonction

>> hessian1 =

             12.6783 % valeur du Hessien de la fonction

>> xMin2 =            % minimum obtenu avec la commande fminbnd
              0.2929

>> funEval2 =
             -3.1690

>> exitTest2 =
              1
```

L'argument `opts`, de type *srtucture*, compte les options d'optimisation spécifiées dans `optimset`. Le champ de la structure `opts` indiqué par `optimset('Display','iter', ...)` affiche des détails pour chaque itération. Si l'on désire afficher uniquement les détails de la dernière itération, on remplacera `'iter'` par `'final'`. Dans le cas où on ne veut afficher aucun détails, on mettra la valeur `'off'`. Le champ indiqué par `optimset(...., 'FunValCheck','on',...)` contrôle si les valeurs de la fonction sont réelles et affiche dans le cas contraire un avertissement quand la commande en question renvoie une valeur complexe ou NaN. On peut suspendre cette vérification, en remplaçant la valeur `'on'` par `'off'`. Le champ d'optimisation `optimset(...., 'TolX', 1e-08)` correspond à la tolérance admise pour la solution approchée. D'autres champs d'optimisation existent comme `MaxFunEvals` qui fixe le nombre maximum d'évaluation de la fonction à optimiser et `MaxIter` fixant également le nombre maximum d'itération. Voir aussi `GradObj`, `OutputFcn`, `PlotFcns`, ... etc dont la description est disponible dans le *help* de Matlab®.

La sortie `xMin1 = 0.2929` est le minimum trouvé. Ce dernier est cherché autour de la valeur initiale `xinit`. La sortie `funEval1 = -3.1690` exprime l'évaluation de la fonction à la dernière itération, c'est-à-dire pour `fx(x = xMin1)`. L'argument `exitTest = 1` signifie que l'algorithme a convergé vers la solution, une valeur négative indiquerai le contraire. L'argument de sortie `output1`, de type *structure*, renvoie les champs suivants :

```
>> output1 =                          % pour la commande fminunc
            iterations: 6
             funcCount: 14
              stepsize: 1
         firstorderopt: 5.9605e-08
             algorithm: [1x38 char]  % 'Quasi-Newton line search'
               message: [1x85 char]  %  Optimization terminated ...

>> output2                            %  pour la commande fminbnd
            iterations: 14
             funcCount: 15
             algorithm: [1x46 char]  % 'golden section search'
               message: [1x111 char] % Optimization terminated ...
```

La fonction a été évaluée 14 fois et l'algorithme converge vers la solution approchée au bout de la $6^{\text{ième}}$ itération. La sortie `stepsize: 1` indique le pas final de l'algorithme moyenne dimension de `Quasi-Newton`. Le mot `Optimization` affiché comme `message`, fait référence au faite que l'algorithme de minimisation fonctionne selon le critère *des moindres carrés*. La commande `fminbnd` présente les mêmes propriétés que celles de `fminunc`, à la différence que `fminbnd` cherche le minimum dans un intervalle, donné en argument d'entrée avec la borne inférieure `lB` (lower Bound) et la borne supérieure `uB` (upper Bound).

Commande `fminsearch`

Nous allons désormais tester la commande `fminsearch` qui s'utilise pour la minimisation de fonctions multidimensionnelles. La syntaxe usuelle de cette commande est analogue à celle de `fminunc`, à la différence près que la commande `fminsearch` ne renvoie ni le *Jacobien* ni le *Hessien* de la fonction à optimiser. Ceci provient du fait que cette commande est basée sur l'algorithme *Simplex*. Nous allons procéder à son implémentation dans l'exercice ci-dessous.

Exercice 2

1) Minimiser la fonction à deux variables définie par :

$$f(x_1, x_2) = x_1^2 + (x_2 - 2)^2 \tag{48}$$

analytiquement puis en utilisant la commande `fminsearch`

Commençons par déterminer, analytiquement, les extremums de la fonction $f(x_1, x_2)$. Le gradient de la fonction s'écrit :

$$\begin{cases} \dfrac{\partial f}{\partial x_1} = 2\,x_1 = 0 \\ \dfrac{\partial f}{\partial x_2} = 2\,(x_2 - 2) = 0 \end{cases} \tag{49}$$

Le vecteur du point critique est donné donc par $\hat{X} = (\hat{x}_1 = 0\,;\, \hat{x}_2 = 2)$. Cherchons désormais la nature de ce point, s'agit-il d'un minimum ou d'un maximum ?. Calculons le déterminant du *Hessien* de f.

$$\begin{vmatrix} \dfrac{\partial^2 f(\hat{X})}{\partial x_1^2} & \dfrac{\partial^2 f(\hat{X})}{\partial x_1\,\partial x_2} \\ \dfrac{\partial^2 f(\hat{X})}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f(\hat{X})}{\partial x_2^2} \end{vmatrix} \Longrightarrow \begin{vmatrix} 2 & 0 \\ 0 & 2 \end{vmatrix} = 4 \tag{50}$$

À partir de ces résultats, il en découle que le point $\hat{X} = (\hat{x}_1 = 0\,;\, \hat{x}_2 = 2)$ est un minimum global. Nous allons résoudre le même système de façon algorithmique en se servant de la commande `fminsearch`. Voici le script Matlab®

