Chapitre[2] Algorithmes d'optimisation

Samir Kenouche - Département des Sciences de la Matière - UMKB

Module : Méthodes Mathématiques et Algorithmes pour la Physique

Résumé

Un processus d'optimisation consiste à choisir entre plusieurs solutions possibles, celle qui est la meilleure. Il s'agit ainsi de minimiser ou de maximiser un critère sur l'ensemble de toutes les solutions admissibles. Nous aborderons les algorithmes d'optimisation usuels à savoir : *Newton*, *quasi-Newton*, la famille des méthodes de *descente de gradient* et les algorithmes génétiques. La programmation des algorithmes abordés dans ce chapitre sera conduite au moyen de scripts Matlab®. Pendant les séances de cours, nous traiterons uniquement des problèmes d'optimisation non contraints. La résolution de problèmes contraints se fera lors des séances de travaux pratiques. La dernière section de ce chapitre est donnée à cet effet.

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 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* ou encore *critère d'optimisation*. Ainsi, l'opération d'optimisation est formalisée selon l'écriture :

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

S. Kenouche est docteur en Physique de l'Université des Sciences et Techniques de Montpellier et docteur en Chimie de l'Université A. Mira de Béjaia.
page web personnelle : http://www.sites.univ-biskra.dz/kenouche
Version corrigée, améliorée et actualisée le 10.10.2020.

Avec $\hat{X} = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)^T$ sont les coordonnées du point critique. On fait appel aux algorithmes d'optimisation afin de résoudre des problèmes de différentes natures, 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 de 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. En effet, pour résoudre un problème d'optimisation convenablement : (1) il faut bien poser le problème (2) choisir le bon algorithme (3) savoir interpréter les résultats qui en découlent.

*A. **Convergence***

L'étude de la convergence d'une méthode numérique est atteinte à travers la suite des itérés $\left\{X^{(k)}\right\}_{k\in\mathbb{N}}$ générés par l'algorithme. Ce dernier est dit convergent si pour tout $X^{(0)} \in \mathbb{R}^n$ nous avons :

$$\lim_{k\to+\infty} \| X^{(k)} - X^* \| = 0 \tag{2}$$

Avec $X^* \in \mathbb{R}^n$ est la solution approchée, de la valeur exacte, déterminée avec une tolérance fixée préalablement. La condition (2) garantit, à partir d'une certaine itération, la satisfaction du critère d'arrêt pour la tolérance exigée. Nous rappelons que le taux de convergence (ou "vitesse" de convergence) et la complexité du problème traité rentrent en ligne de compte lors de l'utilisation d'un algorithme d'optimisation. La convergence ne signifie pas systématiquement l'existence d'un optimum même local. Le schéma numérique adopté doit être à la fois rapide et robuste. Ces deux derniers critères sont contrôlés par l'étude du taux de convergence qui quantifie l'erreur commise à chaque itération, soit :

$$e^{(k)} = \| X^{(k)} - X^* \| \quad \text{tel que} \quad \nabla f(X^*) = 0 \tag{3}$$

L'estimation de l'erreur servira, entre autre, à comparer le taux de convergence des différentes méthodes numériques. En pratique, l'erreur est représentée graphiquement en traçant la norme de l'erreur, soit $\| e^{(k+1)} \|$ en fonction de $\| e^{(k)} \|$ avec une échelle logarithmique. Ainsi, *l'ordre* noté p, de la méthode numérique s'obtient à partir de :

$$\| e^{(k+1)} \| \approx A \ \| e^{(k)} \|^p \Longrightarrow \log \| e^{(k+1)} \| \approx p \ \log \| e^{(k)} \| + \text{cste} \tag{4}$$

L'ordre, p, est quantifié via la pente de l'équation ci-dessus. Il en ressort les conclusions suivantes :

1) Si $p = 1 \Longrightarrow X^{(k)}$ converge linéairement vers la solution approchée. Dans ce cas on gagne la même quantité de précision à chaque itération. Il en résulte :

$$\lim_{k\to+\infty} = \frac{\| X^{(k+1)} - X^* \|}{\| X^{(k)} - X^* \|} = \tau \quad \text{avec} \quad 0 < \tau < 1 \tag{5}$$

   Elle est dite superlinéaire si $\tau = 0$.
2) Si $p = 2 \Longrightarrow X^{(k)}$ converge quadratiquement vers la solution approchée. Dans ce cas on gagne le double de précision à chaque itération.
3) Si $p = 3 \Longrightarrow X^{(k)}$ converge cubiquement vers la solution approchée. Dans ce cas on gagne le triple de précision à chaque itération. Il en résulte :

$$\lim_{k\to+\infty} = \frac{\| X^{(k+1)} - X^* \|}{\| X^{(k)} - X^* \|^p} = \tau \quad \text{avec} \quad \tau \geq 0 \tag{6}$$

D'un point de vue pratique et pour un k suffisamment élevé, la vitesse de convergence d'une méthode itérative est évaluée "empiriquement" au moyen de la relation :

$$K_p(X,k) = \frac{\| X^{(k+2)} - X^{(k+1)} \|}{\| X^{(k+1)} - X^{(k)} \|^p} \tag{7}$$

Bien évidemment, nous visons à ce que la convergence de l'algorithme soit la plus élevée possible afin de tendre vers la solution en un minimum d'itérations pour la tolérance exigée. La convergence quadratique est plus rapide que celle superlinéaire. A son tour cette dernière, est plus rapide que la convergence linéaire. Tenant compte de l'équation (7), il vient que plus $K_p(X,k)$ tend vers zéro plus le taux de convergence de la méthode est élevé.

B. ***Critères d'arrêt***

Les méthodes numériques, dites locales, procèdent de façon itérative. Typiquement, étant donné une valeur initiale, un nouvel itéré est mis à jour afin de converger vers un point stationnaire. Ce processus est réitéré jusqu'à la satisfaction d'un ou plusieurs critères d'arrêt de l'algorithme. D'un point de vue purement théorique, le schéma itératif est infini. En pratique, la suite d'approximations successives est tronquée dés que l'on considère avoir atteint la précision requise. Étant donné une tolérance ϵ, les critères d'arrêt pouvant êtres envisagés sont :

$$\| X^{(k+1)} - X^{(k)} \| < \epsilon \tag{8}$$

$$\| \nabla f(X^{(k)}) \| < \epsilon \tag{9}$$

$$\frac{\| f(X^{(k+1)}) - f(X^{(k)}) \|}{\| f(X^{(k)}) \|} < \epsilon \tag{10}$$

Il est possible d'envisager également un quatrième critère, celui du nombre d'itérations dépassant un seuil fixé préalablement. Il se peut que le critère (23) ne soit pas satisfait même si l'algorithme converge. Les erreurs d'arrondis dues à l'accumulation des opérations arithmétiques peuvent êtres du même ordre de grandeur que le gain de précision obtenu à l'itération en cours. Le critère (8) est recommandé, le schéma itératif est interrompu lorsqu'il ne produit plus de gain significatif en terme de précision. En outre, dans certaines situations la divergence d'un algorithme ne signifie pas forcément l'inexistence de la solution, il faudra juste adapter le nombre d'itérations et/ou la tolérance considérés. D'un point de vue purement pratique, une combinaison de ces critères est prise en compte.

II. Algorithme de Newton

Comme il a été signalé au début de ce chapitre, l'opération d'optimisation s'apparente systématiquement à une minimisation, car trouver le maximum d'une fonction f revient à minimiser la fonction $-f$ selon l'équivalence :

$$\text{Arg}\max_x f(x) \Leftrightarrow \text{Arg}\min_x(-f(x)) \tag{11}$$

Les deux opérations d'optimisation sont pleinement équivalentes. La méthode de Newton est efficiente, notamment parce qu'elle prend en compte la courbure de la fonction-objectif à travers le calcul de sa seconde dérivée. Cette méthode est praticable si à chaque itération, la dérivée seconde est définie ou la matrice hessienne est définie positive pour le cas $X^{(k)} \in \mathbb{R}^n$. 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 $P_{X^{(k)}}(X) : \mathbb{R}^n \longrightarrow \mathbb{R}$, avec :

$$P_{X^{(k)}}(X) = f(X^{(k)}) + (X - X^{(k)})^T \nabla f(X^{(k)}) + \frac{1}{2!}(X - X^{(k)})^T \nabla^2 f(X^{(k)})(X - X^{(k)}) \tag{12}$$

Avec, $\nabla^2 f(X^{(k)})$ est la matrice Hessienne de f en $X = X^{(k)}$ soit $H_f(X^{(k)}) = \nabla^2 f(X^{(k)})$. Posons $U = X - X^{(k)}$, il vient :

$$P_{X^{(k)}}(X+U) = f(X^{(k)}) + U^T \nabla f(X^{(k)}) + \frac{1}{2!} U^T \nabla^2 f(X^{(k)}) U \tag{13}$$

$$\min\{P_{X^{(k)}}(X+U)\} \Longleftrightarrow \nabla P_{X^{(k)}}(X+U) = 0 \tag{14}$$

Une condition suffisante d'optimalité :

$$\nabla P_{X^{(k)}}(X+U) = \nabla f(X^{(k)}) + \nabla^2 f(X^{(k)}) U = 0 \tag{15}$$

$$\nabla f(X^{(k)}) = -U \nabla^2 f(X^{(k)}) \Rightarrow U = -\frac{\nabla f(X^{(k)})}{\nabla^2 f(X^{(k)})} \tag{16}$$

$$X = X^{(k)} - \frac{\nabla f(X^{(k)})}{\nabla^2 f(X^{(k)})} \quad \text{avec} \quad U = X - X^{(k)} \tag{17}$$

On calcule alors un nouveau point, $X^{(k+1)}$, qui minimise $P_{X^{(k)}}$ soit :

$$X^{(k+1)} = X^{(k)} - \frac{\nabla f(X^{(k)})}{\nabla^2 f(X^{(k)})} \tag{18}$$

Ou bien avec une écriture équivalente :

$$X^{(k+1)} = X^{(k)} - H_f(X^{(k)})^{-1} f(X^{(k)}) \tag{19}$$

Ci-dessous l'écriture algorithmique de la méthode de *Newton*

Algorithm 1 Algorithme de Newton

Input : $f(X) \in \mathcal{C}^2$, $X^{(0)} \in \mathbb{R}^n$
$k \longleftarrow 0$

1. **Tant que** critère d'arrêt n'est pas vérifié faire :
   2. Calcul du gradient $G_k \longleftarrow -\nabla f(X^{(k)})$
   3. Calcul de H_f^{-1} $H_k \longleftarrow [H_f(X^{(k)})]^{-1}$
   4. Nouvelle itération $X^{(k+1)} \longleftarrow X^{(k)} + G_k H_k$
   5. Mise à jour : $k \longleftarrow k+1$
6. **Fin**

Appliquons le schéma numérique (19) à la fonction-objectif ci-dessous.

Exercice ❶ ☞ ®

Soit la fonction-objectif de trois variables, $f(X)$ tel que $X \in \mathbb{R}^3$, suivante :

$$f(X) = \frac{1}{4}\,x_3^4 + x_2^3 - \frac{1}{3}\,x_3^3 + 6\,x_1^2 + 3\,x_2^2 + 6\,x_1\,x_2 \tag{20}$$

La tolérance vaut $\epsilon = 10^{-5}$ et pour l'évaluation numérique prendre $X^{(0)} = (-0.6637, 0.1758, 1.5778)^T$.

