Université Polytechnique Mohammed VI
Africa Business School & College of Computing

QFM-M122 - Stochastic Optimization

TP2: Newton avec région de confiance (Cas multidimensionnel)

Étudiante :
Hanan ATMANI

Professeur :
Abdeslam KADRANI

5 Avril 2024

1 Introduction

Ce travail pratique est centré sur l'implémentation de l'algorithme de Newton avec région de confiance pour résoudre un problème d'optimisation sans contraintes de dimension n. L'objectif principal est de mettre en œuvre cet algorithme, similaire à celui étudié dans le TP1, en mettant l'accent sur le calcul du minimum du modèle quadratique de dimension n, cette fois-ci réalisé par le gradient conjugué linéaire. Au cours de ce TP, nous explorerons les étapes nécessaires pour réaliser cette implémentation et évaluerons son efficacité dans la résolution de problèmes d'optimisation de grande dimension.

2 Gradient conjugué linéaire

Dans un premier temps, nous examinerons l'algorithme de descente de gradient simple. Cette méthode consiste à suivre la pente de la fonction pour trouver le minimum local. Nous décrirons le fonctionnement de cet algorithme et son application à notre problème spécifique d'optimisation d'une fonction quadratique $q(\delta) = \frac{1}{2}\delta^T H\delta + c^T\delta$, où H est une matrice définie positive et c est un vecteur constant.
Ensuite, nous nous tournerons vers l'algorithme de gradient conjugué. Contrairement à la descente de gradient simple, cette méthode utilise des directions de recherche conjuguées pour atteindre plus efficacement le minimum de la fonction. Nous discuterons de son fonctionnement et de sa mise en œuvre dans notre programme pour résoudre le même problème d'optimisation.

2.1 Gradient simple

Pour débuter, on pose $q(\delta) = f(x,y) = 2x^2 - 2xy + y^2 + 2x - 2y$ avec $\delta = (x,y)$, $H = \begin{pmatrix} 4 & -2 \\ -2 & 2 \end{pmatrix}$ et $b = \begin{pmatrix} 2 \\ -2 \end{pmatrix}$ La fonction ff est quadratique et donc de classe $C^{+\infty}$. Son gradient est donné par:

$$(\nabla f(x,y))^T = (x,y)\begin{pmatrix} 4 & -2 \\ -2 & 2 \end{pmatrix} + \begin{pmatrix} 2 & -2 \end{pmatrix}$$

Voici ci-dessous le code de la fonction **'Grad_MatSci'** en Scilab, qui met en œuvre l'algorithme de descente de gradient et retourne à chaque itération le nouveau point, la valeur de la fonction f associée à ce point, ainsi que la norme du gradient.

Listing 1: Grad_Mat.sci

```

function [dN,dNQ,niter,L_f] = Grad_Mat(c,Q,eps,MaxIter,verbose,
    delt0)
// r sout le syst me lin aire Q*dN+b=0 par la descente du
    gradient simple.
// on pr sume que Q est la matrice hessienne et c le gradient d'
    une fonction
//    minimiser.
//
//
// Entr e:
```

```
//    c(1,n), Q(n,n) param tres de la fonction quadratique
   minimiser
//         q(delt) = 0.5*delt'*Q*delt + c*delt
//         nabla q(delt) = delt'*Q + c
//    eps tol rance d'arr t
//    MaxIter  nombre maximum d'it rations
//    verbose  imprime les it rations (>0) ou non (<=0)
//    delt0(n,1)  point de d part des it rations
//
// Sortie:
//    dN: la solution  :dN*Q+b=0
//    niter: nombre total d'it rations
//    L_f: liste des valeurs de fonctions au fil des it rations
//

  niter = 0;
  L_f = [];
  [bidon,dim] = size(c)

  delt = delt0;
  deltQ = delt'*Q;
  nablaq = c + deltQ;

  pad = '';
  if verbose >0,
    s='   ';
    for i=1:verbose,
      pad = pad + s
    end
    mprintf("%s iter    q(delt)   ||nabla q(delt)||\n",pad)
    mprintf("%s %3d  %10.7f  %10.7e\n",pad,niter, (c*delt + 0.5*
       deltQ*delt), ...
            norm(nablaq))
  end

  norm2nablaq = nablaq*nablaq';
  pasprecis = norm2nablaq>eps^2;

  while pasprecis & (niter < MaxIter)
    niter = niter + 1;
    p = -nablaq';
    pQ = p'*Q;
    pQp = pQ*p

    theta=norm2nablaq/(pQp);
    if theta<0.0
      warning("Q not positive ")
    end

    delt = delt + theta*p;
```

```
    deltQ = deltQ + theta*pQ;
    nablaq = nablaq + theta*pQ;

    norm2nablaq = nablaq*nablaq'
    pasprecis = norm2nablaq>eps^2;
    if verbose>0,
      mprintf("%s %3d  %10.7f  %10.7e\n",pad,niter, (c*delt + 0.5*
         deltQ*delt), ...
                norm(nablaq))
    end
    L_f(niter) = c*delt + 0.5*deltQ*delt;
  end
  dN = delt;
  dNQ = deltQ;
endfunction
```