```
clear all; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
xinit = [1 1] ; fun = @(x) x(1).^2 + (x(2) - 2).^2;
opts = optimset('Display','iter','FunValCheck','on','TolX',1e-8) ;

[xMin, funEval, exitTest, output] = fminsearch(fun,  xinit, opts) ;
```

Ci-dessous les différentes sorties renvoyées par le script Matlab®.

```
%%%%%%%%%%%%%%%%%%%%%%%%%% Sorties par defaut %%%%%%%%%%%%%%%%%%%%%%%%%%
 Iteration   Func-count     Min f(x)
     0            1                2
     1            3           1.9025
     2            5          1.66563
     3            7          1.38266
     4            9         0.889414
     5           11         0.353447
     6           13        0.0265259
     7           14        0.0265259
     8           16        0.0265259
     9           18        0.0265259

    ...          ...        .........

     63          125      2.15781e-17
     64          127      2.15781e-17

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of
   1.000000e-08 and F(X) satisfies the convergence criteria using OPTIONS.
   TolFun of 1.000000e-04
```

```
>> xMin =

              0.0000    2.0000

>> funEval =

              2.1578e-17

>> exitTest =

              1
```

```
>> output

        iterations: 64
         funcCount: 127
         algorithm: [1x33 char] % 'Nelder-Mead simplex direct search'
           message: [1x194 char] % Optimization terminated: the current x
   satisfies the termination
```

L'algorithme converge vers la solution approchée `Xmin` au bout de 64 itérations. Notons que cette convergence est atteinte car les conditions d'arrêt de l'algorithme sont satisfaites, soit une tolérance `TolX = 1e-8`. En

choisissant par exemple une tolérance de `TolX = 1e-3`, l'algorithme converge vers la solution approchée (`xMin = [-0.0002; 2.0004]`) au bout de 28 itérations seulement. Dans le cas où cette dernière n'est pas spécifiée, la valeur par défaut est `TolX = 1e-6`. On comprend alors que le choix de la tolérance est conditionné par la précision recherchée. Il est important de rappeler aussi que tous ces algorithmes d'optimisation sont basés sur des processus itératifs, donc fortement dépendant du choix de la valeur initiale. Autrement dit, plus la valeur initiale est proche de la solution approchée plus l'algorithme converge rapidement.

Comme il a été mentionné dans la section précédente, la commande `fminunc` s'utilise aussi pour l'optimisation de fonctions à plusieurs variables. Nous allons utiliser cette commande pour optimiser la fonction $f(x_1, x_2) = 2\,x_1^2 + x_1\,x_2 + 2\,x_2^2 - 6\,x_1 - 5\,x_2 + 3$

Voici le script Matlab®

```
clear all; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
xinit = [-1 1] ;
fx = @(x) 2*x(1).^2 + x(1)*x(2) + 2*x(2).^2 - 6*x(1) - 5*x(2) + 3 ;

opts = optimset('LargeScale','off','Display','iter','FunValCheck',...
    'on','MaxIter', 20,'TolX', 1e-5) ;
[x, funEval, exitTest, output, grad, hessian] = fminunc(fx, xinit, opts) ;
```

Les sorties renvoyées par ce script sont :

```
%%%%%%%%%%%%%%%%%%%%%%%% affichage par defaut %%%%%%%%%%%%%%%%%%%%%%%%
Iteration  Func-count       f(x)
     0           3               7
     1           6         -0.123457
     2           9          -3.09662
     3          12          -3.13148
     4          15          -3.13333
     5          18          -3.13333
     6          21          -3.13333
Optimization terminated ...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> x =                    % solution approchee renvoyee