1) Trouver analytiquement l'optimum de la fonction-objectif.
2) Évaluer numériquement la fonction-objectif en utilisant l'algorithme de Newton.
3) Écrire un script Matlab® de l'algorithme de Newton permettant sa minimisation.

Les résolutions analytique et numérique seront menées pendant la séance de cours. Le script Matlab® est donné ci-dessous

```
clear all ; clc ; close all ;

% Samir KENOUCHE - LE 08/08/2019
% ALGORITHME DE NEWTON

fx = @(x) (1/4)*x(3)^4 + x(2)^3 - (1/3)*x(3)^3 + 6*x(1)^2 + ...
    3*x(2)^2 + 6*x(1)*x(2) ;

rng(1110,'twister')  ; % INITIALISATION DU GENERATEUR DES NOMBRES ALEATOIRES
xinit = randn(1,3)' ;  % INITIALISATION ALEATOIRE

it = 0 ; itmax = 20 ;  % NOMBRE D'ITERATIONS
err = 10.^(-5) ;       % TOLERANCE

while it < itmax

    Hx(1,1) = 12 ; Hx(1,2) = 6 ; Hx(1,3) = 0 ;
    Hx(2,1) = 6 ; Hx(2,2) = 6*xinit(2) + 6 ; Hx(2,3) = 0 ;
    Hx(3,1) = 0 ; Hx(3,2) = 0 ; Hx(3,3) = 3*xinit(3)^2 - 2*xinit(3) ;

    dfx = [12*xinit(1) + 6*xinit(2) ; ...
         3*xinit(2)^2+ 6*xinit(1) + 6*xinit(2) ; xinit(3)^3 - xinit(3)^2] ;

    xk = xinit - inv(Hx)*dfx  ;

         if  norm(inv(Hx)*dfx) <= err   % TEST D'ARRET
               solution = xk ;          % OPTIMUM OBTENU
               f_min = fx(solution) ;   % MINIMUM DE LA FONCTION-OBJECTIF
               iteration_max = it ;     % NOMBRE D'ITERATIONS PERMETTANT LA CV
          break
         end

    xinit = xk ; it = it + 1 ;          % MISE A JOUR
end

%% SOLUTION : X* =(0, 0, 1) EN 04 ITERATIONS
```

Il est possible de voir la méthode de *Newton* comme un algorithme de descente de gradient à pas fixe valant l'unité. La famille des algorithmes de descente de gradient fera l'objet des sections suivantes. Nonobstant, la simplicité et la convergence quadratique de l'algorithme de Newton, ce dernier 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 certaines situations, même si la dérivée seconde ou le Hessien pour la cas multidimensionnel existe, son calcul peut se révéler fastidieux ou très laborieux ou encore impossible à atteindre.

***Nota Bene**: Toutes les méthodes numériques ont une structure d'une suite numérique. Ce qui les distinguent, c'est la formule du schéma numérique de la méthode en question. Toute suite de nombres réels (ou de nombre complexe $\mathbb{C}$) est convergente si et seulement si elle vérifie le critère de Cauchy.*

***Définition d'une suite de Cauchy**: Une suite $\left\{X^{(k)}\right\}_{k\in\mathbb{N}}$ est de Cauchy (ou vérifie le critère de Cauchy) si pour un rang $k \in \mathbb{N}$ suffisamment grand nous avons :*

$$\lim_{k\longrightarrow+\infty} \| X^{(k)} - X^{*} \| = 0 \tag{21}$$

Ce critère s'énonce aussi,

$$\forall \epsilon > 0, \exists\, \eta \quad tel\ que \quad k > \eta \quad (ou \quad \eta(\epsilon)) \Rightarrow \| X^{(k)} - X^{*} \| \leq \epsilon$$

Une façon de démontrer le théorème consiste à démontrer que $\left\{X^{(k)}\right\}_{k\in\mathbb{N}}$ est une suite de Cauchy[1]*. Soit $q \in [0\,;\,1[$ et $\forall k \in \mathbb{N}$, nous avons :*

$$\| f(X^{(k)}) - f(X^{(k-1)}) \| \leq q \ \| X^{(k)} - X^{(k-1)} \|$$

$$\Rightarrow \quad \| X^{(k+1)} - X^{(k)} \| \leq q \ \| X^{(k)} - X^{(k-1)} \| \tag{22}$$

D'un autre côté, nous pouvons écrire

$$\| X^{(k)} - X^{(k-1)} \| \leq q \ \| X^{(k-1)} - X^{(k-2)} \| \tag{23}$$

En substituant (23) dans (24) nous obtenons :

$$\Rightarrow \quad \| X^{(k+1)} - X^{(k)} \| \leq q^2 \ \| X^{(k-1)} - X^{(k-2)} \| \tag{24}$$

En procédant par récurrence nous démontrons[2] *:*

$$\Rightarrow \quad \| X^{(k+1)} - X^{(k)} \| \leq q^k \ \| X^{(1)} - X^{(0)} \| \tag{25}$$

Avec $q \in [0\,,\,1[$, la distance $\| X^{(k+1)} - X^{(k)} \|$ tend vers zéro pour k assez grand alors $\left\{X^{(k)}\right\}_{k\in\mathbb{N}}$ est une suite de Cauchy donc elle est convergente.

1. Rappelons que toute suite convergente dans $\mathbb{R}$ est une suite de Cauchy.
2. Pour les puristes, nous opterons plutôt pour $\| X^{(k+1)} - X^{(k)} \| \leq \dfrac{q^k}{1-q} \ \| X^{(1)} - X^{(0)} \|$

A. *Algorithme de quasi-Newton*

L'intérêt d'utiliser l'algorithme de quasi-Newton par rapport à celui de Newton tient aux éléments suivants : (1) le Hessien n'est pas calculé explicitement, ce dernier exige un temps de calcul souvent très élevé (2) la détermination de la direction de descente d_k ne nécessite qu'une multiplication matrice-vecteur, sans se soucier du calcul de gradient (3) les approximations du Hessien $H^{(k+1)}$ sont définies positives, garantissant ainsi que les directions de recherche sont systématiquement descendantes. Le schéma numérique de Broyden-Fletcher-Goldforb-Shanno, communément appelé BFGS est :

$$H^{(k+1)} = H^{(k)} + \left(1 + \frac{y_k^T\, H^{(k)}\, y_k}{y^T\, d_k}\right) \frac{d_k\, d_k^T}{y_k^T\, d_k} - \frac{H^{(k)}\, y_k\, d_k^T + d_k\, y_k^T\, H^{(k)}}{y_k^T\, d_k} \tag{26}$$

Avec,

$$y_k = \nabla f(X^{(k+1)}) - \nabla f(X^{(k)})$$

D'autres versions de cet algorithme existent aussi, nous citerons notamment la formule de Davidon-Fletcher-Powell, communément appelée DFP. L'inconvénient des méthodes de quai-Newton est que l'on exploite pas toute l'information portant sur la forme de la fonction-objectif. Par ailleurs, l'initialisation de $H_0 \in \mathbb{R}^{n\times n}$ se fait très souvent avec la matrice identité $I \in \mathbb{R}^{n\times n}$. La pratique montre que ce choix semble donner des résultats très probants. Ci-dessous, l'écriture algorithmique de la méthode de *Quasi-Newton*

Algorithm 2 Algorithme de Quasi-Newton

Input : $f(X) \in \mathcal{C}^1$, $X^{(0)} \in \mathbb{R}^n$, $H_0 = I_{n\times n}$
$k \longleftarrow 0$

1. **Tant que** critère d'arrêt n'est pas vérifié faire :
2. Calcul du gradient $G_k \longleftarrow -\nabla f(X^{(k)})$
3. Pas optimal $\lambda_k \longleftarrow \text{argmin}(X^{(k)} + \lambda\, G_k)$
4. La descente $d_k \longleftarrow \lambda_k\, G_k\, H_0$
5. Nouvelle itération $X^{(k+1)} \longleftarrow X^{(k)} + d_k$
6. Calcul intermédiaire $y_k \longleftarrow \nabla f(X^{(k+1)}) - \nabla f(X^{(k)})$
7. Approx. de la Hess. $H^{(k+1)} \longleftarrow H^{(k)} + \left(1 + \frac{y_k^T\, H^{(k)}\, y_k}{y^T\, d_k}\right) \frac{d_k\, d_k^T}{y_k^T\, d_k} - \frac{H^{(k)}\, y_k\, d_k^T + d_k\, y_k^T\, H^{(k)}}{y_k^T\, d_k}$
8. Mise à jour : $k \longleftarrow k+1$
9. **Fin**

Exercice ❷ ☞ ®

Nous appliquerons cette méthode pour chercher le minimum de la fonction-objectif à deux dimensions suivante :

$$\begin{cases} f(X) = \dfrac{1}{2}\, x^2 + \dfrac{7}{2}\, y^2 \\ \\ \text{Avec} \quad X^{(0)} = (7.5, 2.2) \end{cases} \tag{27}$$

La tolérance considérée est $\epsilon = 10^{-5}$. Ci-dessous, le script Matlab®

```
close all ; clc ; clear all ;

% Le 12.08.2019 - Samir KENOUCHE
% Algorithme de BFGS - quasi-Newton

n = 2 ; Hinit = eye(n,n) ;
fxy = @(x,y) x.^2*(1/2) + y.^2*(7/2) ; dfx = @(x) x ; dfy = @(y) 7.*y ;
it = 0 ; itmax = 10 ; tol =  1e-5 ; xinit = [7 ; 2] ;

while it < itmax

   d = - Hinit*[dfx(xinit(1)) ; dfy(xinit(2))] ;
   lambdak = -(xinit(1)*d(1) + 7*xinit(2)*d(2))/(d(1).^2 + 7*d(2).^2) ;

   dk = lambdak*d ; xk = xinit + dk ;

   if norm(dk) < tol
     solution = xk   ; % VECTEUR SOLUTION
     nbr_it = it ;     % NOMBRE D'ITERATION PERMETTANT LA CV
       break
   end

   yk = [dfx(xk(1)) ; dfy(xk(2))] - [dfx(xinit(1)) ; dfy(xinit(2))] ;
   Hk = Hinit + (1 + (yk'*Hinit*yk)/(yk'*dk))*(dk*dk')/(yk'*dk) - ...
       (Hinit*yk*dk' + dk*yk'*Hinit)/(yk'*dk) ;

   Hinit = Hk ; xinit = xk ;  it = it + 1 ; % MISE A JOUR
end

solution
```

Les méthodes de Newton et de quasi-Newton peuvent rencontrer des problèmes tels que le calcul du Hessien soit trop complexe ou qu'il n'existe pas. La nécessité d'appliquer une inversion matricielle à chaque itération, ceci peut se révéler rédhibitoire pour des problèmes d'optimisation mobilisant beaucoup de variables. Ces méthodes peuvent donc devenir impraticables. Une alternative consiste à utiliser la famille des algorithmes de descente de gradient. Ces méthodes ne requièrent pas le calcul explicite ou l'approximation du Hessien. Elles n'exigent pas *de facto* le stockage du Hessien ($\mathbb{R}^{n\times n}$), mais seulement un ou quelques vecteurs ($\in \mathbb{R}^n$).

