

Résolution d'un problème d'optimisation différentiable

Marc BOURQUI

Victor CONSTANTIN

Ian SCHORI

Florian 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:

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

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

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
3 % Methodes de descente pour l'optimisation non lineaire %
4 % sans contraintes %
5 % %
6 % BOURQUI Marc %
7 % CONSTANTIN Victor %
8 % SCHORI Ian %
9 % SIMOND Floriant %
10 % %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12 function [x, fx, iterations] = pfp(f, x0, epsilon, useRL, showDetails)
13
14 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
15 % Interface %
16 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
17
18 % nom de la fonction a minimiser, qui est specifiee dans le fichier 'f.m'
19 % et qui est declaree sous forme de string
20 fct = f;
21
22 % point initial
23 x = x0;
24
25 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
26 % Parametres %
27 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
28
29 % pour le critere d'arrêt
30 maxIter = 200 ;
31
32 % initialisation du nombre d'iterations
33 i=1 ;
34
35 % initialisation de la matrice qui stocke tous les iterés
36 % un iteré = une colonne de cette matrice
37
38 stock(:, i) = x0;
39
40 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
41 % Boucle principale %
42 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
43
44
45 % Critere d'arret: x a atteint la precision demandée OU nb iterations max atteint
46
47 while ( normGradient(fct, stock(:, i)) >= epsilon ) && ( i <= maxIter )
48     % mise a jour du nombre d'iterations
```

```

49     i = i+1;
50
51     prev = stock(:,i-1);
52     if showDetails
53         fprintf('Iteration number %d : x = [%f, %f]\n', i, prev(1), prev(2));
54     end
55     % calcul et stockage de la valeur du nouveau x
56     stock(:,i) = pfpInnerLoop(fct, prev, useRL);
57
58 end
59
60 % Calcul de la taille de la matrice contenant tous les x
61 taille = size(stock,2);
62
63 % Evaluation de la fonction en chaque point
64 for i=1:taille
65     valeurstock(i)=feval(fct, stock(:,i));
66 end
67
68
69 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
70 % Affichage des résultats %
71 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
72 if showDetails
73     disp('Valeur de la suite des x :') ;
74     stock'
75 end
76 disp('*****')
77 disp(['Nombre d\'iterations : ' num2str(i-1)])
78 disp(['Valeur de la fonction a l\'optimum : ' num2str(feval(fct, stock(:,i)))] ) ;
79 disp('Valeur de l\'optimum : ')
80 xOptim = stock(:,i)'
81 disp('*****')
82
83 if showDetails
84     % passage au module de visualisation de la fonction et des resultats
85
86     visual3d(fct, stock, valeurstock);
87 end
88 %sprintf('Nombre de fois que la boucle a ete parcourue : %d',i)
89
90 x = xOptim;
91 fx = feval(fct, stock(:,i));
92 iterations = i-1;
93
94 %clear;
95 end

```

Pour `pfp.m`, nous avons réutilisé la structure du corrigé de la série 3. Nous l'avons adapté pour y résoudre la méthode 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 booléen `useRL` qui permet de sélectionner la méthode de détermination du pas (`true` : recherche linéaire, `false` : le pas calculé en (??)).

Listing 2: `pfpInnerLoop.m`

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % Calcul d'un itéré et du pas soit en utilisant %
4 % la recherche linéaire soit la formule de Cauchy %
5 %
6 % BOURQUI Marc %
7 % CONSTANTIN Victor %
8 % SCHORI Ian %

```

```

9 % SIMOND Floriant %
10 % %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12
13 function x = pfpInnerLoop(f, x0, useRL)
14
15     alpha = 1;
16     x = x0;
17
18     [fx, gfx] = feval(f, x);
19     d = -gfx;
20
21     % Calcul du pas
22     if useRL
23         % Avec la recherche linéaire
24         beta1 = 0.5;
25         beta2 = 0.75;
26         lambda = 2;
27         alpha = rechercheLineaire(f, fx, gfx, x, alpha, beta1, beta2, lambda);
28     else
29         %Soit on peut utiliser la fonction dans b) pour calculer le pas
30         alpha = taillepasCauchy(f,x);
31     end
32     x = x + alpha * d;
33 end

```

Cette fonction 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 (??). Le paramètre x0 est l'itéré précédent.