      1.2667    0.9333


>> funEval =              % valeur de la fonction a l'iteration 6

     -3.1333


>> exitTest =             % teste de convergence positif

      1
```

```
>> grad =                % gradient evalue au point x

     1.0e-07 *

      0.9411
     -0.5960


>> hessian =             % hessien evalue au point x

      4.0000    1.0000
      1.0000    4.0000
```

On peut, entre autre, définir explicitement le *gradient* et le *Hessien* de la *fonction-objectif*. Le premier intérêt de cette procédure est de fournir les expressions analytiques, sans approximation, des dérivées. Si ces dernières ne sont pas explicitées, Matlab® calcul une approximation selon la méthode des *différences finies*. L'autre intérêt tient à l'accroissement, notamment pour des systèmes plus complexes, de la vitesse de convergence. Ainsi, le *gradient* et le *Hessien* sont indiqués via la syntaxe `options = optimset('GradObj','on','Hessian','on')`. On commence d'abord par définir la fonction *M-file* suivante.

```
function [fun, Jacobien, Hessien] = myfun(x)

fun = 2*x(1).^2 + x(1)*x(2) + 2*x(2).^2 - 6*x(1) - 5*x(2) + 3 ;

% Compute the objective function value at x
if nargout > 1
% fun called with two output arguments
grad(1) = 4*x(1) + x(2) - 6 ; % Gradient of the function evaluated at x
grad(2) = x(1) + 4*x(2) - 5 ;

Jacobien = grad ;
end

if nargout > 2

hessian(1,1) = 4 ;  hessian(2,1) = 1 ; % Hessian evaluated at x
hessian(2,1) = 1 ;  hessian(2,2) = 4 ;
Hessien = hessian ;

end
return
```

Cette fonction est sauvegardée sous le nom `myfun.m`. L'appel de cette dernière se fait avec le script suivant :

```
clear all; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
xinit = [1 1] ;
opts = optimset('GradObj','on','Hessian','on','Display','iter') ;
[x, fval, exitflag, output] = fminunc(@myfun, xinit, opts)
```

On écrivant `opts = optimset('GradObj','off','Hessian','off')`, Matlab® évalue le gradient et le *Hessien* de la *fonction objectif* par *différences finies*. Le champ d'optimisation `opts = optimset('DerivativeChe` compare le gradient fourni par l'utilisateur et celui évalué par *différences finies*. Le champ d'optimisation `opts = optimset('FinDiffType', 'forward')` (par défaut) stipule que le gradient sera estimé par *différences finies progressives*. En indiquant la valeur `'central'`, dans ce cas, le gradient sera estimé par *différences finies centrées*. Le paramètre d'optimisation `FinDiffType` (type de différences finies) n'est valable que si le paramètre `DerivativeCheck` est activé. D'autres paramètres d'optimisation existent, à l'instar de `DiffMaxChange`, `DiffMinChange`, `FinDiffRelStep`, ... etc. Consulter le `help` de Matlab® pour de amples informations.

Exercice 2

– Justifier de l'existence d'un extremum global des fonctions.

$$\begin{cases} f_1(x_1,x_2) = (1-x_1)^2 + 100\,(x_2 - x_1^2)^2 \\ f_2(x_1,x_2) = 20 + x_1^2 + x_2^2 - 10\,(\cos(2\pi x_1) + \cos(2\pi x_2)) \\ f_3(x_1,x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\ f_4(x_1,x_2) = \dfrac{1}{2} - \sin(x_1^2 + x_2^2) \\ f_5(x_1,x_2) = x_1^3 + x_2^3 - 3\,x_1\,x_2 \\ f_6(x_1,x_2) = x_1^2 - x_1\,x_2 + x_2^2 + 3\,x_1 - 2\,x_2 + 1 \end{cases} \tag{51}$$

– Trouver les extremums des fonctions, analytiquement, ensuite en se servant de la commande `fminsearch`

B. Optimisation sous contraintes

De nombreux problèmes en physique, en chimie, en ingénierie et en économie nécessitent de minimiser une fonction soumise à plusieurs contraintes. On s'intéressera à la résolution de problèmes d'optimisation sous contraintes dont la formulation générale est donnée par :

$$\min_{x\in\mathbb{R}^n} f(x) \tag{52}$$

$$\text{tel que} \begin{cases} h_i(x) = 0, & \{i = 1,2,...,k\} \\ g_j(x) \le 0, & \{j = 1,2,...,l\} \end{cases}$$

Avec, P est un sous-ensemble non vide de $\mathbb{R}^n$ défini par des contraintes d'égalité et/ou d'inégalités de fonctions :

$$P = \{x \in \mathbb{R}^n : h_i(x) = 0, g_j(x) \le 0\} \tag{53}$$

Ainsi, l'ensemble P est appelé domaine des contraintes, $g = (g_1, g_2, ..., g_l)$ sont les contraintes d'inégalité et $h = (h_1, h_2, ..., h_k)$ sont les contraintes d'égalité. Dans cette section, il sera question de présenter l'ensemble de commandes numériques, dédiées à la résolution de problèmes d'optimisation sous contraintes. Toutes ces commandes sont disponibles dans la boite à outil `Optimization Toolbox` de Matlab®.

Commande `linprog`

La commande `linprog` (Linear programming) solutionne un processus d'optimisation, écrit sous une formulation linéaire, en minimisant la quantité :

$$\min_{X\in\mathbb{R}^n} f^T\,x \text{ tel que } \begin{cases} A \times x \le B \\ Aeq \times x = Beq \\ lB \le x \le uB \end{cases} \tag{54}$$

Cette commande utilise deux types d'algorithmes *Large-échelle* (Large-Scale Optimization) et *Moyenne-échelle* (Medium-Scale Optimization). Le premier type est utilisé pour des systèmes complexes en terme de taille et sous certaines conditions qu'on ne va pas détailler ici. On se contentera d'utiliser le deuxième type et plus précisément la méthode *Simplex*. La syntaxe usuelle de la commande `linprog` est :

