TP1 - Recalage d'images - SKANDER RAHAL 4MA- INSA TOULOUSE

Entrée [1]:

```
import numpy as np
from scipy import signal
from scipy import interpolate
from PIL import Image
import matplotlib.pyplot as plt
```

Un problème fréquemment rencontré dans le domaine du traitement d'images est celui du recalage. On dispose de plusieurs images prises à des temps différents, ou par des appareils différents, et on aimerait les mettre en correspondence, c'est-à-dire trouver une déformation du plan, qui assure une correspondence point à point des objets sous-jacents. Donnons quelques exemples d'applications :

- Traitements/retouches d'images. Par exemple, on peut vouloir construire un panoramique à partir d'images de petite taille. Il faut les recaler préalablement.
- Evaluation des déplacements d'objets dans des séquences vidéos (e.g. trouver un défaut de fonctionnement d'un organe, caméras de surveillance, design de robots intelligents ou de systèmes de navigation automatiques ...)
- Couplage d'informations. Par exemple, en imagerie médicale, on obtient une information plus riche en utilisant à la fois une radio et une angiographie. L'une apporte des informations structurelles, l'autre des informations fonctionnelles. Le couplage des deux images donne plus d'information au praticien.
- Beaucoup d'autres applications...

Dans ce TP, nous allons proposer un modèle de recalage assez élémentaire. Les idées constitutives se retrouvent cependant dans presque toutes les techniques récentes.

Entrée [2]:

```
def get_images() :
    n=21
    sigma=0.3
    [X,Y]=np.meshgrid(np.linspace(-1,1,n),np.linspace(-1,1,n), indexing='xy')
    Z=np.sqrt(X*X+Y*Y)
    im1=np.zeros((n,n))
    im1[Z<=.7]=1.
    im1[Z<=.3]=.5
    im1[Z<=.1]=.7
    im2=np.zeros((n,n));
    Z=np.sqrt((X-.3)**2+(Y+.2)**2)
    im2[Z<=.7]=1
    im2[Z<=.3]=.5
    im2[Z<=.1]=.7
    G=np.fft.fftshift(np.exp(-(X**2+Y**2)/sigma**2))
    f=np.real(np.fft.ifft2(np.fft.fft2(G)*np.fft.fft2(im1)))
    g=np.real(np.fft.ifft2(np.fft.fft2(G)*np.fft.fft2(im2)))
    f=f/np.max(f)
    g=g/np.max(g)
    return f,g
```

Entrée [3]:

```
f,g=get_images()
```

1. Formalisation du problème

1.1 Formalisme continu

On modélise les images en niveaux de gris comme des fonctions d'un ensemble borné $\Omega \subset \mathbb{R}$ (typiquement un carré) dans $\mathbb{R}$. La valeur de la fonction en chaque point représente l'intensité lumineuse de l'image.

Soient f et g deux images. On a donc :

$$f : \Omega \subset \mathbb{R}^2 \to \mathbb{R}, g : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$$

En supposant que les images f et g dépendent seulement d'une transformation géométrique qui conserve la luminosité, le problème de recalage peut être formulé comme suit:

> Problème inverse (P_1) :

> Etant donnés f et g dans $H_1(\Omega)$ (les images ont une amplitude bornée et une énergie finie), trouver un champ de vecteurs $u = (u_1, u_2) \in H_1(\Omega)^2$ tel que:
>
> $$f(x + u(x)) = g(x), \forall x \in \Omega.$$

Le problème inverse est mal posé: tout d'abord, l'existence d'une solution n'est pas garantie, et dans le cas où il existe une solution, on n'a pas nécessairement unicité de cette solution. Par exemple, si f et g sont des fonctions constantes, n'importe quel déplacement u est solution

Pour le résoudre, on se propose de le reformuler comme un problème d'optimisation:

> (P_2) On cherche une déformation u du plan qui minimise:
> $$E(u) = \frac{1}{2}\int_\Omega (f(x+u(x)) - g(x))^2 dx = \frac{1}{2}\|f\circ(id+u) - g\|^2.$$

Sans hypothèse supplémentaire, le problème $\min_{u\in H^1(\Omega)^2} E(u)$ n'est a priori pas convexe, toujours mal posé et même éventuellement non différentiable si u et f ne sont pas assez régulières. On pourrait facilement rendre f différentiable (en ajoutant du bruit par exemple à l'image, ce qui revient à convoler f avec une gaussienne) mais il faut également "forcer" u à être différentiable. Pour cela on propose de régulariser le problème de façon à assurer la convexité du problème d'optimisation considéré ainsi que l'existence et l'unicité des solutions.

Pour régulariser le problème inverse, nous allons faire une analogie avec l'élasticité linéaire. La fonction $u = (u_x, u_y)$ représente un champ de déformations. En notant ∂x et ∂y les opérateurs de dérivation partielle par rapport à chacun des axes du plan, on peut définir un potentiel élastique linéarisé :

$$R(u) = \frac{\mu}{2}\int_\Omega \underbrace{(\partial_x u_y + \partial_y u_x)^2(x,y)dxdy}_{R_1(u)=\text{cisaillement}} + \frac{\lambda+\mu}{2}\int_\Omega \underbrace{(\partial_x u_x + \partial_y u_y)^2(x,y)dxdy}_{R_2(u)=\text{variations \ de \ volume}}.$$

En mécanique des structures, μ et λ sont appelées constantes de Lamé. Le paramètre λ n'a pas d'interprétation directe, tandis que le paramètre μ est appelé module de cisaillement.

Le problème d'optimisation à résoudre dans ce TP est le suivant:

> $$(P) \qquad \min_u E(u) + R(u).$$

Q1. A l'aide d'un développement de Taylor, vérifier que le gradient de E s'écrit:

$$\nabla E(u) = (f\circ(id+u) - g)\,\nabla f\circ(id+u)$$

au sens où la différentielle de E est définie par:

$$\langle \nabla E(u), h\rangle = \int_\Omega \langle (f(x+u(x)) - g(x))\nabla f(x+u(x)), h(x)\rangle dx.$$

Réponse:

D'après le développement de Taylor, on sait que:

$$f(x+u(x)+h(x)) = f(x+u(x)) + \langle \nabla f(x+u(x)), h(x)\rangle + o(\|h\|)$$

Donc on a :

$$\begin{aligned}
E(u+h) &= \frac{1}{2}\int_\Omega (f(x+u(x)+h(x)) - g(x))^2 dx \\
&= \frac{1}{2}\int_\Omega (f(x+u(x)) + \langle \nabla f(x+u(x)), h(x)\rangle + o(\|h\|) - g(x))^2 dx \\
&= \frac{1}{2}\int_\Omega [f(x+u(x)) - g(x)]^2 + 2\langle \nabla f(x+u(x)), h(x)\rangle (f(x+u(x)) - g(x)) + o(\|h\|))dx \\
&= E(u) + \int_\Omega \langle (f(x+u(x)) - g(x))\,\nabla f(x+u(x)), h(x)\rangle dx + o(\|h\|)
\end{aligned}$$

Par suite comme:

$$E(u+h) = E(u) + \langle \nabla E(u), h\rangle + o(\|h\|)$$

Donc par identification, on a que:

$$\nabla E(u) = (f\circ(id+u) - g)\,\nabla f\circ(id+u)$$

2. Discrétisation

Pour pouvoir résoudre numériquement le problème (P) (dont les variables de l'optimisation sont des fonctions !), on propose de le discrétiser au préalable.

Soit $1 \le i \le n$ and $1 \le j \le m$. Notons (x_i, y_j) le point de la grille (i,j) et $f_{i,j}$ la valeur de f au point (x_i, y_j). Le produit scalaire sur $V = \mathbb{R}^n \times \mathbb{R}^m$ est défini par:

$$\langle f, g\rangle_V = \sum_{i=1}^n \sum_{j=1}^m f_{i,j} g_{i,j},$$

défini sur $\mathbb{R}^n \times \mathbb{R}^m$.

2.1. Calcul du E et de son gradient

Pour pouvoir calculer E et son gradient, on va avoir besoin d'évaluer $f \circ (Id + u)$ et $\nabla f \circ (id + u)$. C'est ce que fait la fonciton interpol ci-dessous.

