Théorème 3.2.4 — Condition nécessaire et suffisante, cas convexe . Soit f convexe et différentiable sur $\mathbb{R}^n$. Une C.N.S pour que x^* soit un minimum local (donc global) de f est que x^* soit un point critique de f, autrement dit, que:

$$\nabla f(x^*) = 0$$

3.3 Vision générale des méthodes numériques de descente

Dans la partie ci-dessus, la procédure de trouver les extréma et leurs natures reste faisable uniquement pour des fonctions avec peu de variables dont les extréma sont faciles à trouver analytiquement, si non cette procédure n'est plus valable et on fait appel à des méthodes numériques.
On considère désormais une fonction numérique $f : \mathbb{R}^n \to \mathbb{R}$ deux fois différentiable. Le principe général des méthodes de descente est de construire une séquence x_k de points de $\mathbb{R}^n$ qui va converger vers un minimum local x^*. Cette séquence sera construite en choisissant un point initial $x_0 \in \mathbb{R}^n$ puis, pour $k \geq 0$, à répéter tant que nécessaire les conditions suivantes:

1. choisir une direction de descente $d_k \in \mathbb{R}^n$.
2. choisir une taille de pas α_k appelé aussi le pas de descente.
3. poser $x_{k+1} = x_k + \alpha_k d_k$ et continuer avec $k = k+1$.

Le problème est de bien choisir x_0, les directions d_k et les tailles de pas α_k. À chaque étape, d_k et α_k sont choisis tels que $f(x_{k+1}) < f(x_k)$ (sauf si on a atteint le minimum, c'est-à-dire $x_k = x^*$).
On dit que d_k est une direction de descente en x_k quand la fonction:

$$F(\alpha_k) \equiv f(x_k + \alpha_k d_k),\ \alpha_k \geq 0$$

est décroissante en $\alpha_k = 0$, c-à-d, $F'(0) < 0$. Considérons des valeurs petites de $\alpha_k d_k$, le développement de Taylor autour de x_k avec deux termes indique que:

$$f(x_k + \alpha_k d_k) \approx f(x_k) + \alpha_k d_k^T \nabla f(x_k) < f(x_k)$$

Considérons que les valeurs $\alpha_k > 0$, il sera nécessaire de vérifier la condition suivante:

$$d_k^T \nabla f(x_k) < 0$$

Cette condition indique que d_k sera dans le demi-espace opposé au gradient $\nabla f(x_k)$. On introduit l'angle θ_k comme l'angle dont le cosinus est

$$\cos\theta_k = \frac{-d_k^T \nabla f(x_k)}{\| d_k \| \| \nabla f(x_k) \|}$$

Pour que d_k soit une direction de descente, il faut que $-\pi/2 \leq \theta \leq \pi/2$. En pratique, pour éviter des directions de recherche quasi-orthogonales au gradient et garantir une descente significative, on exige la condition:

$$-(\pi/2 - \mu) < \theta_k < (\pi/2 - \mu)$$

avec $\mu > 0$. Graphiquement, l'angle θ_k doit être inclus dans l'intervalle indiqué par la Fig. 3.5

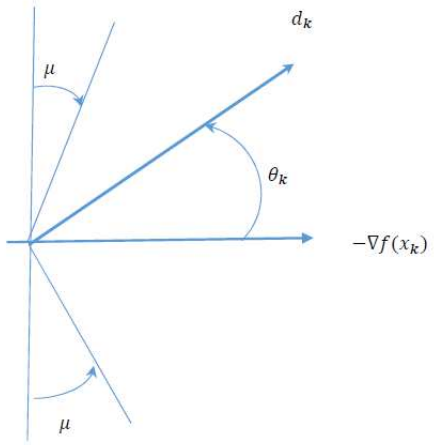

Figure 3.4: Direction de descente

3.4 Méthodes pour déterminer la taille du pas α_k ou recherche linéaire

L'objectif de cette partie est de déterminer une valeur adéquate de α_k dans l'expression $x_{k+1} = x_k + \alpha_k d_k$ qui contrôle la "vitesse" avec laquelle on progresse dans la direction. Le choix de ce paramètre est crucial pour obtenir la convergence vers un minimum.

3.4.1 Méthodes de descente à pas fixe

C'est le choix le plus simple et le moins coûteux, en revanche, il faut faire attention à la valeur e pour assurer une convergence vers un minimum.

▪ **Exemple 3.3** Soit l'exemple de **1D** $f(x) = x^2$. Les figures suivantes illustrent l'effet de la valeur du pas α_k sur la convergence vers le minimum. ▪

3.4.2 Méthodes de descente avec un pas une suite convergente

Ici, le pas α_k est choisi de tel sorte que:

$$\alpha_k \longrightarrow \infty \textbf{ quand } k \longrightarrow \infty$$

Cette méthode souffre du problème de la convergence qui peut être très lente.