```
[x, fval, exitflag, output, lambda] = linprog(f, A, B, Aeq, Beq, lB , uB, x0,
  options)
```

Les quantités `x, f, B, Beq, lB` et `uB` sont des vecteurs. Les arguments `A` et `Aeq` sont des matrices. L'argument `f` désigne le vecteur des coefficients des variables, tel que $f^T\,x = f(1)\,x(1) + f(2)\,x(2) \ldots f(n)\,x(n)$. Les contraintes linéaires de type équation et inéquation sont écrites respectivement selon $Aeq \times x = Beq$ et $A \times x \leq B$. L'argument de sortie `lambda` désigne le *multiplicateur de Lagrange*. Les arguments `lB` et `uB` délimitent l'intervalle de définitions des variables en question. Pour les variables non bornées on mettra `lB = -inf` et `uB = inf`.

Exercice 3

1) Minimiser la fonction définie par :

$$f(x) = -5\,x_1 - 4\,x_2 - 6\,x_3$$

$$\text{tel que} \begin{cases} x_1 - x_2 + x_3 \leq 20 \\ 3\,x_1 + 2\,x_2 + 4\,x_3 \leq 42 \\ 3\,x_1 + 2\,x_2 \leq 30 \\ 0 \leq x_1,\ 0 \leq x_2,\ 0 \leq x_3 \end{cases}$$

2) Maximiser la fonction définie par :

$$f(x) = 14\,x_1 + 6\,x_2$$

$$\text{tel que} \begin{cases} x_1 + x_2 \leq 7.50 \\ 11\,x_1 + 3\,x_2 \leq 0.40 \\ 12\,x_1 + 21\,x_2 \leq 1.50 \\ x_1 \geq 0,\ x_2 \geq 0 \end{cases}$$

Voici le script Matlab®, concernant la minimisation

```
clear all ; clc ;

opts = optimset('LargeScale', 'off','Simplex', 'on') ;
options = optimset(opts, 'Display', 'iter', 'TolFun', 1e-5) ;

f = [-5 ; -4 ; -6] ; A = [1 -1 1 ; 3 2 4 ; 3 2 0] ;
B = [20; 42; 30] ; lB = [0 ; 0 ; 0] ; uB = [inf ; inf ; inf] ;

[x, fval, exitflag, output,lambda] = linprog(f, A, B, [], ...
[],lB, uB, [], options) ;

point_critique = sprintf('%6.4f \n', x)
```

Le champ d'optimisation `opts = optimset('LargeScale', 'off' ...)` signifie qu'on utilisera l'algorithme *Moyenne-échelle* pour résoudre ce problème. Le critère d'arrêt est considéré pour la fonction à travers `options = optimset(... 'TolFun', 1e-5)`, autrement dit l'algorithme s'arrête une fois que la condition $|f(x_{k+1}) - f(x_k)| \leq 1e-5$ est satisfaite. Ci-dessous, l'affichage généré par le script Matlab®.

```
The default starting point is feasible, skipping Phase 1.

Phase 2: Minimize using simplex.
      Iter            Objective              Dual Infeasibility
                        f'*x                    A'*y+z-w-f
       0                    0                       8.77496
       1                  -63                       1.11803
       2                  -78                             0
Optimization terminated.