III. Algorithmes de descente de gradient

Un algorithme de descente de gradient est mis en œuvre suivant les modes de choix des *directions* de descente successives, ensuite par l'amplitude du *pas* effectué dans la direction choisie. La famille des algorithmes de descente de gradient est à la base des processus d'optimisation de problèmes plus au moins complexes. Le terme *descente* vient du fait que cet algorithme cherche l'extremum suivant une direction opposée à celle du gradient de la *fonction-objectif*. Le pas de descente λ_k est soit constant (*méthode de gradient à pas fixe*) ou bien incrémenté selon les méthodes de *Wolfe*, de *gradient à pas optimal* et de *gradient conjugué*. Typiquement, le pas de descente est obtenu au moyen d'une recherche linéaire vérifiant : $f(x^{(k)} + \lambda_k \nabla f(x^{(k)})) < f(x^{(k)})$. Dans cette section, nous rappellerons succinctement la notion de la *dérivée directionnelle*. Cette étape est importante afin de rendre compte du principe de fonctionnement des algorithmes de descente de gradient. Plus de détails sur cette notion sont disponibles *en annexe* (A).

Par définition, 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 :

$$f_{\overrightarrow{d}}(x_0, y_0) = \lim_{\lambda \to 0} \frac{f(x_0 + \lambda\, a, y_0 + \lambda\, b) - f(x_0, y_0)}{\lambda} = \nabla f(x_0, y_0)\ \overrightarrow{d} \tag{28}$$

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* $||\nabla f(x_0, y_0)||$.

Ce théorème peut être prouvé en considérant que $f_{\overrightarrow{d}}(x_0, y_0) = \nabla f(x_0, y_0)\ \overrightarrow{d} = ||\nabla f(x_0, y_0)||\,||\overrightarrow{d}||\cos(\theta) = ||\nabla f(x_0, y_0)||\cos(\theta)$. *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* $f(x,y)$. *Cette condition est prouvée selon :*

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

Théorème*: Soit* f *une fonction* $\mathcal{C}^1$ *sur un ouvert* $\mathcal{D}$ *de* $\mathbb{R}^2$ *et* P_0 *un point d e*$\mathcal{D}$ *tel que* $f(P_0) = k$. *Nous supposons que le* ∇f *est non nul au point* $P_0 = (x_0, y_0)$. *La tangente en* P_0 *à la courbe d'équation* $f(x,y) = c$ *a comme équation :*

$$\nabla f(x_0, y_0) \cdot \vec{P_0 P} = 0 \tag{29}$$

Ou de façon équivalente :

$$\frac{\partial f(x_0, y_0)}{\partial x}\ (x - x_0) + \frac{\partial f(x_0, y_0)}{\partial y}\ (y - y_0) = 0 \tag{30}$$

Démonstration*: Nous travaillerons sur la courbe de niveau* $f(x,y) = 0$. *Le point* $P_0(x_0, y_0)$ *peut se déplacer sur la courbe de niveau de sorte que des coordonnées changent en fonction du temps. Cela nous permet d'écrire* $P_0(x_0 + a\,t,\ y_0 + b\,t)$. *Ainsi nous obtenons l'équation paramétrique (de paramètre* t*) :*

$$f(x_0 + a\,t,\ y_0 + b\,t) = f(x(t), y(t)) = 0 \tag{31}$$

En dérivant (31) *nous obtenons :*

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\ \frac{dx(t)}{dt} + \frac{\partial f}{\partial y}\ \frac{dy(t)}{dt} = 0 \tag{32}$$

Au point $P_0(x_0, y_0)$ *correspondant à* $t = 0$, *il vient :*

$$\frac{df(x_0, y_0)}{dt} = \frac{\partial f(x_0, y_0)}{\partial x}\ \frac{dx(t)}{dt} + \frac{\partial f(x_0, y_0)}{\partial y}\ \frac{dy(t)}{dt} = 0 \tag{33}$$

$$\Rightarrow \quad \frac{df(x_0,y_0)}{dt} = \frac{\partial f(x_0,y_0)}{\partial x}\, a + \frac{\partial f(x_0,y_0)}{\partial y}\, b = 0 \tag{34}$$

$$\Rightarrow \quad \frac{df(x_0,y_0)}{dt} = \nabla f(x_0,y_0)\cdot(a,\, b) \tag{35}$$

Avec a et b sont les coordonnées de la pente de l'équation tangente au point (x_0, y_0). Nous concluons ainsi que le gradient de f est toujours orthogonal (produit scalaire (35) est nul) aux courbes de niveau.

A. *Méthode de gradient à pas fixe*

L'idée de base de cette approche 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 :

$$f(x_{k+1}) < f(x_k) \tag{36}$$

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

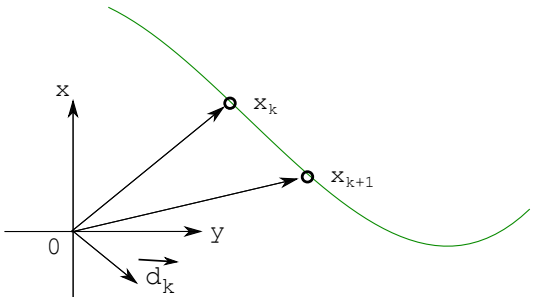

FIGURE 1: Dérivée directionnelle dans la direction $\vec{d}_k$.

Compte tenu de l'Eq. (92) (voir l'annexe), à une dimension on obtient :

$$f_{\vec{d}}(x_k) = \lim_{\lambda\to 0}\frac{f(x_{k+1})-f(x_k)}{\lambda} \Rightarrow f_{\vec{d}}(x_k) = \lim_{\lambda\to 0}\frac{f(x_k+\lambda\, a)-f(x_k)}{\lambda} \tag{37}$$

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,

$$\overrightarrow{x_k x_{k+1}} = \overrightarrow{Ox_{k+1}} - \overrightarrow{Ox_k} \quad \Rightarrow \quad \overrightarrow{Ox_{k+1}} = \overrightarrow{Ox_k} + \overrightarrow{x_k x_{k+1}}$$

$$\Rightarrow \quad \overrightarrow{Ox_{k+1}} = \overrightarrow{Ox_k} + \lambda\, \overrightarrow{d_k} \tag{38}$$

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

$$x^{(k+1)} = x^{(k)} + \lambda\, d_k = x^{(k)} - \lambda \nabla f(x^{(k)}), \qquad \lambda > 0 \tag{39}$$

Ainsi,

$$f(x_{k+1}) < f(x_k) \Leftrightarrow f(x^{(k)} - \lambda \nabla f(x^{(k)}) < f(x_k) \tag{40}$$

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 descente qui est constant dans cette version de l'algorithme. Notons que la vitesse de convergence de cet algorithme est proportionnelle à la valeur de λ. Une valeur initiale pour incrémenter l'algorithme est $\lambda = 0.01$. Cette méthode est itérative, elle a donc besoin bien évidemment d'une valeur initiale pour démarrer l'incrémentation de l'algorithme. Ci-dessous, l'écriture algorithmique de la méthode de gradient à pas fixe.

Algorithm 3 Algorithme de descente de gradient à pas fixe

Input : $f(X) \in \mathcal{C}^1$, $X^{(0)} \in \mathbb{R}^n$, $\lambda \in \mathbb{R}_+^*$
$k \longleftarrow 0$

1. **Tant que** critère d'arrêt n'est pas vérifié faire :
   2. Direction de descente $d_k \longleftarrow -\nabla f(X^{(k)})$
   3. Nouvelle itération $X^{(k+1)} \longleftarrow X^{(k)} + \lambda_k\, d_k$
   4. Mise à jour : $k \longleftarrow k+1$
5. **Fin**

Afin d'illustrer le principe de fonctionnement de cet algorithme, nous chercherons à optimiser une fonction-objectif d'une seule variable.

Exercice ❸ ☞ ®

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 $\mathbb{R}$. Exécutons 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 :

$$x^{(k+1)} = x^{(k)} + \lambda\, d_k = x^{(k)} - \lambda \nabla f(x^{(k)}) = x^{(k)} - \lambda \left[8\,x^{(k)} + e^{x^{(k)}}\right]$$

Commençons par appliquer le schéma numérique ci-dessus,

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

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

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

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

Exercice ❹ ☞ Ⓡ

Soit la fonction-objectif ci-dessous :

$$\begin{cases} f(x) = \cos(2\,x)\,\sqrt{x^2+1} \\ \text{Avec} \quad x_0 = -2,\ x \in [-4, 0] \end{cases} \tag{41}$$

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

Script Matlab®

```
clear all ; close all ; clc ;
% Le 05.08.2019 - 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)
```

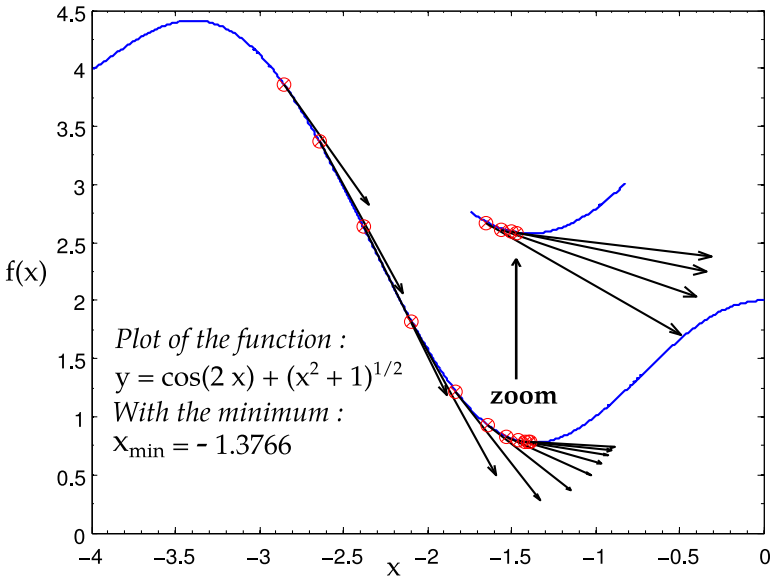

FIGURE 2: Minimisation avec la *descente de gradient à pas fixe*

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. Nous faisons remarquer que le critère d'arrêt $\parallel X^{(k+1)} - X^{(k)} \parallel$ est totalement équivalent à celui de $\parallel \lambda_k \nabla f X^{(k)} \parallel$.

*B. **Méthode de gradient à pas optimal***

L'inconvénient d'utiliser un pas de descente constant ($\lambda = cst$) 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 de la valeur de λ devient délicat dans ce cas de figure. Il faudra par conséquent ajuster la valeur de λ à chaque itération. Cette procédure est appelée *recherche en ligne* (line search) [3]. Nous appliquerons à cet effet la méthode de *gradient à pas optimal*. L'idée de base de cette méthode est de chercher λ_k diminuant d'avantage $f(x^{(k)})$ dans la direction $d_k = -\nabla f(x^{(k)})$, selon :

$$\varphi(\lambda_k) = \underset{\lambda>0}{\operatorname{argmin}}\, f(x^{(k)} - \lambda \nabla f(x^{(k)})) \Leftrightarrow \varphi^{'}(\lambda_k) = 0 \Leftrightarrow \varphi(\lambda_{opt}) \leq \varphi(\lambda_k) \tag{42}$$

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 ? Cherchons la valeur du pas λ_k minimisant une fonction-objectif $f(X^{(k+1)})$, avec $X \in \mathbb{R}^n$, dans la direction de descente d_k. Nous avons :

$$f(X^{(k+1)}) = f(X^{(k)} + \lambda_k\, d_k) \tag{43}$$

Dans ce chapitre, il est question que de fonctions quadratiques. Ainsi la fonction $f(X)$ peut se mettre sous la forme :

$$f(X) = \frac{1}{2}\, X^T\, A\, X - b^T\, X \Rightarrow \nabla f(X) = A\, X - b = g \tag{44}$$

3. Notons qu'il existe d'autres méthodes de recherche linéaire moins restrictives. Nous citerons les méthodes de Armijo, de Goldstein et de Wolf. Ces méthodes autorisent n'ont pas un choix unique du pas optimal, mais un intervalle de valeurs.

