Ecole Polytechnique de Tunisie

Rapport de Projet TP Optimisation

Encadré par :

M. Walid Ben Romdhane

Mme. Yousra Gati

Réalisé par :

IHEB BEN SALEM

HAKIM JEMAA

MOHAMED KARIM ABID

2017/2018

I. Optimisation sans contrainte : La régression linéaire simple

1.1 Etude du problème

1) $E = \sum_{i=1}^{n}(a1 * xi + a0 - yi)^2$; *les xi et les yi sont des données* → E dépends de a0 et a1

2) $E(a) = \sum_{i=1}^{n}(yi - D(xi))^2 = \sum_{i=1}^{n}(a1 * xi + a0 - yi)^2$; *avec* $D: y = a1 * x + a0$

$$= < Ma - m, Ma - m >$$

$$= ||Ma - m||^2 \quad ; avec$$

$$M = \begin{matrix} 1 & x1 \\ 1 & x2 \\ \vdots & \\ 1 & xn \end{matrix} \qquad a = \begin{matrix} a0 \\ a1 \end{matrix} \qquad m = \begin{matrix} y1 \\ y2 \\ \vdots \\ yn \end{matrix}$$

3.a) **La fonction d'erreur :**

Code :

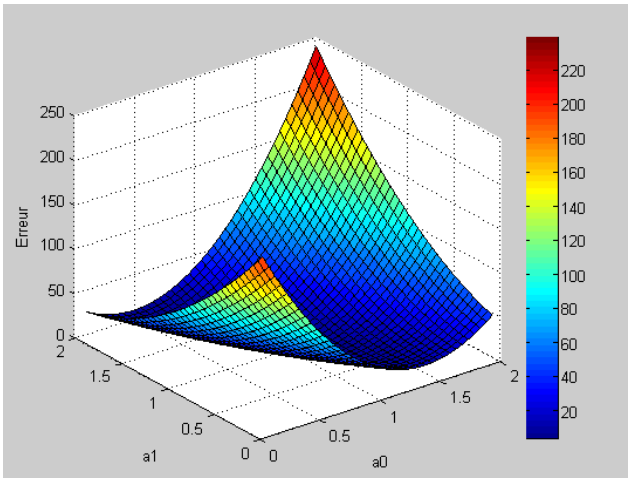
```
function[res]=E(a)
%Chargement des donnees
load Partie1;
%Obtntion du taille de x
[i0,i1]=size(x);
%Construction de M
M=ones(i0,2);
M(:,2)=x;
%Construction de m
m=y;
%calcul de E(a)
K=M*a-m;
res=dot(K,K);
```

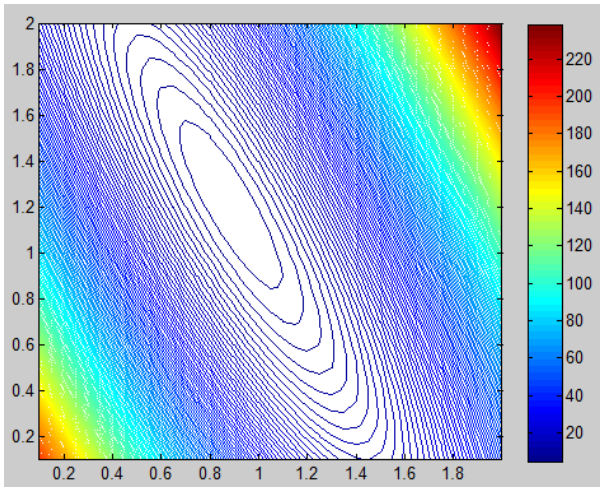
3.b) **Visualisation en 3D et du contour :**

Code :

```
function[]=visualiser()
%Construction d'une grille de points
a0=0.1:0.05:2;
a1=0.1:0.05:2;
[A0,A1]=meshgrid(a0,a1);
[index0,index]=(size(a0));
%Construction de la matrice des erreurs
%en chaque point de la grille
Z=zeros(index,index);
for i=1:index
    for j=1:index
        a=[A0(i,j),A1(i,j)]';
        Z(i,j)=E(a);
    end;
end;
%Generation de l'observation 3D
figure;
surf(A0,A1,Z);
zlabel('Erreur');
xlabel('a0');
ylabel('a1');
colorbar;
%Generation de l'observation du contour
figure;
contour(A0,A1,Z,200);
```

Exécution :





Interprétation :

Visualisation 3D : on peut voir que le minimum existe, car comme indique la barre des couleurs, le bleu foncé présente le 0.

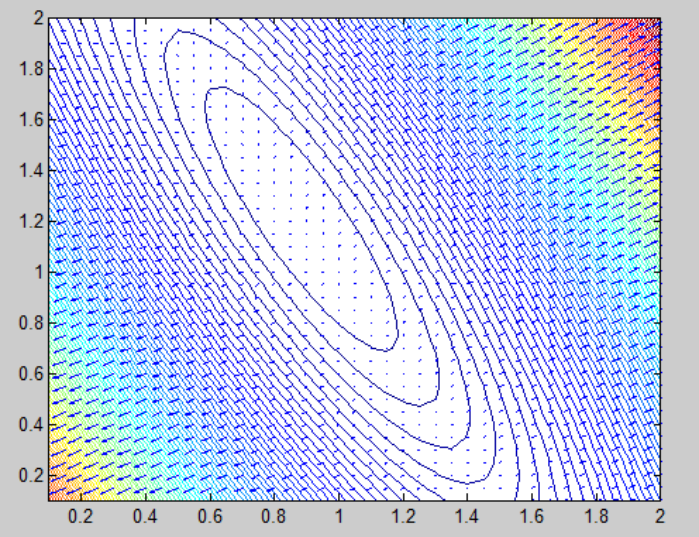
Contour **:** la visualisation du contour présente les lignes de niveaux, c'est-à-dire les points ayant la même valeur, et puisque les lignes de niveaux se rétrécissent et leur couleur tends vers le bleu, ainsi elles tendent vers 0, d'où vers le minimum.

3.c)Visualisation des gradients

Code :

```
function[]=visualiserContourGradiant()
%Construction d'une grille de points
a0=0.1:0.05:2;
a1=0.1:0.05:2;
[A0,A1]=meshgrid(a0,a1);
[index0,index]=(size(a0));
Z=zeros(index,index);
for i=1:index
    for j=1:index
        a=[A0(i,j),A1(i,j)]';
        Z(i,j)=E(a);
    end;
end;
%traçage du contour
contour(A0,A1,Z,100);
%calcul des gradiants
[px,py]=gradient(Z);
hold on;
%traçage du champ de gradiant
quiver(A0,A1,px,py);
hold off;
```

Exécution :



Interprétation :

Le champ de gradient est un champ sortant de la zone du rétrécissement des lignes de niveaux où

Les valeurs des gradients sont faibles et devient plus importants en sortant de cette zone.

Ainsi le minimum a un gradient nul.

D'autre part la fonction d'erreur est différentiable en a, convexe (d'hessienne positive), ainsi le minimum a un gradient nulle, on a ainsi l'existence et l'unicité de ce minimum d'après le graphique aussi, ainsi les résultats théoriques se conforment aves celles expérimentaux.

4) Autre forme du problème de minimisation

$$E(a) =< Ma - m, Ma - m >=< Ma, Ma > -2 < m, Ma > +< m, m >$$

$$= 2(1/2 < tMM, a > -< tMm, a >)+< m, m >$$

$$= 2\left(\frac{1}{2} < A, a > -< b, a >\right)+< m, m > \quad avec\ A = tMM\ ; b = tMm$$

$$= 2 * F(a)+< m, m > \quad avec\ F(a) = \left(\frac{1}{2} < A, a > -< b, a >\right)$$

Or <m,m> est une constante ,ainsi minimiser E revient à minimiser F.

Code :

Construction des matrices A et b :

```
function[A,b]=construire()
load Partie1;
%Construction de M et m
[i0,i1]=size(x);
M=ones(i0,2);
M(:,2)=x;
m=y;
%Construction de Aet b
A=M'*M;
b=M'*m;
```

La fonction F

```
function[res]=F(a)
[A,b]=construire();
res=0.5*dot(A*a,a)-dot(b,a);
```

5) gradient et Hessienne de f

$$F(a) = \left(\frac{1}{2} < A, a > -< b, a >\right)$$

$\rightarrow DF(a) = Aa - b$

$\rightarrow DDF(a) = A$

Code :

gradF(a)

```
function[res]=DF(a)
[A,b]=construire();
res=A*a-b;
```

HessF(a)

```
function[res]=DDF(a)
 [A,b]=construire();
res=A;
```

6) Le point minimisant F :

$\text{gradF(a)} = 0 \Leftrightarrow \text{A.x} = \text{b}\ \{\text{or A inversible } (\det(\text{A}) \neq 0)\} \Leftrightarrow x = A^{-1} * b$

Code et exécution :

```
>> inv(A)*b
```

```
ans =

    0.8844
    1.2083
```

II. Algorithme de résolution

1. Implémentation des algorithmes

1.a)Newton

Code :

```
function[sol,n]=NewtonP(a)
%Construction des variables initiales
%pour avoir la condition norm(y-x)>eps vérfié
%pour entre dans le boucle
x=a/2;
y=a;
%initialisation du compteur
n=0;
    while norm(y-x)>eps
     %incrementation du compteur
     n=n+1;

     x=y;
     y=y-inv(DDF(y))*DF(y);
    end
sol=x;
```

Test pour a= [3, 0.25] :

```
>> [sol,n]=NewtonP(a)

sol =

    0.8844
    1.2083


n =

     3
```

1.b) Gradient à pas fixe

Code :

```
function[sol,n]=GradFixeP(a)
%Construction des variables initiales
 %pour avoir la condition norm(y-x)>eps vérfié
%pour entrer dans le boucle
 x=a/2;
 y=a;
 %Initialisation du compteur
 n=0;
 %choix du pas
 d=0.01;
 %algorithme
    while norm(y-x)>eps
      %incrementation du compteur
      n=n+1;

      x=y;
      y=y-d*DF(y);
    end
sol=y;
```

Test pour a=[3 , 0.25] :

```
>> [sol,n]=GradFixeP(a)

sol =

    0.8844
    1.2083


n =

   480
```