3.4.3 Méthodes de descente à pas optimal (pas exact)

Il s'agit d'un problème de recherche linéaire où il faut choisir le pas à chaque itération: α_k évolue avec les itérations. Dans le cas optimal, le pas est choisi de façon à minimiser la fonction d'une variable α:

$$f(x_k + \alpha_k d_k) = min_{\alpha>0} f(x_k + \alpha d_k)$$

La dérivée de la fonction $f(x_k + \alpha d_k)$ par rapport à α donne:

$$\begin{aligned} f(x_k + \alpha d_k)'_\alpha &= \frac{\partial f(x_k+\alpha d_k)}{\partial x_k(1)} d_k(1) + \frac{\partial f(x_k+\alpha d_k)}{\partial x_k(2)} d_k(2) + \cdots + \frac{\partial f(x_k+\alpha d_k)}{\partial x_k(n)} d_k(n) \\ &= \nabla f(x_k + \alpha d_k)^T d_k \end{aligned}$$

Alors en $\alpha = 0$, on a:

$$f(x_k + \alpha d_k)'_\alpha = \nabla f(x_k)^T d_k < 0 \ \textit{(résultat d'une méthode de descente)}$$

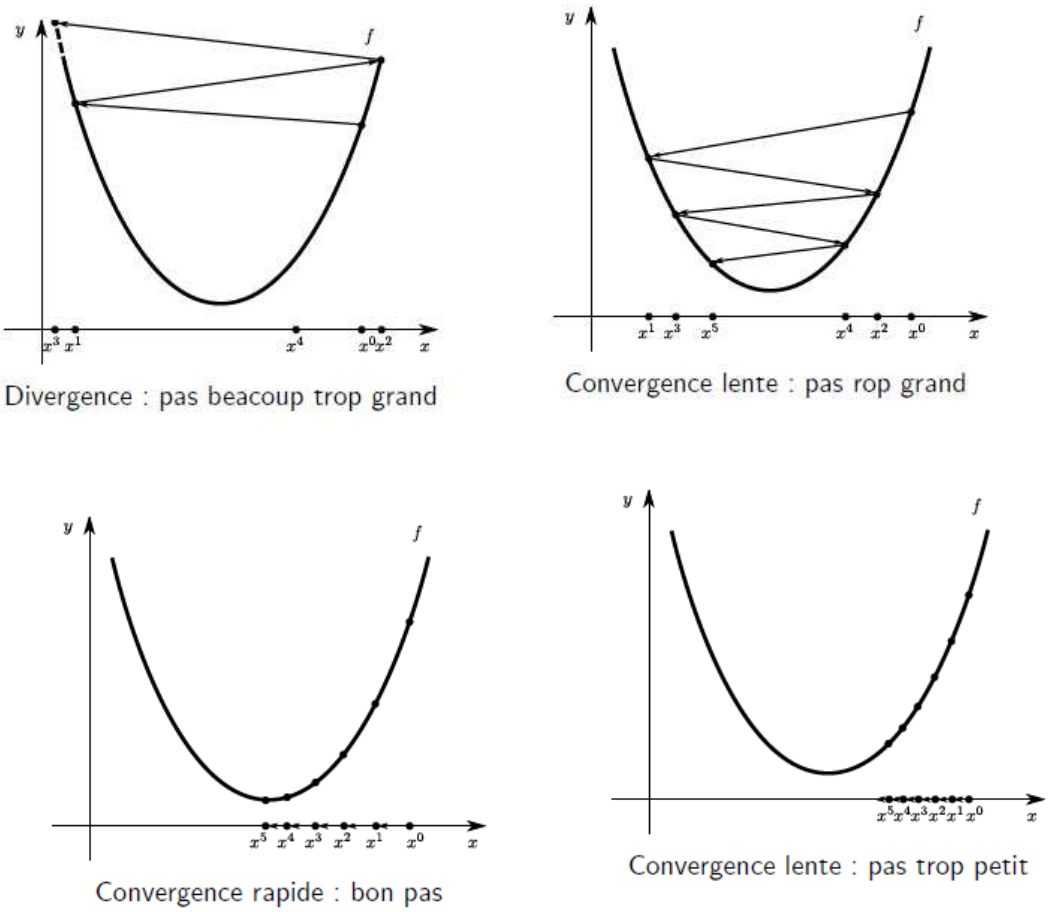

Figure 3.5: Effet du pas α

La figure Fig. 3.6 montre un possible aspect de $F(\alpha)$ où le minimum est situé au point α^*. La détermination de ce point se calcule en posant

$$F(\alpha^*)' = 0 \Leftrightarrow \nabla f(x_k + \alpha^* d_k)^T d_k = 0$$

.

3.4.4 Méthode de descente à pas approximatif

Dans de nombreux cas pratiques, il n'est pas possible de trouver analytiquement la valeur optimale du pas α_k. Une possibilité est alors de fixer sa valeur en satisfaisant certains critères faciles à vérifier et qui consiste à éviter des valeurs très grandes et très petites.

Critère de Armijo et Goldstein

Ce critère permet d'éviter une valeur très grande de α en obligeant $F(\alpha)$ à être au-dessous d'une droite $\lambda(\alpha)$) dont la pente est une petite fraction de $F'(0)$ (voir Fig. 3.7).

$$F(\alpha) \geq \lambda(\alpha) = F(0) + \rho\alpha F'(0) = f(x_k) + \rho\alpha\nabla f(x_k)^T d_k,\ 0 < \rho < \frac{1}{2} \text{ (critère de Armijo)}$$

Pour éviter une valeur très petite de α, *Goldstein* oblige que $F(\alpha)$ soit au-dessus d'une droite $\varphi(\alpha)$ où la pente est supérieure à la pente d'origine:

$$F(\alpha) \leq \varphi(\alpha) = F(0) + (1-\rho)\alpha F'(0) = f(x_k) + (1-\rho)\alpha\nabla f(x_k)^T d_k \text{ (critère de Goldstein)}$$