Avec A une matrice symétrique définie positive. Le schéma numérique de l'algorithme de descente de gradient impose :

$$X^{(k+1)} = X^{(k)} - \lambda_k \nabla f(X^{(k)}) \tag{45}$$

Nous cherchons un pas optimal tel que :

$$\lambda_k = \underset{\lambda \geq 0}{\operatorname{argmin}} f(X^{(k)} - \lambda \nabla f(X^{(k)})) \tag{46}$$

Ainsi,

$$\frac{\partial f(X^{(k+1)})}{\partial \lambda}(\lambda = \lambda_k) = 0 \Leftrightarrow \nabla f(X^{(k+1)})\ \nabla f(X^{(k)}) = 0 \tag{47}$$

Tenant compte de (44) il vient

$$\begin{aligned}
\nabla f(X^{(k+1)})\ \nabla f(X^{(k)}) &= \Big[A\, X^{(k+1)} - b\Big]^T\ \nabla f(X^{(k)}) = 0 \\
&= \Big[A\ \Big[X^{(k)} - \lambda_k \nabla f(X^{(k)})\Big] - b\Big]^T\ \nabla f(X^{(k)}) = 0 \\
&= \Big[A\, X^{(k)} - b - A\, \lambda_k \nabla f(X^{(k)})\Big]^T\ \nabla f(X^{(k)}) = 0 \\
&= (A\, X^{(k)} - b)^T \nabla f(X^{(k)}) - \lambda_k \nabla f(X^{(k)})^T\, A \nabla f(X^{(k)}) = 0
\end{aligned}$$

$$\Rightarrow \lambda_k = \frac{(A\, X^{(k)} - b)^T \nabla f(X^{(k)})}{\nabla f(X^{(k)})^T\, A \nabla f(X^{(k)})} = \frac{\nabla f(X^{(k)})^T \nabla f(X^{(k)})}{\nabla f(X^{(k)})^T\, A \nabla f(X^{(k)})} \tag{48}$$

La formule générale de la méthode de descente de gradient à pas optimal dans la direction d_k est :

$$X^{(k+1)} = X^{(k)} + \frac{d_k^T\, g_k}{d_k^T\, A\, d_k}\, d_k \quad \text{avec} \quad g_k = \nabla f(X^{(k)}) \quad \text{et} \quad d_k^T = \nabla f(X^{(k)})^T \tag{49}$$

Algorithm 4 Algorithme de descente de gradient à pas optimal

Input : $f(X) \in \mathcal{C}^1$, $X^{(0)} \in \mathbb{R}^n$
$k \longleftarrow 0$

1. **Tant que** critère d'arrêt n'est pas vérifié faire :
   2. Direction de descente $d_k \longleftarrow -\nabla f(X^{(k)})$
   3. Trouver un pas tel que $\lambda_k \longleftarrow \min\limits_{\lambda_k \geq 0} f(X^{(k)} - \lambda \nabla f(X^{(k)}))$
   4. Nouvelle itération $X^{(k+1)} \longleftarrow X^{(k)} + \lambda_k\, d_k$
   5. Mise à jour : $k \longleftarrow k + 1$
6. **Fin**

Exercice ❺ ☞ ®

Illustrons la méthode de gradient à pas optimal pour la fonction-objectif de deux variables suivante :

$$f(X) = 4\,(x^2 + y^2) - 2\,x\,y - 6\,(x + y) \quad \text{avec} \quad X = (x, y)^T \tag{50}$$

Le gradient s'écrit :

$$\nabla f(X) = \begin{pmatrix} 8\,x - 2\,y - 6 \\ -2\,x + 8\,y - 6 \end{pmatrix} \tag{51}$$

Et le Hessien,

$$H(f) = \nabla^2 f(X) = \begin{pmatrix} 8 & -2 \\ -2 & 8 \end{pmatrix} \tag{52}$$

La Matrice Hessienne est symétrique définie positive. La fonction $f(X)$ est strictement convexe et unimodale. Désormais notons $d_k = (d_1, d_2)^T$ la direction de gradient, soit :

$$\nabla f(X) = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} 8\,x - 2\,y - 6 \\ -2\,x + 8\,y - 6 \end{pmatrix} \tag{53}$$

Nous avons aussi :

$$\varphi(X + \lambda_k\,\nabla f(X)) = 4\,(x + \lambda_k\,d_1)^2 + 4\,(y + \lambda_k\,d_2)^2 - 2\,(x + \lambda_k\,d_1)\,(y + \lambda_k\,d_2) - 6\,(x + \lambda_k\,d_1 + y + \lambda_k\,d_2)$$

calculons la dérivée :

$$\begin{aligned} &\varphi^{'}(\lambda_k) = 0 \\ &\varphi^{'}(\lambda_k) = 8\,(x + \lambda_k\,d_1)\,d_1 + 8\,(y + \lambda_k\,d_2)\,d_2 - 2\,(x + \lambda_k\,d_1)\,d_2 - 2\,(y + \lambda_k\,d_2)\,d_1 - 6\,(d_1 + d_2) = 0 \\ &\Longrightarrow \lambda_k = \frac{6\,(d_1 + d_2) + 2\,(x\,d_2 + y\,d_1) - 8\,(x\,d_1 + y\,d_2)}{8\,(d_1^2 + d_2^2) - 4\,d_1\,d_2} \end{aligned}$$

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

$$(x^{(k+1)}, y^{(k+1)}) = (x^{(k)}, y^{(k)}) - \frac{6\,(d_1 + d_2) + 2\,(x\,d_2 + y\,d_1) - 8\,(x\,d_1 + y\,d_2)}{8\,(d_1^2 + d_2^2) - 4\,d_1\,d_2}\,(\underbrace{8\,x^{(k)} - 2\,y^{(k)} - 6}_{d_1}, \underbrace{-2\,x^{(k)} + 8\,y^{(k)} - 6}_{d_2})$$

Script Matlab® :

```
clc ; clear all ; close all ;

% Le 08/08/2019 - Samir KENOUCHE
% METHODE DE GRADIENT A PAS OPTIMAL

 fxy = @(x,y) 4*(x.^2 + y.^2) - 2*x*y - 6*(x+y) ;  itmax = 5 ; it = 0 ;
 dfx = @(x,y) 8*x - 2*y - 6 ; dfy = @(x,y) -2*x + 8*y - 6 ;

while it < itmax

   xinit = [.3 .4] ;
   df = - [dfx(xinit(1),xinit(2)) dfy(xinit(1),xinit(2))] ;

   lambdak = (6*(df(1) + df(2)) + 2*(xinit(1)*df(2) + xinit(2)*df(1)) - ...
         8*(xinit(1)*df(1)+ xinit(2)*df(2)))/(8*(df(1).^2 + df(2).^2) - ...
         4*df(1)*df(2)) ;       % PAS OPTIMAL EXACTE

   xk = xinit + lambdak*df      % NOUVEL ITERE

   xinit = xk ; it = it + 1 ;  % MISE A JOUR

end
```

```
f_min = fxy(xk(1),xk(2)) ; % MIN DE LA FONCTION-OBJECTIF

%%%%%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%%%%%%%
x = -10:0.3: 8 ; y = -10:0.3:8 ; [xgrid, ygrid] = meshgrid(x,y) ;
zgrid = fxy(xgrid,ygrid) ; figure('color',[1 1 1]) ;
contourf(xgrid,ygrid,zgrid) ; xlabel('x') ; ylabel('y') ; hold on ;
plot(xk(1),xk(2),'rx') ; plot(xk(1),xk(2),'ro') ;
```

Exercice ❻ ☞ ®

Nous appliquerons cette méthode pour chercher le minimum de la fonction :

$$\begin{cases} f(x,y) = \dfrac{1}{2}\,x^2 + \dfrac{7}{2}\,y^2 \\ \text{Avec} \quad x_0 = (7.5, 2.2) \end{cases} \tag{54}$$

Avec un raisonnement analogue que précédemment, nous obtenons l'expression analytique du pas optimal

$$\lambda_k = \frac{-(x^{(k)}\,d_1 + 7\,y^{(k)}\,d_2)}{d_1^2 + 7\,d_2^2} \tag{55}$$

Il en résulte le schéma numérique :

$$(x^{(k+1)}, y^{(k+1)}) = (x^{(k)}, y^{(k)}) - \frac{(x^{(k)}\,d_1 + 7\,y^{(k)}\,d_2)}{d_1^2 + 7\,d_2^2}\,(\underbrace{x^{(k)}}_{d_1}, \underbrace{7\,y^{(k)}}_{d_2}) \tag{56}$$

Script Matlab® :

```
clear all ; close all ; clc ;

% Le 09/08/2019 - Samir KENOUCHE
% GRADIENT A PAS OPTIMAL

fxy = @(x,y) x.^2*(1/2) + 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 =  fxy(xgrid, ygrid) ; 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))] ;
   lambdak = -(xinit(1)*dk(1) + 7*xinit(2)*dk(2))/(dk(1).^2 + 7*dk(2).^2) ;

   xk = xinit + lambdak*dk ;

  %%%%%%%%%%%%%%%%%%%%%%%%%%%% TEST D'ARRET %%%%%%%%%%%%%%%%%%%%%%%%%%%%
      if norm(lambdak*dk) <= tol
         solution = xk ;           % SOLUTION OBTENUE
         nbre_it = it ;            % NOMBRE D'ITERATION PERMETTANT LA CV
```