Entrée [4]:

```
def interpol(f,ux,uy) :
    # function that computes f \circ Id+u and interpolates it on a mesh
    nx,ny=f.shape
    ip=interpolate.RectBivariateSpline(np.arange(nx),np.arange(ny),f)
    [X,Y]=np.meshgrid(np.arange(nx),np.arange(ny), indexing='ij')
    X=X+ux
    Y=Y+uy
    return np.reshape(ip.ev(X.ravel(),Y.ravel()),(nx,ny))
```

2.2. Calcul de R et de son gradient

On discrétise également les opérateurs de dérivation partielles par différences finies ; par exemple la dérivée partielle par rapport à x est donnée par:

$$\begin{cases} (\partial_x f)_{i,j} = f_{i+1,j} - f_{i,j} \text{ si } i < n \\ (\partial_x f)_{n,j} = 0 \end{cases}$$

On peut alors écrire :

$$R(u) = \frac{\mu}{2} \sum_i (\partial_x u_y + \partial_y u_x)^2(i) + \frac{\lambda + \mu}{2} \sum_i (\partial_x u_x + \partial_y u_y)^2(i).$$

où:

- $u_x \in \mathbb{R}^n$ et $u_y \in \mathbb{R}^n$ sont les discrétisations des composantes du champ de vecteurs u sur la grille choisie et $\partial_x : \mathbb{R}^n \to \mathbb{R}^n$
- $\partial_y : \mathbb{R}^n \to \mathbb{R}^n$ représentent des opérateurs de différences finies.

On peut ré-écrire $R(u) = \frac{1}{2} R_1(u) + \frac{1}{2} R_2(u)$ avec :

$$R_1(u) = \langle A_1 u, A_1 u \rangle, \qquad R_2(u) = \langle A_2 u, A_2 u \rangle.$$

Q2. Donner les formules de discrétisation des opérateurs ∂_y, $\partial_x^\top$ et $\partial_y^\top$. Implémenter ces opérateurs ci-après.

On sait que :

$$\langle \partial_x^T u, v \rangle = \langle u, \partial_x v \rangle$$

donc :

$$\begin{aligned}
\langle \partial_x^T u, v \rangle &= \sum_{i=1}^{n} \sum_{j=1}^{m} (\partial_x^T u)_{i,j} . v_{ij} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{m} u_{i,j} . (\partial_x v)_{ij} \\
&= \sum_{i=1}^{n-1} \sum_{j=1}^{m} u_{ij} (v_{i+1,j} - v_{ij}) \\
&= \sum_{i=1}^{n-1} \sum_{j=1}^{m} u_{ij} v_{i+1,j} - \sum_{i=1}^{n-1} \sum_{j=1}^{m} u_{ij} v_{ij} \\
&= \sum_{i=2}^{n} \sum_{j=1}^{m} u_{i-1,j} v_{ij} - \sum_{i=1}^{n-1} \sum_{j=1}^{m} u_{ij} v_{ij} \\
&= \sum_{i=2}^{n-1} \sum_{j=1}^{m} u_{i-1,j} v_{ij} - \sum_{i=2}^{n-1} \sum_{j=1}^{m} u_{ij} v_{ij} - \sum_{j=1}^{m} u_{1j} v_{1j} + \sum_{j=1}^{m} u_{n-1,j} v_{nj} \\
&= \sum_{i=2}^{n-1} \sum_{j=1}^{m} (u_{i-1,j} - u_{ij}) v_{i,j} + \sum_{j=1}^{m} u_{n-1,j} v_{n,j} - \sum_{j=1}^{m} u_{1j} v_{1j}
\end{aligned}$$

Par identification, on a que:

$$\partial_x^T u = \begin{cases} u_{i-1,j} - u_{ij} \text{ si } i \geq 2 \text{ et } i \leq n-1 \\ u_{n-1,j} \text{ si } i = n \\ -u_{1,j} \text{ si } i = 1 \end{cases}$$

De plus,

$$\partial_y^T u = \begin{cases} u_{i,j-1} - u_{ij} \text{ si } j \geq 2 \text{ et } j \leq m-1 \\ u_{i,m-1} \text{ si } j = m \\ -u_{i,1} \text{ si } j = 1 \end{cases}$$

Entrée [5]:

```
def dx(im) :
    d=np.zeros(im.shape)
    d[:-1,:]=im[1:,:]-im[:-1,:]
    return d
def dy(im) :
    d=np.zeros(im.shape)
    d[:,:-1]=im[:,1:]-im[:,:-1]
    return d
def dyT(im) :
    d=np.zeros(im.shape)
    d[:,1:-1]=im[:,0:-2]-im[:,1:-1]
    d[:,-1]=im[:,-2]
    d[:,0]=-im[:,0]
    return d
def dxT(im) :
    d=np.zeros(im.shape)
    d[1:-1,:]=im[0:-2,:]-im[1:-1,:]
    d[-1,:]=im[-2,:]
    d[0,:]=-im[0,:]
    return d
```

Q3. On écrit $R(u)$ comme un opérateur de V^2 dans $\mathbb{R}$. Montrer que $R(u)$ peut s'écrire sous la forme:

$$R(u) = \frac{1}{2}\langle A \begin{pmatrix} u_x \\ u_y \end{pmatrix}, \begin{pmatrix} u_x \\ u_y \end{pmatrix} \rangle_{V^2},$$

et donnez l'expression des matrices A_1 et A_2 en fonction des opérateurs ∂_x, $\partial_x^\top$, ∂_y et $\partial_y^\top$.

On sait que:

$$R(u) = \frac{\mu}{2}\sum_i (\partial_x u_y + \partial_y u_x)^2(i) + \frac{\lambda+\mu}{2}\sum_i (\partial_x u_x + \partial_y u_y)^2(i).$$

De plus,

$$R(u) = \frac{1}{2}R_1(u) + \frac{1}{2}R_2(u) = \frac{1}{2}\langle A_1 u, A_1 u\rangle + \frac{1}{2}\langle A_2 u, A_2 u\rangle$$

Donc à partir de $R(u)$,on pourra retrouver l'expression de A_1 et A_2 en fonction des opérateurs ∂_x, $\partial_x^\top$, ∂_y et $\partial_y^\top$.

Pour A_1:

$$R_1(u) = \|A_1 u\|^2 = \left\| A_1 \begin{pmatrix} u_x \\ u_y \end{pmatrix} \right\|^2 = \mu \|\partial_x u_y + \partial_y u_x\|^2 = \mu \| \begin{pmatrix} \partial_y & \partial_x \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} \|^2$$

Donc,

$$A_1 = \sqrt{\mu} \begin{pmatrix} \partial_y & \partial_x \end{pmatrix}$$

Pour A_2:

$$R_2(u) = \|A_2 u\|^2 = \left\| A_2 \begin{pmatrix} u_x \\ u_y \end{pmatrix} \right\|^2 = (\lambda+\mu) \|\partial_x u_x + \partial_y u_y\|^2 = (\lambda+\mu) \| \begin{pmatrix} \partial_x & \partial_y \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} \|^2$$

Donc,

$$A_2 = \sqrt{\lambda+\mu} \begin{pmatrix} \partial_x & \partial_y \end{pmatrix}$$

Q4. Donner l'expression du gradient de R.

On sait que : $R(u) = \frac{1}{2}R_1(u) + \frac{1}{2}R_2(u)$, on a $\nabla R(u) = \frac{1}{2}\nabla R_1(u) + \frac{1}{2}\nabla R_2(u)$.

Donc , on calcule de façon séparée ∇R_1 et ∇R_2 :

Pour $\nabla R_1(u)$:

$$\begin{aligned} \nabla R_1(u) &= 2A_1^T A_1 u \\ &= 2\mu \begin{pmatrix} \partial_y^T \\ \partial_x^T \end{pmatrix} (\partial_y \ \partial_x) \begin{pmatrix} u_x \\ u_y \end{pmatrix} \\ &= 2\mu \begin{pmatrix} \partial_y^T \partial_y & \partial_y^T \partial_x \\ \partial_x^T \partial_y & \partial_x^T \partial_x \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} \\ &= 2\mu \begin{pmatrix} \partial_y^T \partial_y u_x + \partial_y^T \partial_x u_y \\ \partial_x^T \partial_y u_x + \partial_x^T \partial_x u_y \end{pmatrix} \end{aligned}$$

Pour $\nabla R_2(u)$:

$$\begin{aligned}\nabla R_2(u) &= 2A_2^T A_2 u \\ &= 2(\lambda+\mu)\begin{pmatrix}\partial_x^T \\ \partial_y^T\end{pmatrix}(\partial_x\ \partial_y)\begin{pmatrix}u_x \\ u_y\end{pmatrix} \\ &= 2(\lambda+\mu)\begin{pmatrix}\partial_x^T\partial_x & \partial_x^T\partial_y \\ \partial_y^T\partial_x & \partial_y^T\partial_y\end{pmatrix}\begin{pmatrix}u_x \\ u_y\end{pmatrix} \\ &= 2(\lambda+\mu)\begin{pmatrix}\partial_x^T\partial_x u_x + \partial_x^T\partial_y u_y \\ \partial_y^T\partial_x u_x + \partial_y^T\partial_y u_y\end{pmatrix}\end{aligned}$$