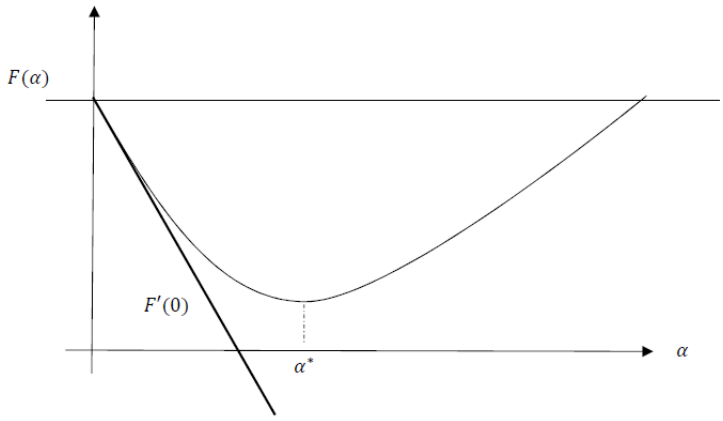

Figure 3.6: Allure de $F(\alpha)$

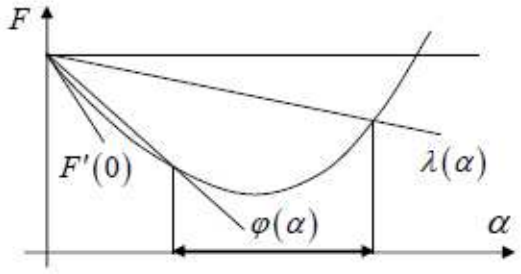

Figure 3.7: Critère de Armijo et Goldstein

Critère de Wolf

Encore une fois, ce critère est aussi composé par deux majorations. La première condition permettant des valeurs non très grandes est similaire au critère de *Armijo*. La deuxième condition oblige que la pente soit algébriquement grande que la pente d'origine.

$$F'(\alpha) \geq \beta F'(0),\ \rho < \beta < 1$$

Donc, *Wolf* se considère plus grand que *Goldstein* La méthode innexacte la plus étendue pour calculer l'amplitude du pas α est connue par *backtrakink*.

Algorithme de backtracking

L'idée de l'algorithme consiste à réduire progressivement la valeur du paramètre α jusqu'à en trouver une qui satisfait les critères de *Armijo* et *Goldstein*, ou juste un des deux, de préférence celui de *Armijo*. En pratique, on commence par un pas complet $\alpha = 1$, et on va le réduire à travers le facteur $\beta \cdot \alpha$ avec $\beta \in [0,1]$

R Cet algorithme est valable uniquement pour des directions de descente.

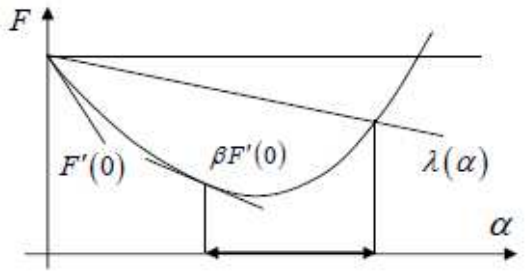

Figure 3.8: Critère de Wolf

3.5 Méthode pour déterminer les directions de descente

Les différentes méthodes de descente diffèrent par le choix du pas α et de la direction d_k; mais toutes doivent vérifier $f(x_{k+1}) \leq f(x_k)$.

3.5.1 Méthode de la plus forte pente ou méthode du gradient

Dans la méthode de la plus forte pente, la direction est choisie pour être opposée au gradient de la fonction (plus forte pente vient du fait que $\cos(\theta_k) = 1$), c'est-à-dire:

$$d_k = -\nabla f(x_k)$$

Cette direction peut être combinée avec la recherche linéaire, que se soit exacte ou approximative. En plus que cette méthode paraît logique à la première vue, en général cette méthode converge très lentement, en particulier proche de l'optimum. En revanche, c'est une méthode simple à implémenter.

■ **Exemple 3.4 — méthode du gradient à pas optimal.** Soit la fonction à minimiser:

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

Il est clair que la fonction est convexe, donc le minimum local est aussi un minimum global. Pour la méthode de la plus forte pente, la direction est la suivante:

$$d_k = -\nabla f(x_k) = \begin{bmatrix} x_{1_k} \\ x_{2_k} - 2 \end{bmatrix}$$

Comme c'est déjà démontré auparavant, le pas optimal se calcule en résolvant l'équation:

$$\begin{aligned} f(x_k + \alpha^* d_k)^T \cdot d_k = 0 \quad &\Leftrightarrow \quad \nabla f(x_{k+1})^T \cdot \nabla f(x_k) = 0 \\ &\Leftrightarrow \quad \begin{bmatrix} x_{1_k} - \alpha^* x_{1_k} & x_{2_k} - \alpha^*(x_{2_k} - 2) - 2 \end{bmatrix} \cdot \begin{bmatrix} x_{1_k} \\ x_{2_k} - 2 \end{bmatrix} = 0 \\ &\Leftrightarrow \quad \alpha^* = 1 \end{aligned}$$

■

■ **Exemple 3.5 — méthode du gradient à pas approximative (méthode de backtracking).** Soit la fonction à minimiser:

$$f(x_1, x_2) = e^{x_1 + 3x_2 - 0.1} + e^{x_1 - 3x_2 - 0.1} + e^{-x_1 - 0.1}$$

On déclare d'abord une fonction *function* qui donne la valeur de la fonction ainsi que le gradient.

```
function [f g]= objfun_min1(x)
%sdfdf
A = [1 3; 1 -3; -1 0];
b = -0.1*[1; 1; 1];
f = sum(exp(A*x+b));
if nargout<2
    return
end
g = A'*exp(A*x+b);
%fprintf('%4.0f %13.8e %13.8e %13.8e %13.8e\n',i,x,f,alpha);
```