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

 plot(xk(1), xk(2),'xw','MarkerSize',7) ; hold on ;
 plot(xk(1), xk(2),'ow','MarkerSize',7) ; hold on ;
 quiver(xk(1), xk(2), -dfx(xk(1)),- dfy(xk(2)),...
     echelle,'Color', 'r', 'LineWidth',1) ;

 xinit = xk ;  it = it + 1 ;      % MISE A JOUR
end

f_min = fxy(solution(1),solution(2)) ; % MIN DE LA FONCTION-OBJECTIF
```

L'affichage graphique généré par ce script est :

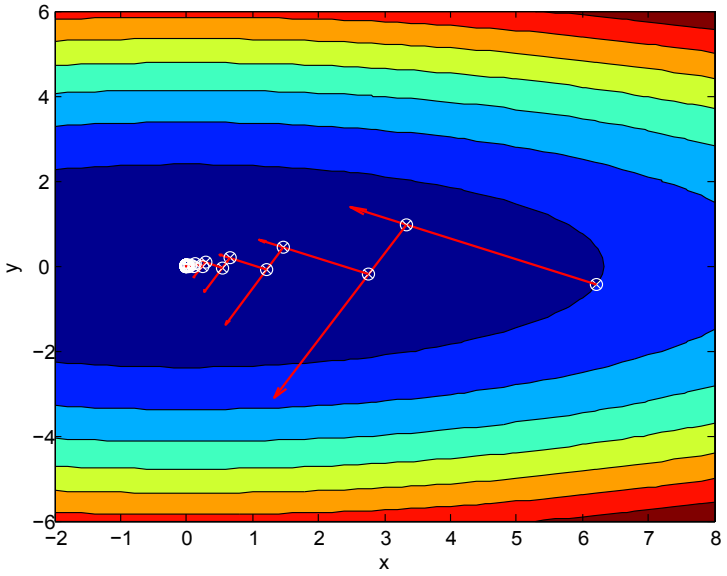

FIGURE 3: Minimisation par la méthode de la *descente de gradient à pas optimal*

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

$$\varphi(\lambda) = d_k \, \nabla f(x^{(k+1)}) \tag{57}$$

La recherche du minimum impose de calculer la dérivée :

$$\varphi^{'}(\lambda) = 0 \Leftrightarrow < d_k | \nabla f(x^{(k+1)}) >= 0 \Leftrightarrow - < \nabla f(x^{(k)}) | \nabla f(x^{(k+1)}) >= 0 \tag{58}$$

Ainsi, le produit scalaire des deux gradients est nul, par conséquent les directions de descente successives calculées par l'algorithme sont orthogonales. Ceci explique pourquoi la convergence suit une trajectoire en zigzag à angles droits. 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 à un problème spécifique.

C. *Méthode de gradient conjugué*

Cette méthode procède par la détermination successive de directions de recherche et de longueurs de descente. Contrairement aux autres méthodes de descente qui rendent compte uniquement du comportement local de la fonction-objectif qui se matérialise par une structure en zigzag. La méthodes du gradient conjugué permet de s'affranchir de cette structure en déterminant des directions de recherche différentes des directions précédentes.

Définition: Soit une matrice $A \in \mathbb{R}^{n\times n}$ définie positive. Deux directions d_{k+1} et d_k de $\mathbb{R}^n$ sont conjuguées par rapport à la matrice A si

$$d_{k+1}^T\, A\, d_k = 0 \qquad \forall k \tag{59}$$

En outre, si $A = I$ (matrice identité), les directions conjuguées d_{k+1} et d_k sont orthogonales. Nous cherchons ainsi d_{k+1} dans le plan formé par les directions orthogonales d_k et g_{k+1} soit :

$$d_{k+1} = -g_{k+1} + \beta_k\, d_k \qquad \beta_k \in \mathbb{R} \quad et \quad g_{k+1}, d_k \in \mathbb{R}^n \tag{60}$$

Cherchons l'expression de β_k en combinant les relations (59) et (60), soit :

$$(-g_{k+1} + \beta_k\, d_k)^T\, A\, dk = 0 \Rightarrow -g_{k+1}^T\, A\, d_k + \beta_k\, d_k^T\, A\, d_k = 0$$

$$\Rightarrow \beta_k = \frac{g_{k+1}^T\, A\, d_k}{d_k^T\, A\, d_k} \tag{61}$$

L'expression du pas optimal a déjà été déterminée précédemment (48) :

$$\lambda_k = \frac{\nabla f(X^{(k)})^T\, \nabla f(X^{(k)})}{\nabla f(X^{(k)})^T\, A\, \nabla f(X^{(k)})} \tag{62}$$

Algorithm 5 Algorithme de descente de gradient conjugué

Input : $f(X) \in \mathcal{C}^1$, $X^{(0)} \in \mathbb{R}^n$, $\nabla f(X^{(0)}) \in \mathbb{R}^n$, A tel que $\forall M \in \mathbb{R}^n, M \neq 0 : M^T\, A\, M > 0$
$k \longleftarrow 0$

1. **Tant que** critère d'arrêt n'est pas vérifié faire :
   2. Pas optimal $\lambda_k = \dfrac{d_k^T\, d_k}{d_k^T\, A\, d_k}$
   3. Nouvelle itération $X^{(k+1)} \longleftarrow X^{(k)} + \lambda_k\, d_k$
   4. Calcul du scalaire $\beta_k = \dfrac{g_{k+1}^T\, A\, d_k}{d_k^T\, A\, d_k}$
   5. Direction conjuguée $d_{k+1} = -g_{k+1} + \beta_k\, d_k$
   6. Mise à jour : $k \longleftarrow k+1$
7. **Fin**

Soulignons qu'il existe plusieurs méthodes pour calculer le paramètre β_k. Dans l'équation (61), nous avons utilisé la méthode de *Fletcher-Reeves*. Nous citerons également les méthodes de *Polack-Ribière* et de *Hestenes-Stiefel*.

Exercice ❼ ☞ ®

Illustrons la méthode de gradient conjugué pour la fonction-objectif de deux variables suivante :

$$f(X) = 5\, x^2 + \frac{y^2}{2} - 3\,(x+y) \quad \text{avec} \quad X = (x,y)^T \tag{63}$$

Prendre comme vecteur initial $X^{(0)} = [-2, 1]^T$ et une tolérance $\epsilon = 10^{-4}$. Le critère d'arrêt est :

$$\| \frac{f(X^{(k+1)}) - f(X^{(k)})}{f(X^{(k)})} \| < \epsilon$$

Script Matlab® :

```
clc ; clear all ; close all ;

% Le 10/08/2019 - Samir KENOUCHE
% METHODE DU GRADIENT CONJUGUE

 fxy = @(x,y) 5*x.^2 + y.^2*(1/2) - 3*(x+y) ;  itmax = 15 ; it = 0;
 dfx = @(x) 10*x - 3 ; dfy = @(y) y - 3 ; tol = 1e-04 ;

 A = [10 0 ; 0 1] ;  b = [3 ; 3] ; xinit = [-2 ; 1] ;
   dkinit =  - [dfx(xinit(1)) ; dfy(xinit(2))] ; xtest = zeros(2,itmax);

while it < itmax

   lambdak = (dkinit'*dkinit)/(dkinit'*A*dkinit) ;

   xmin = xinit + lambdak*dkinit ;

   if abs((fxy(xmin(1), xmin(2)) - fxy(xinit(1),...
           xinit(2)))/fxy(xinit(1), xinit(2))) < tol ;

         solution = xmin   ; % VECTEUR SOLUTION
         nbr_it = it ;       % NOMBRE D'ITERATION PERMETTANT LA CV
       break
   end

   betak = ([dfx(xmin(1)) ; dfy(xmin(2))]'*A*dkinit)/(dkinit'*A*dkinit)  ;
   dk =  - [dfx(xmin(1)) ; dfy(xmin(2))] + betak*dkinit
   xinit = xmin ; dkinit = dk ; it = it + 1 ; % MISE A JOUR
end

f_min = fxy(solution(1),solution(2)) ; % MIN DE LA FONCTION-OBJECTIF

%%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%%%%
x = -20:0.3: 18 ; y = -20:0.3:18 ; [xgrid, ygrid] = meshgrid(x,y) ;
zgrid = fxy(xgrid,ygrid) ; figure('color',[1 1 1]) ;
contourf(xgrid,ygrid,zgrid) ;  xlabel('x') ; ylabel('y') ; hold on ;
plot(solution(1),solution(2),'rx') ; plot(solution(1),solution(2),'ro') ;