Par suite, on retrouve donc:

$$\nabla R(u) = \frac{1}{2}\nabla R_1(u) + \frac{1}{2}\nabla R_2(u) = \mu\begin{pmatrix}\partial_y^T\partial_y u_x + \partial_y^T\partial_x u_y \\ \partial_x^T\partial_y u_x + \partial_x^T\partial_x u_y\end{pmatrix} + (\lambda+\mu)\begin{pmatrix}\partial_x^T\partial_x u_x + \partial_x^T\partial_y u_y \\ \partial_y^T\partial_x u_x + \partial_y^T\partial_y u_y\end{pmatrix}$$

2.3. Implémentation de la fonction objectif $E + R$

Entrée [6]:

```
def objective_function(f,g,ux,uy,lamb,mu) :
    fu=interpol(f,ux,uy)
    #calcul de E
    E=(1/2)*np.linalg.norm(fu-g)**2
    R=(mu/2)*np.linalg.norm(dx(uy)+dy(ux))**2+((lamb+mu)/2)*np.linalg.norm(dx(ux)+dy(uy))**2
    obj=E+R
    return obj,fu
```

3. Un algorithme de gradient

Une itération de la méthode de descente de gradient est de la forme:

$$u_{k+1} = u_k - s_k(\nabla E(u) + \nabla R(u))$$

Q5. Compléter la fonction RecalageDG implémentant la descente de gradient et utilisant l'algorithme de recherche linéaire par rebroussement proposé ci-dessous.

Entrée [7]:

```
def linesearch(ux,uy,step,descentx,descenty,obj_old,f,g,lamb,mu) :
    step=2*step
    tmpx=ux-step*descentx
    tmpy=uy-step*descenty
    obj,fu=objective_function(f,g,tmpx,tmpy,lamb,mu)
    while obj >obj_old and step > 1.e-8:
        step=0.5*step
        tmpx=ux-step*descentx
        tmpy=uy-step*descenty
        obj,fu=objective_function(f,g,tmpx,tmpy,lamb,mu)
    return tmpx,tmpy,step
```

Entrée [8]:

```
def RecalageDG(f,g,lamb,mu,nitermax,stepini) :
    ux=np.zeros(f.shape)
    uy=np.zeros(f.shape)
    CF=[]
    step_list=[]
    niter=0
    step=stepini
    dfx=dx(f)
    dfy=dy(f)
    while niter < nitermax and step > 1.e-8 :
        niter+=1
        obj,fu=objective_function(f,g,ux,uy,lamb,mu)
        CF.append(obj)
        # Gradient of E at point u
        gradEx=(fu-g)*interpol(dfx,ux,uy)
        gradEy=(fu-g)*interpol(dfy,ux,uy)


        # Gradient of R at point u
        gradRx=mu*(dyT(dy(ux))+dyT(dx(uy)))+(lamb+mu)*(dxT(dx(ux))+dxT(dy(uy)))
        gradRy=mu*(dxT(dy(ux))+dxT(dx(uy)))+(lamb+mu)*(dyT(dx(ux))+dyT(dy(uy)))


        # Gradient of E+R at point u
        gradx=gradEx+gradRx
        grady=gradEy+gradRy


        ux,uy,step=linesearch(ux,uy,step,gradx,grady,obj,f,g,lamb,mu)
        step_list.append(step)
        if (niter % 3 ==0) :
            print('iteration :',niter,' cost function :',obj,'step :',step)
    return ux,uy,np.array(CF),np.array(step_list),niter
```

Q5. Ecrire un compte-rendu des expériences réalisées et des résultats obtenus. Commentez.

Afin que l'algorithme de descente de gradient converge avec un nombre d'itéarations le plus minimal possible,j'ai décidé de créér une fonction qui permet de calculer les valeurs de λ et μ les plus optimales possibles. Pour cela , pour différentes valeurs de λ ,j'ai calculé le nombre d'itérations associé et on remarque que l'algorithme converge lorsque le nombre d'itérations est égale à 2589. La valeur de λ associé est égale à 0.11111111111111116 , c'est bel et bien la valeur optimale de λ.

Entrée [9]:

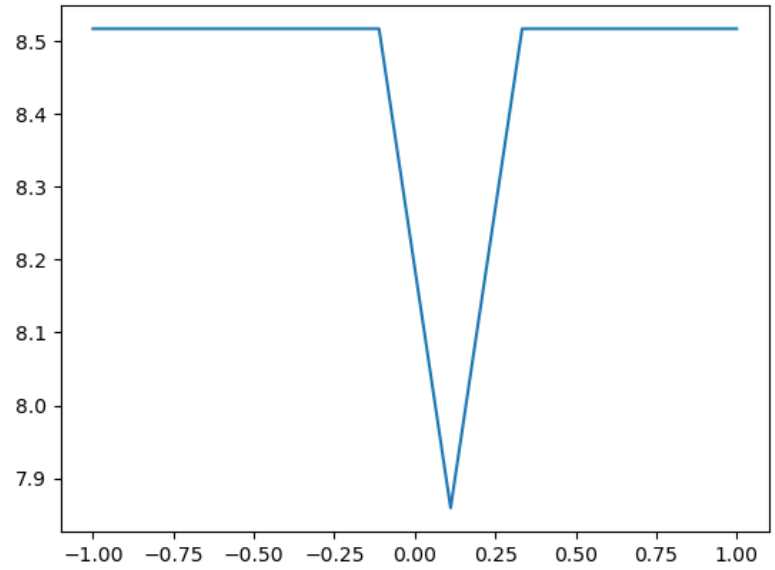
```
vect_lambd=np.linspace(-1,1,10)
vect_mu=np.copy(0.1*vect_lambd)
niter=np.zeros(10)
for i in range(10):
    ux,uy,CF,step,niter[i]=RecalageDG(f,g,vect_lambd[i],vect_mu[i],5000,0.1)
```

```
iteration : 3  cost function : 18.531415660085198 step : 0.8
iteration : 6  cost function : -324.4483031346749 step : 6.4
iteration : 9  cost function : -1883674384904.5137 step : 51.2
iteration : 12  cost function : -1.3387703798151107e+28 step : 409.6
iteration : 15  cost function : -1.0809136426302752e+50 step : 3276.8
iteration : 18  cost function : -1.6330452127191533e+78 step : 26214.4
iteration : 21  cost function : -1.2102010731799808e+112 step : 209715.2
iteration : 24  cost function : -3.0084284692686946e+151 step : 1677721.6
iteration : 27  cost function : -2.286547593053499e+196 step : 13421772.8
iteration : 30  cost function : -5.029789638804229e+246 step : 107374182.4
iteration : 33  cost function : -3.0896728151973952e+302 step : 858993459.2
iteration : 36  cost function : -inf step : 6871947673.6
iteration : 39  cost function : -inf step : 54975581388.8
iteration : 42  cost function : -inf step : 439804651110.4
iteration : 45  cost function : -inf step : 3518437208883.2
iteration : 48  cost function : nan step : 28147497671065.6
iteration : 51  cost function : nan step : 225179981368524.8
iteration : 54  cost function : nan step : 1801439850948198.5
iteration : 57  cost function : nan step : 1.4411518807585588e+16
```

Entrée [10]:

```
plt.plot(vect_lambd,np.log(niter))
print(niter)
```

```
[5000. 5000. 5000. 5000. 5000. 2589. 5000. 5000. 5000. 5000.]
```



Entrée [11]:

```
k=np.argmin(niter)
print(k,'indice qui correspond au nombre d''itérations minimale de l''algo de descente de gradient')
print(niter[k],'correspond au nombre d''itérations minimale de l''algo de descente de gradient')
lambd_opt=vect_lambd[k]
print(lambd_opt,'la valeur optimale de lambda associé à niter minimale')
```

```
5 indice qui correspond au nombre ditérations minimale de lalgo de descente de gradient
2589.0 correspond au nombre ditérations minimale de lalgo de descente de gradient
0.11111111111111116 la valeur optimale de lambda associé à niter minimale
```

Par la suite , on prendra λ égale à 0.11111111111111116 et $\mu = 0.1\lambda$ Dans ce cas, on obtient ci-dessous, une fonction finale égale à la fonction target.

