Résolution d'un problème d'optimisation différentiable

Marc Bourqui Victor Constantin Ian Schori Floriant Simond

January 4, 2013

Énoncé du problème

Trouver (une approximation de) la solution du problème suivant en appliquant le théorème de la plus forte pente:

$$\min_{x \in \mathbb{R}^2} (x_1 - 2)^4 + (x_1 - 2)^2 x_2^2 + (x_2 + 1)^2 \tag{1}$$

Réponses aux questions

(a) Implémenter la méthode de plus forte pente (Algorithme 11.3) à l'aide du logiciel MATLAB. Déterminer la taille du pas en appliquant la recherche linéaire, Algorithme 11.2 (les deux conditions de Wolfe).

Listing 1: pfp.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                        %
% Methodes de descente pour l'optimisation non lineaire  %
% sans contraintes                                       %
%                                                        %
% BOURQUI Marc                                           %
% CONSTANTIN Victor                                      %
% SCHORI Ian                                             %
% SIMOND Floriant                                        %
%                                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x, fx, iterations] = pfp(f, x0, epsilon, useRL, showDetails)

%%%%%%%%%%%%%%%%
% Interface    %
%%%%%%%%%%%%%%%%

% nom de la fonction a minimiser, qui est specifiee dans le fichier 'f.m'
% et qui est declaree sous forme de string
fct = f;

% point initial
x = x0;

%%%%%%%%%%%%%%%%%%
% Parametres     %
%%%%%%%%%%%%%%%%%%

% pour le critère d'arrêt
maxIter = 200    ;

% initialisation du nombre d'iterations
i=1 ;

% initialisation de la matrice qui stocke tous les iterés
% un iteré = une colonne de cette matrice

stock(:,i) = x0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Boucle principale            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Critere d'arret: x a ateint la precision demandée OU nb iterations max ateint

while ( normGradient(fct,stock(:,i)) >= epsilon ) && ( i <= maxIter )
    % mise a jour du nombre d'iterations
```

```
    i = i+1;

    prev = stock(:,i-1);
    if showDetails
        fprintf('Iteration number %d : x = [%f, %f]\n', i, prev(1), prev(2));
    end
    % calcul et stockage de la valeur du nouveau x
    stock(:,i) = pfpInnerLoop(fct, prev, useRL);

end

% Calcul de la taille de la matrice contenant tous les x
taille = size(stock,2);

% Evaluation de la fonction en chaque point
for i=1:taille
    valeurstock(i)=feval(fct,stock(:,i));
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Affichage des résultats  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if showDetails
    disp('Valeur de la suite des x :') ;
    stock'
end
disp('*****************************************')
disp(['Nombre d''iterations :                    ' num2str(i-1)])
disp(['Valeur de la fonction a l''optimum : ' num2str(feval(fct,stock(:,i)))] ) ;
disp('Valeur de l''optimum : ')
xOptim = stock(:,i)'
disp('*****************************************')

if showDetails
    % passage au module de visualisation de la fonction et des resultats

    visual3d(fct, stock, valeurstock);
end
%sprintf('Nombre de fois que la boucle a ete parcourue : %d',i)

x = xOptim;
fx = feval(fct,stock(:,i));
iterations = i-1;

%clear;
end
```

Pour `pfp.m`, nous avons réutilisé la structure du corrigé de la série 3. Nous l'avons adapté pour y résoudre la màthode de la plus forte pente, selon l'algorithme 11.3. La fonction, son gradient et sa hessienne sont placés dans un fichier que nous avons nommé `f.m`. De plus, nous avons ajouté un booléan useRL qui permet de sélectionner la méthode de détermination du pas (true : recherche linéaire, false : le pas calculé en (b)).

Listing 2: pfpInnerLoop.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                  %
% Calcul d'un itéré et du pas soit en utilisant    %
% la recherche linéaire soit la formule de Cauchy  %
%                                                  %
% BOURQUI Marc                                     %
% CONSTANTIN Victor                                %
% SCHORI Ian                                       %
```

```
% SIMOND Florian                                                 %
%                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function x = pfpInnerLoop(f, x0, useRL)

    alpha = 1;
    x = x0;

    [fx, gfx] = feval(f, x);
    d = -gfx;

    % Calcul du pas
    if useRL
        % Avec la recherche linéaire
        beta1 = 0.5;
        beta2 = 0.75;
        lambda = 2;
        alpha = rechercheLineaire(f, fx, gfx, x, alpha, beta1, beta2, lambda);
    else
        %Soit on peut utiliser la fonction dans b) pour calculer le pas
        alpha = taillepasCauchy(f,x);
    end
    x = x + alpha * d;
end
```