% VERIFICATION : A * solution - b = 0 Avec XMIN = (0.30 ; 3.0)
% ATTENTION AUX ERREURS D'ARRONDI
```

La méthode de gradient conjugué appliquée à une fonction-objectif d'ordre n converge au plus en n itérations. Notons toutefois que les erreurs d'arrondi dans le calcul des directions conjuguées font que nous n'obtenons pas toujours la solution exacte en n itérations. Cette méthode réclame un nombre d'itérations proportionnel

à $\sqrt{\text{cond}(A)}$. Contrairement aux autres méthodes de gradient, exigeant un nombre d'itérations proportionnel à $\text{cond}(A)$.

IV. **Algorithmes génétiques**

Cette classe[4] d'algorithmes est dite évolutionniste, car fondée sur les mécanismes Darwiniens de la sélection naturelle. Le fonctionnement de ces algorithmes est comme suit : partant d'une population initiale d'individus, choisis aléatoirement, la performance relative de chaque individu est mesurée au moyen d'une fonction *fitness*. Les valeurs de cette fonction sont classées selon un ordre décroissant. Nous créons ainsi une nouvelle population qui a hérité du bagage génétique des meilleurs individus de la population initiale. Cette nouvelle population est crée en se servant des opérations : *sélection*, *croisement* et *mutation*. Ce processus est réitéré jusqu'à la sélection du meilleur individu, constituant la solution du problème d'optimisation.

Avant de décrire les différents mécanismes du fonctionnement de ces algorithmes, nous donnerons, chemin faisant, quelques définitions élémentaires. Un individu I (ou chromosome) est codé sous forme d'une chaine binaire constituée de m gènes, $I = \{g_1, g_2, g_3, \cdots, g_m\}$ tel que $\forall i \in [1, m], g_i \in V = \{0, 1\}$. Important ! il existe également le codage réel tel que $g_i \in \mathbb{R}$. Toutefois, dans cette section nous nous focaliserons exclusivement sur le codage binaire. Pour une population constituée de n individus $\mathcal{P} = \{I_1, I_2, I_3, \cdots, I_n\}$ et chaque individu est codé en m gènes, nous obtenons :

$$\mathcal{P} = \begin{cases} I_1 = (g_1^1, g_2^1, g_3^1, \cdots, g_m^1\} \\ I_2 = (g_1^2, g_2^2, g_3^2, \cdots, g_m^2\} \\ I_3 = (g_1^3, g_2^3, g_3^3, \cdots, g_m^3\} \\ \vdots \\ I_n = (g_1^n, g_2^n, g_3^n, \cdots, g_m^n\} \end{cases} \tag{64}$$

Chaque individu I_i (appelé aussi chromosome ou encore génome) de la population $\mathcal{P}$ est formé de m gènes, choisis de manière totalement aléatoire. Dans le cas du codage binaire, la fonction *fitness* (ou efficacité) $f \in \mathbb{R}_+^n$. Cette fonction sert à classer les individus en fonction de leur performance. Elle peut être vue également comme une mesure d'adaptation des individus à leur environnement. En pratique, on utilise une fonction de décodage δ permettant le passage du système binaire au système décimal, soit :

$$\delta : \{0, 1\}^m \longmapsto \mathbb{R}$$

$$f : \delta\,\{0, 1\}^m \longmapsto \mathbb{R}_+^*$$

Dans sa formulation la plus simple, la fonction δ est définie selon :

$$\delta(I_i) = \sum_{j=1}^{m} g_j\, 2^{m-j} \quad \text{tel que} \quad \delta(I_i) \in \mathbb{R}_+^* \tag{65}$$

Plus de détails sur la conversion des systèmes de numération sont donnés en annexe (B). Dans ce qui suit, nous décrirons succinctement les trois mécanismes de base de la sélection naturelle. Ensuite, nous prendrons un exemple d'application numérique afin de faciliter la compréhension des différentes notions inhérentes aux algorithmes génétiques.

4. Les algorithmes génétiques font partie des méthodes dites de recherche globale : cela signifie la possibilité d'atteindre un optimum, ou du moins un meilleur point, même si ce dernier n'est pas dans le voisinage immédiat de l'itéré précédent. Toutes les autres méthodes d'optimisation vues précédemment sont dites de recherche locale : Le nouvel itéré est recherché dans le voisinage immédiat (d'où la notion de localité) de l'itéré précédent.

a) ***Sélection****:* Le processus de sélection naturelle permet aux gènes les mieux adaptés de se reproduire plus souvent et de contribuer davantage aux générations futures. Dans une population $\mathcal{P} = \{I_1, I_2, I_3, \cdots, I_n\}$, chaque individu est évalué par la fonction *fitness* $R_w^i = f(\delta(I_i))$. En pratique, nous associons à chaque individu, une probabilité P_s^i, d'être sélectionné.

$$P_s^i = \frac{R_w^i}{\sum\limits_{i\in\mathcal{P}} R_w^i} \tag{66}$$

Il existe une panoplie de processus de sélection. Nous avons pris la méthode la plus intuitive. Autrement dit, la probabilité de reproduction d'un individu dépend de sa valeur au regard de l'ensemble des valeurs de la population.

b) ***Croisement****:* Nous vérifions d'abord s'il y a croisement (ou reproduction) selon une probabilité de croisement typiquement $P_c \in V(0.85, 1)$. Il existe plusieurs types de croisement, nous considérons ici le croisement dit arithmétique. Soient une population de deux individus $\{I_1, I_2\}$ et un nombre aléatoire $\alpha \in V(0, 1)$. S'il y a croisement les rejetons $r_i = \{r_1, r_2, r_3\}$ (enfants) de la nouvelle génération s'obtiennent selon :

$$r_i = \begin{cases} r_1 = \alpha\, I_1 + \alpha\, I_2 \\ r_2 = (1-\alpha)\, I_1 + \alpha\, I_2 \\ r_3 = \alpha\, I_1 + (1-\alpha)\, I_2 \end{cases} \tag{67}$$

L'opération de croisement atteint chaque gène de chaque individu, selon le processus ci-dessous.

$$r_1 = \begin{pmatrix} \alpha\, g_1^1 + \alpha\, g_1^2 \\ \alpha\, g_2^1 + \alpha\, g_2^2 \\ \alpha\, g_3^1 + \alpha\, g_3^2 \\ \vdots \\ \alpha\, g_m^1 + \alpha\, g_m^2 \end{pmatrix} \tag{68}$$

$$r_2 = \begin{pmatrix} (1-\alpha)\, g_1^1 + \alpha\, g_1^2 \\ (1-\alpha)\, g_2^1 + \alpha\, g_2^2 \\ (1-\alpha)\, g_3^1 + \alpha\, g_3^2 \\ \vdots \\ (1-\alpha)\, g_m^1 + \alpha\, g_m^2 \end{pmatrix} \tag{69}$$

$$r_3 = \begin{pmatrix} \alpha\, g_1^1 + (1-\alpha)\, g_1^2 \\ \alpha\, g_2^1 + (1-\alpha)\, g_2^2 \\ \alpha\, g_3^1 + (1-\alpha)\, g_3^2 \\ \vdots \\ \alpha\, g_m^1 + (1-\alpha)\, g_m^2 \end{pmatrix} \tag{70}$$

Les individus de la nouvelle génération $\{r_1, r_2, r_3\}$ ont hérité en théorie les meilleurs gènes des individus de la génération précédente.

*c) **Mutation**:* Nous vérifions d'abord s'il y a mutation selon une probabilité de mutation typiquement $P_m \in V(10^{-4}, 10^{-1})$. Pour un gène ayant une probabilité de mutation P_m, nous obtenons de façon totalement aléatoire :

$$g_i^{'} = \begin{cases} g_i + \psi[\max(g_i) - g_i] & \text{1ère possibilité} \\ \\ g_i - \psi[\max(g_i) - g_i] & \text{2ème possibilité} \end{cases} \tag{71}$$

La fonction ψ est définie selon l'expression :

$$\psi(x) = x\,\beta\left(\frac{gT - gt}{gT}\right)^b \tag{72}$$

Avec β est un nombre aléatoire $\in V(0,1)$, gt est la génération courante, gT est la génération maximale et b est le degré d'extinction.

Exercice ❽ ☞ ®

En guise d'application numérique, soit à maximiser la fonction-objectif à une seule variable :

$$x^* \in \underset{x\in\mathbb{R}^n}{\text{argmax}}\ 17\,(x - x^2) \tag{73}$$

Si l'on souhaite la minimiser on prendra tout simplement :

$$x^* \in \underset{x\in\mathbb{R}^n}{\text{argmin}}\ -\ 17\,(x - x^2) \tag{74}$$

Prendre une probabilité de croisement $P_c = 0.75$ et une probabilité de mutation $P_m = 0.001$. La population initiale est formée de six individus. Ces derniers sont constitués de quatre gènes. Cet exemple simple vise à illustrer concrètement le fonctionnement des opérations de sélection, de croisement, de mutation et l'utilisation des différents paramètres des algorithmes génétiques permettant d'atteindre l'optimum. Cet exercice sera intégralement résolu lors de la séance de cours.

Exercice ❾ ☞ ®

Pour des problèmes d'optimisation plus complexes, nous utiliserons la fonction Matlab® prédéfinie `ga`. En guise d'application de cette fonction, nous souhaitons optimiser la fonction-objectif précédente (63) dont nous rappelons la formule :

$$f(X) = 5\,x^2 + \frac{y^2}{2} - 3\,(x + y) \quad \text{avec} \quad X = (x, y)^T$$

La syntaxe de la fonction Matlab® prédéfinie `ga` est donnée dans le script Matlab® ci-dessous :

```
clc ; clear all ;

% Samir KENOUCHE - Unconstrained optomization using algorithm genetic
% Le 14/09/2019

addpath('C:\Users\kenouche') ;

myfitness = @(x) 5*x(1).^2 + x(2).^2*(1/2) - 3*(x(1)+x(2)) ;

cineq = [] ; ceq = [] ;   % unconstrained problem
```

```
rng(1037,'twister') ;      % Control random number generation
xinit = randn([1 2]) ;     % Initialisation
nbrevars = 2 ;             % Number of variables
LB = [-inf -inf] ;         % Lower bound
UB = [+inf +inf] ;         % Upper bound

options = gaoptimset('CreationFcn', @gacreationlinearfeasible, ...
    'PlotFcns', @gaplotbestf, 'CrossoverFraction', 0.75, ...
    'InitialPopulation',xinit,'MutationFcn',@mutationadaptfeasible, ...
    'EliteCount', 2,'PopulationSize',50, 'TolFun', 1e-03,'Display','iter') ;

[xoptimal,fval,exitflag,output,population,scores] = ga(myfitness, ...
    nbrevars,[],[],[],[],LB,UB, cineq, ceq, options) ;

disp(output.message) ;
str = ['LES POINTS STATIONNAIRES OBTENUS : ' num2str(xoptimal)]
% LES POINTS STATIONNAIRES OBTENUS : 0.3  3
```

En conclusion, nous soulignons que les trois processus d'un algorithme génétique sont régit par un certain nombre de paramètres fixés préalablement. Ces derniers conditionnent la réussite de l'optimisation du problème étudié. Parmi ces paramètres, nous citerons (1) La taille de la population $\mathcal{P}$, et la taille de la chaine de codage des individus. Si cette population est trop grande, le temps de calcul de l'algorithme peut se révéler très contraignant. En revanche, si la taille de la population est petite, l'algorithme convergera vraisemblablement vers un mauvais individu. (2) Le choix de la valeur de la probabilité de croisement P_c, celle-ci dépend de la forme de la fonction *fitness*. En pratique, le choix de cette probabilité se fait de façon heuristique. Plus P_c est élevée, plus la population subit des transformations profondes. (3) Le choix de la valeur de la probabilité de mutation P_m. Elle est généralement faible puisqu'une valeur élevée risque de conduire à une solution divergente.

Une dernière notion inhérente aux algorithmes génétiques est celle de *l'élitisme*. Cette opération consiste à conserver le meilleur individu dans la génération ultérieure. Après le croisement, on compare le meilleur individu de la génération courante avec le meilleur individu de la génération antérieure. A l'issue de cette comparaison, le meilleur individu est conservé au détriment de l'autre qui est éliminé.

Exercice ⓾ ☞ ®

1) Minimiser en utilisant l'algorithme génétique la fonction-objectif définie par :

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

$$\hat{X} \in \underset{X\in\mathbb{R}^n}{\operatorname{argmin}} \quad f(X) \text{tel que } \begin{cases} x_1x_2 + x_1 - x_2 + 1.5 \leq 0 \\ 10 - x_1\,x_2 \leq 0 \\ 0 \leq x_1 \leq 1,\ 0 \leq x_2 \leq 13 \end{cases}$$

D'abord nous commençons par écrire les contraintes dans un fichier `M-file`, qu'on appellera ensuite depuis le script principal.

```
function [cineq, ceq] = myconstraint(x)
  cineq = [1.5 + x(1)*x(2) + x(1) - x(2) ;
  -x(1)*x(2) + 10] ;
  ceq = [] ;
return
```

Ce fichier est sauvegardé obligatoirement sous le nom `myconstraint.m` et doit être dans le même répertoire que le script principal ci-dessous :

```
clc ; clear all ;

% Samir KENOUCHE - Constrained optomization using algorithm genetic
% Le 15/09/2019
addpath('C:\Users\kenouche') ;

myfitness = @(x) 100*(x(1)^2 - x(2))^2 + (1 - x(1))^2 ;

rng(1403,'twister') ;   % Control random number generation
xinit = randn([1 2]) ;  % initialisation
nbrevars = 2 ;          % Number of variables
LB = [0 0] ;            % Lower bound
UB = [1 13] ;           % Upper bound

options = gaoptimset('CreationFcn', @gacreationlinearfeasible, ...
    'PlotFcns', @gaplotbestf, 'CrossoverFraction', 0.75, ...
    'InitialPopulation',xinit,'MutationFcn',@mutationadaptfeasible, ...
    'EliteCount', 1,'PopulationSize',50, 'TolFun', 1e-03, ...
    'Display','iter') ;

[xopti,fval,exitflag,output,population,scores] = ga(myfitness, nbrevars, ...
    [],[],[],[],LB,UB, @myconstraint, options) ;

disp(output.message)
str = ['Les points statonnaires obtenus : ' num2str(xopti)]
%% Les points stationnaires obtenus : 0.812202  12.3122
```

V. Travaux pratiques avec des fonctions Matlab prédéfinies

A. *Optimisation sans contraintes*

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

$$\hat{X} = \underset{X \in \mathcal{D}}{argmin}\ f(X) \tag{75}$$

Avec, $\mathcal{D}$ est un sous-ensemble de $\mathbb{R}^n$. Les variables $X = (x_1, x_2, ..., x_n)^T$ sont appelées *variables d'optimisation* ou *variables de décision*. La fonction f, à valeurs réelles, définie par $f : \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}$ est la *fonction-objectif* ou *fonction de coût*. Dans cette section, on abordera les fonctions Matlab® prédéfinies, dédiées à la résolution de problèmes d'optimisation sans contraintes. Toutes ces fonctions sont disponibles dans la boite à outil *Optimization Toolbox* du logiciel. Les fonctions Matlab® prédéfinies destinées à la minimisation de fonctions d'une seule variable sont `fminbnd` et `fminunc`. Notons que cette dernière peut également être utilisée pour les fonctions de plusieurs variables. Ces commandes présentent une syntaxe très similaires.

```
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 fonction `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 fonction `optimset`, dont ses différents paramètres seront décrits dans l'exercice ci-dessous. L'argument `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'itérations 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'itérations 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 à minimiser.