1.c) Gradient à pas optimale

Code :

```
function[sol,n]=GradOptP(a)
%Construction des variables initiales
%pour avoir la condition norm(y-x)>eps vérfié
%pour entrer dans le boucle
x=a/2;
y=a;
%Construction de A et !b
[A,b]=construire();
%Initialisation du compteur
n=0;
%algorithme
    while norm(y-x)>eps
     %incrementation du compteur
     n=n+1;

     x=y;
     r=A*x-b;
     d=dot(r,r)/dot(A*r,r);
     y=y-d*DF(y);

    end
sol=x;
```

Test pour a= [3, 0.25] :

```
>> [sol,n]=GradOptP(a)

sol =

    0.8844
    1.2083


n =

    39
```

1.d) Gradient conjugué

Code :

```
function[res,n]=GrapConjugP(a)
%Construction des variables initiales
r1=DF(a);
d1=r1;
a1=a;
a0=a1/2;
%Construction de A et !b
[A,b]=construire();
%Initialisation du compteur
n=0;
%algorithme
while norm(a0-a1)>eps
    %incrementation du compteur
    n=n+1;
    a0=a1;
    d0=d1;
    r0=r1;

    p0=dot(r0,r0)/dot(A*d0,d0);

    a1=a0-p0*d0;

    r1=DF(a1);

    b0=dot(r1,r1)/dot(r0,r0);
    d1=r1+b0*d0;

end;
res=a1;
```

Test pour a= [3, 0.25] :

```
>> [sol,n]=GrapConjugP(a)

sol =

    0.8844
    1.2083


n =

     4
```

Interprétation :

On peut remarquer à partie cet exemple que l'ordre de la rapidité des algorithmes est comme suivant :

Newton> Gradient conjugué > Gradient à pas optimale > Gradient à pas fixe

2) Comparaison des algorithmes

2.a) trajectoire des itérations

-L'implémentation est similaire pour les 4 méthodes, on va juste la présenter la méthode de Newton.

-On va implémenter 4 méthodes :

1.NewtonPdonnees : qui retourne le vecteur a de chaque itération.

2.visualiserContour : trace les lignes de niveaux de F. (implémenté une seule fois)

3.TrajNewton : trace la trajectoire des itérations. (qui utilise NewtonPdonnees)

4.TrajNewtoneMultiple : trace la trajectoire des itérations de plusieurs exemples(qui utilise TrajNewton)

Code visualiserContour :

(le même dans 1.1 3. b)

```
function[]=visualiserContour()
a0=0.1:0.05:2;
a1=0.1:0.05:2;
[A0,A1]=meshgrid(a0,a1);
[index0,index]=(size(a0));
Z=zeros(index,index);
for i=1:index
    for j=1:index
        a=[A0(i,j),A1(i,j)]';
        Z(i,j)=E(a);
    end;
end;
%res=Z;
contour(A0,A1,Z,50);%,'ShowText','on');
```

Newton

Code NewtonMultiple :

```
function[]=TrajNewtonMultiple()
%On va visualiser 4 examples
%On divise la figuire sur 4 parties
subplot(2,2,1);
TrajNewton([.1,1.4]');
title('[0.1,1.4]');

subplot(2,2,2);
TrajNewton([1.7,.2]');
title('[1.7,0.2]');

subplot(2,2,3);
TrajNewton([1.98,1.8]');
title('[1.98,1.8]');

subplot(2,2,4);
TrajNewton([.2,.2]');
title('[0.2,0.2]');
```

Code TrajNewton :

```
function[]=TrajNewton(a)
visualiserContour();
hold on;
donnees=NewtonPdonnees(a);
plot(donnees(:,1),donnees(:,2),'r.-');
hold on;
```

Code NewtonPdonnees :

```
function[sol]=NewtonPdonnees(a)
x=a/2;
y=a;
n=0;
%initialisation la matrice initiale des donnees avec
%une taille assez grande qu'on va le régler eventuellement
%dans la matrice donneesRes qui est le resultat
donnees=zeros(700,2);

%ajout du vecteur de depart
donnees(1,:)=a;
%algorithme
    while norm(y-x)>eps
     n=n+1;
     x=y;
     y=y-inv(DDF(y))*DF(y);
     %ajout du vecteur calculé
     donnees(n+1,:)=y;
    end;
%%initialisation la matrice initiale de donnees
%de taille le nombre d'iteration
donneesRes=zeros(n+1,2);
%l'ajout des donnees
for i=1:n+1
    donneesRes(i,:)=donnees(i,:);
end;
sol=donneesRes;
```

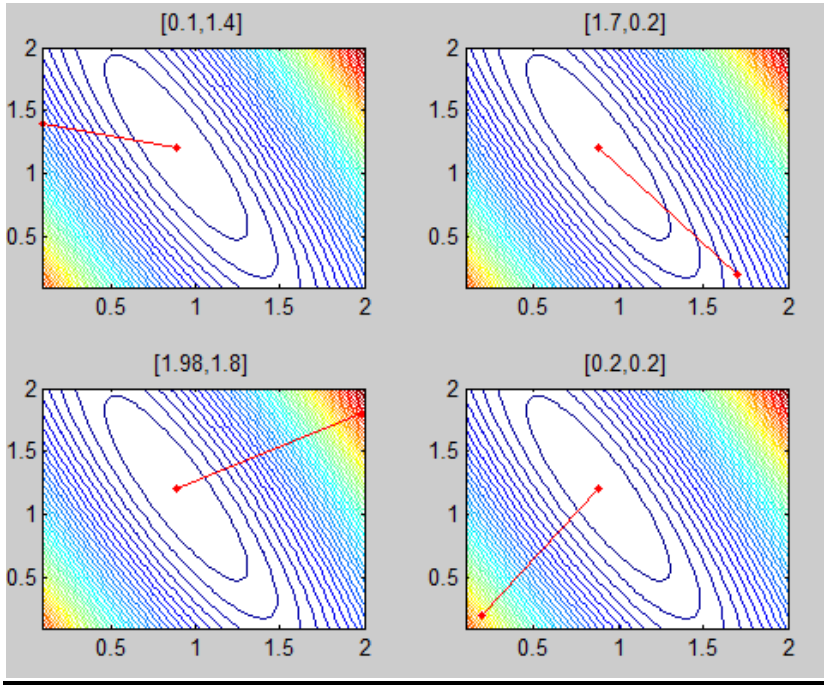
Visualisation des itérations sur 4 exemples de chaque méthode :

-Le choix des points:

-[0.1,1.4] et [1.7,0.2] ont une erreur faible

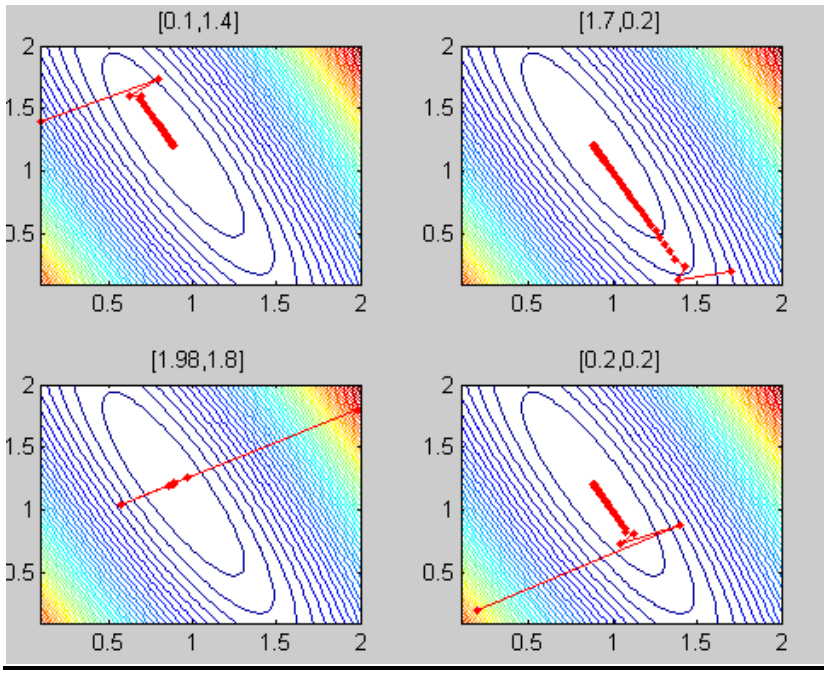
-[1.98,1.8] et [0.2,0.2] ont une erreur importante

1.Newton



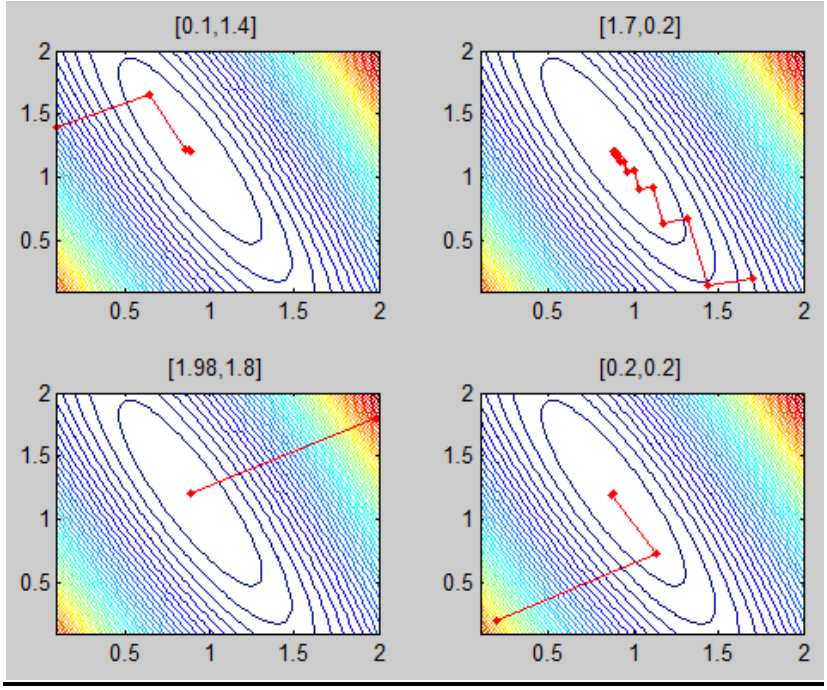
Commentaire : La méthode de Newton atteint le minimum rapidement en une seule itération.