Cette foction effectue une itération de l'algorithme de la plus forte pente en utilisant la méthode de calcul du pas spécifiée. Le choix est effectué à l'aide du booléen useRL qui permet de choisir entre la recherche linéaire et la méthode indiquée au point (b). Le paramètre x0 est l'itéré précedent.

Listing 3: rechercheLineaire.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Implémente la recherche linéaire                              %
%                                                               %
% BOURQUI Marc                                                  %
% CONSTANTIN Victor                                             %
% SCHORI Ian                                                    %
% SIMOND Florian                                                %
%                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function alpha = rechercheLineaire(f, fx, gfx, x, alpha0, beta1, beta2, lambda)
    alpha = alpha0;
    alphal = 0;
    alphar = inf;

    [fxad, fgxad] = feval(f, x + alpha * -gfx);

    while (fxad > fx + alpha * beta1 * gfx' * -gfx) || (fgxad' * -gfx < beta2 * ...
        gfx' * -gfx)
        if fxad > fx + alpha * beta1 * gfx' * -gfx
            alphar = alpha;
            alpha = (alphal + alphar)/2;
        elseif fgxad' * -gfx < beta2 * gfx' * -gfx
            alphal = alpha;
            if alphar < inf
                alpha = (alphal + alphar)/2;
            else
                alpha = lambda * alpha;
            end
        end

        [fxad, fgxad] = feval(f, x + alpha * -gfx);
```

```
32      end
33  end
```

Cette fonction implémente la recherche linéaire d'après l'algorithme 11.2. Le paramètre fx est la fonction évaluée en x, et le paramètre gfx est son gradient en x. Nous avons choisi de les passer en paramètres pour ne pas devoir les recalculer. Mais pour plus de modularité, on peut déterminer fx et gfx en ajoutant [fx, gfx] = feval(f, x); avant la boucle while.

(b) Implémenter une fonction qui donne la taille du pas suivant:

$$\alpha_k = \frac{\bigtriangledown f(x_k)^T \bigtriangledown f(x_k)}{\bigtriangledown f(x_k)^T \bigtriangledown^2 f(x_k) \bigtriangledown f(x_k)} \tag{2}$$

Quelle est la nature de ce pas? D'où cette formule vient-elle?

Listing 4: taillepasCauchy.m