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

Exercice ❶ ☞ ®

1) Minimiser la fonction-objectif définie par :

$$\begin{cases} f(x) = (x-1) \times \exp(-x^2 + 2\,x + 1) \\ \\ x \in [-4, 4] \end{cases} \tag{76}$$

en utilisant les fonctions prédéfinies `fminbnd` et `fminunc`

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 OBTENU

>> funEval1 =
            -3.1690
             % VALEUR DE LA FONCTION A LA DERNIERE ITERATION

>> exitTest1 =
             1
             % TEST DE CONVERGENCE POSITIF

>> grad1 =

         -5.9605e-08
             % VALEUR DU GRADIENT DE LA FONCTION-OBJECTIF

>> hessian1 =

             12.6783
             % VALEUR DU HESSIEN DE LA FONCTION-OBJECTIF

>> xMin2 =
             0.2929
```

```
              % MINIMUM OBTENU AVEC fminbnd

>> 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 fonction 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*. `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).

Nous allons désormais tester la fonction `fminsearch` qui s'utilise pour la minimisation de fonctions multi-dimensionnelles. Sa syntaxe usuelle est analogue à celle de `fminunc`, à la différence près que `fminsearch` ne renvoie ni le *Jacobien* ni le *Hessien* de la fonction à optimiser. Ceci provient du fait que cette fonction est

basée sur l'algorithme *Simplex*. Nous allons procéder à son implémentation dans l'exercice ci-dessous.

Exercice ❷ ☞ ®

1) Minimiser la fonction-objectif de deux variables définie par :

$$f(x_1, x_2) = x_1^2 + (x_2 - 2)^2 \tag{77}$$

analytiquement puis en utilisant la fonction prédéfinie `fminsearch`

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

$$\begin{cases} \dfrac{\partial f}{\partial x_1} = 2\,x_1 = 0 \\ \dfrac{\partial f}{\partial x_2} = 2\,(x_2 - 2) = 0 \end{cases} \tag{78}$$

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.

$$\begin{vmatrix} \dfrac{\partial^2 f(\hat{X})}{\partial x_1^2} & \dfrac{\partial^2 f(\hat{X})}{\partial x_1\,\partial x_2} \\ \dfrac{\partial^2 f(\hat{X})}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f(\hat{X})}{\partial x_2^2} \end{vmatrix} \Longrightarrow \begin{vmatrix} 2 & 0 \\ 0 & 2 \end{vmatrix} = 4 > 0 \tag{79}$$

À partir de ces résultats, il en découle que le point $\hat{X} = (\hat{x}_1 = 0\,;\, \hat{x}_2 = 2)$ est le minimum recherché. Nous résoudrons le même système de façon algorithmique en se servant de la fonction `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.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%% SORTIE 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 fonction `fminunc` s'utilise aussi pour l'optimisation de fonctions à plusieurs variables. Nous allons l'utiliser pour optimiser la fonction $f(x_1, x_2) = 2\,x_1^2 + x_1\,x_2 + 2\,x_2^2 - 6\,x_1 - 5\,x_2 + 3$

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) ;
```

Il est possible de 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® calcule 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)
```

En écrivant `opts = optimset('GradObj','off','Hessian','off')`, Matlab® évalue le gradient et le Hessien de la fonction-objectif par la méthode des *différences finies* (DF). Le champ d'optimisation `opts = optimset('DerivativeCheck','on')` compare le gradient fourni par l'utilisateur à celui évalué par *DF*. 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 ❸ ☞ Ⓢ

– Justifier de l'existence d'un extremum des fonctions-objectif suivantes :

$$\begin{cases} f_1(x_1,x_2) = (1-x_1)^2 + 100\,(x_2 - x_1^2)^2 \\ f_2(x_1,x_2) = 20 + x_1^2 + x_2^2 - 10\,(\cos(2\pi x_1) + \cos(2\pi x_2)) \\ f_3(x_1,x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\ f_4(x_1,x_2) = \dfrac{1}{2} - \sin(x_1^2 + x_2^2) \\ f_5(x_1,x_2) = x_1^3 + x_2^3 - 3\,x_1\,x_2 \\ f_6(x_1,x_2) = x_1^2 - x_1\,x_2 + x_2^2 + 3\,x_1 - 2\,x_2 + 1 \end{cases} \tag{80}$$

– Trouver les extremums des fonctions ci-dessus, analytiquement, ensuite en se servant de la fonction prédéfinie `fminsearch`

B. Optimisation avec contraintes

De nombreux problèmes en physique, en chimie, en ingénierie et en économie nécessitent de minimiser une fonction-objectif soumise à plusieurs contraintes. Dans ce qui suit, nous nous intéresserons à la résolution de problèmes d'optimisation sous contraintes dont la formulation mathématique générale est donnée par :

$$\hat{X} = \underset{X\in\mathbb{R}^n}{argmin}\ f(X) \tag{81}$$

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

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

$$P = \{x \in \mathbb{R}^n : h_i(x) = 0, g_j(x) \leq 0\} \tag{82}$$

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

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

$$\underset{X\in\mathbb{R}^n}{min}\ f^T\,x \text{ tel que} \begin{cases} A \times x \leq B \\ Aeq \times x = Beq \\ lB \leq x \leq uB \end{cases} \tag{83}$$

`linprog` s'utilise avec 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 `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 définition des variables en question. Pour les variables non bornées, on mettra `lB = -inf` et `uB = inf`.

Exercice ❹ ☞ ®

1) Minimiser la fonction-objectif définie par :

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

$$\hat{X} = \underset{X\in\mathbb{R}^3}{argmin}\, f(X) \quad \text{tel que} \begin{cases} x_1 - x_2 + x_3 \leq 20 \\ 3\,x_1 + 2\,x_2 + 4\,x_3 \leq 42 \\ 3\,x_1 + 2\,x_2 \leq 30 \\ 0 \leq x_1,\ 0 \leq x_2,\ 0 \leq x_3 \end{cases}$$

2) Maximiser la fonction-objectif définie par :

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

$$\hat{X} = \underset{X\in\mathbb{R}^2}{argmin}\, f(X) \quad \text{tel que} \begin{cases} x_1 + x_2 \leq 7.50 \\ 11\,x_1 + 3\,x_2 \leq 0.40 \\ 12\,x_1 + 21\,x_2 \leq 1.50 \\ x_1 \geq 0,\ x_2 \geq 0 \end{cases}$$

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.

```
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 constate 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)
```

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
```

Désormais on se servira de la fonction `fmincon`. La topologie générale d'un processus d'optimisation avec contraintes sur les variables, peut s'écrire suivant la notation compacte suivante :

$$\hat{X} \in \underset{X\in\mathbb{R}^n}{argmin} f(X) \quad \text{tel que} \quad \begin{cases} A \times x \leq B \\ Aeq \times x = Beq \\ C(x) \leq x \\ Ceq(x) = x \\ lB \leq x \leq uB \end{cases} \tag{84}$$

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 `fmincon`. Cette fonction 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 `fmincon` 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 fonctions vues 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 ❺ ☞ ®

– On se propose de minimiser la fonction-objectif définie par :

$$f(x_1, x_2) = 2\,x_1^2 + x_1\,x_2 + 2\,x_2^2 - 6\,x_1 - 6\,x_2 + 15$$

$$\hat{X} = \underset{X\in\mathbb{R}^2}{argmin}\, f(X) \quad \text{tel que} \quad \begin{cases} x_1 + 2\,x_2 \leq 5 \\ 4\,x_1 \leq 7 \\ x_2 \leq 2 \\ -2\,x_1 + 2\,x_2 = -1 \end{cases}$$

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 `optimset('LevenbergMarquardt','on', ...)` indique que l'opération d'optimisation sera menée par le biais de l'algorithme de *Levenberg-Marquardt*. Le choix de cet algorithme peut se faire également avec la syntaxe `optimset('NonlEqnAlgorithm','lm', ...)`. On peut aussi faire appel à l'algorithme de *Gauss-Newton*, en utilisant la syntaxe `optimset('NonlEqnAlgorithm','gn', ...)`. Les différentes sorties renvoyées par ce script sont énumérées ci-dessous.

```
%%%%%%%%%%%%%%%%%%%%%%% 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
```

Nous résolvons dans l'exercice ci-dessous, un problème d'optimisation avec contraintes non-linéaires.

Exercice ❻ ☞ ®

– On se propose de minimiser le fonction-objectif définie par :

$$f(x_1, x_2) = \exp(x_1)\,(4\,x_1^2 + 2\,x_2^2 + 4\,x_1\,x_2 + 2\,x_2 + 1)$$

$$\hat{X} = \underset{X \in \mathbb{R}^2}{argmin}\ f(X) \quad \text{tel que} \begin{cases} 2 + x_1\,x_2 - x_1 - x_2 \leq 0 \\ -x_1\,x_2 \leq 10 \end{cases}$$

On commence d'abord par écrire le fichier `M-file` correspondant aux contraintes non-linéaires. Ce fichier bien entendu est sauvegardé 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 ❼ ☞ Ⓢ

– On se propose de minimiser les fonctions-objectif définies par :

$$f(x_1, x_2) = \frac{1}{2}(x_1 - 3)^2 + \frac{1}{2}(x_2 - 1)^2$$

$$\hat{X} = \underset{X \in \mathbb{R}^2}{argmin}\, f(X) \quad \text{tel que} \begin{cases} x_1 + x_2 - 1 \leq 0 \\ x_1 - x_2 - 1 \leq 0 \\ -x_1 + x_2 - 1 \leq 0 \\ -x_1 - x_2 - 1 \leq 0 \end{cases}$$

La fonction `quadprog` (Quadratic programming) solutionne un processus d'optimisation, écrit sous une formulation quadratique qui minimise la quantité :

$$\underset{X \in \mathbb{R}^n}{min}\, f(X) = \frac{1}{2}\, x^T\, H\, x + f^T\, x \text{ tel que } \begin{cases} A \times x \leq B \\ Aeq \times x = Beq \\ lB \leq x \leq uB \end{cases} \tag{85}$$

La syntaxe usuelle de cette fonction est :

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

L'argument `H` est le Hessien 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 fonction `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 ❽ ☞ Ⓡ

– On se propose de minimiser les fonctions-objectif définies par :

$$f(x_1, x_2) = x_1^2 + 4\, x_1 + 5\, x_2$$

$$\hat{X} = \underset{X \in \mathbb{R}^2}{argmin}\, f(X) \quad \text{tel que} \begin{cases} 2\, x_1 + x_2 \geq 10 \\ 3\, x_1 + 6 x_2 \leq 80 \\ 5\, x_1 + 7\, x_2 \leq 50 \\ x_1, x_2 \geq 0 \end{cases}$$

$$f(x_1, x_2, x_3) = x_1^2 + x_1\,x_2 + 2\,x_2^2 + 2\,x_3^2 + 2\,x_2\,x_3 + 4\,x_1 + 6\,x_2 + 12\,x_3$$

$$\hat{X} = \underset{X\in\mathbb{R}^3}{argmin}\, f(X) \quad \text{tel que} \begin{cases} x_1 + x_2 + x_3 \geq 6 \\ -x_1 - x_2 + 2\,x_3 \geq 2 \\ 0 \leq x_1, x_2, x_3 \leq 100 \end{cases}$$

– Réécrire la fonction-objectif selon la notation générale décrite par l'Eq. (85).
– Trouver les coordonnées du point critique en utilisant `quadprog`.

Réécrivons d'abord ce système selon la notation générale décrite par l'Eq. (85).

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

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

D'après la définition sur les contraintes d'inégalités, elles doivent être écrites, inférieure ou égale à une constante. 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
```