Les différentes étapes du l'algorithme de la méthode de backtracking sont présentées par le programme qui suit: ∎

```
function [x f]=plus_forte_pente(fun,x0)
rho=0.1;
beta=0.5;
f1=0;
maxit=100;
x=x0;

for i=1:maxit
    [f g]=fun(x);
    if(abs(f-f1)<1e-10)
        break
    end
    d=-g;
    alpha=1;
    for k=1:10 %backtracking
        xnew=x+alpha*d;
        fxnew=fun(xnew);
        if fxnew < f+alpha*rho*g'*d
            break
        else
            alpha = alpha*beta;
        end
    end
    x=x+alpha*d;
    f1=f;
    fprintf('%4.0f %13.8e %13.8e %13.8e %13.8e\n',i,x,f,alpha);
end
return
```

L'exécution de ce programme dans l'espace de travail de *matlab* donne les résultats illustrés par la Fig. 3.9:

La figure Fig. 3.11 montre comment évolue les itérations $x_1(k)$ en fonction de $x_2(k)$ dans le tracé des contours.

3.5.2 Méthode du gradient conjugué

La méthode du gradient conjugué linéaire est une méthode qui résout deux problèmes équivalents possédant la même solution unique. Ces deux problèmes sont le système

```
[x f]=plus_forte_pente(@objfun_min1,[-1;1])

 1 -1.26517900e+00 -2.50497831e-01 9.16207023e+00 6.25000000e-02
 2 -6.29000734e-01 6.51924176e-02 3.86828053e+00 2.50000000e-01
 3 -4.50514899e-01 -7.72284882e-02 2.68052760e+00 2.50000000e-01
 4 -4.21089848e-01 2.38719166e-02 2.60419641e+00 1.25000000e-01
 5 -3.97610304e-01 -8.05335008e-03 2.56942254e+00 1.25000000e-01
 6 -3.65030711e-01 1.39821003e-02 2.56295544e+00 2.50000000e-01
 7 -3.59263955e-01 -5.78404615e-03 2.56080796e+00 1.25000000e-01
 8 -3.55227870e-01 2.43800662e-03 2.55966300e+00 1.25000000e-01
 9 -3.52463501e-01 -1.04150522e-03 2.55939647e+00 1.25000000e-01
10 -3.48696568e-01 1.93956544e-03 2.55931730e+00 2.50000000e-01
11 -3.48020112e-01 -8.46703041e-04 2.55929408e+00 1.25000000e-01
12 -3.47557873e-01 3.70439501e-04 2.55927350e+00 1.25000000e-01
13 -3.47243091e-01 -1.62316050e-04 2.55926873e+00 1.25000000e-01
14 -3.47028931e-01 7.11957301e-05 2.55926742e+00 1.25000000e-01
15 -3.46737604e-01 -1.33695923e-04 2.55926699e+00 2.50000000e-01
16 -3.46685147e-01 5.87394995e-05 2.55926683e+00 1.25000000e-01
17 -3.46649462e-01 -2.58117184e-05 2.55926673e+00 1.25000000e-01
18 -3.46625190e-01 1.13436901e-05 2.55926671e+00 1.25000000e-01
19 -3.46592176e-01 -2.13150939e-05 2.55926670e+00 2.50000000e-01
20 -3.46586231e-01 9.36927894e-06 2.55926670e+00 1.25000000e-01
21 -3.46582187e-01 -4.11844758e-06 2.55926670e+00 1.25000000e-01
22 -3.46579437e-01 1.81036718e-06 2.55926670e+00 1.25000000e-01
23 -3.46577566e-01 -7.95799575e-07 2.55926670e+00 1.25000000e-01
x =
  -0.3466
  -0.0000
f =
   2.5593
```

Figure 3.9: Résultat de Exemple. 3.5

d'équation linéaires:

$$Ax = b$$

et le problème de minimisation d'une fonction quadratique:

$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

Pour ces deux problèmes, la matrice A est considérée carrée, symétrique et définie positive. Une des propriétés intéressantes de la méthode du gradient conjuguée est la création d'une suite de directions linéairement indépendants non-nulles de $\mathbb{R}^n$ $\{d_0, d_1, \cdots, d_{n-1}\}$, possédant la propriété d'être *mutuellement conjuguées* par rapport à la forme quadratique $f(x)$. Rappelons que cette propriété s'exprime par:

$$d_i^T A d_j = 0 \ \textit{pour tout} \ i \neq j$$

Grâce à ces directions, il est possible de minimiser $f(x)$ en seulement n pas (c-à-d en n étapes), en minimisant $J(x)$ successivement selon les n directions. Il est facile de trouver que:

$$\nabla f(x) \equiv g = Ax - b = -r = -d, \textit{et le Hessien est défini par} \ \nabla^2 f(x) = A$$

Il est clair que le gradient g est opposé au vecteur *des résidus* r. La valeur optimale du pas α peut être calculée analytiquement grâce à l'orthogonalité entre la direction de recherche et le gradient évalué au nouveau point.

$$d_k^T \nabla f(x_k + \alpha d_k) = d_k^T g_{k+1} = d_k^T A(x_k + \alpha d_k) + d_k^T b = d_k^T g_k + \alpha d_k^T A d_k = 0$$