L'exécution de cette fonction se fait dans le fichier `test1.sce`. Nous lui passons la matrice H, le vecteur b, ainsi qu'un vecteur initial et le nombre maximal d'itérations. De plus, nous spécifions uniquement une tolérance ϵ comme critère d'arrêt.

Listing 2: test1.sce

```



// reproduit les calculs du tableau de  f(x,y)=2x^2-2xy+y^2+2x-2y

H=[4 -2;-2 2];
b=[2 -2];
n=2;
delt0=[10;5];

xstar = -H\b';  // Solution optimale

verbose = 1;
MaxIter = 20;
exec ("Grad_Mat.sci",0);

[dNG,dNQ,ngc,L_f] = Grad_Mat(b,H,1e-8,MaxIter,verbose,delt0);
```

Le fichier d'exécution du programme `test1.sce` affiche les résultats suivants :

```
-->exec('/home/asus/TP2_stochastique/TP2_Etud(2)/TP2_Etud/test1.sce', -1)
   iter    q(delt)   ||nabla q(delt)||
     0  135.0000000   3.4176015e+01
     1  19.7783784  5.7267917e+00
     2   2.1745662  5.2214865e+00
     3  -0.5149828  8.7495180e-01
     4  -0.9258980  7.9775017e-01
     5  -0.9886785  1.3367706e-01
     6  -0.9982703  1.2188202e-01
     7  -0.9997357  2.0423474e-02
     8  -0.9999596  1.8621403e-02
     9  -0.9999938  3.1203431e-03
    10  -0.9999991  2.8450187e-03
    11  -0.9999999  4.7673287e-04
    12  -1.0000000  4.3466821e-04
    13  -1.0000000  7.2836294e-05
    14  -1.0000000  6.6409562e-05
    15  -1.0000000  1.1128089e-05
    16  -1.0000000  1.0146199e-05
    17  -1.0000000  1.7001738e-06
    18  -1.0000000  1.5501585e-06
    19  -1.0000000  2.5975629e-07
    20  -1.0000000  2.3683661e-07

-->
```

Fig. 1: Évolution de la fonction objectif et de la norme du gradient avec l'algorithme de descente de gradient simple

2.2 Gradient conjugué linéaire avec matrice

Dans cette section, nous allons implémenter l'algorithme du gradient conjugué à l'aide de la fonction `GC_Mat.sci`. Cette fonction retourne, à chaque itération, la valeur de f dans le nouveau point, la norme du gradient de f dans ce point, ainsi que le pas utilisé.