2.Gradiant à pas fixe



Commentaire : La méthode gradient à mas fixe prends beaucoup d'itérations pour s'approcher peu à peu à la solution

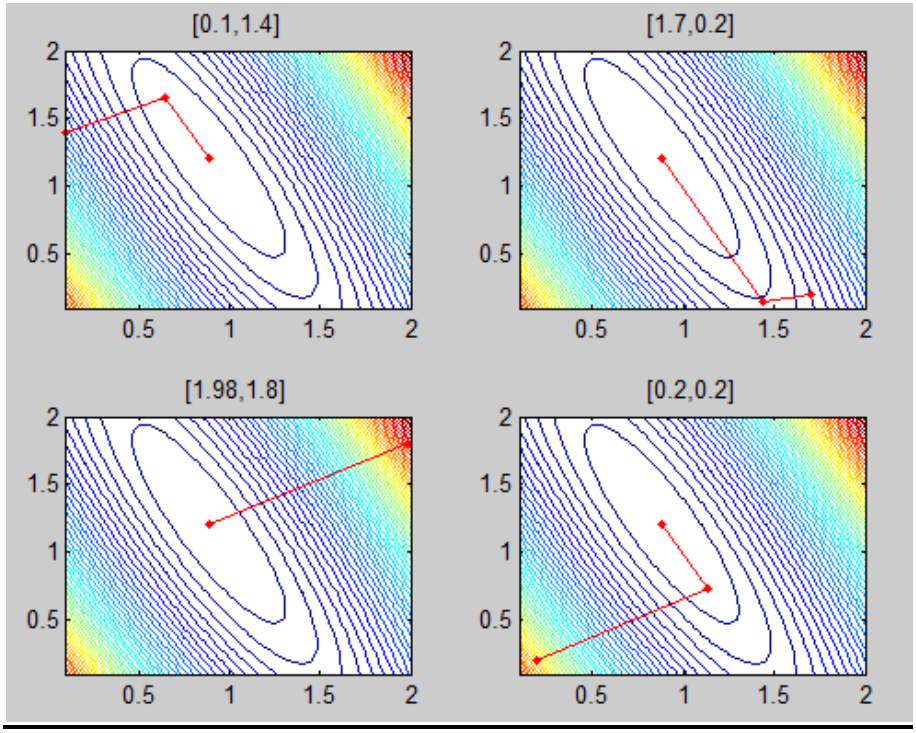
3.Gradiant à pas optimale :



Commentaire : La méthode gradient à pas optimale prends mois d'itérations pour s'approcher à la solution .

On remarque qu'elle tends vers la solutions plus rapidement pour les points d'erreur grande (qui se situent dans les lignes de niveaux rouges).

4.Gradiant conjugué :



Commentaire : La méthode du gradient conjugué atteint la solution en 2 itérations.

2.b) Les courbe du nombre d'itération n nécessaire pour obtenir une précision eps donnée en

fonction de -log10(eps).

-L'implémentation est similaire pour les 4 méthodes, on va juste la présenter pour la méthode de Newton.

-On va implémenter 3 méthodes :

1.CourbeErreurNewtondonnees : retourne une matrice qui contient des couples (erreur,nombre d'iteration) pour une erreur appartient à (10^-16,10^-1)

2.CourbeErreurNewton : trace la courbe des nombres d'itérations pour un vecteur a en fonction de –log10(erreur).

3.CourbeErreurNewtonMultipleTogetherTitle : trace la courbe de 4 points différents.

NB : On a changé les méthodes de résolution pour prendre aussi à part que le point d'initialisation , l'erreur de précision e : NeswtonP(a)→ NeswtonP(a,e)

Code CourbeErreurNewtondonnees :

```
function[sol]=CourbeErreurNewtondonnees(a)
%Construction de la matrice d'erreur
e=linspace(10^-16,10^-1);
%initialisation de la matrice des nombres d'iterations
nbr=zeros(1,100);
%la remplir
for i=1:100
    [sol,n]=NewtonPe(a,e(1,i));
    nbr(1,i)=n;
end;
sol=nbr;
```

Code : CourbeErreurNewton

```
function[]=CourbeErreurNewton(a)
e=linspace(10^-16,10^-1);
semilogx(-e,CourbeErreurNewtondonnees(a));
```

Code :CourbeErreurNewtonMultipleTogether

```
]function[]=CourbeErreurNewtonMultipleTogether()

 subplot(2,2,1);
 CourbeErreurNewton([.1,1.4]');
 title('[0.1,1.4]');
 subplot(2,2,2);
 CourbeErreurNewton([1.7,.2]');
 title('[1.7,0.2]');
 subplot(2,2,3);
 CourbeErreurNewton([1.98,1.8]');
 title('[1.98,1.8]');
 subplot(2,2,4);
 CourbeErreurNewton([.2,.2]');
 title('[0.2,0.2]');
```

Visualisations :

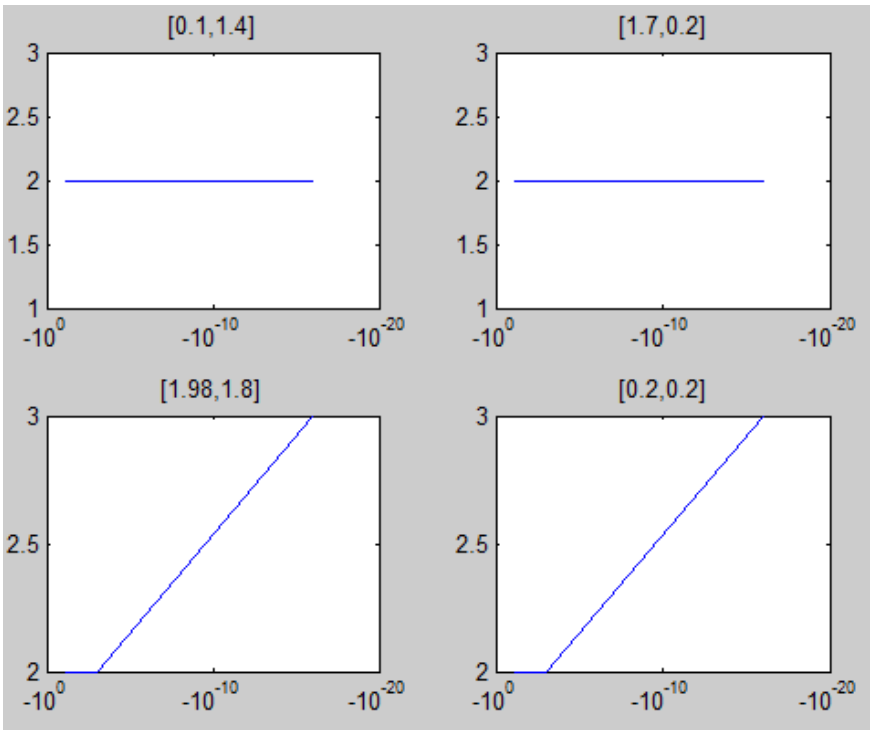
-Le choix des points :

-[0.1,1.4] et [1.7,0.2] ont une erreur faible

-[1.98,1.8] et [0.2,0.2] ont une erreur importante

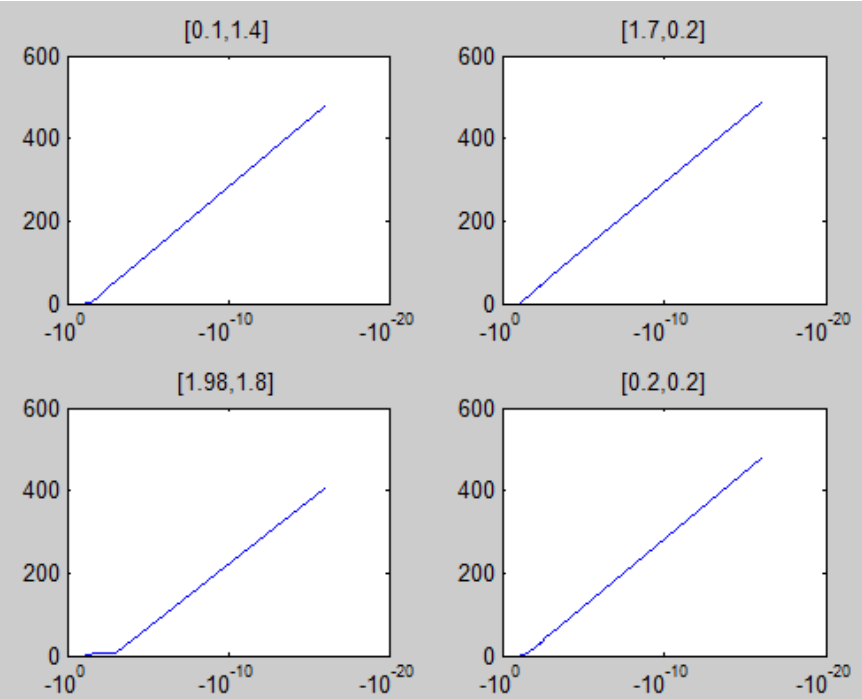
Commentaire général :plus la précision augmente , plus le nombre d'itérations augmente