```
1  function pas = taillepasCauchy(f, x)
2      [~, fgx, fghx] = feval(f,x);
3      top = fgx' * fgx;
4      bottom = fgx' * fghx * fgx;
5
6      pas = top / bottom;
7
8  end
```

Comme f n'est pas une fonction linéaire, on peut l'approximer par un modèle quadratique. On va chercher à miniser ce modèle dans la direction de la plus forte descente, à savoir $\bigtriangledown f$. Le point qui minimise ce modèle est le point de Cauchy et se détermine de la manière suivante :

$$x_C = x_k - \alpha_C \bigtriangledown f(x_k) \tag{3}$$

où

$$\alpha_C = \underset{\alpha \in \mathbb{R}_0^+}{\operatorname{argmin}}\ m_{x_k}(x_k - \alpha \bigtriangledown f(x_k)) \tag{4}$$

Sachant f convexe, α_C peut être calculé par (2).

(c) Comparer le comportement de l'algorithme en utilisant les pas (a) et (b).

Pour comparer ces deux méthodes, nous avons choisi d'utiliser comme critère le nombre d'itérations que prend chacun.

Pour pouvoir repondre aux questions c et d nous avons créé un fichier MatLab, `comparator.m`, qui nous permet de lancer deux méthodes l'une après l'autre et qui enregistre les résultats. Il génère aussi un tableau contenant le nombre d'itérations pour chaque méthode et des graph, ceci dans le but de nous faciliter la comparaison.