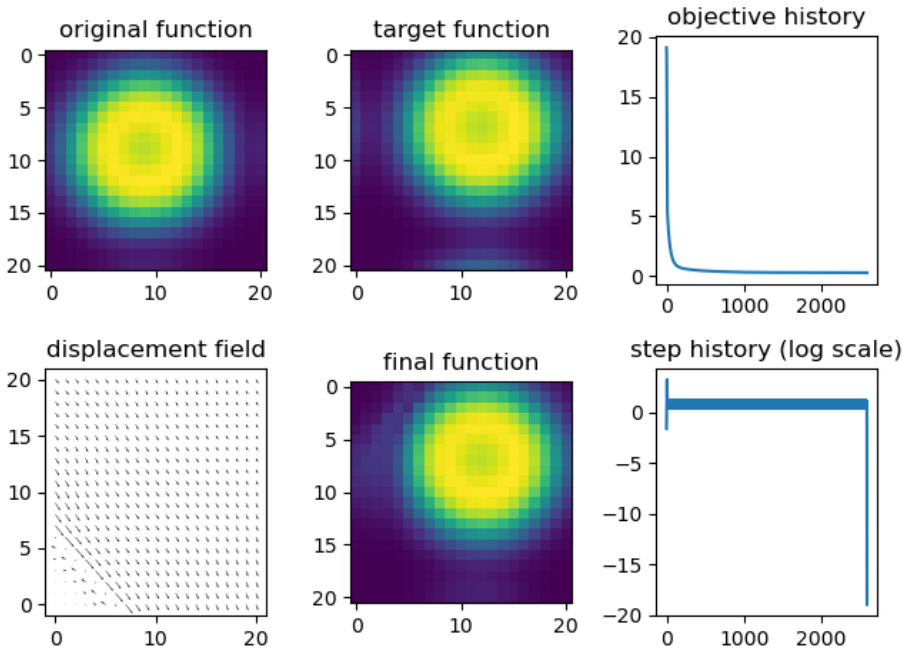
Entrée [12]:

```
lamb=0.11111111111111116
mu=0.011111111111111116
nitermax=5000

step0 =0.1
ux,uy,CF,step,niter=RecalageDG(f,g,lamb,mu,nitermax,step0)
```

```
iteration : 3  cost function : 18.618896932611236 step : 0.8
iteration : 6  cost function : 14.988649458581124 step : 6.4
iteration : 9  cost function : 6.253350155244529 step : 3.2
iteration : 12  cost function : 5.66641590608094 step : 1.6
iteration : 15  cost function : 5.050452267479189 step : 1.6
iteration : 18  cost function : 4.675066440198215 step : 1.6
iteration : 21  cost function : 4.358644438079138 step : 1.6
iteration : 24  cost function : 4.091246554549395 step : 1.6
iteration : 27  cost function : 3.7588666350411453 step : 3.2
iteration : 30  cost function : 3.5086018793583884 step : 3.2
iteration : 33  cost function : 3.2857513951420403 step : 1.6
iteration : 36  cost function : 3.0896040684373647 step : 1.6
iteration : 39  cost function : 2.9214371596841993 step : 1.6
iteration : 42  cost function : 2.7168285722684975 step : 3.2
iteration : 45  cost function : 2.5562696565858003 step : 3.2
iteration : 48  cost function : 2.413063559119502 step : 1.6
iteration : 51  cost function : 2.286922684436102 step : 1.6
iteration : 54  cost function : 2.179025394311027 step : 1.6
iteration : 57  cost function : 2.0453938808930547 step : 3.2
```

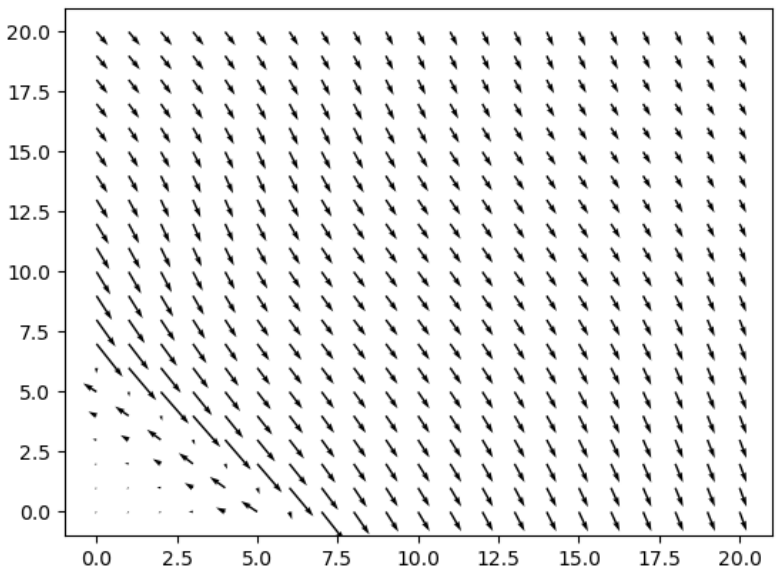
Entrée [13]:

```
fig, ax = plt.subplots(2,3)
ax[0,0].imshow(f)
ax[0,0].set_title('original function')
ax[0,1].imshow(g)
ax[0,1].set_title('target function')
ax[1,0].quiver(ux,uy)
ax[1,0].set_title('displacement field')
ax[1,1].imshow(interpol(f,ux,uy))
ax[1,1].set_title('final function')
ax[0,2].plot(CF)
ax[0,2].set_title('objective history')
ax[1,2].plot(np.log(step))
ax[1,2].set_title('step history (log scale)')

plt.tight_layout()
plt.show()
```



Entrée [14]:

```
plt.quiver(ux, uy)
plt.show()
```



Le graphe ci-dessus contenant l'ensemble des vecteurs (ux,uy) résume le déplacement de notre fonction. Cela correspond bien aux graphes de la fonction originale et la fonction objective. En effet, on a bien qu'en bas à gauche, l'angle ne change pas d'une image à une autre (vecteur nul dans le graphe de quiver) cependant on note des vecteurs non nuls sur le reste du graphe de quiver, ceci explique bien le déplacement de l'objet entre les deux images.

4. Algorithme de moindres carrés.

On souhaite maintenant implémenter un algorithme de second ordre pour résoudre le problème $(P) \quad \min_u E(u) + R(u)$ afin d'accélérer la convergence de l'algorithme. Pour cela, on va reformuler le problème (P) en un problème de moindres carrés et appliquer l'algorithme de Levenberg-Marquardt.

Soit:

$$\Psi(u) = \begin{pmatrix} f \circ (Id + u) - g \\ \sqrt{\mu}(\partial_x u_y + \partial_y u_x) \\ \sqrt{\mu + \lambda}(\partial_x u_x + \partial_y u_y) \end{pmatrix},$$

où $f \circ (id + u)$ est l'interpolation de $x \mapsto f(x + u(x))$ sur la grille. Minimiser $E(u) + R(u)$ est équivalent à résoudre le problème suivant:

$$\min_u \|\Psi(u)\|_2^2.$$

Il s'agit maintenant d'un problème de moindres carrés que l'on va résoudre à l'aide de l'algorithme de Levenberg Marquardt :

$$u_{k+1} = u_k - H_k^{-1} J_\Psi(u_k)^\top \Psi(u_k) \quad \text{avec} \quad H_k = J_\Psi(u_k)^\top J_\Psi(u_k) + \varepsilon Id$$

Q6. Calculer la matrice jacobienne de Ψ, notée $J_\Psi(u)$.

On sait que:

$$J_\psi(u) = \begin{pmatrix} \nabla_{\psi_1}(u)^T \\ \nabla_{\psi_2}(u)^T \\ \nabla_{\psi_3}(u)^T \end{pmatrix}$$

De plus,

$$\psi(u) = \begin{pmatrix} \psi_1(u) \\ \psi_2(u) \\ \psi_3(u) \end{pmatrix}$$

et

$$\psi(u) = \begin{pmatrix} f \circ (Id + u) - g \\ \sqrt{\mu}(\partial_x u_y + \partial_y u_x) \\ \sqrt{\mu + \lambda}(\partial_x u_x + \partial_y u_y) \end{pmatrix}$$

Afin de déterminer $J_\psi(u)$, on calcule les différentes valeurs $\nabla_{\psi_1}, \nabla_{\psi_2}, \nabla_{\psi_3}$:

$$\begin{aligned} \nabla_{\psi_2}(u) &= \sqrt{\mu}\nabla(\partial_x u_y + \partial_y u_x) \\ &= \sqrt{\mu}\nabla \begin{pmatrix} \partial_y \\ \partial_x \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} \end{aligned}$$

Donc , par suite , on a bien que :

$$\nabla_{\psi_2} = \sqrt{\mu} \begin{pmatrix} \partial_y \\ \partial_x \end{pmatrix}$$