a. Newton



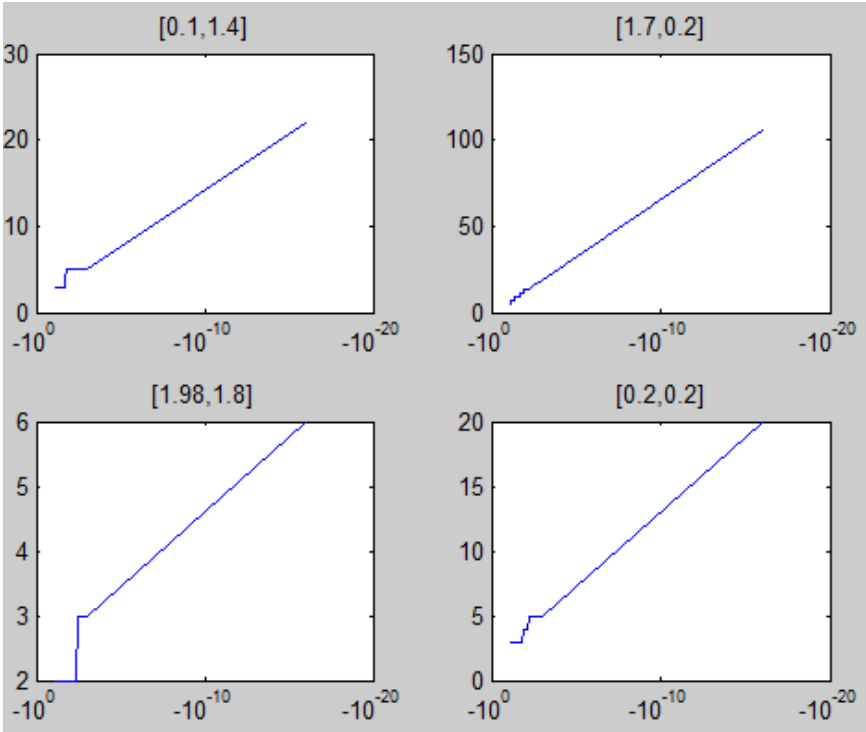
Commentaire : La méthode de Newton prend un nombre très faible d'itérations : entre 2 et 3 itérations même si la précision est importante (10.e-16)

b. Gradient pas fixe



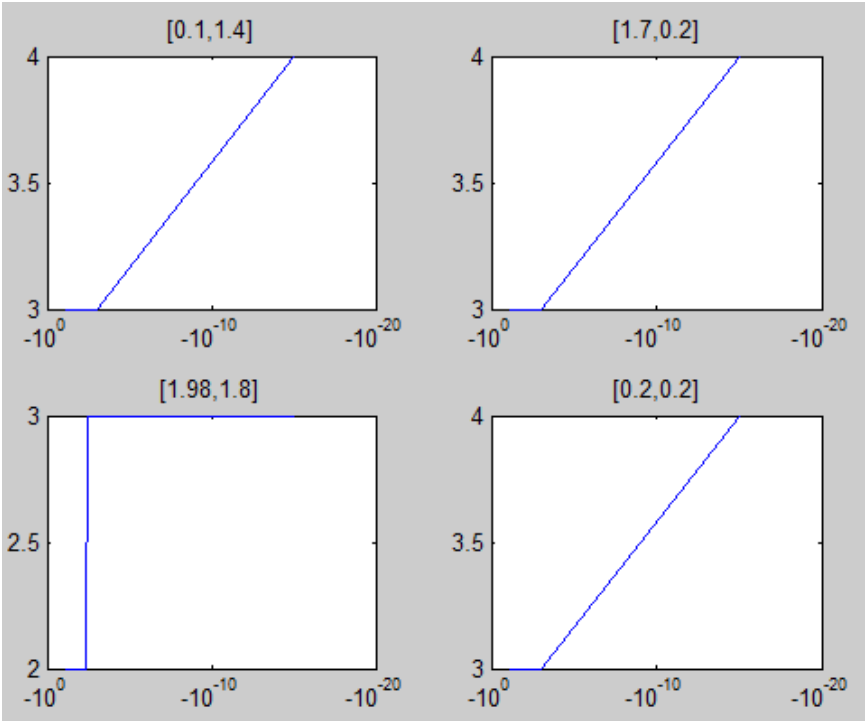
Commentaire : La méthode Gradient pas fixe prends un nombre assez important d'itérations(> 400 et 500 au moyenne) qui augmente en fonction de la précision.

a. Gradient pas optimale



Commentaire : La méthode gradient pas optimale prends un nombre inferieur à celui à pas fixe, qui peut atteint 6 , et peut être grand à 100 itérations par exemple.

b. Gradient conjugué



Commentaire : La méthode gradient conjugué prends un nombre d'itérations entre 2 et 4 quelque soit la précision (4 pour une grande précision)

Interprétation :

D'âpres ces deux questions on peut constater que :

1. Newton : tends rapidement vers la solution et prends au maximum 3 itérations
2. Gradient à pas fixe : prends un nombre important d'itérations et s'approche peu à peu
3. Gradient à pas optimale : prends un nombre moyen d'itérations, s'approche plus rapidement à la solution que la méthode à gradient fixe
4. Gradient conjugué : s'approche rapidement vers la solution, prends au maximum 4 solutions

Conclusion : Comparaison entre ces 4 algorithmes :

Newton >Gradient conjugué > Gradient à pas optimale > Gradient à pas fixe

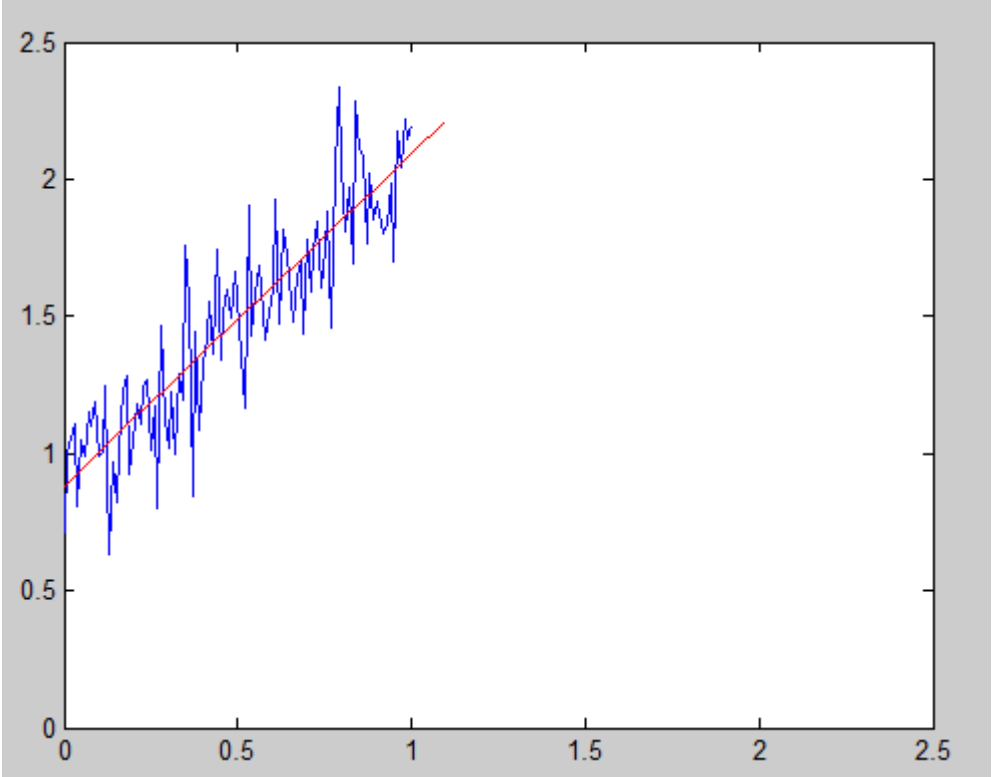
→Il est préférable d'utiliser la méthode de Newton ou celle du gradient conjugué pour un coût faible.

3. Visualisation du nuage de point et la droite de régression optimale sur un même graphique

Code :

```
function[]=plotesol()
load Partie1;
plot(x,y);
axis([0 2.5 0 2.5]);
hold on;
abs=0:.1:1.1;
%ord=1.2*abs+0.85;
ord=1.2083*abs+0.8844;
plot(abs,ord,'r-');
hold off;
```

Exécution :



2 Optimisation avec contrainte : la régression linéaire multiple

2.1 Etude du problème :

Dans le cas précédent, nous avons exprimé la variable y comme fonction linéaire d'une seule variable. Lorsque y s'exprime comme fonction linéaire de plusieurs variables, on parle de régression linéaire multiple. La variable$y$ est alors estimée par l'expression suivante :

$$\hat{y} = a_0 + a_1x_1 + a_2x_2 + ... + a_nx_n \quad (1)$$

On dit que $x_1, x_2, \ldots, x_n$ sont les variables indépendantes (ou explicatives) et que y est la variable dépendante (ou à expliquer). On note :

- $X = [x_0, x_1, x_2, \ldots, x_n]$ la matrice des données relatives aux variables explicatives contenant $n + 1$ vecteurs colonnes $[x_j] = [x_{i,j}]$. Remarquer que X commence avec un vecteur x_0 qui n'apparait pas dans l'équation 1. C'est un vecteur colonne rempli de 1.
- $y = [y_i]$ le vecteur colonne contenant les observations relatives à la variable à expliquer
- $a = [a_0, a_1, a_2, \ldots, a_n]^T$ le vecteur colonne contenant les coefficients

1) Ecrire $\widehat{Y}$ en fonction de X et de a

$\widehat{Y}$=Xa

2) Ecrire la fonction d'erreur quadratique E(a). Calculer son gradient et sa hessienne et montrer qu'elle est convexe.

E(a)=$\|Xa - y\|_2^2$. En faisant un calcul similaire à celui dans la partie 1:

E(a) = <Xa-y, Xa-y > => E(a) = <Xa,Xa>-2<Xa,y>-<y,y> => E(a)=< t(X)Xa, a> -2<a, t(X)y> -<y,y>

t(X) est la transposé de la matrice X

Donc $\nabla E(a) = 2Aa - 2b$ et hess(E)= $\nabla^2 E(a)$=2A avec A=t(X)X et b=t(X)y

<$\nabla E(x) - \nabla E(y)$,x-y> = <A(x-y),(x-y) >= <X(x-y),X(x-y)> >0 donc la fonction E est convexe

3) Pour des observations données X et y, déterminer la formule qui permet d'obtenir les paramètres qui minimisent l'erreur

$\nabla E(a) = 0$ => $2Aa - 2b=0$ =>t(X)Xa= t(X)y => $a=X^{-1}t(X)^{-1}$ (t(X)y)

4) Utiliser Matlab pour calculer ces paramètres avec la formule de la question précédente. Que remarquez-vous ?

```
>> inv(X'*X)*(X'*y)
Warning: Matrix is singular to working precision.

ans =

   Inf
   Inf
   Inf
   Inf
   Inf
   Inf
   Inf
```

Le vecteur a explosé avec ses composants qui tendent vers l'infini. Ceci est expliqué par le fait que la matrice A= t(X)X n'est pas inversible.