Listing 3: GC_Mat.sci

```
function [dN, dNQ, niter, L_f] = GC_Mat(c, Q, eps, MaxIter, verbose
    , delt0)
    // Entr e:
    //    c(1,n), Q(n,n) param tres de la fonction quadratique
       minimiser
    //        q(delt) = 0.5*delt'*Q*delt + c*delt
    //        nabla q(delt) = delt'*Q + c
    //    eps tol rance d'arr t
    //    MaxIter  nombre maximum d'it rations
    //    verbose  imprime les it rations (>0) ou non (<=0)
    //    delt0(n,1)  point de d part des it rations
    //
    // Sortie:
    //    dN: la solution  :dN*Q+b=0
    //    niter: nombre total d'it rations
    //    L_f: liste des valeurs de fonctions au fil des
       it rations

    // init
```

```
    niter = 0;
    L_f = [];
    [bidon,dim] = size(c);

    // Let's compute grad_Q
    delt = delt0;
    deltQ = delt'*Q;
    nablaq = c + deltQ;

    // Let's display infos
    pad = '';
    if verbose > 0,
        s='    ';
        for i=1:verbose,
            pad = pad + s;
        end
        mprintf("%s iter    q(delt)  ||nabla q(delt)||  theta\n",
           pad);
        mprintf("%s %3d  %10.7f    %10.7e   %10.7f\n", pad, niter,
           (c*delt + 0.5*deltQ*delt), ...
            norm(nablaq), 0);
    end

    // Let's compute ||  q  (x)||
    norm2nablaq = nablaq*nablaq';
    pasprecis = norm2nablaq > eps^2;

    // Let's compute d_0
    d = -nablaq';
    b = 0;

    while pasprecis & (niter < MaxIter)
        // d_k
        d = -nablaq' + (b*d);
        dQ = d'*Q;
        dQd = dQ*d;

        // theta_k
        theta = (-nablaq*d) / dQd;
        if theta < 0.0
            warning("Q not positive ");
        end

        // delt_k+1
        delt = delt + theta*d;

        // update nabla_q, beta, deltQ
        nablaq = nablaq + theta*dQ;
        b = (nablaq*Q*d) / dQd;
        deltQ = deltQ + theta*dQ;
```

```
        // update stop conditions
        norm2nablaq = nablaq*nablaq';
        pasprecis = norm2nablaq > eps^2;
        niter = niter + 1;

        if verbose > 0,
            mprintf("%s %3d  %10.7f    %10.7e   %10.7f\n", pad, niter, (c*delt + 0.5*deltQ*delt), ...
            norm(nablaq), theta);
        end
        L_f(niter) = c*delt + 0.5*deltQ*delt;
    end
    dN = delt;
    dNQ = deltQ;
endfunction
```

On fait l'appel à la fonction `GC_Mat.sci` dans le fichier `test12.sce`, en lui passant comme arguments la matrice H, le vecteur b et le vecteur initial, ainsi que la seule tolérance ϵ.

Listing 4: test12.sce

```
// reproduit les calculs du tableau de f(x,y)=2x^2-2xy+y^2+2x-2y

H = [4 -2; -2 2];
b = [2 -2];
n = 2;
delt0 = [10; 5];

xstar = -H\b';  // Solution optimale

verbose = 1;
MaxIter = 20;
exec("GC_Mat.sci", 0);

[dNG, dNQ, ngc, L_f] = GC_Mat(b, H, 1e-8, MaxIter, verbose, delt0);
```

L'exécution de ce programme (`test12.sce`) donne les résultats suivants :

```
-->exec('/home/asus/TP2_stochastique/TP2_Etud(2)/TP2_Etud/GC_Mat.sci', -1)

-->exec('/home/asus/TP2_stochastique/TP2_Etud(2)/TP2_Etud/test12.sce', -1)
   iter    q(delt)   ||nabla q(delt)||   theta
     0  135.0000000     3.4176015e+01    0.0000000
     1   19.7783784     5.7267917e+00    0.1972973
     2   -1.0000000     2.3914936e-15    1.2671233

-->
```

Fig. 2: Évolution de la fonction objectif et de la norme du gradient avec l'algorithme de descente Gradient conjugué

En appelant les deux fonctions `GC_Mat.sci` et `Gard_Mat.sci` dans le fichier `exemple12.sci`, cette fois en changeant les dimensions à $n = 15$, nous commençons par générer une matrice aléatoire symétrique définie positive de taille $n \times n$. Il est important de noter que la matrice générée H vérifie que toutes ses valeurs propres sont triples, ce qui signifie qu'il y a en tout 5 valeurs propres. Nous générons également un vecteur aléatoire b de taille n. Nous passons ensuite ces deux fonctions en argument, ainsi que le nombre maximal d'itérations = 20, la seule tolérance et le vecteur initial.