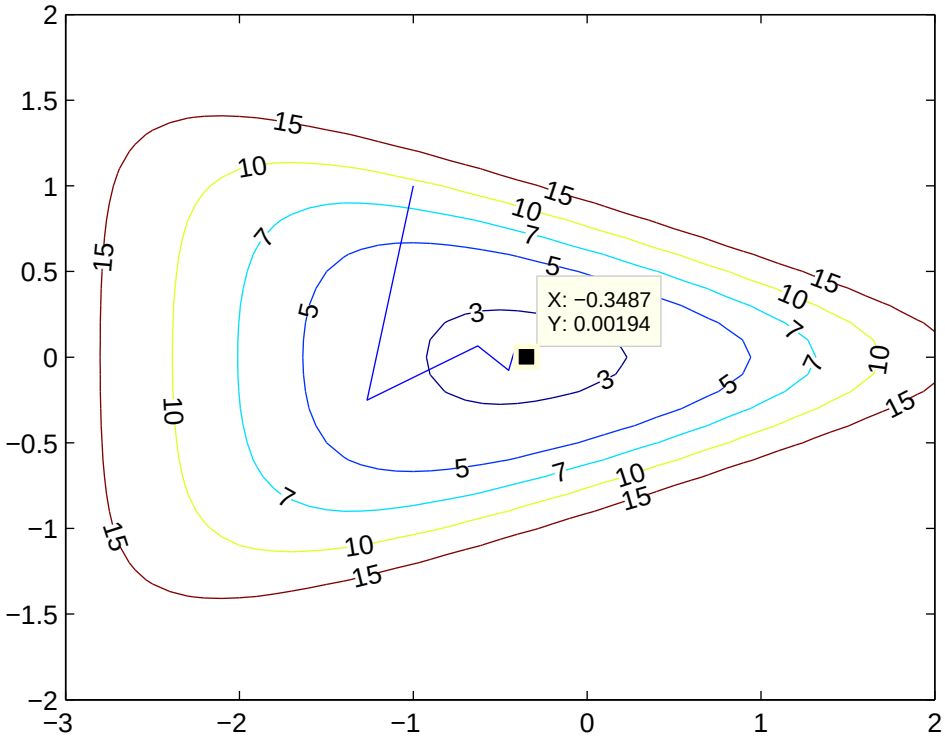

Figure 3.10: Résultats graphiques de l'Exemple. 3.5

$$\alpha = -\frac{d_k^T g_k}{d_k^T A d_k}$$

Dans la méthode du gradient de la plus forte pente (*steepest descent method*), c-à-d $d_k = -g_k$, on a:

$$\alpha = \frac{g_k^T g_k}{g_k^T A g_k}$$

Dans la méthode du gradient conjugué, la direction de recherche d_k est une combinaison du gradient g_k et de la direction de recherche antérieure d_{k-1}.

$$d_k = -g_k + \beta_{k-1} d_{k-1}$$

Le scalaire β_{k-1} se détermine de telle sorte que la nouvelle direction de recherche soit conjuguée à l'antérieure par rapport à la matrice A.

$$d_k A d_{k-1} = 0$$

A partir des deux expressions précédentes, on obtient la valeur de β_{k-1}:

$$\beta_{k-1} = \frac{g_k^T A d_{k-1}}{d_{k-1}^T A d_{k-1}}$$

Algorithme du gradient conjugué

■ **Exemple 3.6** Dans le code matlab suivant, une comparaison entre l'algorithme du gradient conjugué et le calcul exact d'un système de 4 dimensions:

Algorithm 1 Méthode du gradient conjugué

Entrées: x_0, A, b.

$g_0 \leftarrow Ax_0 - b$
$d_{-1} \leftarrow 0$
$d_0 \leftarrow -g_0$
for $k = 1$ to m **do**
  $g_k \leftarrow Ax_k - b$
  $\beta_{k-1} \leftarrow \frac{g_k^T A d_{k-1}}{d_{k-1}^T A d_{k-1}}$
  $d_k \leftarrow -g_k + \beta_{k-1} d_{k-1}$
  $\alpha \leftarrow -\frac{d_k^T g_k}{d_k^T A d_k}$
  $x_{k+1} = x_k + \alpha_k d_k$
end for

```
clc
n=4;
A=rand(n,n);
A=A'*A;
b=rand(n,1);
xv=1./diag(A);
x=zeros(n,1);gv=A*xv-b;
g=zeros(n,1);
pv=zeros(n,1);
p=zeros(n,1);
nit=0;
error=1e06;
while error>1e-06
    nit=nit+1;
    if nit==1
        p=-gv;
    else
            beta=gv'*A*pv/(pv'*A*pv);
            p=-gv+beta*pv;
    end
            alpha=-p'*gv/(p'*A*p);
            x=xv+alpha*p;
            g=A*x-b;
            error=norm(g);
            xv=x;
            gv=g;
            pv=p;
end
             disp(A\b); disp('  ');disp(x);
```

■

Exercice 3.3 Faire un programme qui permet de calculer le minimum de la fonction $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ par la méthode du gradient conjugué avec α calculé par le processus de backtracking. ■

3.5.3 Méthode de Newton

La méthode de Newton n'est pas une méthode d'optimisation à proprement dit, c'est une méthode faite pour résoudre l'équation $f(x) = 0$, où $f : \mathbb{R}^n \to \mathbb{R}$. En d'autre terme, la méthode de Newton s'obtient par le développement en série de Taylor d'ordre deux:

$$f(x) = f(x_k) + (x - x_k)^T \nabla f(x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k)$$

Imposons la condition de la valeur stationnaire et que la matrice Hessienne soit inversible :

$$\nabla f(x) = \nabla f(x_k) + \nabla^2 f(x_k)(x - x_k) = 0$$

$$\Rightarrow x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

où $d_k = -[\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$ est appelée *direction de Newton.*

Avantages de la méthode

- Pour des fonctions quadratiques convexes, le minimum s'obtient en un seul pas.
- A la proximité du minimum x^*, la convergence est garantie et elle est quadratique.