>> point_critique =       % solution
                    0.0000
                   15.0000
                    3.0000
>> fval =
                   -78
>> exitflag =
                    1
```

On voit que l'algorithme *Simplex* converge vers la solution approchée au bout de trois itérations. Ci-dessous, le script Matlab® relatif à la maximisation.

```
clear all ; clc ;

opts = optimset('LargeScale', 'off','Simplex', 'on') ;
options = optimset(opts, 'Display', 'iter', 'TolFun', 1e-5) ;

f = [-14 ; -6] ; A = [1 1 ; 11 3 ; 12 21] ;
B = [7.50 ; 0.40 ; 1.50] ; lB = [0 ; 0] ; uB = [inf ; inf] ;

[x, fval, exitflag, output,lambda] = linprog(f, A, B, [], ...
[],lB, uB, [], options) ;

point_critique = sprintf('%6.4f \n', x)
```

Ainsi, maximiser la fonction $14\,x_1 + 6\,x_2$ revient à minimiser $-14\,x_1 - 6\,x_2$. Ci-dessous, l'affichage généré par ce script.

```
The default starting point is feasible, skipping Phase 1.

Phase 2: Minimize using simplex.
      Iter            Objective              Dual Infeasibility
```

```
                        f'*x                    A'*y+z-w-f
       0                   0                       15.2315
       1            -0.509091                      2.18182
       2                -0.64                            0
Optimization terminated.

>> point_critique =       % point critique du maximum
                    0.0200
                    0.0600
>> fval =
                   -0.6400

>> exitflag =
                    1
```

Commande `fmincon`

La topologie générale d'un processus d'optimisation sous contraintes sur les variables, peut s'écrire suivant la notation compacte suivante :

$$\hat{X} \in \underset{X\in\mathbb{R}^n}{argmin} f(X) \quad \text{tel que} \quad \begin{cases} A \times x \leq B \\ Aeq \times x = Beq \\ C(x) \leq x \\ Ceq(x) = x \\ lB \leq x \leq uB \end{cases} \tag{55}$$

Les quantités x, B, Beq, lB, et uB sont des vecteurs. A et Aeq sont des matrices, $f(x)$, $C(x)$ et $Ceq(x)$ sont des fonctions pouvant êtres également des fonctions non-linéaires. Afin de résoudre des problèmes d'optimisation sous contraintes, on se servira de la commande `fmincon`. Cette commande sert à optimiser des *fonctions-objectifs* multidimensionnelles non-linéaires. Par défaut, l'algorithme d'optimisation est basé sur la méthode **SQP** (Sequential Quadratic Programming). La syntaxe usuelle de cette commande s'écrit selon :

```
[x, fval, exitflag, output, lambda, grad, hessian] = fmincon(fun, x0, A, B,
  Aeq, Beq, lB, uB, @nonlcon, options)
```

Les contraintes linéaires de type équation et inéquation sont écrites respectivement selon $Aeq \times x = Beq$ et $A \times x \leq B$. Les contraintes non-linéaires de type équation et inéquation sont données respectivement par $Ceq(x) = x$ et $C(x) \leq x$. La fonction *M-file* `@nonlcon` contient les contraintes non-linéaires de type équation et inéquation. Les autres arguments en entrée et en sortie ont la même signification que ceux des commandes vue précédemment. Il est très important de souligner que si un ou plusieurs arguments ci-dessus sont manquants, on doit les remplacer par un ensemble vide `[]`. L'ordre d'apparition des arguments en entrée est important, on doit toujours commencer par les contraintes de type inégalité même si ce type de contrainte est vide. Nous commencerons dans un premier temps par résoudre un problème d'optimisation sous contraintes linéaires.

Exercice 4

– On se propose de minimiser le fonction définie par :

$$\underset{X\in\mathbb{R}^n}{min} \quad f(x_1, x_2) = 2\,x_1^2 + x_1\,x_2 + 2\,x_2^2 - 6\,x_1 - 6\,x_2 + 15$$

$$\text{tel que} \begin{cases} x_1 + 2\,x_2 \leq 5 \\ 4\,x_1 \leq 7 \\ x_2 \leq 2 \\ -2\,x_1 + 2\,x_2 = -1 \end{cases}$$

Voici le script Matlab®

```
clc ; clear all ;
%%%%%%%%%%%%%%%%%%%% Optimisation sous contraintes %%%%%%%%%%%%%%%%%%%%
fun = @(x) 2*x(1).^2 + x(1)*x(2) + 2*x(2).^2 - 6*x(1) - 6*x(2) + 15 ;