Afin de comparer les pas, il faut mettre compareSteps = true dans `compartor.m`. Pour générer le tableau 1 , nous avons défini arbitrairement 10 vecteurs x dans le fichier `x.m`.

Afin d'avoir plus de valeurs nous avons défini une fonction en colimaçon dans le fichier `xsnake.m`, qui génère une suite de points commencant à la solution ($(2,-1)$; $(3,-1)$; $(3,0)$; $(-1,0)$; $(1,-2)$; ...). Nous l'avons utilisé pour générer beaucoup de points afin d'avoir des valeurs pour nos graph.

Dans la figure 2 nous avons un premier graph avec 500 valeurs.

Ce graphique n'est pas très explicite, mais on observe tout de même la tendance générale de la méthode de plus forte pente avec la recherche linéaire qui nécessite beaucoup d'itérations lorsque la distance à la solution est grande. L'utilisation du pas de Cauchy permet d'améliorer grandement l'efficaticité de l'algorithme, en particulier pour de grandes distances. Toutefois, on remarque deux tendances, dont l'une tend à être inefficace tandis que l'autre reste efficace.

k	x_0		$x_{optim,rchlin}$		$f(x_{optim,rchlin})$	n_{rchlin}	$x_{optim,cauchy}$		$f(x_{optim,cauchy})$	n_{cauchy}
1	0	0	2	-1	1.7309e-14	8	2	-1	2.1765e-11	11
2	1	1	2	-1	1.5584e-16	7	2	-1	1.5015e-11	9
3	5	5	2	-1	8.4792e-23	11	2	-1	1.5649e-11	13
4	-5	-5	2	-1	7.4719e-12	25	2	-1	2.5613e-13	13