Ci-dessous les sorties correspondantes :

```
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 revient 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 = []`. Ci-dessous 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 de 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 constater, le champ `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 fonction `quadprog`, on prendra `-H` et `-f`.

Exercice ❾ ☞ Ⓢ

– Minimiser les fonctions-objectif définies par :

$$f(x_1, x_2) = (x_1 - 2)^2 + (x_2 - 2)^2$$

$$\hat{X} = \underset{X \in \mathbb{R}^2}{argmin}\, f(X) \quad \text{tel que} \begin{cases} x_1 + 2\,x_2 \leq 3 \\ 3\,x_1 + 2x_2 \geq 3 \\ x_1 - 2\,x_2 \leq 2 \\ x_1, x_2 \geq 0 \end{cases}$$

$$f(x_1, x_2, x_3) = x_1^3 + x_2^3 + x_3^3$$

$$\hat{X} = \underset{X \in \mathbb{R}^3}{argmin}\, f(X) \quad \text{tel que} \begin{cases} x_1^3 + x_2^3 + x_3^3 = 1 \\ 2\,x_3^3 - x_2^2 \leq 0 \\ x_1 \geq 0 \\ x_3 \leq 0 \end{cases}$$

– Réécrire la fonction-objectif selon la notation générale décrite par l'Eq. (85).
– Trouver les coordonnées du point critique en utilisant `quadprog`. Pour la fonction à trois variables, prenez le vecteur des valeurs initiales $(1, 0, -1)$.

Pour conclure cette section, il recommandé également dans le même sillage de consulter les fonctions d'optimisation prédéfinies `fseminf`, `fminimax` et `fgoalattain`.

Annexe A
Dérivée directionnelle

Nous souhaitons quantifier le taux de variation de la fonction $f : \mathbb{R}^2 \longmapsto \mathbb{R}$ lorsqu'elle passe d'un 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 $\overline{P_0P}$. Soit $\overrightarrow{d} = a\,\vec{i} + b\,\vec{j}$ un vecteur unitaire ayant la même direction que $\overrightarrow{P_0P}$.

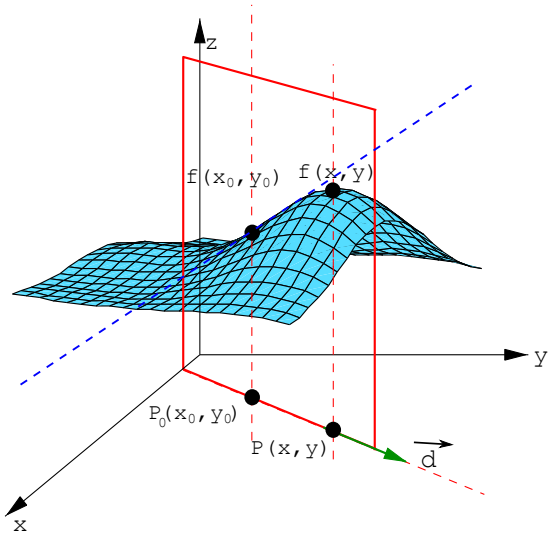

Figure 4: Dérivée directionnelle au point P_0 dans la direction $\vec{d}$.

Sur la figure ci-dessus :

$$\begin{aligned} \overrightarrow{P_0P} // \overrightarrow{d} \Rightarrow \overrightarrow{P_0P} &= \lambda\,\overrightarrow{d} \quad \text{avec} \quad \lambda \in \mathbb{R}_+^* \\ \Rightarrow \overrightarrow{P_0P} &= \lambda\,(a\,\vec{i} + b\,\vec{j}) = \lambda\,a\,\vec{i} + \lambda\,b\,\vec{j} \end{aligned} \tag{86}$$

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

$$\overrightarrow{P_0P} = \overrightarrow{OP} - \overrightarrow{OP_0} \tag{87}$$
$$= (x\,\vec{i} + y\,\vec{j}) - (x_0\,\vec{i} + y_0\,\vec{j}) \tag{88}$$
$$= x\,\vec{i} + y\,\vec{j} - x_0\,\vec{i} - y_0\,\vec{j} \tag{89}$$
$$= (x - x_0)\,\vec{i} + (y - y_0)\,\vec{j} \tag{90}$$

Par identification des équations (86) et (90), il vient :

$$\begin{cases} x - x_0 = \lambda\,a \\ y - y_0 = \lambda\,b \end{cases} \Rightarrow \begin{cases} x = x_0 + \lambda\,a \\ y = y_0 + \lambda\,b \end{cases} \tag{91}$$

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 :

$$f_{\overrightarrow{d}}(x_0, y_0) = \lim_{\lambda \to 0} \frac{f(x_0 + \lambda\,a, y_0 + \lambda\,b) - f(x_0, y_0)}{\lambda} \tag{92}$$

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* $||\nabla f(x_0, y_0)||$.

Ce théorème peut être prouvé en considérant que $f_{\overrightarrow{d}}(x_0, y_0) = \nabla f(x_0, y_0)\ \overrightarrow{d} = ||\nabla f(x_0, y_0)||\,||\overrightarrow{d}||\cos(\theta) = ||\nabla f(x_0, y_0)||\cos(\theta)$. 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 $f(x, y)$. Cette condition est prouvée selon :

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

Comme exemple d'application, calculons 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. La solution est donnée ci-dessous :

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

Annexe B
Codage binaire

Dans ce système de numération binaire, on utilise la base 2. En prenant par exemple le nombre 55, celui-ci se décompose en : $55 = 32 + 16 + 4 + 2 + 1$.

$55 = 1 \times 32 + 1 \times 16 + 1 \times 4 + 1 \times 2 + 1 \times 1$
$55 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
$55 = 1 \times 2^5 + 1 \times 2^4 + \mathbf{0} \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
$55 = 110111$

On utilise également la notation $55_{(10)} = 110111_{(2)}$. Il est possible d'arriver au même résultat, en procédant par des divisions successives du nombre 55 suivant la base 2. La condition d'arrêt correspond à un quotient nul, ensuite on remonte les restes des divisions, selon :

$$\begin{array}{r|l} 55 & 2 \\ \hline 4 & 27 \\ \hline 15 & \\ 14 & \\ \hline \mapsto\to\ 1 & \end{array} \qquad \begin{array}{r|l} 27 & 2 \\ \hline 2 & 13 \\ \hline 07 & \\ 6 & \\ \hline \mapsto\to\ 1 & \end{array} \qquad \begin{array}{r|l} 13 & 2 \\ \hline 12 & 6 \\ \hline \mapsto\to\ 1 & \end{array} \qquad \begin{array}{r|l} 6 & 2 \\ \hline 6 & 3 \\ \hline \mapsto\to\ 0 & \end{array}$$

$$\begin{array}{r|l} 3 & 2 \\ \hline 2 & 1 \\ \hline \mapsto\to\ 1 & \end{array} \qquad \begin{array}{r|l} 1 & 2 \\ \hline 0 & 0 \\ \hline \mapsto\to\ 1 & \end{array}$$

On obtient $55_{(10)} = 110111_{(2)}$ (de la droite vers la gauche). On peut proposer l'algorithme ci-dessous pour transformer un nombre de la base décimale, par exemple le nombre 125, vers la base binaire.

```
%%%%%%%%%%%%%%%%% CONVERSION DECIMAL VERS LE BINAIRE %%%%%%%%%%%%%%%%%
clc ; clear all ;
% Samir KENOUCHE - le 09/08/2019

decNumber = 125 ; base = 2 ; binNumber = [];
quot = 1 ;

while quot ~= 0

    quot = floor(decNumber/base) ; reste = decNumber - quot*base  ;
    decNumber = floor(decNumber/base);
    binNumber = [reste binNumber] ;

end

binNumber
```

Ce script renvoie le résultat suivant `binNumber = 1111101`.

$$
\begin{array}{r|l}
1\,2\,5 & 2 \\
\hline
1\,2 & 6\,2 \\
\hline
0\,5 & \\
4 & \\
\hline
\mapsto\!\!\rightarrow\ 1 &
\end{array}
\qquad
\begin{array}{r|l}
6\,2 & 2 \\
\hline
6 & 3\,1 \\
\hline
0\,2 & \\
2 & \\
\hline
\mapsto\!\!\rightarrow\ 0 &
\end{array}
\qquad
\begin{array}{r|l}
3\,1 & 2 \\
\hline
2 & 1\,5 \\
\hline
1\,1 & \\
1\,0 & \\
\hline
\mapsto\!\!\rightarrow\ 1 &
\end{array}
\qquad
\begin{array}{r|l}
1\,5 & 2 \\
\hline
1\,4 & 7 \\
\hline
\mapsto\!\!\rightarrow\ 1 &
\end{array}
$$

$$
\begin{array}{r|l}
7 & 2 \\
\hline
6 & 3 \\
\hline
\mapsto\!\!\rightarrow\ 1 &
\end{array}
\qquad
\begin{array}{r|l}
3 & 2 \\
\hline
2 & 1 \\
\hline
\mapsto\!\!\rightarrow\ 1 &
\end{array}
\qquad
\begin{array}{r|l}
1 & 2 \\
\hline
0 & 0 \\
\hline
\mapsto\!\!\rightarrow\ 1 &
\end{array}
$$

On retrouve ainsi le même résultat $1255_{(10)} = 1111101_{(2)}$. À l'inverse, la conversion du système binaire vers le système décimal se fera, suivant :

$110111_{(2)} = 1 \times 2^{(6-1)} + 1 \times 2^{(6-2)} + 0 \times 2^{(6-3)} + 1 \times 2^{(6-4)} + 1 \times 2^{(6-5)} + 1 \times 2^{(6-6)}$.
$110111_{(2)} = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$.
$110111_{(2)} = 32 + 16 + 0 + 4 + 2 + 1 = 55_{(10)}$

Cette conversion est implémentée dans le script Matlab® ci-dessous :

```
%%%%%%%%%%%%%%%%% CONVERSION BINAIRE VERS LE DECIMAL %%%%%%%%%%%%%%%%%
clc ; clear all ;
% Samir KENOUCHE - le 09/08/2019

binNumber = [1 1 0 1 1 1] ; base = 2 ; decNumber = 0 ;

for i = 1 : numel(binNumber)

   decNumber = decNumber + binNumber(i)*base^(numel(binNumber) - i) ;
end

decNumber
```

Pour la conversion de nombres fractionnaires on écrira :

$01101.010_{(2)} = 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}$
$01101,010_{(2)} = 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 + 0 \times 0.5 + 1 \times 0.25 + 0 \times 0.12$
$01101.010_{(2)} = 13.25_{(10)}$

Pour la conversion d'un nombre fractionnaire d'un système décimal vers le binaire, on procèdera comme suit :

$55.8625_{(10)} \stackrel{?}{=} xxxxx_{(2)}$

On commence d'abord par écrire la partie entière en binaire : $55_{(10)} = 110111_{(2)}$. Ensuite on multiplie par 2 la partie fractionnaire :

$0.8625 \times 2 = 1.725 = 1 + 0.725$
$0.7250 \times 2 = 1.450 = 1 + 0.450$

$0.4500 \times 2 = 0.900 = 0 + 0.900$
$0.9000 \times 2 = 1.800 = 1 + 0.800$
$0.8000 \times 2 = 1.600 = 1 + 0.600$ ...

À partir de cet exemple, on note que la partie entière est codée, par des divisions successives par 2, sur un nombre donné de bits. La partie fractionnaire est codée sur un nombre donné de bits en multipliant successivement par 2 jusqu'à ce que la partie fractionnaire soit nulle ou le nombre de bits considéré est atteint. Cette conversion donne :

$$55.8625_{(10)} = 110111.11011_{(2)} = 11011.111011_{(2)}\, 2^1$$