2.2 Méthode de pénalisation

Le problème constaté lors de la question précédente est appelé “explosion des coefficients”. Pour éviter ce problème, on pénalise la norme du vecteur a en ajoutant un terme à la fonction d’erreur. Le nouveau problème d’optimisation s’écrit alors :

$$\min_a \|Xa - y\|_2^2 + \lambda \|a\|_2^2 \qquad (2)$$

où λ est un réel positif.

1. Expliquer comment la pénalisation permet d'éviter l'explosion des coefficients.

On ajouter le terme : lambda*<a,a> .

Le problème devient Min ½<Aa,a>-<b,a> avec A=t(X)X-lambda*I(n+1) et b=t(X)y

Donc A devient inversible et on résout le problème rencontré dans 2.1 . Le minimum est en $a=A^{-1}b$

2. Parmi les méthodes utilisées dans la première partie, laquelle est la mieux adaptée pour résoudre le problème (2) ? Ecrire les fonctions Matlab qui permettent de l'implémenter

La méthode la plus adapté est celle de newton car la fonction qu'on cherche à minimiser et convexe.

(<∇E(x)–∇E(y),x-y> = <A(x-y),(x-y)> = <X(x-y),X(x-y)>+lambda*<x-y,x-y> >0 donc la fonction E est convexe)

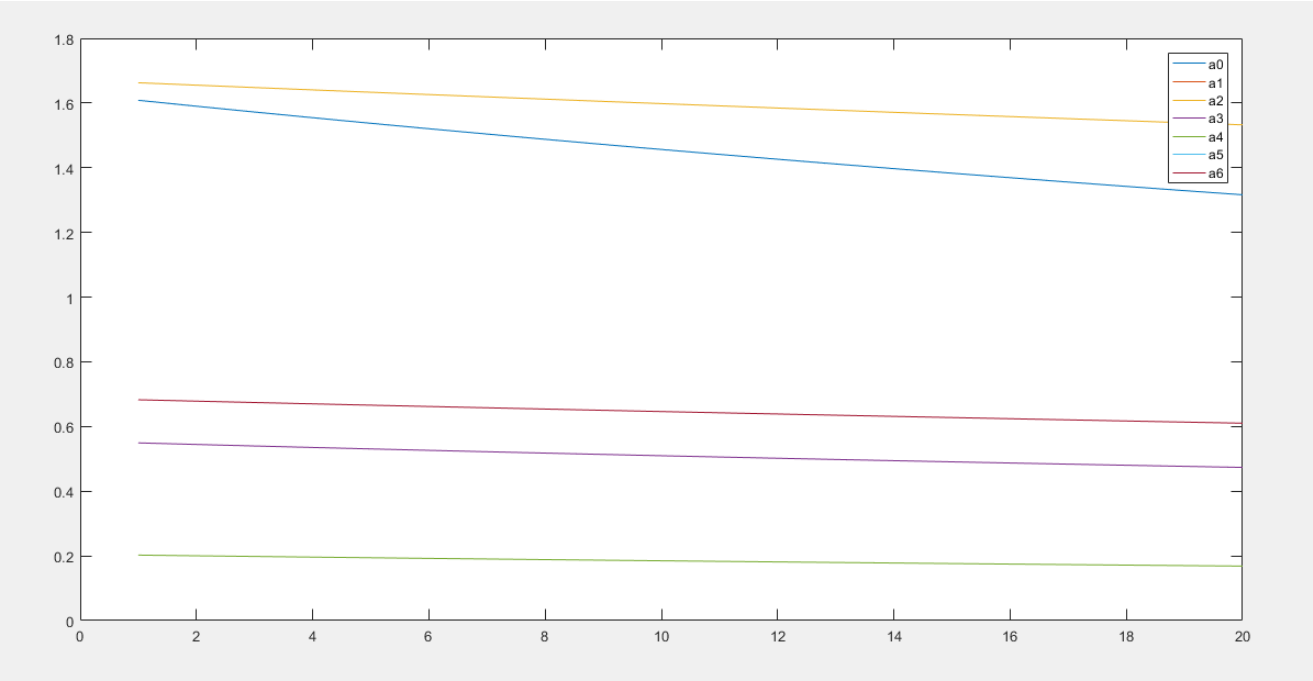
```
function a=penalisationNewton(lambda,x0,epsilon)
load partie2;
a0=x0+2; %pour assurer l'entrée dans la boucle while
a1=x0;
A=X'*X+lambda*eye(length(X'*X));
b=X'*y;
    while norm(a1-a0)>epsilon
     a0=a1;
     a1=a0-inv(A)*(A*a0-b);
     % car A est le gradient deuxieme de notre fonction et A*a-b est son gradient
    end
a=a1;
end
```

3. Tracer la variation des valeurs des coefficients en fonction de λ. Commenter.
4. Soit $\hat{y}_{opt}(\lambda)$ l'estimation de y obtenue en utilisant les paramètres optimaux pour une valeur donnée de λ. Tracer l'évolution des variables y et $\hat{y}_{opt}(\lambda)$ sur un même graphique en faisant varier la valeur de λ.

Code de la question 3 et 4:

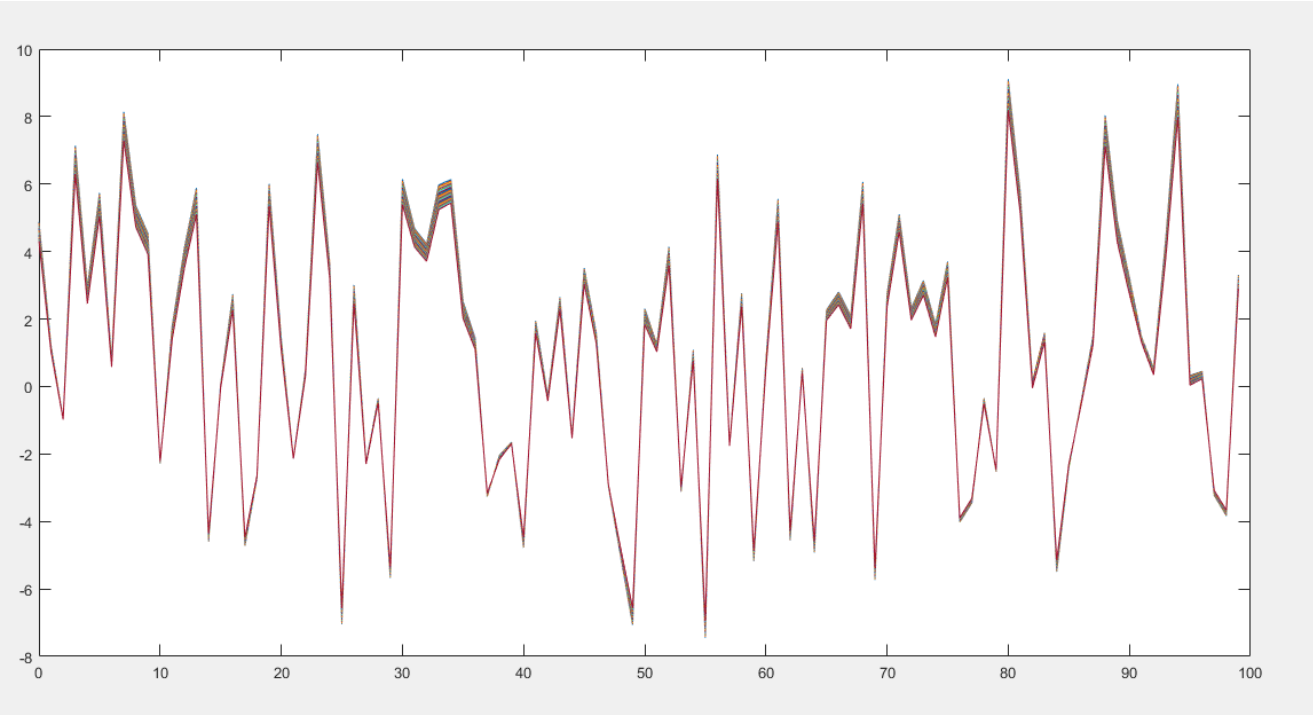
```
function TracagePenalisaton()
load partie2;
lambda=1:1:20;
indices=0:1:(length(y)-1);
for i=1:length(lambda)

    a(:,i)=penalisationNewton(lambda(i),[1;1;1;1;1;1;1],0.1);
    % a est la matrice conteant pour colonnes les vecteur a pour differents lambda
    Yopt(i,:)=X*a(:,i);% Yopt est la matrice contenant en chaque ligne un vecteur yopt pour un lambda donnée
end
ylabel('a');
xlabel('Parametre de pénalisation lambda');
plot(lambda,a) %tracage de a(lambda)
legend('a0','a1','a2','a3','a4','a5','a6')
figure;
ylabel('y');
xlabel('Parametre de pénalisation lambda');
plot(indices,y)
hold()
plot(indices,Yopt) %tracage de yopt(lambda)
end
```

3)