Listing 5: exemple12.sce

```
// G n rateur de probl mes

n = 15;

// valeurs propres extr mes
lambda_1 = 1;
lambda_n = 20;
range = lambda_n-lambda_1;

lambda = lambda_1:range/(n/3-1):lambda_n
// Matrice diagonale avec les valeurs propres  tales  entre lambda_1 et
// lambda_n r p t s trois fois

Lambda = diag([lambda lambda lambda]);

// G n rons une matrice de rotation "al atoire".
M=rand(n,n);
[Q,R] = qr(M);

// H est une rotation de la matrice diagonale Lambda
H = Q*Lambda*Q';

// un vecteur b al atoire
b = rand(1,n);
xstar = -H\b';  // Solution optimale
delt0=zeros(n,1)
verbose = 1;
MaxIter = 20;
disp('la taille de la matrice Q est :', size(H));
disp( 'Gradient conjugu ')
exec("GC_Mat.sci", 0);

[dNG, dNQ, ngc, L_f] = GC_Mat(b, H, 1e-8, MaxIter, verbose, delt0);
disp('Gradient Simple')
exec("Grad_Mat.sci", 0);

[dNG, dNQ, ngc, L_f] = Grad_Mat(b, H, 1e-8, MaxIter, verbose, delt0);
```

Ce programme donne les résultats suivants :

```
"Gradient conjugué"
  iter     q(delt)    ||nabla q(delt)||   theta
    0    0.0000000     1.8409154e+00     0.0000000
    1   -0.2314451     1.7240281e+00     0.1365873
    2   -0.5187774     1.6286601e+00     0.1933418
    3   -0.7156395     1.2555373e+00     0.1484332
    4   -0.8202909     4.8273779e-01     0.1327748
    5   -0.8324486     1.6425443e-15     0.1043422

"Gradient Simple"
  iter     q(delt)    ||nabla q(delt)||
    0    0.0000000   1.8409154e+00
    1   -0.2314451   1.7240281e+00
    2   -0.3596343   1.2520271e+00
    3   -0.4499414   1.3189869e+00
    4   -0.5218387   1.0117364e+00
    5   -0.5796844   1.0685819e+00
    6   -0.6267141   8.2326575e-01
    7   -0.6649743   8.6964872e-01
    8   -0.6961178   6.7016201e-01
    9   -0.7214691   7.0792445e-01
   10   -0.7421063   5.4554203e-01
   11   -0.7589058   5.7628257e-01
   12   -0.7725814   4.4409621e-01
   13   -0.7837139   4.6912043e-01
   14   -0.7927764   3.6151468e-01
   15   -0.8001536   3.8188555e-01
   16   -0.8061590   2.9428953e-01
   17   -0.8110476   3.1087235e-01
   18   -0.8150272   2.3956518e-01
   19   -0.8182668   2.5306435e-01
   20   -0.8209040   1.9501705e-01
```

Fig. 3: Comparaison entre l'algorithme du gradient conjugué et l'algorithme du gradient simple

Interprétation:
Suite à la résolution du problème de minimisation de $f(x, y)$ pour $n = 2$, nous remarquons une nette différence de performance entre l'algorithme du gradient simple et celui du gradient conjugué. En effet, tandis que le gradient simple a nécessité 20 itérations pour se rapprocher de la solution, le gradient conjugué a atteint cette solution en seulement 2 itérations. Cette disparité est encore plus prononcée lorsque $n = 15$: le gradient conjugué converge vers la solution après seulement 5 itérations, tandis que le gradient simple reste significativement éloigné de la solution même après 20 itérations. Ces observations soulignent l'efficacité remarquable du gradient conjugué.