Limitations de la méthode

- Comme le Hessien est symétrique, il est nécessaire de calculer $n(n+1)/2$ dérivées secondaires, une chose qui n'est pas toujours facile ou possible.
- A chaque itération, il faut résoudre un système de n équations linéaires avec n variables, ce qui exige $O(n^3)$ opérations arithmétiques.
- La méthode de Newton *ne contient pas une convergence globale.* Si on part d'un point trop loin du point stationnaire, la matrice Hessienne risque d'être non définie positive ou inversible.

Modifications simples de la méthode

- Utiliser la même Hesienne dans plusieurs itérations pour réduire l'effort du calcul:

$$x_{k+1} = x_k - [\nabla^2 f(x_0)]^{-1} \nabla f(x_k)$$

- Utiliser un pas réduit en introduisant le facteur α_k pour garantir la descente de la fonction d'objective, c'est l'exemple de garantir le critère de Armijo (backtracking):

$$x_{k+1} = x_k - \alpha_k [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

Exercice 3.4 Refaire l'exemple 3.5 par la méthode Newton avec un pas approximatif vérifiant le critère de Armijo. ■

Combinaison de la méthode de Newton avec du gradient

Un algorithme hybride Newton/gradient pour des points de départ loins où la matrice Hesienne n'est pas définie positive, pourrait améliorer les prestations de la méthode de Newton.

Exercice 3.5 Appliquer la méthode hybride Newton/gradient avec un pas approximatif vérifiant le critère de Armijo pour minimiser la fonction: $f(x) = 100(x_2 - x_1^2)^2 + (1 + x_1)^2$. Utiliser la commande *chol* pour vérifier l'inversibilité de la matrice Hesienne. ■

Si le Hessien n'est pas défini positif, il existe des modifications apportées à cette matrice pour qu'elle soit définie positive. Parmi ces méthodes, on cite la méthode de $Levenberg - Marquardt$.

3.5.4 Méthode de Levenberg-Marquardt

Cette méthode consiste à régulariser la matrice Hessienne par un facteur μ:

$$x_{k+1} = x_k - \alpha[\mu I + \nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

où μ est un scalaire positif. Le processus s'initialise par une petite valeur de μ et si la matrice $[\mu I + \nabla^2 f(x_k)]$ n'est pas définie positive, on augmente μ par un facteur β et on tente de nouveau.

■ **Exemple 3.7** Dans le présent exemple, on traite le problème d'optimisation sous matlab à travers trois programmes type *function*:

- *levmarq.m*
- *grad.m*
- *hesslocal.m*

Le premier *levmarq.m* est chargé de la recherche du point stationnaire en employant l'expression:

$$x_{k+1} = x_k - \alpha d_k$$

La méthode permet d'obtenir la direction d_k en appliquant une réflexion du gradient point x_k par le biais:

$$d_k = [\mu I + \nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

L'algorithme développé utilise le coefficient $\mu = 0.001$ et gère la syntaxe suivante:

$$[fopt, xopt, iter] = levmarq(funcion, x0, epsilong, epsilonf)$$

où les paramètres d'entrée sont:

- *function*: c'est le nom de la fonction de coût, qui doit avoir uniquement un paramètre x comme entrée.
- x_0: est le point initial.
- *epsilong*: est la valeur minimale de tolérance du gradient de la fonction et qui définie aussi la condition d'arrêt de l'algorithme. La série suivante de code matlab montre le contenu du l'algorithme développé.

```
function [fopt,xopt,iter] = levmarq(funcion,x0,epsilong,
    epsilonf)
i=1;
tam=size(x0);n=tam(1,1);
epsg(i)=100;
epsf(i)=100;
itermax=1000;
l=0.001;
alpha=1;

f0=feval(funcion,x0);F=f0;
g0=grad(funcion,x0);G=g0;

while abs(epsg(i)) > epsilong | abs(epsf(i)) > epsilonf & i
    < itermax
H=hesslocal(funcion,x0);
dk=(-1)*(linsolve((H+(l*eye(n))),G));
alpha = fminsearch(@(alfa) funobj(x0+alfa*dk), alpha);
x1=x0+alpha*dk;
```

```
18 f0=F; g0=G;
19 x0=x0+alpha*dk;
20 F=feval(funcion,x0);
21 G=grad(funcion,x0);
22 i=i+1;
23 epsf(i)=abs(F-f0)/abs(f0);
24 epsg(i)=norm(G);
25 end
26
27 xopt=x0;
28 fopt=F;
29 iter=i;
30 epsf(1)=[];
31 epsg(1)=[];
32 plot([2:iter],epsf,'ro-');
33 xlabel('Iteration');
34 ylabel('Variation relative de la function');
35 figure
36 plot([2:iter],epsg,'b*-');
37 xlabel('Iteration');
38 ylabel('Magnitud del gradiente');
```