5	-10	10	2	-1	1.0878e-20	15	2	-1	1.703e-11	19
6	20	-10	2	-1	1.0794e-11	30	2	-1	2.4096e-11	16
7	-50	10	2	-1	2.5332e-22	18	2	-1	1.4666e-11	17
8	60	30	2	-1	5.1019e-16	29	2	-1	9.4145e-12	20
9	-80	100	1.9819	58.213	3507.3	200	2	-1	2.425e-12	193
10	-80	2	2	-1	2.499e-14	18	2	-1	1.0285e-12	21

Figure 1: Recherche linéaire vs. pas de Cauchy

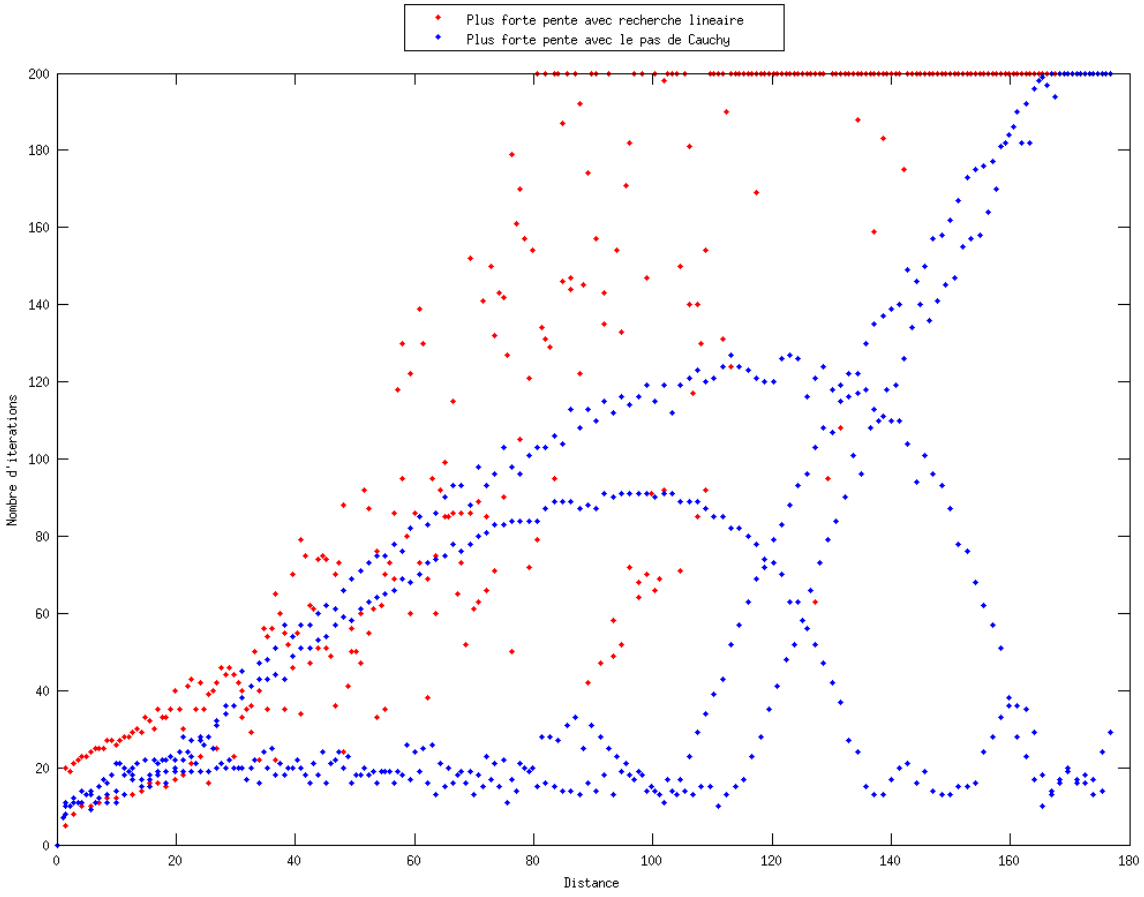
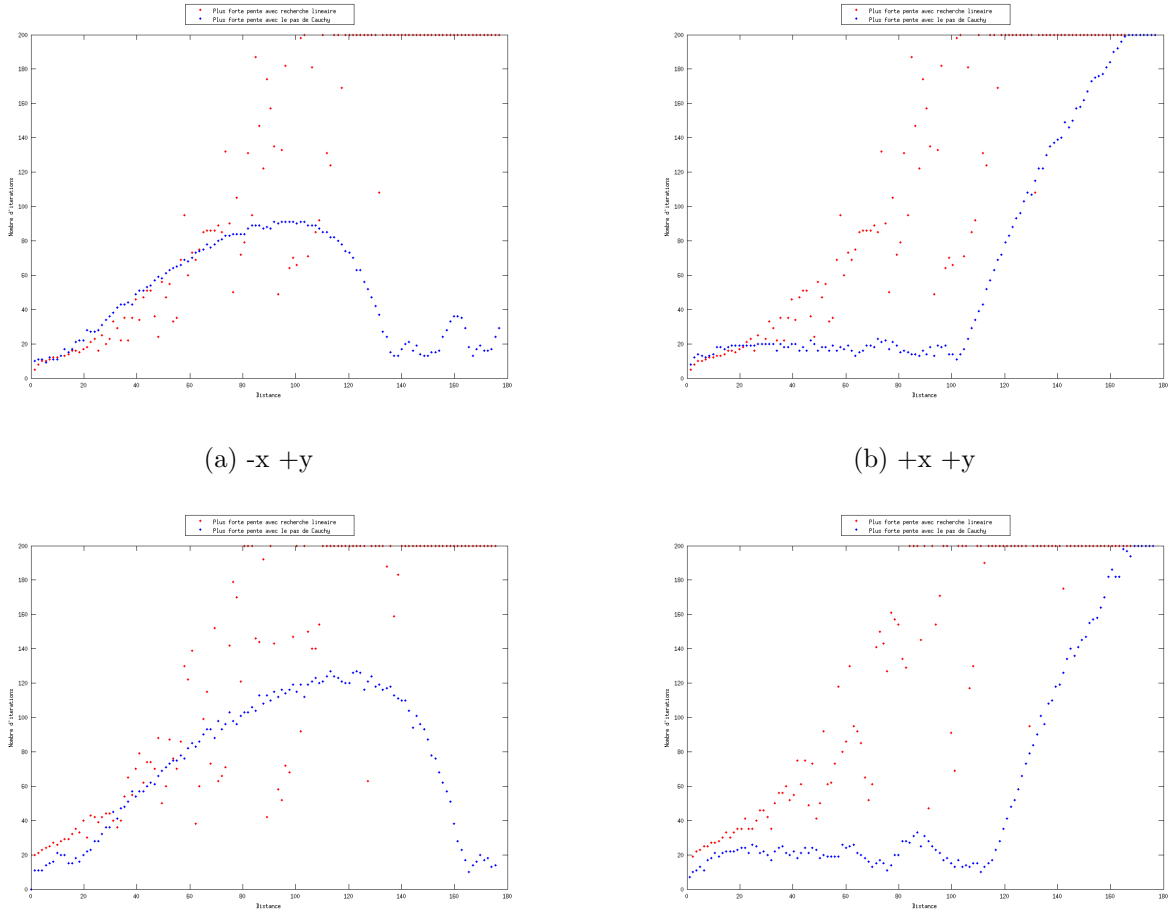


Figure 2: Recherche linéaire vs. quasi-Newton

Nous avons séparé les résultats selon les différences de coordonnées du point de départ par rapport à la solution. Nous avons représenté les valeurs sur la figure 3.

On remarque que lorsque x est diminuée, l'alogrithme utilisant le pas de Cauchy est bien plus efficace que la recherche linéaire, en particulier pour de grandes distances à la solution. Alors que lorsque x est augmenté, cette méthode reste généralement meilleure que la recherche linéaire. Cependant, son efficacité est grandement réduite pour de grandes distances.



(a) -x +y

(b) +x +y

(c) -x -y

(d) +x -y

Figure 3: Graphs comparaison recherche linéaire vs. Cauchy par quadrant