2.3 Gradient conjugué linéaire sans matrice

Dans cette étape, il s'agit de remplacer toutes les occurrences de l'utilisation directe de la matrice par un appel à une fonction $Hv(x, v)$. Actuellement, la variable x est superflue car la matrice hessienne de la fonction q est constante. Pour tester cette approche, on exécute le fichier "test2.sce", où la variante est codée dans "`GC_Hv.sci`".

Le code Scilab de la fonction `GC_Hv.sci` est le suivant :

Listing 6: GC_Hv.sci

```
/function [dN, dNQ, niter, L_f] = GC_Hv(c, Hv, x, eps, MaxIter,
   verbose, delt0)
```

```
// Entr e:
//    c(1,n), Q(n,n) param tres de la fonction quadratique minimiser
//         q(delt) = 0.5*delt'*Q*delt + c*delt
//         nabla q(delt) = delt'*Q + c
//    eps tol rance d'arr t
//    MaxIter  nombre maximum d'it rations
//    verbose  imprime les it rations (>0) ou non (<=0)
//    delt0(n,1)  point de d part des it rations
//
// Sortie:
//    dN: la solution  :dN*Q+b=0
//    niter: nombre total d'it rations
//    L_f: liste des valeurs de fonctions au fil des it rations

  // init
  niter = 0;
  L_f = [];
  [bidon,dim] = size(c)

  // Let's compute grad_Q
  delt = delt0;
  deltQ = Hv(x, delt);
  nablaq = c + deltQ;

  // Let's display infos
  pad = '';
  if verbose >0,
    s='   ';
    for i=1:verbose,
      pad = pad + s
    end
    mprintf("\n%s iter    q(delt)  ||nabla q(delt)||  theta\n",pad)
    mprintf("%s %3d  %10.7f    %10.7e   %10.7f\n", pad, niter, (c*delt + 0.5*deltQ*delt), ...
        norm(nablaq), 0)
  end

  // Let's compute || q (x)||
  norm2nablaq = nablaq*nablaq';
  pasprecis = norm2nablaq>eps^2;

  // Let's compute d_0
  p = -nablaq'
  b = 0

  while pasprecis & (niter < MaxIter)
    // d_k
    p = -nablaq' + (b*p);
    pQ = Hv(x, p);
```

```
    pQp = pQ*p;

    // theta_k
    theta=norm2nablaq/pQp;
    if theta<0.0
      warning("Q not positive ")
    end

    // delt_k+1
    delt = delt + theta*p;

    // update nabla_q, beta, deltQ
    nablaq = nablaq + theta*pQ;
    nablaq_T = nablaq'

    nablaq_Q = Hv(x, nablaq_T)
    b = nablaq_Q*p / pQp
    deltQ = deltQ + theta*pQ;

    // update stop conditions
    norm2nablaq = nablaq*nablaq';
    pasprecis = norm2nablaq>eps^2;
    niter = niter + 1;

    if verbose>0,
     mprintf("%s %3d  %10.7f    %10.7e   %10.7f\n", pad, niter, (c*delt + 0.5*deltQ*delt), ...
        norm(nablaq), theta)
    end
    L_f(niter) = c*delt + 0.5*deltQ*delt;
  end
  dN = delt;
  dNQ = deltQ;
endfunction
```

Dans le fichier "`test2.sce`", nous appelons les fonctions `Hv_GC.sci` et `GC_Mat.sci`.