Dans la ligne 16 du programme, la fonction prédéfinie *fminsearch* sert à trouver le pas optimal à chaque étape.
Le deuxième programme script, permet de calculer la matrice Hessienne d'une fonction donnée dans un point défini. La syntaxe est la suivante:

$$H = hesslocal(function, x_0)$$

Dans la suivante liste, le contenu de ce programme est montré.

```
1  function [H] = hesslocal(funcion,x0)
2  tam=size(x0);
3  n=tam(1,1);
4  H=zeros(n,n);
5  for i=1:n
6  deltaveci=zeros(n,1);
7  deltai=abs(0.001*x0(i)/norm(x0));
8  deltaveci(i)=deltaveci(i)+deltai;
9  for j=1:n
10 if i == j
11 H(i,j)=(feval(funcion,x0+deltaveci)-2*feval(funcion,x0));
12 H(i,j)=(H(i,j)+feval(funcion,x0-deltaveci))/(deltai*deltai);
13 elseif i < j
14 deltavecj=zeros(n,1);
15 deltaj=abs(0.01*x0(j)/norm(x0));
16 deltavecj(j)=deltaveci(j)+deltaj;
17 H(i,j)=(feval(funcion,x0+deltaveci+deltavecj));
18 H(i,j)=H(i,j)+(feval(funcion,x0-deltaveci-deltavecj));
19 H(i,j)=H(i,j)-(feval(funcion,x0+deltaveci-deltavecj));
20 H(i,j)=H(i,j)-(feval(funcion,x0-deltaveci+deltavecj));
```

```
H(i,j)=(H(i,j))/((4*deltai*deltaj));
 else
H(i,j)=H(j,i);
end
end
end
```

Le dernier script est chargé du calcul du vecteur gradient d'une fonction multivariable en un point donné. La syntaxe de cette commande est décrit par le programme suivant:

```
function [G] = grad(funcion,x0)
n=size(x0);
 n=n(1,1);
G=zeros(n,1);
for i=1:n
deltavec=zeros(n,1);
delta=abs(0.01*x0(i)/norm(x0));
 deltavec(i)=deltavec(i)+delta;
G(i)=(feval(funcion,x0+deltavec)-feval(funcion,x0-deltavec))
    /(2*delta);
end
```

La fonction d'objectif pour le problème planifié est:

```
function y=funobj(x)
y=x(1)^2+2*x(2)^2+2*x(3)^2+2*x(1)*x(2)+2*x(2)*x(3);
```

L'appelation et l'exécution de la fonction donne les résultats suivants:

```
>> [fop,xopt,ite]=levmarq('funobj',[2;4;10],1e-5,1e-5)

fop =

   1.9339e-37


xopt =

   1.0e-18 *

    0.0833
   -0.0002
    0.3055


ite =

    13
```

Figure 3.11: Résultats du programme *levmarq.m*

■

3.5.5 Méthode de Quasi-Newton

Souvent, dans la pratique, l'inverse du Hessien (on le note par H^{-1}) est très difficile à évaluer dans le cas où la fonction f n'est pas analytique. On souhaite donc ne pas calculer

exactement le Hessien, mais simplement en évaluer une approximation. Parmi les méthodes d'approximation du Hessien sont retenues ici: la méthode *BFGS* pour *Broyden-Fletcher-Goldfarb-Shanno* et la méthode *DFP* pour *Davidon - Fletcher - Powell*.
Dans ce qui suit, on note $s_k = x_{k+1} - x_k$ et $y_k = \nabla f_{k+1} - \nabla f_k$ et on choisit une matrice H_0 définie positive (pour des raisons de convexité): la matrice d'identité est habituellement choisie.

Actualisation du Hessien par la méthode BFGS

Dans cette approche, l'approximation du Hessien est donnée par:

$$H_{k+1} = H_k + \frac{y_k^T y_k}{y^T s_k} - \frac{H_k s_k s_k^T H_k^T}{s_k^T H_k s_k}$$

Actualisation du Hessien par la méthode DFP

Dans cette approche, l'approximation du Hessien est donnée par:

$$H_{k+1} = H_k + \frac{s_k^T s_k}{s^T y_k} - \frac{H_k y_k y_k^T H_k^T}{y_k^T H_k y_k}$$

■ **Exemple 3.8** Dans le code matlab suivant, les deux méthodes: BFGS et DFP sont programmées avec la recherche approximative basée sur backtracking.

```
function [x i f nfeval]=quasi_newton(fun,x,methode)
rho=0.01;
beta=0.1;
[f g]=fun(x);
H=eye(length(x));
eps1=1e-5;
eps2=1e-8;
maxit=1000;nfeval=1;
for i=1:maxit
    xp=x; gp=g;
    p=-H*g;
    nh=norm(p);
    if norm(g,inf)<eps1 || nh<=eps2*(eps2+norm(x))
        break
    end
    alpha=1;
    for k=1:10          %backtracking recherche lineaire
        xnew=x+alpha*p;
        fxnew=fun(xnew);
        nfeval=nfeval+1;
        if  fxnew < f+alpha*rho*g'*p
            break
        else
            alpha=alpha*beta;
        end
    end
    x=x+alpha*p;
    s=x-xp;
    ns=norm(s);
    [f g]=fun(x);
```

```
31      y=g-gp;
32      ys=y'*s;
33      nfeval=nfeval+1;
34      if ys>sqrt(eps)*ns*norm(y)
35          v=H*y;
36          yv=y'*v;
37          if methode==1
38              H=H-(v/yv)*v'+(s/ys)*s';
39          else
40              H=H+(1+yv/ys)*(s*s')/ys-(s*v'+v*s')/ys;
41          end
42      end
43  end
```