On notera aussi que l'algorithme utilisant le pas de Cauchy change de comportement selon le décalage sur x. En effet, jusqu'à la centaine, lorsque que x est dédéuit, l'algorithme perd en efficacité, alors que lorsque x est ajouté, l'algorithme est stable. Mais à partir d'une distance supérieure à la centaine, l'inverse se produit.

(d) Comparer la methode de plus forte pente et la methode quasi-Newton (qui est déjà implementée – Série 3).

D'une manière générale, la méthode de la plus forte pente est plus efficace que la méthode de quasi-Newton lorsque la distance entre le point de départ et la solution est petite.

Pour se faire une idée du comportement des deux méthodes, nous avons générés 350 valeurs de départ d'après notre méthode xsnake. Le résultat est présenté sur la figure 5.

Au premier coup d'oeil, on observe que la méthode quasi-Newton est plus efficace que la méthode de plus forte pente. Toutefois, la méthode de plus forte pente est en moyenne une fois sur deux plus efficace à proximité de la solution.

Pour distinguer plus clairement ces différences, nous avons séparé les résultats selon les 4 mêmes cadrans qu'au point c.

On remarque que lorsque l'on s'éloigne de la solution en augmentant y, la méthode de plus forte pente est plus appropriée à proximité de la solution. Mais à partir d'une distance à la solution de 40 environ, la méthode de quasi-Newton devient plus performante.