A = [1 2 ; 4 0 ; 0 1] ; B = [5 7 2] ;
Aeq = [-2 2] ; Beq = -1 ; xinit = [-1 1/2] ;

options = optimset('LevenbergMarquardt','on','Display','iter', ...
    'TolX', 1e-4) ;

[x, fval, exitflag, output, lambda, grad, hessian] = fmincon(fun, ...
    xinit, A, B, Aeq, Beq, [], [], [], options)
```

Le champ d'optimisation désigné par `optimset('LevenbergMarquardt','on', ...)` stipule que l'opération d'optimisation sera résolue par le biais de l'algorithme de *Levenberg-Marquardt*. Le choix de cet algorithme peut se faire également avec la syntaxe `optimset('NonlEqnAlgorithm','lm', ...)`. Dans le même type d'algorithme, on peut aussi faire appel à celui de *Gauss-Newton*, en utilisant la syntaxe `optimset('NonlEqnAlgorithm','gn' ...)`. Le schéma numérique de ces algorithmes est détaillé dans le chapitre 11 (*Algorithmes d'ajustement*, section *Régression non-linéaire*). Les différentes sorties renvoyées par ce script sont énumérées comme suit

```
%%%%%%%%%%%%%%%%%%%%%%%% affichage par defaut %%%%%%%%%%%%%%%%%%%%%%%%
Iter F-count         f(x)   constraint
    0       3           20            4
    1       6       8.4375    1.332e-15
    2       9      8.05206            0
    3      12       7.9875     2.22e-16

Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
No active inequalities.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> x =                  % coordonnees du point critique

      1.4500    0.9500

>> fval =

      7.9875

>> exitflag =

      1
```

```
>> output =

         iterations: 3
          funcCount: 12
      lssteplength: 1
           stepsize: 0.1607
          algorithm: [1x44 char]
      firstorderopt: [1x1 double]
    constrviolation: [1x1 double]
            message: [1x144 char]

>> lambda =

          lower: [2x1 double]
          upper: [2x1 double]
          eqlin: 0.3750
      eqnonlin: [0x1 double]
       ineqlin: [3x1 double]
    ineqnonlin: [0x1 double]

>> grad =

      0.7500
     -0.7500

>> hessian =

      2.6500    2.3500
      2.3500    2.6500
```

On résoudra, dans l'exercice ci-dessous, un problème d'optimisation sous contraintes non-linéaires.

Exercice 5

– On se propose de minimiser le fonction définie par :

$$\min_{X\in\mathbb{R}^n} \quad f(x_1,x_2) = \exp(x_1)\,(4\,x_1^2 + 2\,x_2^2 + 4\,x_1\,x_2 + 2\,x_2 + 1)$$

$$\text{tel que } \begin{cases} 2 + x_1\,x_2 - x_1 - x_2 \leq 0 \\ -x_1\,x_2 \leq 10 \end{cases}$$

On commence d'abord par écrire la `fonction M-file` correspondant aux contraintes non-linéaires. Cette fonction bien entendu est sauvegardée sous le nom `mycontr.m`. Voici le script :

```
function [C, Ceq] = mycontr(x)
% fonction definissant les contraintes non lineaires

C = [2 + x(1)*x(2) - x(1) - x(2) ; -x(1)*x(2) - 10] ; % inequation
Ceq = [] ; % pas de contraintes non lineaires en equation
```

```
return
```

Ci-dessous le script Matlab® du programme appelant.

```
clc ; clear all ;

fun = @(x) exp(x(1))*(4*x(1)^2 + 2*x(2)^2 + 4*x(1)*x(2) + 2*x(2) + 1)

xinit = [-1 1] ;

options = optimset('LevenbergMarquardt','on','Display','iter', ...
    'TolX', 1e-4) ;

[x, fval, exitflag, output, lambda, grad, hessian] = fmincon(fun, ...
    xinit, [], [], [], [], [], [], @mycontr, options)
```

Les arguments de sorties renvoyés sont :

```
%%%%%%%%%%%%%%%%%%%%%% affichage par defaut %%%%%%%%%%%%%%%%%%%%%%
Iter F-count         f(x)   constraint
    0       3       1.8394            1
    1       6      1.98041      -0.1839
    2       9      1.63636       0.2596
    3      12      0.34132        4.524
    4      15     0.530719       0.3344
    5      18     0.447582     -0.08164
    6      21     0.123147        2.352
    7      24    0.0575464       0.3957
    8      27    0.0335016       0.1153
    9      30    0.0331726    0.0001285
   10      33     0.033173    1.589e-10

Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-06):
  lower      upper     ineqlin   ineqnonlin
                                        1
                                        2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> x =

     -9.0990    1.0990

>> fval =

      0.0332

>> exitflag =
```

```
       1

>> output =

         iterations: 10
          funcCount: 33
       lssteplength: 1
           stepsize: [1x1 double]
          algorithm: [1x44 char]
      firstorderopt: [1x1 double]
    constrviolation: [1x1 double]
            message: [1x144 char]

>> lambda =

          lower: [2x1 double]
          upper: [2x1 double]
          eqlin: [0x1 double]
      eqnonlin: [0x1 double]
        ineqlin: [0x1 double]
    ineqnonlin: [2x1 double]

>> grad =

    0.0255
   -0.0034

>> hessian =

    0.0280    0.0035
    0.0035    0.0096
```


Exercice d'application

– On se propose de minimiser les fonctions définies par :

$$\min_{X\in\mathbb{R}^2} \quad f(x_1,x_2)=\frac{1}{2}(x_1-3)^2+\frac{1}{2}(x_2-1)^2$$

$$\text{tel que} \begin{cases} x_1+x_2-1\leq 0 \\ x_1-x_2-1\leq 0 \\ -x_1+x_2-1\leq 0 \\ -x_1-x_2-1\leq 0 \end{cases}$$

Commande `quadprog`

La commande `quadprog` (Quadratic programming) solutionne un processus d'optimisation, écrit sous une formulation quadratique, en minimisant la quantité :

$$\min_{X\in\mathbb{R}^n} f(X)=\frac{1}{2}\,x^T\,H\,x+f^T\,x \text{ tel que } \begin{cases} A\times x\leq B \\ Aeq\times x=Beq \\ lB\leq x\leq uB \end{cases} \tag{56}$$

La syntaxe usuelle de cette commande est :

```
[x, fval, exitflag, output, lambda] = quadprog(H, f, A, B, Aeq, Beq, lb, ub,
  x0, options)
```

L'argument `H` est le *Hessien* (matrice carrée) de la *fonction-objectif* et `f` représente le vecteur des coefficients de la partie linéaire de la *fonction-objectif*. Ces deux arguments sont obligatoires tandis que les autres sont optionnels. Ces derniers ont la même signification que ceux de la commande `fmincon`, pareil également pour les arguments de sortie. L'ordre d'apparition des arguments doit être respecté, si un argument optionnel n'est pas utilisé, il faudra le remplacer par l'ensemble vide `[]`.

Exercice 6

– On se propose de minimiser les fonctions définies par :

$$\min_{X\in\mathbb{R}^2} \quad f(x_1,x_2) = x_1^2 + 4\,x_1 + 5\,x_2$$

$$\text{tel que} \begin{cases} 2\,x_1 + x_2 \geq 10 \\ 3\,x_1 + 6x_2 \leq 80 \\ 5\,x_1 + 7\,x_2 \leq 50 \\ x_1, x_2 \geq 0 \end{cases}$$

$$\min_{X\in\mathbb{R}^3} \quad f(x_1,x_2,x_3) = x_1^2 + x_1\,x_2 + 2\,x_2^2 + 2\,x_3^2 + 2\,x_2\,x_3 + 4\,x_1 + 6\,x_2 + 12\,x_3$$

$$\text{tel que} \begin{cases} x_1 + x_2 + x_3 \geq 6 \\ -x_1 - x_2 + 2\,x_3 \geq 2 \\ 0 \leq x_1, x_2, x_3 \leq 100 \end{cases}$$

– Réécrire la *fonction objective* selon la notation générale décrite par l'Eq. (56).
– Trouver les coordonnées du point critique en utilisant la commande `quadprog`.

Réécrivant d'abord ce système selon la notation générale décrite par l'Eq. (56).

$$\min_{X\in\mathbb{R}^2} \frac{1}{2} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} -2 & -1 \\ 3 & 6 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 10 \\ 80 \\ 50 \end{bmatrix}$$

D'après la définition sur les contraintes d'inégalités, elles doivent être écrites, inférieure ou égale à une constante. Ainsi, la contrainte $2\,x_1 + x_2 \geq 10$ est réécrite sous la forme $-2\,x_1 - x_2 \leq 10$. Voici le script Matlab®, pour la fonction à deux variables

```
clc ; clear all ;