Listing 3: rechercheLineaire.m

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 % Implémente la recherche linéaire %
3 % %
4 % BOURQUI Marc %
5 % CONSTANTIN Victor %
6 % SCHORI Ian %
7 % SIMOND Floriant %
8 % %
9 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
10
11 function alpha = rechercheLineaire(f, fx, gfx, x, alpha0, beta1, beta2, lambda)
12     alpha = alpha0;
13     alphas = 0;
14     alphas = inf;
15
16     [fxad, fgxad] = feval(f, x + alpha * -gfx);
17
18     while (fxad > fx + alpha * beta1 * gfx' * -gfx) || (fgxad' * -gfx < beta2 * ...
19         gfx' * -gfx)
20         if fxad > fx + alpha * beta1 * gfx' * -gfx
21             alphas = alpha;
22             alpha = (alpha + alphas)/2;
23         elseif fgxad' * -gfx < beta2 * gfx' * -gfx
24             alphas = alpha;
25             if alphas < inf
26                 alpha = (alpha + alphas)/2;
27             else
28                 alpha = lambda * alpha;
29             end
30         end
31
32     [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:

$$\alpha_k = \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_k)^T \nabla^2 f(x_k) \nabla f(x_k)} \quad (2)$$

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 minimiser ce modèle dans la direction de la plus forte descente, à savoir ∇f . Le point qui minimise ce modèle est le point de Cauchy et se détermine de la manière suivante :

$$x_C = x_k - \alpha_C \nabla f(x_k) \quad (3)$$

où

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

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

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

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

Pour pouvoir répondre aux questions ?? et ?? 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 ?? , 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 commençant à la solution $((2, -1); (3, -1); (3, 0); (-1, 0); (1, -2); \dots)$. Nous l'avons utilisé pour générer beaucoup de points afin d'avoir des valeurs pour nos graph.

Dans la figure ?? 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'efficacité 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