Par contre, lorsque l'on s'éloigne selon $-y$, la méthode de plus forte pente est quasiment constamment plus inefficace.

k	x_0		$x_{optim,pfp}$		$f(x_{optim,pfp})$	n_{pfp}	$x_{optim,qN}$		$f(x_{optim,qN})$	n_{qN}
1	0	0	2	-1	1.7309e-14	8	2.0004	-1.0001	1.3149e-07	10
2	1	1	2	-1	1.5584e-16	7	1.9999	-1.0003	7.2551e-08	8
3	5	5	2	-1	8.4792e-23	11	1.9999	-1	1.0205e-08	15
4	-5	-5	2	-1	7.4719e-12	25	2	-1	2.5406e-10	20
5	-10	10	2	-1	1.0878e-20	15	2	-1	1.8655e-10	22
6	20	-10	2	-1	1.0794e-11	30	2	-1	3.6242e-11	25
7	-50	10	2	-1	2.5332e-22	18	2	-1	1.4196e-10	33
8	60	30	2	-1	5.1019e-16	29	2	-1	3.4393e-10	33
9	-80	100	1.9819	58.213	3507.3	200	2	-1	2.3289e-09	36
10	-80	2	2	-1	2.499e-14	18	2.0001	-0.99997	1.2099e-08	35

Figure 4: Plus forte pente vs. Quasi-Newton

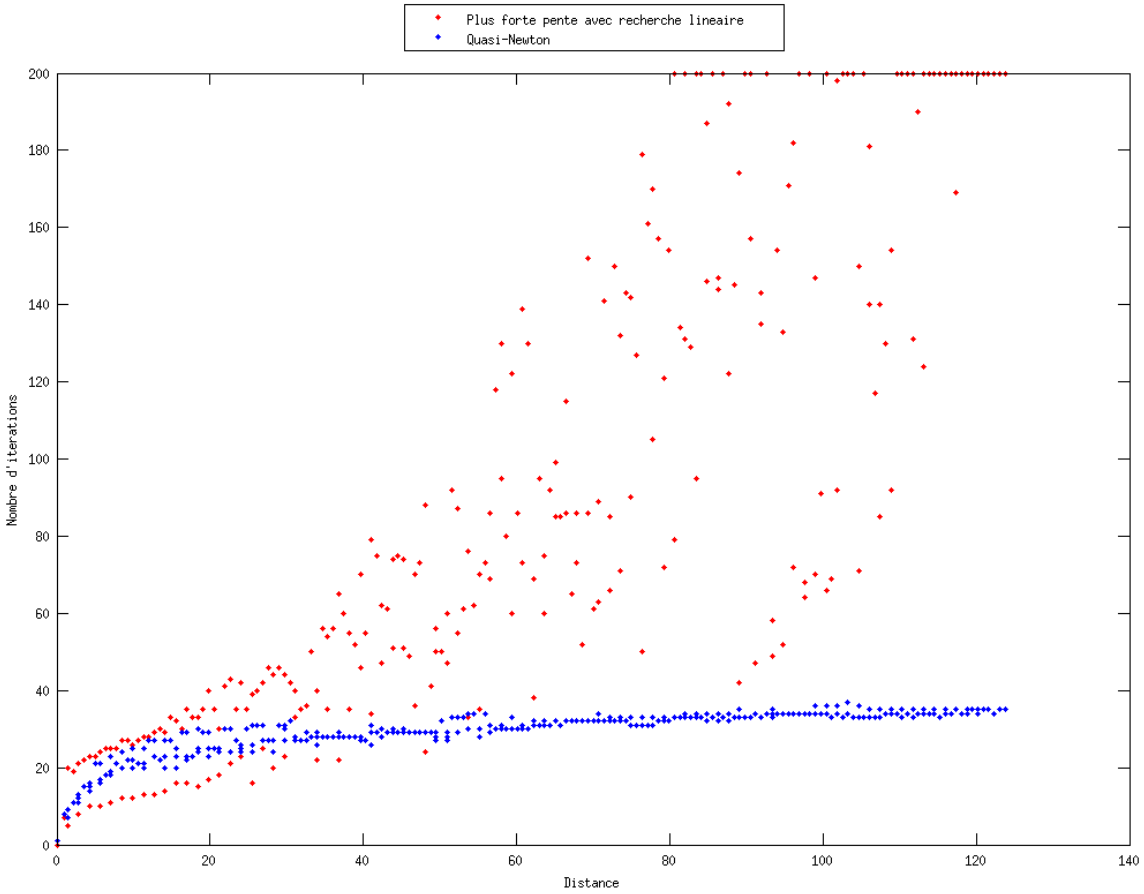
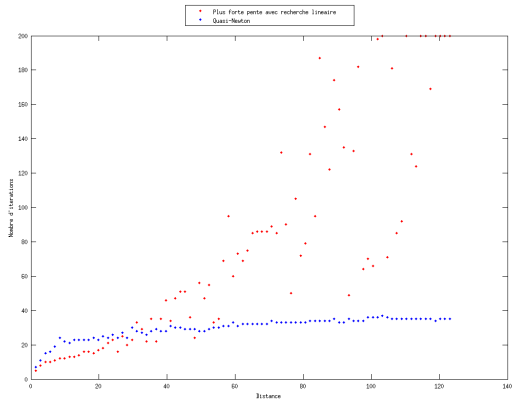
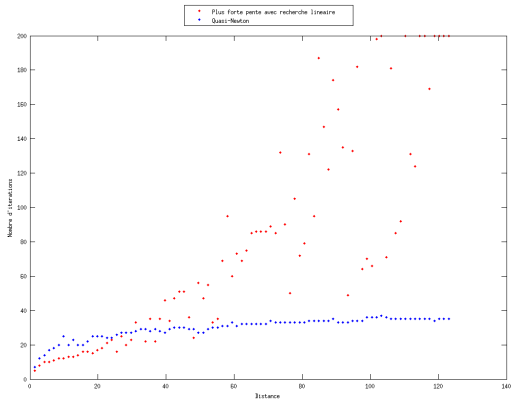


Figure 5: PFP VS Quasi-Newton

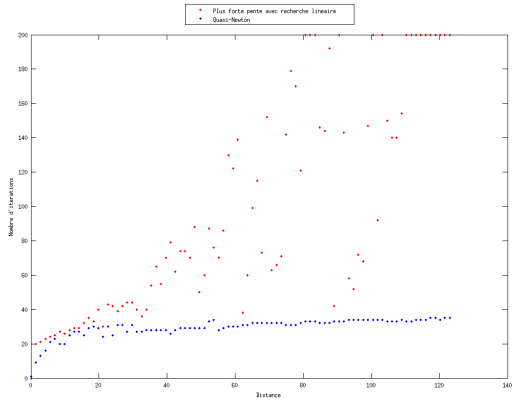
La méthode de quasi-Newton semble se stabiliser assez rapidement et le nombre d'itérations augmente très légérement. Alors que la méthode de plus forte pente donne des résultats aléatoires.



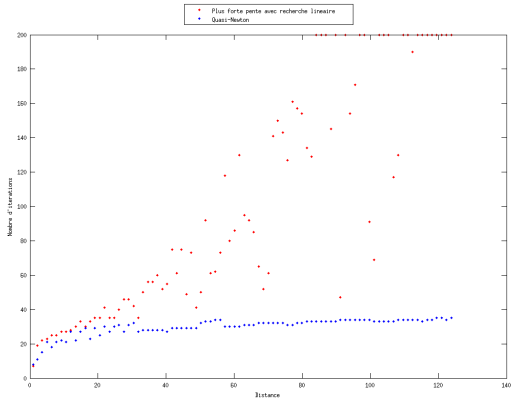
(a) -x +y



(b) +x +y



(c) -x -y



(d) +x -y

Figure 6: Graphs comparaison plus forte pente vs. Quasi-Newton par quadrant