Le programme est évalué sur la fonction d'objective suivante:

```
1  function [f g] = objfun_min3(x)
2  f = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
3  if nargout<2,
4      return,
5  end
6  g = [-200*(x(2)-x(1)^2)*2*x(1)-2*(1-x(1));200*(x(2)-x(1)^2)];
```

```
>> [x i f nf]=quasi_newton(@objfun_min3,[-1.2;1],1)

x =

    1.0000
    1.0000


i =

   247


f =

   1.7976e-10


nf =

   506
```

Figure 3.12: Résultats du programme quasi-Newton appliquant la méthode de DFP

■

3.5.6 Problème des moindres carrés

Le problème des moindres carrés minimise la différence entre un ensemble de données et une fonction modèle qui approxime ces données. Étant donné un ensemble de données $d(t_i, y_i)$ et un modèle $\phi(x, t_i)$, alors la différence entre les fonctions donne l'équation $r_i(x) = \phi(x, t_i) - y_i$, où y_i est le composant y donné au point t_i. La fonction d'objective du

```
>> [x i f nf]=quasi_newton(@objfun_min3,[-1.2;1],2)

x =

    1.0000
    1.0000


i =

    40


f =

   3.0678e-14


nf =

    88
```

Figure 3.13: Résultats du programme quasi-Newton appliquant la méthode de BFGS

problème des moindres carrés est donc:

$$f(x) = \frac{1}{2}\sum r_i^2(x) = \frac{1}{2}R^T(x)R(x)$$

La minimisation de $f(x)$ permet d'approximer plus le modèle théorique aux points observés. Chaque r_i est appelé le résidu et qui est une fonction continue de $\mathbb{R}^n \to \mathbb{R}$. Cette équation inclut la somme de tous les résidus r_i du vecteur *résiduel* r de m composants donné par le vecteur:

$$r = (r_1(x), r_2(x), \cdots, r_m(x))^T$$

Lors du calcul du gradient de $f(x)$, il est nécessaire de trouver le gradient de chaque vecteur résiduel. La Jacobienne $J(x)$ est une matrice de tous les $\nabla r_i(x)$:

$$J(x) = \left[\frac{\partial r_j}{\partial x_i}\right]_{j=1,\cdots,m;i=1,\cdots,n} = \begin{bmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_m(x)^T \end{bmatrix}$$

Le gradient et le Hessien peuvent être exprimés en terme du Jacobien:

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x)\nabla r_j = J^T(x)R(x)$$

$$\nabla^2 f(x) = \sum_{j=1}^{m} \nabla r_j(x)^T \nabla r_j(x) + \sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x) = J^T J + S(x)$$

La matrice Hessienne comporte donc deux termes, le premier ne faisant intervenir que la matrice Jacobienne de f. Si on utilise les deux termes, on peut appliquer une variante de la méthode de Newton à la fonction f. Cependant, le second terme comporte une somme de matrices Hessiennes des composantes de f, possiblement fastidieux à manipuler. C'est pourquoi on peut envisager une approximation de la méthode de Newton qui n'utilisera

que le premier terme J^TJ. C'est la méthode nommée $Gauss-Newton$. Il est clair que la matrice J^TJ n'est inversible que si $m>n$. De plus, il est souvent possible de régulariser le terme à inverser en ajoutant un multiple de l'identité, idée connue sous le nom de méthode Levenberg-Marquardt.

3.5.7 Méthode de Gauss-Newton

Donc, l'algorithme de Gauss-Newton consiste en l'itération:

$$x_{k+1} = x_k - (J^TJ)^{-1}R(x)^TJ$$

La condition bien sûr est que J^TJ soit inversible.

3.5.8 Méthode de Levenberg-Marquardt

L'algorithme de Levenberg–Marquardt consiste en l'itération:

$$x_{k+1} = x_k - (J^TJ+\lambda I)^{-1}R(x)^TJ$$

où $\lambda>0$ est un paramètre de régularisation. Plus λ est grand, plus l'itération ressemble à un algorithme de descente du gradient alors que de petites valeurs de λ font que l'algorithme se rapproche de celui de Gauss–Newton.

■ **Exemple 3.9** Ici, un ensemble de données de la population des USA (en million) et l'année correspondante:

Année	Population
1815	8.5
1825	10
1835	14.7
1845	19.7
1855	26.7
1865	35.2
1875	44.4
1885	55.9

Table 3.1: Population des USA

Cet ensemble de données est noté par $d(t_i,y_i)$ où t_i est l'année et y_i est la population. La fonction du modèle est approximée par $\phi(x,t)=x_1e^{x_2t}$ où l'année 1815 est notée par $t=1$, 1825 notée par $t=2$ est ainsi de suite. La condition initial est $x_0=(6,0.3)$.
La fonction résiduelle:

$$\begin{aligned} R(x) &= \phi(x,t)-d(t_i,y_i) \\ &= \begin{pmatrix} 6e^{0.3(1)}-8.5 \\ 6e^{0.3(2)}-11 \\ \vdots \\ 6e^{0.3(8)}-55.9 \end{pmatrix} \end{aligned}$$

$$X_1 = X_0 - (J^T(X_0)J(X_0))^{-1}J^T(X_0)R(X_0)$$

avec

$$J(X_0) = \nabla R(X_0) = (e^{x_2t}\ tx_1e^{x_2t})$$

■