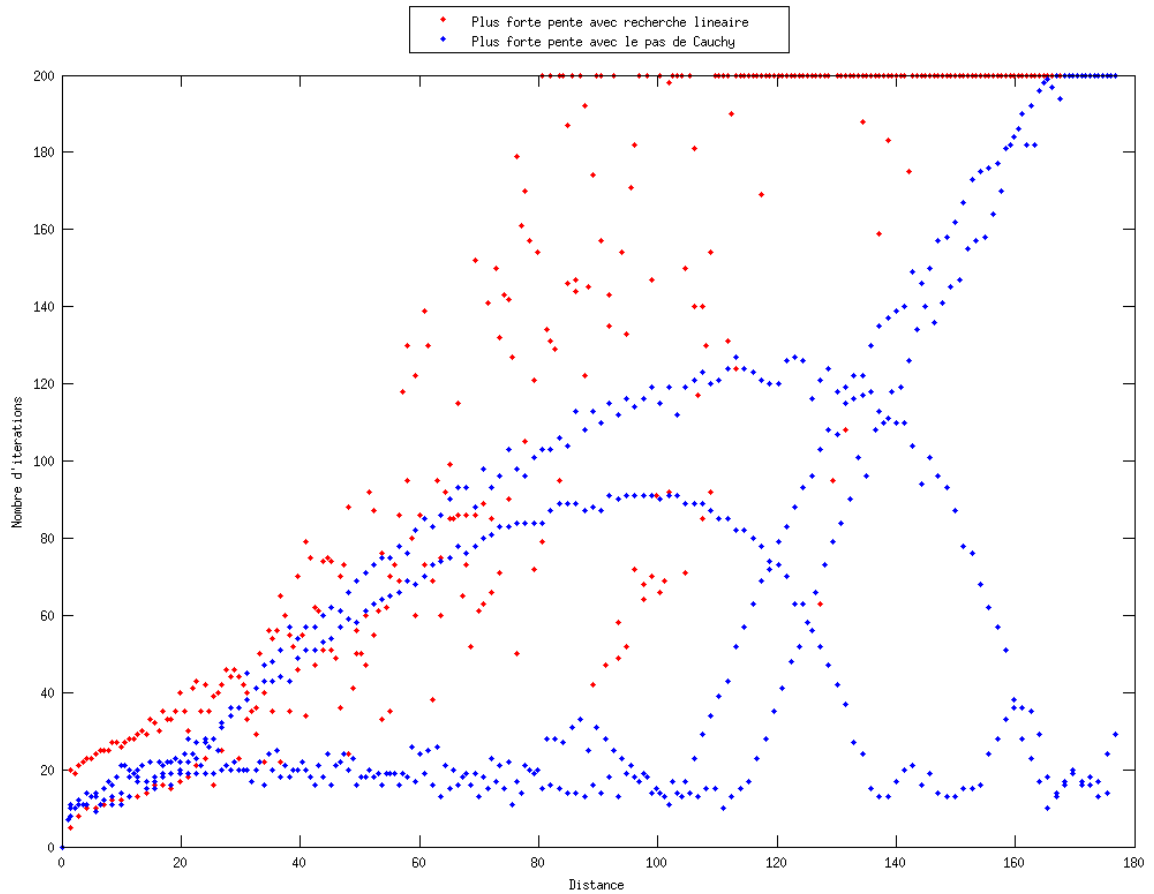
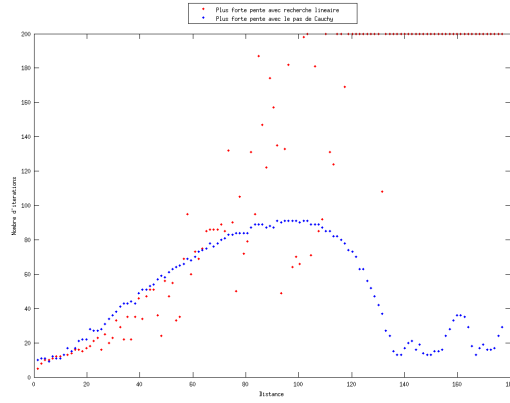


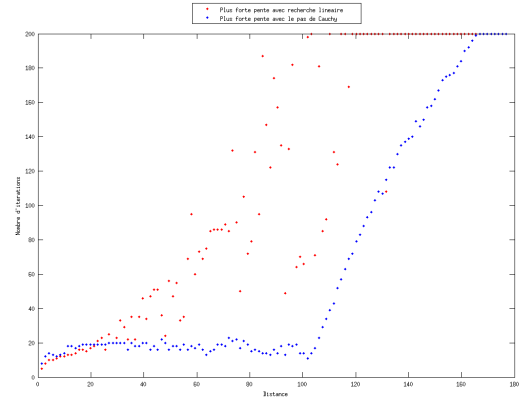
Figure 2: Recherche linéaire vs. quasi-Newton

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 ??.

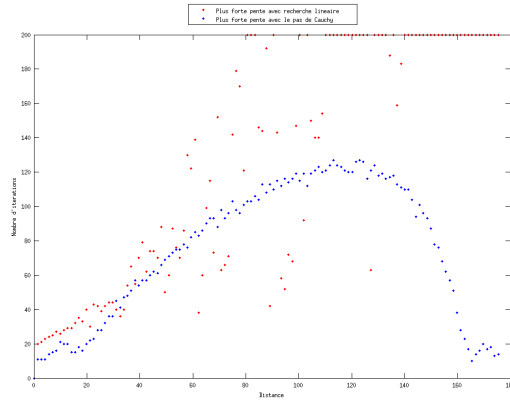
On remarque que lorsque x est diminuée, l'algorithme utilisant le pas de Cauchy est bien plus efficace que la recherche linéaire, en particulier pour de grandes distances à la solution. Alors que lorsque x est augmenté, cette méthode reste généralement meilleure que la recherche linéaire. Cependant,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