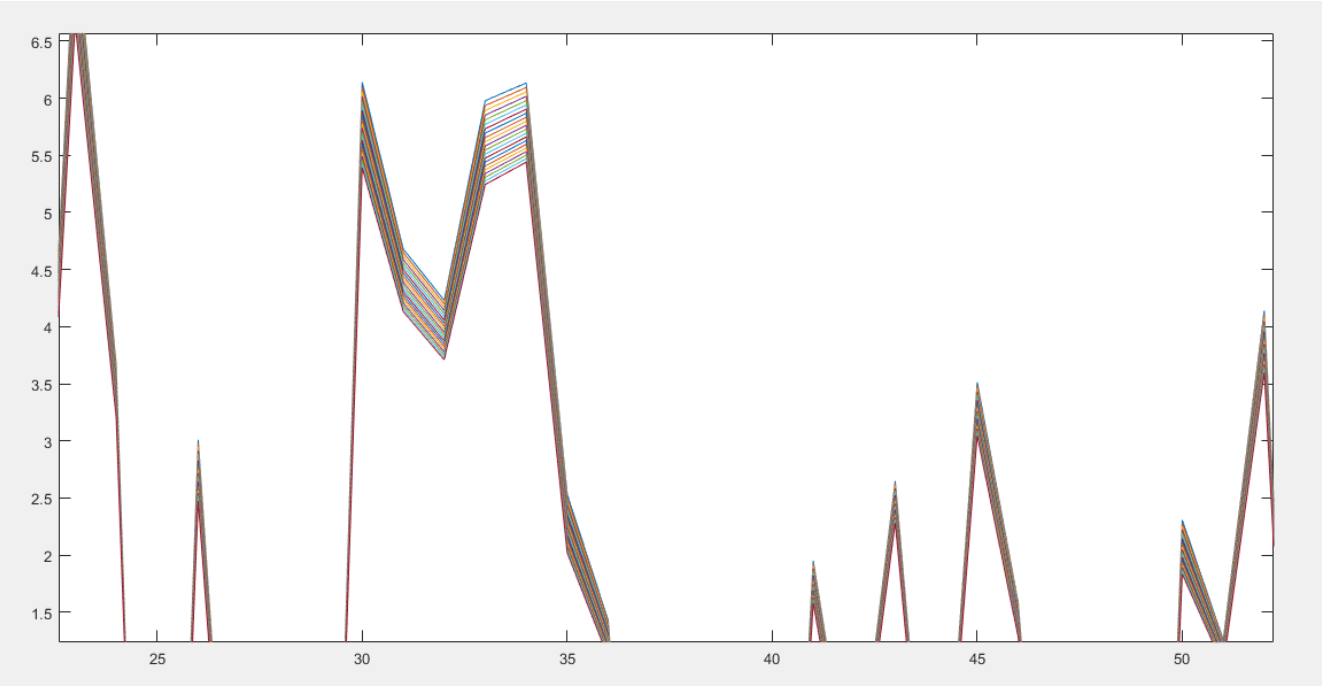


La figure de a1 et a2 sont confondu. De meme pour a5 et a6

4)



Lorsqu'on effectue un agrandissement :


6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
25
30
35
40
45
50

III) Méthode de gradient accéléré : Système de premier ordre

***) But de la partie**

Le but de cette partie est d'utiliser des algorithmes d'optimisation pour minimiser l'erreur entre les valeurs théoriques et pratiques et trouver les coefficients a et b pour réaliser la régression exponentielle et approcher au plus la solution.

1) Calcul du gradient

La fonction d'erreur E est la suivante :

$$E = \log\left(\sum_{i=1}^{n}(y_i - \hat{y}_i)^2\right)$$

avec :

y_i : la valeur mesurée de y à l'instant t_i

$\hat{y}_i$: la valeur estimée par le modèle $\hat{y}_i = a\left(1 - e^{bt_i}\right)$

C'est une fonction donc qui dépend de deux variables a et b

$$E(a,b) = \log(\sum_{i=1}^{n}(y_i - \mathrm{a} * (1 - e^{b*t_i})^2)$$

Le gradient de E nous donne un vecteur dont les deux composantes sont :

$$\frac{\partial E}{\partial a} = \frac{\sum_{i=1}^{n} -2 * (1 - e^{b*t_i}) * (\mathrm{yi} - \mathrm{a} * (1 - e^{b*t_i})}{\sum_{i=1}^{n}(y_i - \mathrm{a} * (1 - e^{b*t_i})^2)}$$

$$\frac{dE}{db} = \frac{\sum_{i=1}^{n} 2 * a * t_i * e^{b*t_i} * (y_i - \mathrm{a} * (1 - e^{b*t_i})}{\sum_{i=1}^{n}(y_i - \mathrm{a} * (1 - e^{b*t_i})^2)}$$

2) Implémentation

Cette question est une simple implémentation sur Matlab de la fonction **erreur.m** et de la fonction **grad_erreur.m** qui renvoie le gradient de la fonction erreur

```
function [E]=erreur(a,b,t,y)
somme=0;
for i=1:length(t)
    somme=somme+(y(i)-a.*(1-exp(b.*t(i))))^2;
end
E=log(somme);
end
```

Pour l'exécution de la fonction **erreur**, on peut soit faire rentrer une liste de valeurs des t et y ou bien on fait

Load partie3.mat et on exécute comme suit

```
>> load partie3.mat
>> erreur(2,5,t,y)

ans =

  101.8508
```

En ce qui concerne la fonction **grad_erreur**, L'exécution est la suivante

```
function [G]=grad_erreur(a,b,t,y)
G=zeros([2 1]);
E=erreur(a,b,t,y);
for i=1:length(t)
    G(1)=G(1)+(-2).*(1-exp(b.*t(i))).*(y(i)-a.*(1-exp(b.*t(i))));
    G(2)=G(2)+2.*a.*t(i).*exp(b.*t(i)).*(y(i)-a.*(1-exp(b.*t(i))));
end
G=G/exp(E);
end
```

Même principe pour l'exécution :

```
>> grad_erreur(1,2,t,y)

ans =

    2.0000
   19.5925
```

3) Traçage

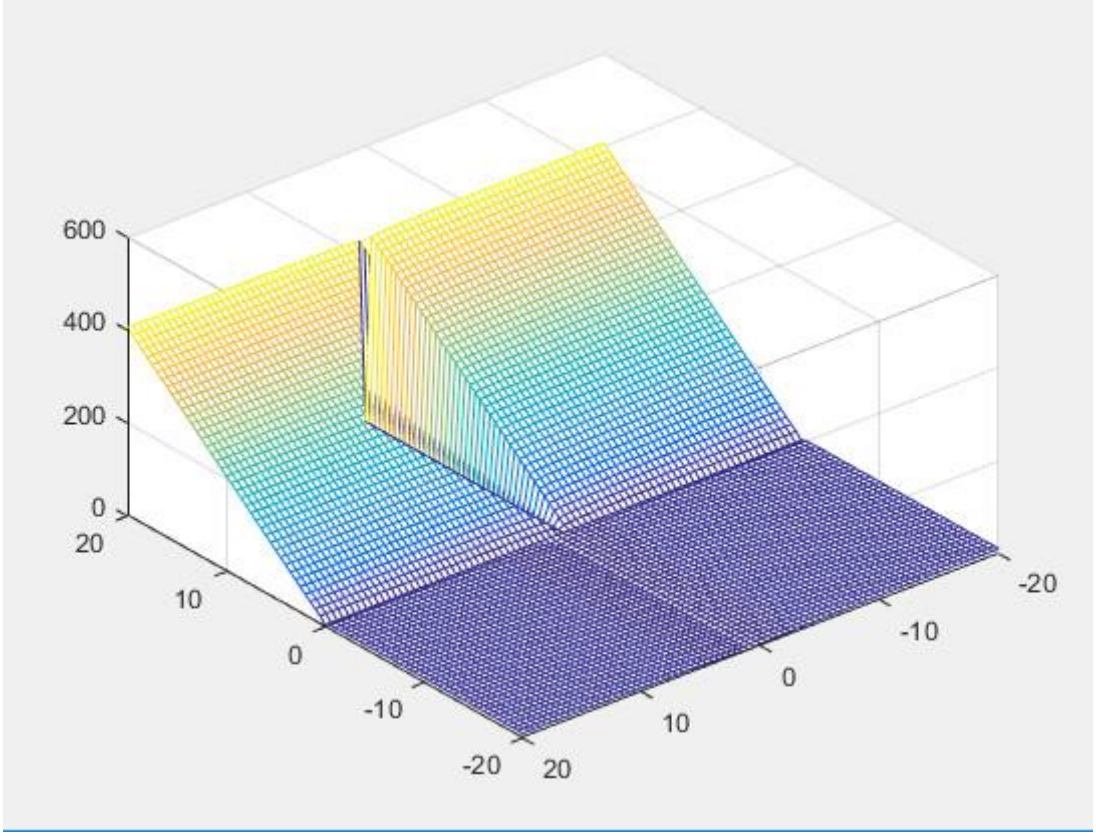
Il s'agit de la visualisation du contour et une visualisation en 3D de la courbe.

Pour faire ainsi :

```
%load partie3.mat
a=-20:0.05:20;
b=a;
[A,B]=meshgrid(a,b);
M=zeros(size(A));
l=size(A);
for k=1:l(1)
    for j=1:l(1)
        M(k,j)=erreur(A(k,j),B(k,j),t,y);
    end
end
mesh(A,B,M);
contour(A,B,M);
```

Pour visualiser en 3D, on utilise **mesh**

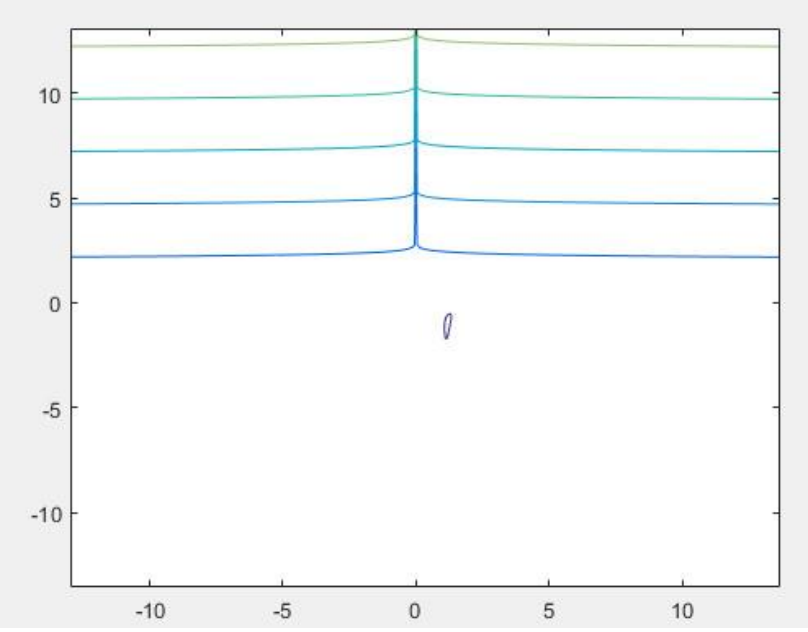
```
mesh(A,B,M);
```



On remarque qu'il y'a un creux dans la fonction ;

Pour la visualisation des contours et des lignes de niveaux on utilise **contour**

```
contour(A,B,M);
```



Remarque :

Le petit cercle que nous voyons dans la figure est celui qui contient les coordonnées de la solution de convergence des algorithmes présentés si dessous.