Calculons ensuite ∇_{ψ_3} :

$$\begin{aligned} \nabla_{\psi_3}(u) &= \sqrt{\mu + \lambda}\nabla(\partial_x u_x + \partial_y u_y) \\ &= \sqrt{\mu + \lambda}\nabla \begin{pmatrix} \partial_x \\ \partial_y \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} \end{aligned}$$

Donc par suite , on a bien que :

$$\nabla_{\psi_3} = \sqrt{\mu + \lambda} \begin{pmatrix} \partial_x \\ \partial_y \end{pmatrix}$$

Enfin , on calculera ∇_{ψ_1} : on sait que $(J_{\psi_1}(u))^T = \nabla_{\psi_1}(u)$ et $J_{Id+u}(u) = \nabla^T(Id + u)(u) = Id$

$$\begin{aligned} J_{\psi_1}(u) &= J_f(Id + u) \times J_{Id+u}(u) \\ &= \nabla^T f(Id + u) \times Id \\ &= \nabla^T f(Id + u) \end{aligned}$$

On obtient donc:

$$J_\psi(u) = \begin{pmatrix} df_x(Id + u) & df_y(Id + u) \\ \sqrt{\mu}\partial_y & \sqrt{\mu}\partial_x \\ \sqrt{\lambda + \mu}\partial_x & \sqrt{\lambda + \mu}\partial_y \end{pmatrix}$$

Q7. Implémenter les fonctions JPsi, JTPsi et JTJ qui calculent respectivement:

- le produit de $J_\psi(u)$ par une direction $v = (v_x, v_y) \in V^2$,
- le produit de $J_\Psi(u)^\top$ par $\phi = (\phi_1, \phi_2, \phi_3) \in V^3$,
- le produit de $(J_\Psi(u)^\top J_\Psi(u) + \epsilon I)$ par une direction $v = (v_x, v_y) \in V^2$.

Entrée [15]:

```
def JTPsi(phi,df,lamb,mu) :
    ux=phi[0]*df[0]+np.sqrt(mu)*dyT(phi[1])+np.sqrt(lamb+mu)*dxT(phi[2])
    uy=phi[0]*df[1]+np.sqrt(mu)*dxT(phi[1])+np.sqrt(lamb+mu)*dyT(phi[2])
    return [ux,uy]


def JPsi(vx,vy,df,lamb,mu) :
    JPsi0=df[0]*vx+df[1]*vy
    JPsi1=np.sqrt(mu)*dy(vx)+np.sqrt(mu)*dx(vy)
    JPsi2=np.sqrt(mu+lamb)*dx(vx)+np.sqrt(mu+lamb)*dy(vy)
    return [JPsi0,JPsi1,JPsi2]

def JTJ(vx,vy,df,lamb,mu,epsilon) :
    uxs,uys=JTPsi(JPsi(vx,vy,df,lamb,mu),df,lamb,mu)
    uxs=uxs+epsilon*vx
    uys=uys+epsilon*vy
    return uxs,uys
```

Calculons maintenant la direction de recherche d_k comme solution du système linéaire:

$$(J_\Psi(u_k)^\top J_\Psi(u_k) + \epsilon I)\begin{pmatrix} d_x \\ d_y \end{pmatrix} = -J_\Psi(u_k)^\top \Psi(u_k)$$

Pour cela, on vous donne l'algorithme suivant qui par la méthode du gradient conjugué calcule une solution $d = (d_x, d_y) \in V^2$ du problème:

$$(J_\Psi(u_k)^\top J_\Psi(u_k) + \epsilon I)\begin{pmatrix} d_x \\ d_y \end{pmatrix} = b$$

Entrée [16]:

```
def CGSolve(u0x,u0y,lamb,mu,b,epsilon,df) :
    nitmax=100;
    ux=u0x; uy=u0y; #point de départ de l'algorithme
    # Computes JTJu
    Ax,Ay=JTJ(ux,uy,df,lamb,mu,epsilon);
    rx=b[0]-Ax
    ry=b[1]-Ay
    px=rx
    py=ry
    rsold=np.linalg.norm(rx)**2+np.linalg.norm(ry)**2
    for i in range(nitmax) :
        Apx,Apy=JTJ(px,py,df,lamb,mu,epsilon);
        alpha=rsold/(np.vdot(rx[:],Apx[:])+np.vdot(ry[:],Apy[:]))
        ux=ux+alpha*px
        uy=uy+alpha*py
        rx=rx-alpha*Apx
        ry=ry-alpha*Apy
        rsnew=np.linalg.norm(rx)**2+np.linalg.norm(ry)**2
        if np.sqrt(rsnew)<1e-10 :
            return [ux,uy]
        px=rx+rsnew/rsold*px
        py=ry+rsnew/rsold*py
        rsold=rsnew
    return [ux,uy]
```

Q8. Compléter l'algorithme RecalageGN implémentant la méthode de Levenberg-Marquardt.

Entrée [17]:

```
def psi(fu,g,lamb,mu,ux,uy):
    psi1=fu-g
    psi2=np.sqrt(mu)*dy(ux)+np.sqrt(mu)*dx(uy)
    psi3=np.sqrt(lamb+mu)*dx(ux)+np.sqrt(lamb+mu)*dy(uy)
    return [psi1,psi2,psi3]
def RecalageGN(f,g,lamb,mu,nitermax,stepini,epsi) :
    ux=np.zeros(f.shape)
    uy=np.zeros(f.shape)
    descentx=np.zeros(f.shape)
    descenty=np.zeros(f.shape)
    #raise ValueError('To complete if necessary')
    CF=[]
    step_list=[]
    niter=0
    step=stepini
    dfx=dx(f)
    dfy=dy(f)
    while niter < nitermax and step > 1.e-8 :
        niter+=1
        obj,fu=objective_function(f,g,ux,uy,lamb,mu)
        CF.append(obj)
        # Gradient of F at point u
        #raise ValueError('Compute b here')
        #raise ValueError('Compute dfx,dfy here')
        dfx1=interpol(dfx,ux,uy)
        dfy1=interpol(dfy,ux,uy)
        df=[dfx1,dfy1]
        psi1=psi(fu,g,lamb,mu,ux,uy)
        b=JTPsi(psi1,df,lamb,mu)
        [descentx,descenty]=CGSolve(descentx,descenty,lamb,mu,b,epsi,df)
        ux,uy,step=linesearch(ux,uy,step,descentx,descenty,obj,f,g,lamb,mu)
        step_list.append(step)
        # Display
        if (niter % 3 ==0) :
            print('iteration :',niter,' cost function :',obj,'step :',step)
    return ux,uy,np.array(CF),np.array(step_list),niter
```

Q9. Tester le nouvel algorithme et comparer sa vitesse de convergence avec celle de l'algorithme de gradient.

Afin que l'algorithme des moindres carrés converge avec un nombre d'itéarations le plus minimal possible,j'ai décidé de créér une fonction qui permet de calculer les valeurs de λ et μ les plus optimales possibles. Pour cela , pour différentes valeurs de λ ,j'ai calculé le nombre d'itérations associé en Log.

Entrée [18]:

```
nitermax=5000
epsi=0.1
step0=0.01
v_lamb=np.linspace(0,60,30)
v_mu=np.copy(v_lamb)
niter=np.zeros(30)
for i in range (30):
    ux,uy,CF,step,niter[i]=RecalageGN(f,g,v_lamb[i],v_mu[i],nitermax,step0,epsi)
```