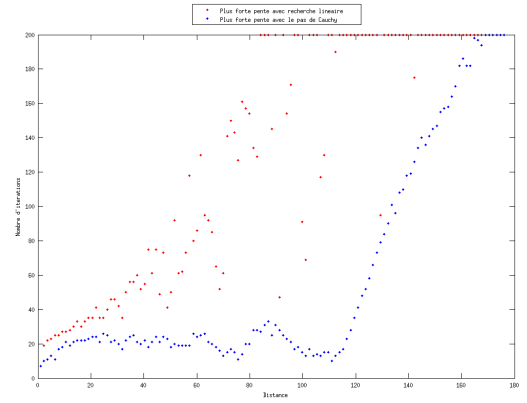
(a) $-x + y$



(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 déduit, 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 ??.

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 ??.

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

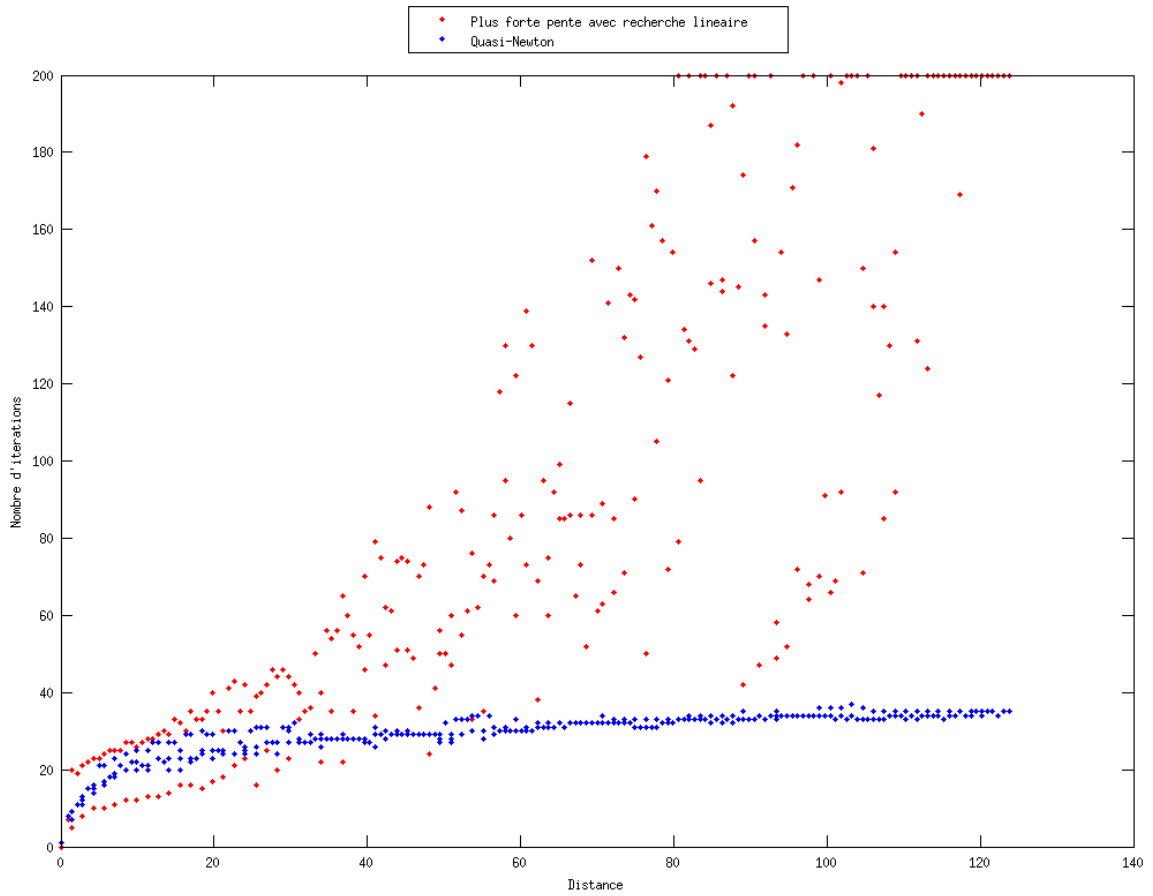
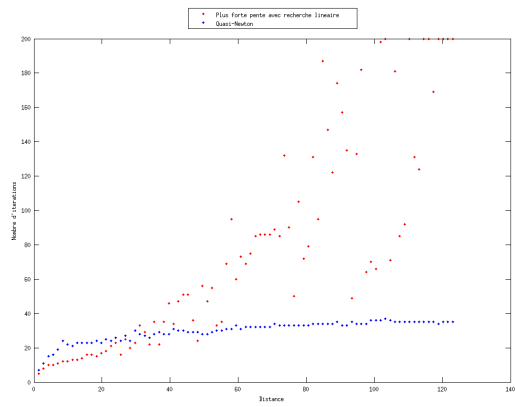
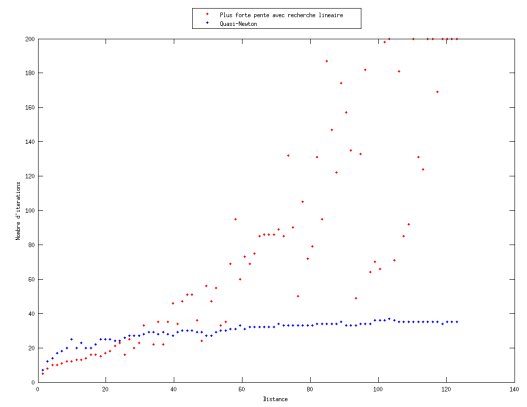