4) Convexité

Cette fonction n'est pas convexe. En effet pour le vérifier, et puisqu'il s'agit d'une fonction dans IR2 on peut utiliser le corollaire suivant :
F est convexe ssi $\forall\ (x, y)$ appartenant à IR2 on a $<\nabla x - \nabla y, x - y > \geq 0$
Or dans notre cas nous avons testé pour une valeur de deux vecteurs x et y

```
%load partie3.mat
x0=[1 5];
v0=[5 3];
x1=grad_erreur(x0(1),x0(2),t,y);
v1=grad_erreur(v0(1),v0(2),t,y);
x=(v0-x0)*(v1-x1);
x
```

Et le résultat est négatif

```
>> question4

x =

   -6.1460
```

Donc la fonction n'est pas convexe contrairement à ce qui peut sembler graphiquement

5) Implémentation des méthodes de minimisation

Dans cette question, on va implémenter 3 méthodes qui sont la méthode du gradient à pas fixe,

Méthode d'inertie et la méthode du gradient accéléré.

a) Méthode du gradient à pas fixe

```
function [L]=methodepasfixe(x0,eps,pas,t,y)
x1=x0;
x2=x1-pas*grad_erreur(x1(1),x1(2),t,y);
L=[x0,x2];
while norm(x2-x1) > eps
    x1=x2;
    x2=x1-pas*grad_erreur(x1(1),x1(2),t,y);
    L=[L,x2];
end
end
```

L'exécution est la suivante :

```
>> methodepasfixe([1;2],0.00001,0.0005,t,y)

ans =

  Columns 1 through 14

    1.0000    0.9990    0.9980    0.9970    0.9960    0.9950    0.9940    0.9930    0.9920    0.9910    0.9900    0.9889    0.9879    0.9869
    2.0000    1.9902    1.9804    1.9706    1.9608    1.9510    1.9412    1.9315    1.9217    1.9119    1.9021    1.8923    1.8825    1.8727

  Columns 15 through 28
```

Après plusieurs lignes on trouve

```
 Columns 603 through 611

   1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991
  -0.8917   -0.8917   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918
```

1.1991 Et -0.8918

b) Méthode d'inertie

```
function [L]=methode_inertie(x0,v0,pas,pasv,eps,t,y)
x1=x0;
v1=v0;
v2=pasv*v1+pas*grad_erreur(x1(1),x1(2),t,y);
x2=x1-v2;
L=[x0,x2];
while norm(x2-x1) > eps
    x1=x2;
    v1=v2;
    v2=pasv*v1+pas*grad_erreur(x1(1),x1(2),t,y);
    x2=x1-v2;
    L=[L,x2];
end
end
```

Pour l'exécution nous avons bien évidemment le même résultat mais avec un nombre différent d'itérations

```
>> methode_inertie([1;2],[0;1],0.0005,0.0004,0.00001,t,y)

ans =
```

Après quelques lignes

```
 Columns 603 through 611

   1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991
  -0.8917   -0.8917   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918
```

1.1991 Et -0.8918

c) Méthode du gradient accéléré

```
function [L]=grad_accelere(x0,v0,pas,pasv,eps,t,y)
v=pasv*v0+pas*grad_erreur(x0(1),x0(2),t,y);
x=x0-v;
L=[x0,x];
while(norm(x-x0)>eps)
    x0=x;
    v=pasv*v+pas.*grad_erreur(x(1)-pas*v(1),x(2)-pas*v(2),t,y);
    x=x-v;
    L=[L,x];

end

end
```

L'exécution du code nous donne exactement le même nombre d'itérations que le gradient d'inertie avec bien sur le même résultat final.

```
>> grad_accelere([1;2],[0;1],0.0005,0.0004,0.00001,t,y)

ans =


Columns 603 through 611

  1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991    1.1991
 -0.8917   -0.8917   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918   -0.8918
```

1.1991 Et -0.8918

Remarque :

Les 3 algorithmes convergent vers la même valeur ce qui est logique (1.1991 Et -0.8918).Ce qui diffère entre les 3 algorithmes est le nombre d'itérations mais cette différence est plus visible avec d'autres exemples tels que celui utilisé dans la question 7

6) Traçage des courbes

a) On recherche des paramètres qui nous permettent de voir la convergence des algorithmes :

Remarque importante : après plusieurs tests, on remarque qu'on n'a pas convergence si le pas est supérieur à une certaine valeur qui est de 10^-4.

```
load partie3.mat
x0=[1;0];
v0=[1;2];
L1=methodepasfixe(x0,0.000001,0.0001,t,y);
L2=methode_inertie(x0,v0,0.0004,0.09,0.000001,t,y);
L3=grad_accelere(x0,v0,0.0004,0.09,0.000001,t,y);

L1
L2
L3
```

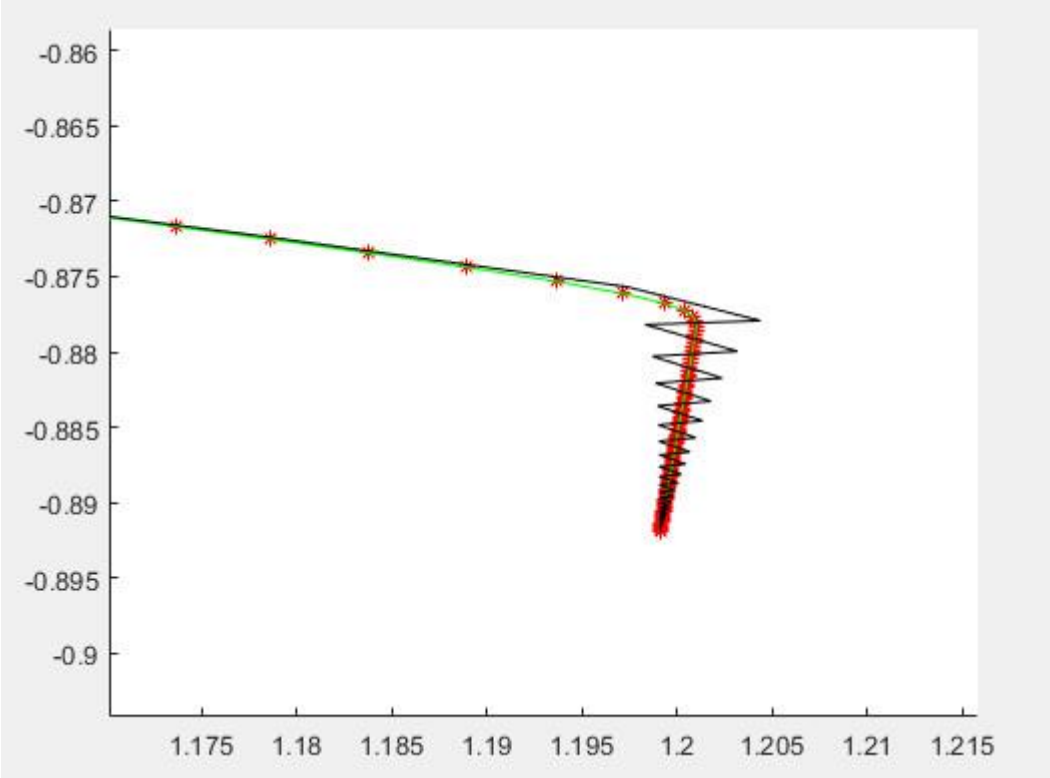
La chaine d'exécution est très longue et impossible à afficher en photo mais le résultat est toujours pour les 3 algorithmes 1.1991 Et -0.8918

b) I) On trace la trajectoire des itérations sur la figure représentant les courbes

Ceci est le code pour la méthode du gradient accéléré. Les deux autres se font de la même manière.

```
a=-10:0.5:10;
b=a;
[A,B]=meshgrid(a,b);
M=zeros(size(A));
L=size(A);
for i=1:L(1)
    for j=1:L(1)
        M(i,j)=erreur(A(i,j),B(i,j),t,y);
    end
end
x0=[1;0];
v0=grad_erreur(x0(1),x0(2),t,y);

K=grad_accelere(x0,v0,0.0004,0.09,0.000001,t,y);
x1=K(1,:);
x2=K(2,:);
hold on
contour(A,B,M);
plot(x1,x2,'r');
```

Graphiquement, nous obtenons



Remarque

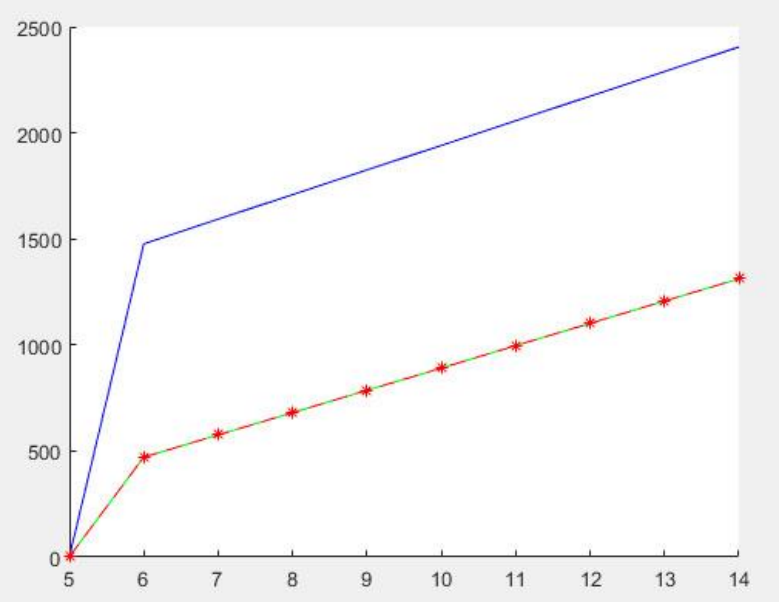
L'interprétation du chemin pris par les algorithmes se fera dans la question 7)b comme annoncé dans l'énoncé du TP grâce au zoom.