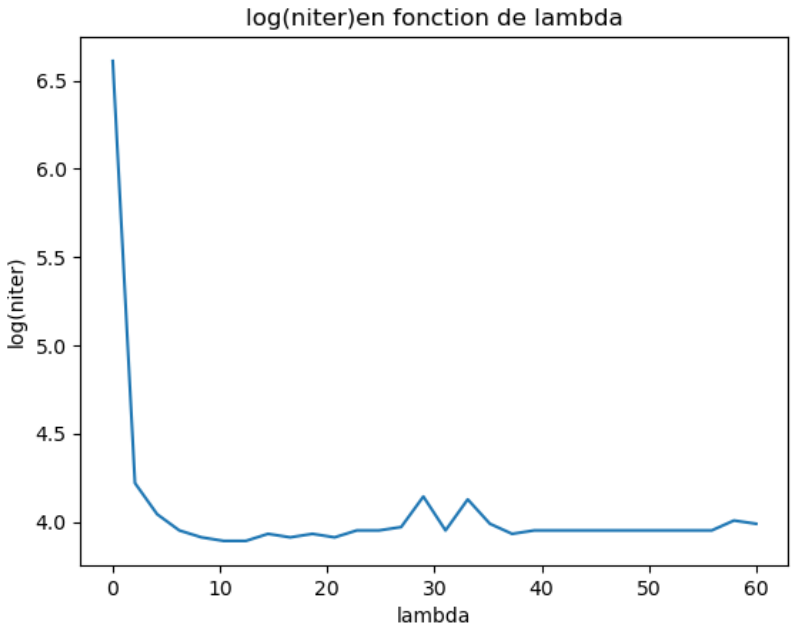
```
iteration : 3  cost function : 18.73324891787389 step : 0.08
iteration : 6  cost function : 15.390488694830228 step : 0.64
iteration : 9  cost function : 3.581852271528054 step : 5.12
iteration : 12  cost function : 1.6262697498641454 step : 5.12
iteration : 15  cost function : 1.291069312350728 step : 10.24
iteration : 18  cost function : 1.2220570540253008 step : 5.12
iteration : 21  cost function : 1.156019983316257 step : 10.24
iteration : 24  cost function : 1.129399108665922 step : 10.24
iteration : 27  cost function : 1.110816675420118 step : 10.24
iteration : 30  cost function : 1.0962568942912683 step : 10.24
iteration : 33  cost function : 1.086495035159741 step : 5.12
iteration : 36  cost function : 1.0730801953354112 step : 5.12
iteration : 39  cost function : 1.0635328650999776 step : 5.12
iteration : 42  cost function : 1.0552463888102417 step : 10.24
iteration : 45  cost function : 1.050322878697209 step : 10.24
iteration : 48  cost function : 1.046099117142508 step : 10.24
iteration : 51  cost function : 1.0424646481708506 step : 10.24
iteration : 54  cost function : 1.0393617573366596 step : 10.24
iteration : 57  cost function : 1.0367334809515498 step : 10.24
```

On trace ci-dessous la fonction log(niter) en fonction de lambda.

Entrée [19]:

```
plt.show()
plt.title('log(niter)en fonction de lambda')
plt.xlabel('lambda')
plt.ylabel('log(niter)')
plt.plot(v_lamb,np.log(niter))
print(niter)
```

```
[743.  68.  57.  52.  50.  49.  49.  51.  50.  51.  50.  52.  52.  53.
  63.  52.  62.  54.  51.  52.  52.  52.  52.  52.  52.  52.  52.  52.
  55.  54.]
```



Entrée [20]:

```
k=np.argmin(niter)
print(k, 'est l''indice qui correspond à la valeur de niter minimale')
print(niter[k],'représente la valeur de niter minimale')
lambda_opt=v_lamb[k]
print(lambda_opt,'représente la valeur de lambda optimale')
```

```
5 est lindice qui correspond à la valeur de niter minimale
49.0 représente la valeur de niter minimale
10.344827586206897 représente la valeur de lambda optimale
```

On remarque que l'algorithme converge lorsque le nombre d'itérations est égale à 49. La valeur de λ associé est égale à 10.344827586206897 , c'est bel et bien la valeur optimale de λ.

Entrée [21]:

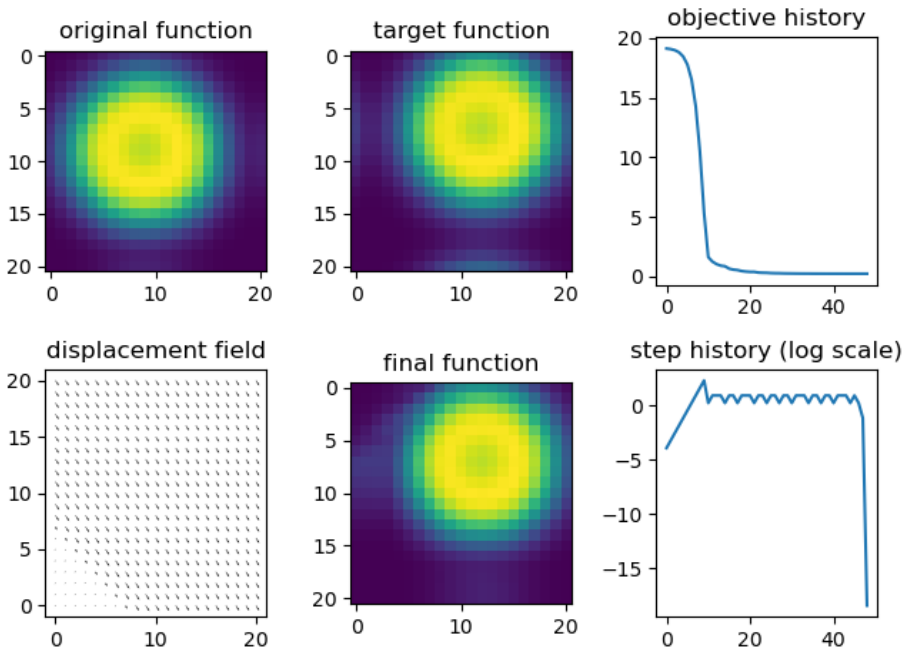
```
epsi=0.1
nitermax=100
lamb=10.344827586206897
mu=10.344827586206897
ux,uy,CF,step,niter=RecalageGN(f,g,lamb,mu,nitermax,step0,epsi)
```

```
iteration : 3  cost function : 18.980738143768278 step : 0.08
iteration : 6  cost function : 17.77995806584432 step : 0.64
iteration : 9  cost function : 10.48750056552422 step : 5.12
iteration : 12  cost function : 1.2581385111202377 step : 2.56
iteration : 15  cost function : 0.8733688909132206 step : 1.28
iteration : 18  cost function : 0.5515327259787275 step : 1.28
iteration : 21  cost function : 0.3995331304334949 step : 2.56
iteration : 24  cost function : 0.32636696278754185 step : 2.56
iteration : 27  cost function : 0.28871259437604946 step : 2.56
iteration : 30  cost function : 0.26969172596087443 step : 2.56
iteration : 33  cost function : 0.25955411629296304 step : 2.56
iteration : 36  cost function : 0.2544248218702207 step : 2.56
iteration : 39  cost function : 0.25171996219395804 step : 2.56
iteration : 42  cost function : 0.25044111273447456 step : 2.56
iteration : 45  cost function : 0.2500618820130859 step : 1.28
iteration : 48  cost function : 0.2496309942056984 step : 0.32
```

On remarque que l'algorithme des moindres carrés converge plus rapidement comparant à l'algorithme de descente de gradient. En effet ,le nombre d'itérations de l'algorithme des moindres carrés est égale à 48 qui est très inférieure au nombre d'itérations de l'algorithme de descente de gradient qui est égale à 2589.

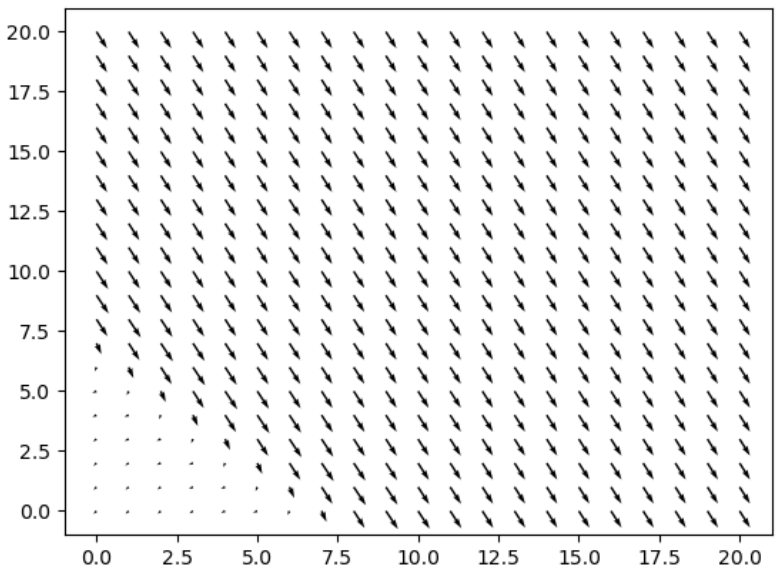
Entrée [22]:

```
fig, ax = plt.subplots(2,3)
ax[0,0].imshow(f)
ax[0,0].set_title('original function')
ax[0,1].imshow(g)
ax[0,1].set_title('target function')
ax[1,0].quiver(ux,uy)
ax[1,0].set_title('displacement field')
ax[1,1].imshow(interpol(f,ux,uy))
ax[1,1].set_title('final function')
ax[0,2].plot(CF)
ax[0,2].set_title('objective history')
ax[1,2].plot(np.log(step))
ax[1,2].set_title('step history (log scale)')

plt.tight_layout()
plt.show()
```