Listing 7: test2.sce

```
// G n rateur de probl mes

n = 15;

// valeurs propres extr mes
lambda_1 = 1;
lambda_n = 20;
range = lambda_n-lambda_1;

lambda = lambda_1:range/(n/3-1):lambda_n
// Matrice diagonale avec les valeurs propres   tales   entre lambda_1 et
// lambda_n r p t s trois fois
```

```

Lambda = diag([lambda lambda lambda]);

// G n rons une matrice de rotation "al atoire".
M=rand(n,n);
[Q,R] = qr(M);

// H est une rotation de la matrice diagonale Lambda
H = Q*Lambda*Q';

// un vecteur b al atoire
b = rand(1,n);
delt0=zeros(n,1);


verbose = 1;
MaxIter = 20;

exec ("GC_Mat.sci",0);

[dNCGMat,dNQ,ngc,L_fGC] = GC_Mat(b,H,1e-8,MaxIter,verbose,delt0);



function [Hv] = Hv(x,v)
  Hv = v'*H;
endfunction

exec ("GC_Hv.sci",0);

x = zeros(1,n);

[dNCGHv,dNQ,ngc,L_fGC] = GC_Hv(b,Hv,x,1e-8,MaxIter,verbose,delt0);

norm(dNCGMat - dNCGHv) // devrait  tre  exactement 0
```

L'exécution de ce test donne les résultats suivants :

```
-->exec('/home/asus/TP2_stochastique/TP2_Etud(2)/TP2_Etud/test2.sce',
    iter     q(delt)   ||nabla q(delt)||   theta
      0    0.0000000     2.0106758e+00     0.0000000
      1   -0.5304557     3.0562035e+00     0.2624189
      2   -1.2954099     2.1039992e+00     0.1637951
      3   -1.5413387     9.7541282e-01     0.1111087
      4   -1.5932970     3.5179655e-01     0.1092215
      5   -1.5997393     1.1900254e-15     0.1041096

    iter     q(delt)   ||nabla q(delt)||   theta
      0    0.0000000     2.0106758e+00     0.0000000
      1   -0.5304557     3.0562035e+00     0.2624189
      2   -1.2954099     2.1039992e+00     0.1637951
      3   -1.5413387     9.7541282e-01     0.1111087
      4   -1.5932970     3.5179655e-01     0.1092215
      5   -1.5997393     1.2858435e-15     0.1041096

-->
```

Fig. 4: Évolution des résultats de l'exécution du test

Interperêtation
L'interprétation révèle que la fonction `Hv` calcule efficacement le produit entre un vecteur et une matrice. Tant `GC_Mat.sci` que `Hv_GC.sci` effectuent essentiellement la même tâche. Cependant, `Hv_GC.sci` présente l'avantage de ne pas nécessiter la matrice Q comme argument.

3 Région de confiance

3.1 Adaptation de GC_Hv

Dans cette section, nous aborderons l'adaptation de l'algorithme du gradient conjugué linéaire pour son utilisation au sein de l'algorithme de région de confiance. Alors que dans l'implémentation initiale, la matrice sous-jacente H était supposée définie positive, ce n'est plus nécessairement le cas dans le contexte de la région de confiance. Cette adaptation requiert deux modifications principales :

1. Calcul d'un pas de déplacement maximal pour rester dans la région de confiance. Le prochain point doit satisfaire une contrainte de la forme $k\delta + \theta p \leq \Delta$, où δ est la distance de déplacement, p est la direction de recherche, θ est un paramètre ajustable, et Δ est le rayon de la région de confiance. Nous devons déterminer la plus grande valeur de θ assurant cette condition, appelée θ_{Max}.

2. Détection d'une direction de courbure négative. Nous vérifions si le produit de la direction p par la matrice hessienne H est négatif ($p^T Hp < 0$). Cette détection est directement liée au signe de θ.

Si la valeur calculée de θ dans l'algorithme du gradient conjugué est trop grande (supérieure à θ_{Max}) ou négative, nous devons ajuster θ à θ_{Max} et terminer le calcul du gradient conjugué. Pour évaluer l'efficacité de cette adaptation, nous utiliserons le fichier "test3.sce", qui suppose que votre variante est codée dans "GC_TR.sci". Trois instances de problèmes seront résolues, où la taille de la région de confiance sera ajustée pour valider que votre implantation semble fournir des résultats corrects.
Le code Scillab de la fonction `GC_TR.sci` est suivan:

Listing 8: GC_TR.sci

```

function [dN, dNQ, niter, L_f] = GC_TR(c, Hv,x, eps, MaxIter, Delta
    ,verbose)
    // Entr e:
    //    c(1,n), Q(n,n) param tres de la fonction quadratique
       minimiser
    //          q(delt) = 0.5*delt'*Q*delt + c*delt
    //          nabla q(delt) = delt'*Q + c
    //    eps tol rance d'arr t
    //    MaxIter  nombre maximum d'it rations
    //    verbose  imprime les it rations (>0) ou non (<=0)
    //    delt0(n,1)  point de d part des it rations
    //
    // Sortie:
    //    dN: la solution  :dN*Q+b=0
    //    niter: nombre total d'it rations
    //    L_f: liste des valeurs de fonctions au fil des
       it rations

    // init
    niter = 0;
    L_f = [];
    [bidon,dim] = size(c);
    n = length(x)
    delt = zeros(n, 1);
    deltQ =Hv(x,delt);
    nablaq = c + deltQ;
    p= -nablaq';
    b = 0;

    // Let's display infos
    pad = '';
    if verbose > 0,
        s='   ';
        for i=1:verbose,
            pad = pad + s;
        end
        mprintf("%s iter    q(delt)   ||nabla q(delt)||  theta\n",
           pad);
        mprintf("%s %3d  %10.7f    %10.7e   %10.7f\n", pad, niter,
           (c*delt + 0.5*deltQ*delt), ...
            norm(nablaq), 0);
    end

    norm2nablaq = nablaq*nablaq';
    pasprecis = norm2nablaq>eps^2;
    sortie = %f

    // Boucle
```

```
    while pasprecis & (niter < MaxIter) & ~sortie
        // count iteration
        niter = niter + 1;

        // refresh variable
        pQ =Hv(x,p);
        pQp = pQ*p;

        // compute theta_max
        a = p'*p;
        b=2*delt'*p ;
        c = delt'*delt - Delta^2;
        disc= b^2-4*a*c;
        if disc < 0 then
            sortie = %t
        else
            theta1= (-b-sqrt(disc))/(2*a);
            theta2= (-b+sqrt(disc))/(2*a);
            theta=max(theta1, theta2);
        end

        // sortie
        sortie=Hv(x,p)*p <= 0

        if ~sortie then
            thetaGC=(-nablaq*p) / pQp;

            // BLOC INESISTANT DANS L'ALGO DE BASE
            if thetaGC<0 | thetaGC>theta then
                thetaGC = theta
            end
            // -----------------------------------

            sortie=thetaGC>theta;
            if ~sortie then
                delt=delt+thetaGC*p;

                // update nablaq & deltQ
                deltQ = deltQ + thetaGC*pQ;
                nablaq = nablaq + thetaGC*pQ;

                b = (Hv(x, nablaq')*p) / pQp;
                p= -nablaq' +b*p;
            end
        end

        // update stop conditions
        norm2nablaq = nablaq*nablaq';
        pasprecis = norm2nablaq > eps^2;
        if verbose > 0,
```

```
                mprintf("%s %3d  %10.7f    %10.7e   %10.7f\n", pad, niter, (c*delt + 0.5*deltQ*delt), ...
                norm(nablaq), theta);
            end
            // L_f(niter) = c*delt + 0.5*deltQ*delt;
        end

        //disp("sortie")
        /*if sortie then
            disp('sortie happens')
            delt = delt + (thetaGC*p)
        end*/
        dN = delt;
        dNQ = deltQ;
endfunction
```

Dans le fichier `test3.sce`, nous appelons la fonction `GC_TR.sci` pour résoudre trois instances de problèmes, où la taille de la région de confiance est ajustée pour valider que votre implantation semble fournir des résultats corrects.