6 ii) On va tracer le nombre d'itérations en fonction de la précision epsilon

```
% on choisit une liste de valeurs au lieu d'utiliser le semilog pour avoir un
%graphe plus parlant
% on s'arrete a 10^-5 parceque a partir de  moins de 10^-4 l'algorithme ne
% converge pas
epsilon=[10-5 10^-6 10^-7 10^-8 10^-9 10^-10 10^-11 10^-12 10^-13 10^-14 ];
logchain=5:1:14;
N1=zeros(1,10);
N2=zeros(1,10);
N3=zeros(1,10);
for i=1:10
    X1=grad_accelere([1;0],grad_erreur(1,0,t,y),0.0001,0.09,epsilon(i),t,y);
    X2=methode_inertie([1;0],grad_erreur(1,0,t,y),0.0001,0.09,epsilon(i),t,y);
    X3=methodepasfixe([1;0],epsilon(i),0.0001,t,y);
    l1=size(X1);
    l2=size(X2);
    l3=size(X3);
    N1(i)=l1(2);
    N2(i)=l2(2);
    N3(i)=l3(2);
end
hold on
plot(logchain,N3,'b');
plot(logchain,N2,'g');
plot(logchain,N1,'r--*');
```

On va prendre une liste d'epsilon qui commence de 10^-5 parce qu'il faut qu'on soit au-dessous de 10^-4 pour le pas (vérifiée expérimentalement)

Graphe :



Pas fixe plus lente et nombre d'iterations reste constant

Commentaire

Bleu : méthode pas fixe

Rouge : méthode d'inertie

Vert : méthode accéléré

Les deux dernières méthodes se confondent comme tous les résultats presque précédemment. La méthode du pas fixe est plus lente que les deux autres. On remarque aussi que plus epsilon grandit les courbes ont une certaine tendance a être parallèle entre elle donc la différence du nombre d'itérations reste constante.

7) Finalisation

Question a)

Pour le point de départ $x^0 = [-2; -1]$, $\rho = 0,0004$, $\eta = 0.09$ et $\varepsilon = 10^{-6}$

```
%load partie3.mat
x0=[-2;-1];
v0=[-1;2];
pas=0.0004;
pasv=0.09;
eps=10^-6;
tic
L1=methodepasfixe(x0,eps,pas,t,y);
toc
tic
L2=methode_inertie(x0,v0,pas,pasv,eps,t,y);
toc
tic
L3=grad_accelere(x0,v0,pas,pasv,eps,t,y);
toc
l1=size(L1);
l2=size(L2);
l3=size(L3);
l1(2)
l2(2)
l3(2)
% la methode d'inertie et accelere sont de meme rapidité
% on choisit pour le reste la methode d'inertie
```

Le résultat de l'exécution est le suivant :

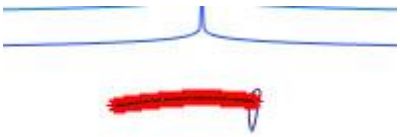
```
>> question7
Elapsed time is 0.364606 seconds.
Elapsed time is 0.193868 seconds.
Elapsed time is 0.136718 seconds.

ans =

        6552


ans =

        5546


ans =

        5546
```

Commentaire : la méthode d'inertie et du gradient accéléré ont la même nombre d'itérations mais ce nombre ne montre en rien la rapidité de de chaque algorithme. Pour cela nous devons voir le temps d'exécution avec tic et tac. On remarque que la plus rapide est la méthode du gradient accéléré.

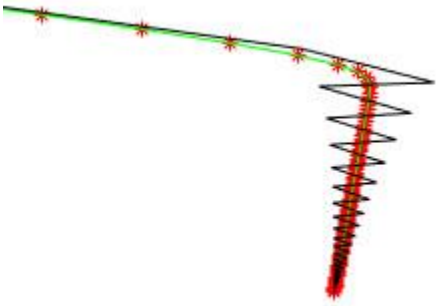
Question b)

```
a=-10:0.05:10;
b=a;
[A,B]=meshgrid(a,b);
M=erreur(A,B,t,y);
M=zeros(size(A));
L=size(A);
for i=1:L(1)
    for j=1:L(1)
        M(i,j)=erreur(A(i,j),B(i,j),t,y);
    end
end
X1=grad_accelere([-2;-1],grad_erreur(-2,-1,t,y),0.0001,0.09,0.000001,t,y);
X2=methode_inertie([-2;-1],grad_erreur(-2,-1,t,y),0.0001,0.09,0.000001,t,y);
X3=methodepasfixe([-2;-1],0.0004,0.000001,t,y);
l1=X1(1:1,:);
l2=X1(2:2,:);
l3=X2(1:1,:);
l4=X2(2:2,:);
l5=X3(1:1,:);
l6=X3(2:2,:);
hold on
contour(A,B,M);
plot(l1,x2l2,'r*');
plot(l3,l4,'g');
plot(l5,l6,'k');
```

Le résultat de la compilation est le suivant :



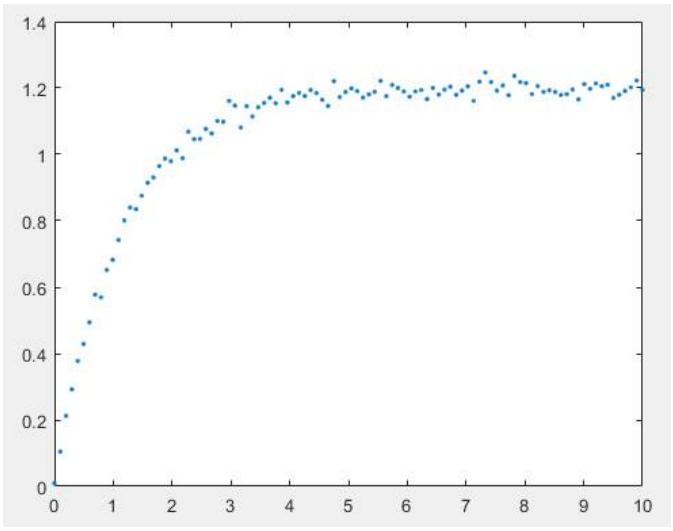
En faisant un zoom on remarque que la méthode du gradient à pas fixe prend plus de temps à être exécuter en prenant un chemin plus long (en noir) alors que les deux autres méthode se confondent pratiquement.



Question c)

Ceci est initialement la courbe sortie avec la commande

plot(t,y,'.');



Il s'agit d'un nuage de points.

Pour réaliser une animation, deux méthodes s'offrent à nous :

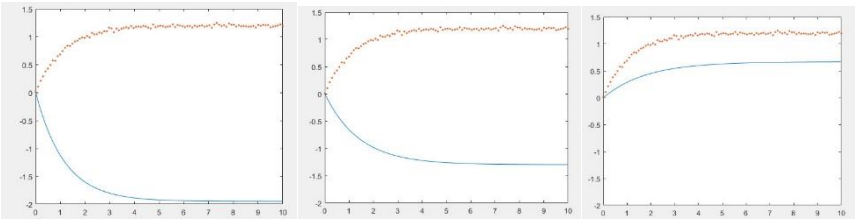
Soit une réalisation plus ou moins compliqué en utilisant wait mais ceci nous obligera à coordonner les threads dans Matlab, ou bien on peut utiliser

```
X=grad_accelere([-2;-1],[1;-1],0.0001,0.09,0.000001,t,y);
s=size(X);
k=s(2);
for i=1:200:k
    K=X(1,i)*(1-exp(X(2,i)*t));
    plot(t,K,t,y,'.');
    ylim([-2 1.5]);
    F(i)=getframe;
end
K=X(1,k)*(1-exp(X(2,k)*t));
plot(t,K,t,y,'.');
ylim([-1.5 1.5]);
F(k)=getframe;
```

Explication

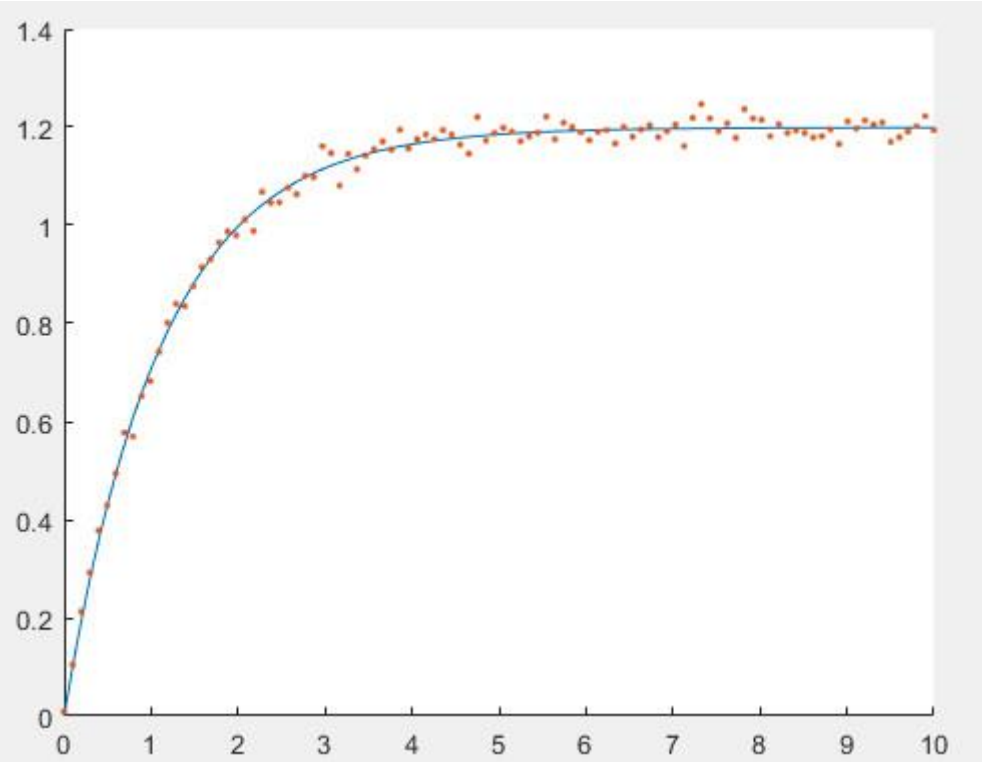
Ylim : on limite grâce à elle le plan d'affichage et getFrame on fait des captures des figures courantes.

Une animation se produit alors



Commentaire :

Plus l'indice augmente plus on se rapproche de l'allure voulu de la courbe et au final on obtient :



Conclusion

On atteint le but de la partie 3 et celui du TP d'approcher par des méthodes d'optimisation le nuage de points et y fonder une courbe.