Entrée [23]:

```
plt.quiver(ux, uy)
plt.show()
```



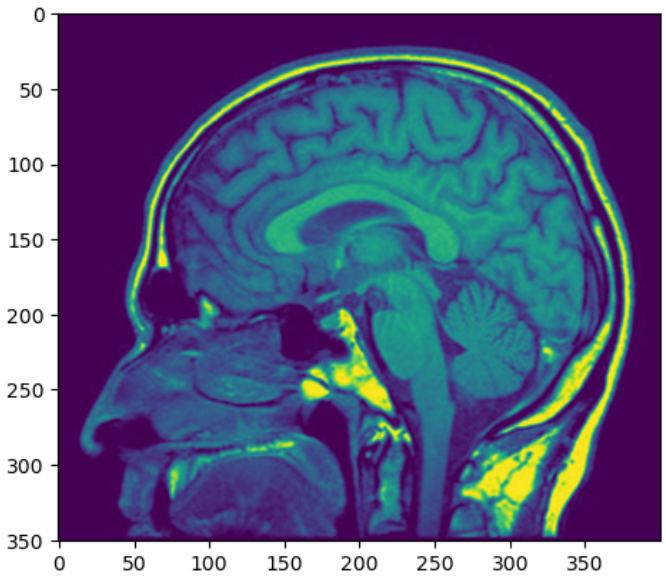
Le graphe ci-dessus contenant l'ensemble des vecteurs (ux,uy) résume le déplacement de notre fonction. Cela correspond bien aux graphes de la fonction originale et la fonction objective. En effet, on a bien qu'en bas à gauche, l'angle ne change pas d'une image à une autre (vecteur nul dans le graphe de quiver) cependant on note des vecteurs non nuls sur le reste du graphe de quiver, ceci explique bien le déplacement de l'objet entre les deux images.

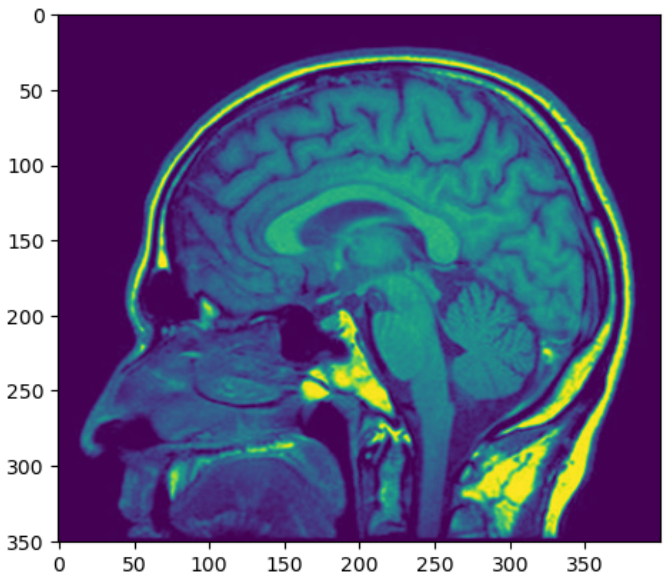
5. Jeu des différences

Maintenant que vous avez implémenté et testé les deux algorithmes sur l'image-jouet proposée, voyons que cela donne sur une image IRM d'un cerveau. Saurez-vous détecter les différences/déplacements entre les deux images ?

Entrée [24]:

```
im1=Image.open('IRM1.png')
im2=Image.open("IRM2.png")
plt.imshow(plt.imread('IRM1.png'))
plt.show()
plt.imshow(plt.imread('IRM2.png'))
plt.show()
```





Entrée [25]:

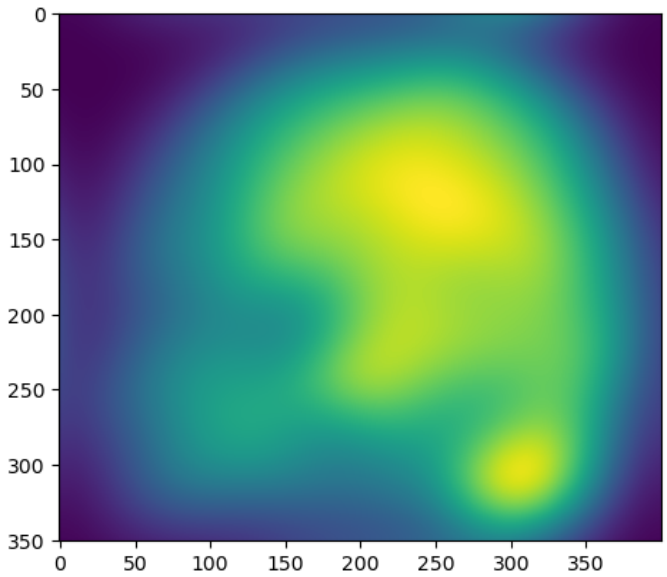
```
[n,m]=im1.size
sigma=0.2
[X,Y]=np.meshgrid(np.linspace(-1,1,n),np.linspace(-1,1,m), indexing='xy')
Z=np.sqrt(X*X+Y*Y)
G=np.fft.fftshift(np.exp(-(X**2+Y**2)/sigma**2))
f=np.real(np.fft.ifft2(np.fft.fft2(G)*np.fft.fft2(im1)))
g=np.real(np.fft.ifft2(np.fft.fft2(G)*np.fft.fft2(im2)))
f=f/np.max(f)
g=g/np.max(g)
```

Entrée [26]:

```
plt.imshow(f)
```

Out[26]:

```
<matplotlib.image.AxesImage at 0x7fae982dd340>
```

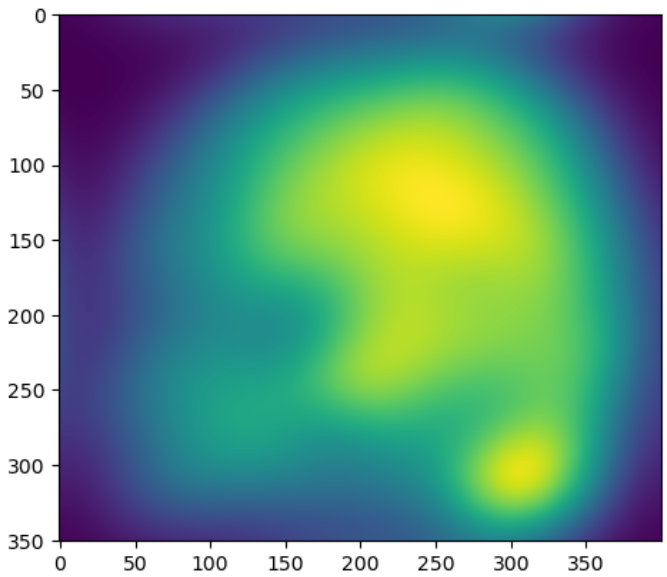


Entrée [27]:

```
plt.imshow(g)
```

Out[27]:

```
<matplotlib.image.AxesImage at 0x7faea8092be0>
```



Entrée [28]:

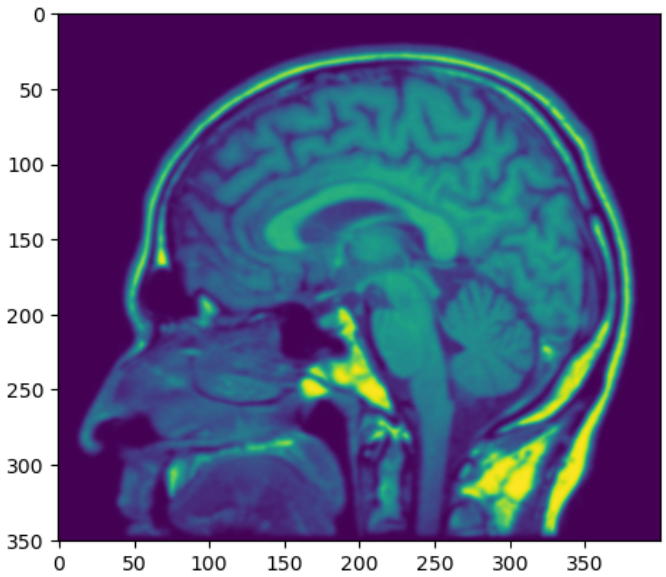
```
[n,m]=im1.size
sigma=0.01
[X,Y]=np.meshgrid(np.linspace(-1,1,n),np.linspace(-1,1,m), indexing='xy')
Z=np.sqrt(X*X+Y*Y)
G=np.fft.fftshift(np.exp(-(X**2+Y**2)/sigma**2))
f=np.real(np.fft.ifft2(np.fft.fft2(G)*np.fft.fft2(im1)))
g=np.real(np.fft.ifft2(np.fft.fft2(G)*np.fft.fft2(im2)))
f=f/np.max(f)
g=g/np.max(g)
```