Figure 5: PFP VS Quasi-Newton

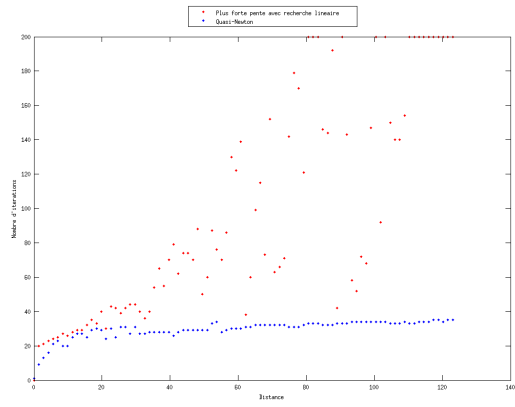
La méthode de quasi-Newton semble se stabiliser assez rapidement et le nombre d'itérations augmente très légèrement. Alors que la méthode de plus forte pente donne des résultats aléatoires.



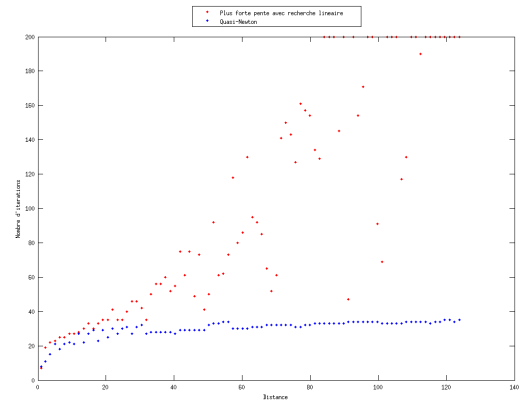
(a) $-x + y$



(b) $+x + y$



(c) $-x - y$



(d) $+x - y$

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