Listing 9: test3.sce

```



 // G n rateur de probl mes

n = 15;

// valeurs propres extr mes
lambda_1 = 1;
lambda_n = 20;
range = lambda_n-lambda_1;

lambda = lambda_1:range/(n-1):lambda_n
// Matrice diagonale avec les valeurs propres   tales   entre lambda_1 et lambda_n
Lambda = diag(lambda);

// G n rons une matrice de rotation "al atoire".
M=rand(n,n);
[Q,R] = qr(M);

// H est une rotation de la matrice diagonale Lambda
H = Q*Lambda*Q';

// un vecteur b al atoire
b = rand(1,n);
delt0=zeros(n,1);


function [Hv] = Hv(x,v)
```

```
  Hv = v'*H;
endfunction

verbose =0;
MaxIter = 20;

x = zeros(1,n);


exec ("GC_Hv.sci",0);
[dNCGHv,dNQ,ngc,L_fGC] = GC_Hv(b,Hv,x,1e-8,MaxIter,verbose,delt0);

exec ("GC_TR.sci",0);
Delta = (1.01)*norm(dNCGHv);
[dNTR,dNQ,ngc] = GC_TR(b ,Hv, x, 1e-8, MaxIter, Delta, verbose);
n1=norm(dNCGHv-dNTR) // devrait tre exactement 0
disp('*******Test1*******')
disp(n1)


Delta = (0.99)*norm(dNCGHv);
[dNTR,dNQ,ngc] = GC_TR(b ,Hv, x, 1e-8, MaxIter, Delta, verbose);
n2=norm(dNCGHv-dNTR) // devrait tre proche de 0
disp('*******Test2*******')
disp(n2)

Delta = (0.5)*norm(dNCGHv);
[dNTR,dNQ,ngc] = GC_TR(b ,Hv, x, 1e-8, MaxIter, Delta, verbose);
n3=norm(dNCGHv-dNTR) // devrait tre loin de 0
disp('*******Test3*******')
disp(n3)
```

L'exécution de ce code produit les résultats suivants

```
"*******Test1*******"

 6.213D-16

"*******Test2*******"

 0.0201458

"*******Test3*******"

 1.0239112

-->
```

Fig. 5: Résultats de l'exécution du code test3.sce

Interprétation

Les résultats montrent que, à mesure que la taille de la région de confiance augmente, nous nous éloignons de la solution exacte. Cela suggère que l'ajustement de la région de confiance doit être effectué avec précaution pour obtenir des résultats précis

3.2 Test de l'algorithme

Listing 10: Rosenbrock.sci

```


dimension = 2;
function [val, df] = fdf(x)
  dim = size(x)(1);
  vec = zeros(1,dim);
  cost = 0;
  for i = 1:dim-1,
      cost = cost + (1 - x(i))^2 + 100*(x(i+1) - x(i)^2)^2;
      vec(i) = -2*(1 - x(i)) - 400*x(i)*(x(i+1) - x(i)^2);
  end
  vec(dim) = 200*(x(dim) - x(dim-1)^2);

  val = cost;
  df = vec;
endfunction

// fonction h retournant la hessienne  de f
function [dff] = Hv(x,v)
  dim = size(v)(1);
  G = zeros(dim,dim);
  for i = 1:dim-1,
      G(i,i) = 2 - 400*(x(i+1) - 3*x(i)^2);
      G(i,i+1) = -400*x(i);
  end
  G(dim,dim) = 200;
  G(dim,dim-1) = -400*x(dim-1);

  dff = v'*G;
endfunction

verbose = 1;
MaxIter = 20;
exec ("GC_TR.sci",-1);
c = zeros(1,dimension);
Delta = 0.9;
x0 = rand(1,dimension);
[dNTR,dNQ,ngc] = GC_TR(c, Hv,x0, 1e-8, MaxIter, Delta,verbose);
```

4 Conclusion

En conclusion, nous avons examiné l'adaptation de l'algorithme du gradient conjugué pour l'utilisation dans l'algorithme de région de confiance. Nous avons constaté que la modification des paramètres, notamment la taille de la région de confiance, peut avoir un impact significatif sur les performances de l'algorithme. En ajustant correctement la région de confiance. Cependant, une augmentation excessive de la taille de la région de confiance peut conduire à un éloignement de la solution exacte. Par conséquent, il est essentiel de trouver un équilibre approprié dans le réglage des paramètres pour garantir la convergence vers la solution optimale.