Entrée [29]:

```
plt.imshow(f)
```

Out[29]:

```
<matplotlib.image.AxesImage at 0x7faecb46a1c0>
```

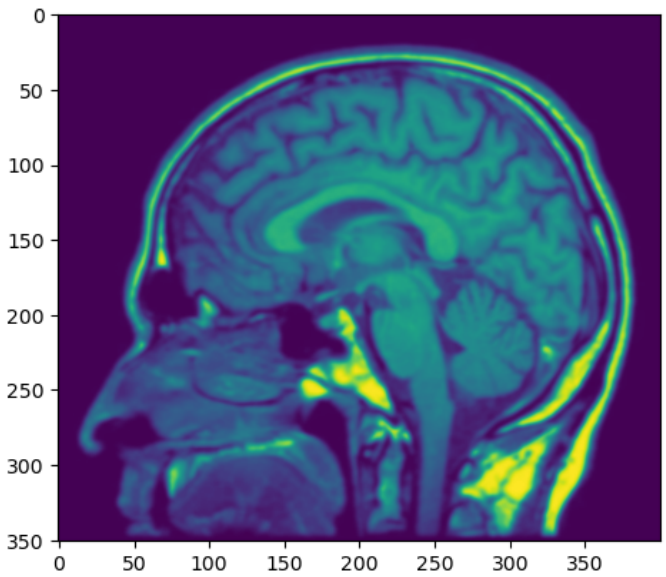


Entrée [30]:

```
plt.imshow(g)
```

Out[30]:

```
<matplotlib.image.AxesImage at 0x7faecc307b50>
```



Tout d'abord, on choisit l'algorithme des moindres carrés afin de minimiser le nombre d'itérations. Dans le cas de l'algorithme de descente de gradient, cela peut prendre un temps exorbitant afin d'assurer la convergence.

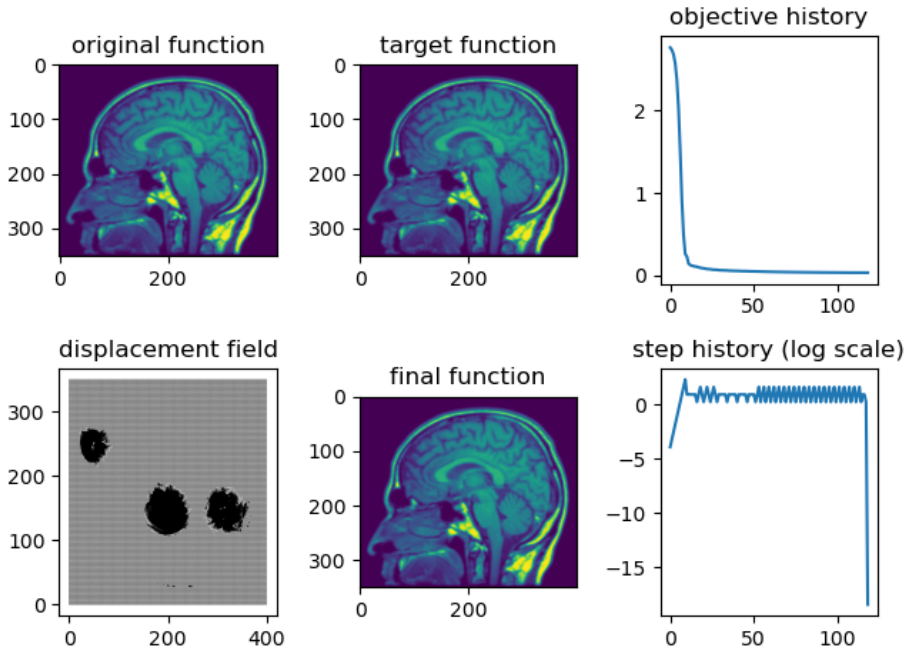
Entrée [31]:

```
epsi=0.01
step0=0.01
nitermax=500
lamb=0.0001
mu=0.0001
ux,uy,CF,step,niter=RecalageGN(f,g,lamb,mu,nitermax,step0,epsi)
```

```
iteration : 3  cost function : 2.6839462465073978 step : 0.08
iteration : 6  cost function : 2.0284033105056767 step : 0.64
iteration : 9  cost function : 0.4869730160177394 step : 5.12
iteration : 12  cost function : 0.14085249407064304 step : 2.56
iteration : 15  cost function : 0.11114494221439977 step : 2.56
iteration : 18  cost function : 0.09599077292026556 step : 2.56
iteration : 21  cost function : 0.08524067632652496 step : 1.28
iteration : 24  cost function : 0.07494773124600218 step : 2.56
iteration : 27  cost function : 0.06839865099956086 step : 5.12
iteration : 30  cost function : 0.06304705330287422 step : 2.56
iteration : 33  cost function : 0.05992059326327939 step : 2.56
iteration : 36  cost function : 0.05680617009055569 step : 2.56
iteration : 39  cost function : 0.05453130160712284 step : 2.56
iteration : 42  cost function : 0.05219698175710828 step : 2.56
iteration : 45  cost function : 0.0504764877290117 step : 2.56
iteration : 48  cost function : 0.048622042171788143 step : 2.56
iteration : 51  cost function : 0.047285060553058866 step : 2.56
iteration : 54  cost function : 0.04575826567442521 step : 5.12
iteration : 57  cost function : 0.04423446554510761 step : 5.12
iteration : 60  cost function : 0.04289661089552428 step : 5.12
iteration : 63  cost function : 0.04172232128468914 step : 5.12
iteration : 66  cost function : 0.040667382445765024 step : 5.12
iteration : 69  cost function : 0.039717800418627 step : 5.12
iteration : 72  cost function : 0.03885101466387786 step : 5.12
iteration : 75  cost function : 0.038063186722724254 step : 5.12
iteration : 78  cost function : 0.03733721874404187 step : 5.12
iteration : 81  cost function : 0.03667072269523035 step : 5.12
iteration : 84  cost function : 0.03605318902864886 step : 5.12
iteration : 87  cost function : 0.03548599699444661 step : 5.12
iteration : 90  cost function : 0.03496226155423998 step : 5.12
iteration : 93  cost function : 0.03448201993166862 step : 5.12
iteration : 96  cost function : 0.034037257722704994 step : 5.12
iteration : 99  cost function : 0.033626532262733176 step : 5.12
iteration : 102  cost function : 0.03324340270993657 step : 5.12
iteration : 105  cost function : 0.03288837562762685 step : 5.12
iteration : 108  cost function : 0.03256368590798059 step : 5.12
iteration : 111  cost function : 0.03229706787543414 step : 5.12
iteration : 114  cost function : 0.03207509024548387 step : 5.12
iteration : 117  cost function : 0.0319044489989712 step : 2.56
```

Entrée [32]:

```
fig, ax = plt.subplots(2,3)
ax[0,0].imshow(f)
ax[0,0].set_title('original function')
ax[0,1].imshow(g)
ax[0,1].set_title('target function')
ax[1,0].quiver(ux,uy)
ax[1,0].set_title('displacement field')
ax[1,1].imshow(interpol(f,ux,uy))
ax[1,1].set_title('final function')
ax[0,2].plot(CF)
ax[0,2].set_title('objective history')
ax[1,2].plot(np.log(step))
ax[1,2].set_title('step history (log scale)')

plt.tight_layout()
plt.show()
```



En choisissant des valeurs de lambda et de mu élevées (par exemple: λ=μ=8), on remarque que même avec l'algorithme des moindres carrés (qui converge plus rapidement que celui de descente de gradient) , cela prend aussi un temps énorme ainsi qu'un nombre d'itérations assez conséquent avant d'assurer la convergence. C'est pour cela que j'ai décidé de choisir des valeurs λ=μ=0.0001. La valeur de σ agit aussi sur la pertinence des résultats obtenus. En effet , pour σ=0.2, les deux images f et g sont presque identiques et on observe une tâche colorée et floue au niveau du crâne.Cela ne nous aide pas à determiner avec precision les endroits là où a lieu les changements. Cependant, en réduisant la valeur de σ à 0.01, on pourra distinguer les différences entre les deux images. Cela est bel et bien représenté dans la figure 'displacement filed'. En effet les tâches en noir représentent les endroits où on a eu du changement (différences entre la fonction initiale et finale). De plus , en réduisant avec précaution la valeur de ϵ, le nombre d'itérations diminue par conséquent (il faut toujours avoir un nombre d'itération suffisant pour assurer la convergence).