H = [2 0 ; 0 0] ; f = [4 ; 5] ;
A = [-2 -1 ; 3 6 ; 5 7] ; B = [10 ; 80 ; 50] ; lB = [0 ; 0] ;
uB = [inf ; inf] ;

options = optimset('LargeScale','off','TolX', 1e-7) ;
[x, fval, exitflag, output, lambda] = quadprog(H, f, A, B, [],[], ...
    lB, uB, [], options) ;

if exitflag > 0

   disp('L''algorithme a converge vers la solution :')
   point_critique = x
else

    disp('L''algorithme n''a pas converge !')

end
```

Les sorties renvoyées sont comme suit

```
Optimization terminated.
L'algorithme a converge vers la solution :

point_critique =

     0
     0
```

Notons qu'on peut réécrire ce script en considérant l'inégalité $x_1, x_2 \geq 0$ comme deux contraintes séparées selon $-x_1 \leq 0$ et $-x_2 \leq 0$ ce qui renvient à écrire deux nouvelles lignes `[-1 0; 0 -1]` dans la matrice A et `[0; 0]` dans le vecteur B. Dans ce cas `lB = []` et `uB = []`. Voici le script.

```
clc ; clear all ;

H = [2 0 ; 0 0] ; f = [4 ; 5] ;
A = [-2 -1 ; 3 6 ; 5 7 ; -1 0 ; 0 -1] ; B = [10 ; 80 ; 50 ; 0 ; 0] ;

options = optimset('LargeScale','off','TolX', 1e-7) ;
[x, fval, exitflag, output, lambda] = quadprog(H, f, A, B, [],[], ...
    [], [], [], options) ;

if exitflag > 0

   disp('L''algorithme a converge vers la solution :')
   point_critique = x
else
```

```
    disp('L''algorithme n''a pas converge !')

end
```

Script Matlab® pour la fonction à trois variables

```
clc ; clear all ;

H = [2 1 0 ; 1 4 2 ; 0 2 4] ; f = [4 ; 6 ; 12] ;
A = [-1 -1 -1 ; 1 1 -2] ; B = [-6 ; -2] ; lB = [0 ; 0 ; 0] ;
uB = [100 ; 100 ; 100] ; xinit = [1 ; 1 ; 1] ;

options = optimset('Diagnostics','on','TolX', 1e-7) ;
[x, fval, exitflag, output, lambda] = quadprog(H, f, A, B, [],[], ...
    lB, uB, xinit,options) ;

if exitflag > 0

   disp('L''algorithme a converge vers la solution :')
   point_critique = x
else

    disp('L''algorithme n''a pas converge !')

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Diagnostic Information
Number of variables: 3

 Number of linear inequality constraints:    2
 Number of linear equality constraints:      0
 Number of lower bound constraints:          3
 Number of upper bound constraints:          3

Algorithm selected
   medium-scale: active-set
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
End diagnostic information
```

Comme on peut le voir, le champs `optimset('Diagnostics','on', ...)` renvoie des informations sur la *fonction-objectif*, comme le nombre de variables, le nombre d'équation et d'inéquation ... etc.

```
Optimization terminated.
L'algorithme a converge vers la solution :

>> point_critique =
```

```
3.3333
0.0000
2.6667
```

Afin de maximiser la *fonction-objectif* au moyen de la commande `quadprog`, on prendra `-H` et `-f`.

Exercice 7

– On se propose de minimiser les fonctions définies par :

$$\min_{X\in\mathbb{R}^2} \quad f(x_1,x_2) = (x_1-2)^2 + (x_2-2)^2$$

$$\text{tel que} \begin{cases} x_1 + 2\,x_2 \leq 3 \\ 3\,x_1 + 2x_2 \geq 3 \\ x_1 - 2\,x_2 \leq 2 \\ x_1, x_2 \geq 0 \end{cases}$$

$$\min_{X\in\mathbb{R}^3} \quad f(x_1,x_2,x_3) = x_1^3 + x_2^3 + x_3^3$$

$$\text{tel que} \begin{cases} x_1^3 + x_2^3 + x_3^3 = 1 \\ 2\,x_3^3 - x_2^2 \leq 0 \\ x_1 \geq 0 \\ x_3 \leq 0 \end{cases}$$

– Réécrire la *fonction-objectif* selon la notation générale décrite par l'Eq. (56).
– Trouver les coordonnées du point critique en utilisant la commande `quadprog`. Pour la fonction à trois variable prenez le vecteurs des valeurs initiales $(1, 0, -1)$.

Voir également, dans le même sillage, les commandes d'optimisation `fseminf`, `fminimax` et `fgoalattain`.