Proyecto2_DanielVallejo_ImágenesBiomédicas

April 17, 2022

1 Proyecto 2. Estimación de tensores de difusión

1.1 Daniel Vallejo Aldana // Procesamiento de imágenes biomédicas

Intorducción

La difusión de protones de hidrógeno es un fenómeno ampliamente utilizado en resonancia magnética. Dicha técnica determina el cambio de posición de los protones de hidrógeno a través de las diferentes direcciones g que corresponden a la dirección del gradiente magnético activado durante un determinado tiempo δ.

Para cada boxel, el valor de la señal de resonancia en la dirección g está dado de la siguiente forma

$$S_g = S_0 exp^{-bg^t Dg}$$

Donde el parámetro b depende de parámetros de máquina δ (Tiempo de encendido de gradiente), Δ (Tiempo del experimento, es decir, el tiempo en el que se deja mover a las partículas) y G. Este último valor hubo que convertirlo a T/mm^2 por la convención de unidades usadas aquí.

De lo anterior, se quiere estimar el tensor de difusión D que corresponde a una matriz de 9×9, la cual es simétrica y definida positiva es decir que cumple $x^t Dx > 0$ para $x \neq 0$. Sea ∇ la matriz de gradientes donde su valor de b es un determinado valor en este caso es 700. Entonces el sistema de ecuaciones a resolver es de la forma.

$$\frac{\log(S_\nabla / S_0)}{-\overrightarrow{b}} = \mathbf{G}d$$

Donde $\mathbf{G}$ es una matriz calculada de acuerdo a la forma descrita en las notas. Con los valores $d \in \mathbb{R}^6$ podemos construir nuestra matriz $D \in Mat_{3\times 3}(\mathbb{R})$ que corresponde al tensor de difusión.

1.1.1 1.Ejecutar y entender el código de dipy para el cálculo de tensores de difusión, asegurándose de entender todas las variables presentes en especial las variables gtab

[1]:
```
import numpy as np
from dipy.io.image import load_nifti, save_nifti,load_nifti_data
from dipy.io.gradients import read_bvals_bvecs
from dipy.core.gradients import gradient_table
#Algoritmos de reconstrucción de datos crudos
import dipy.reconst.dti as dti
```

```
#Precargar datos de una base de datos ya construida
from dipy.data import get_fnames
import matplotlib.pyplot as plt
from numpy import array
import math
from typing import Union,Optional
from dipy.data import get_sphere
from dipy.viz import window, actor
from dipy.segment.mask import median_otsu
from dipy.reconst.dti import fractional_anisotropy, color_fa
```

[2]:
```
hardi_fname, hardi_bval_fname, hardi_bvec_fname = get_fnames('stanford_hardi')
```

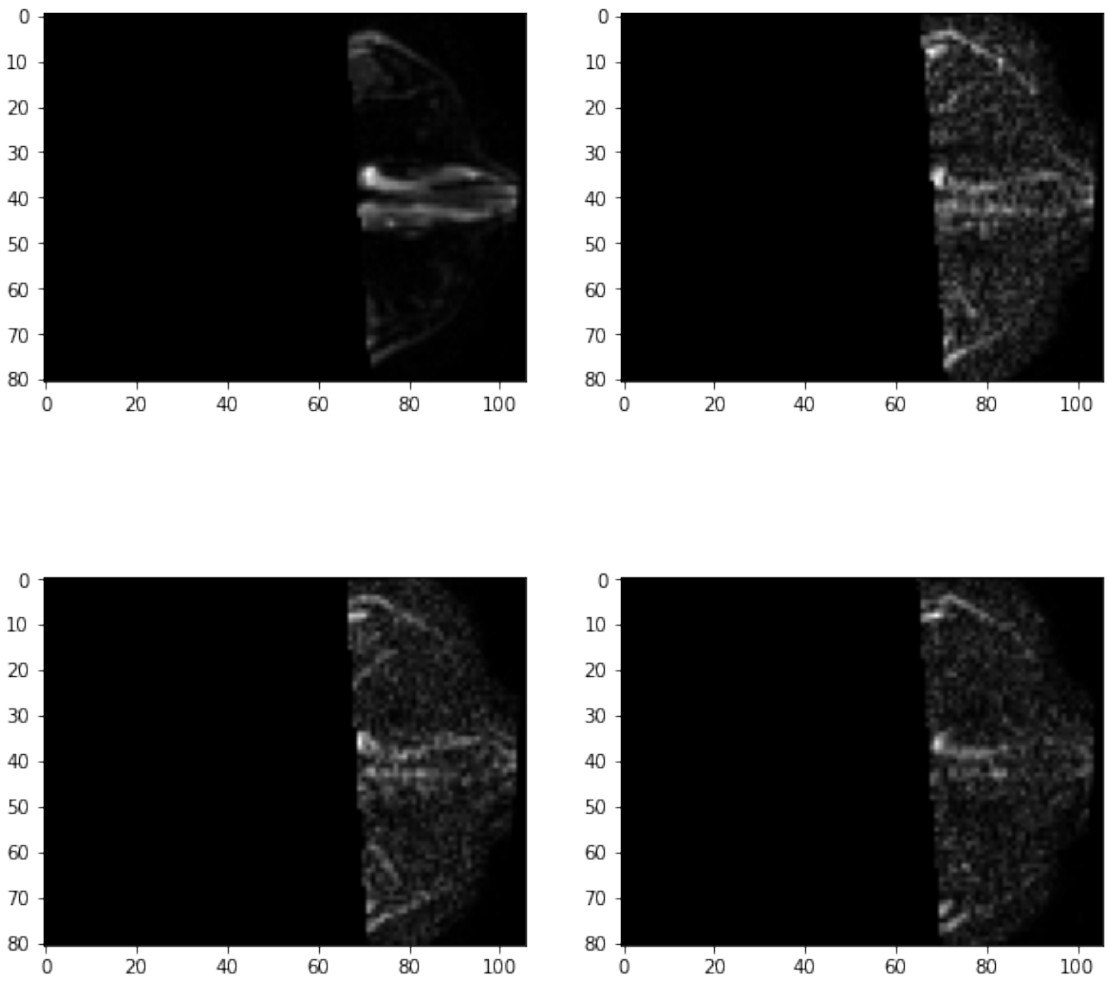
[3]:
```
data, affine = load_nifti(hardi_fname)
bvals, bvecs = read_bvals_bvecs(hardi_bval_fname, hardi_bvec_fname)
gtab = gradient_table(bvals, bvecs)
print('data.shape (%d, %d, %d, %d)' % data.shape)
```

```
data.shape (81, 106, 76, 160)
```

La anterior forma de los datos corresponde a 76 tajadas del cerebro con 160 direcciones de gradiente diferentes, el tamaño de una imágen es de 81×106. Veamos a continuación algunos ejemplos de una tajada del cerebro en diferentes direcciones de gradiente

[4]:
```
fig,ax=plt.subplots(2,2,figsize=(10,10))
ax[0][0].imshow(data[:,:,0,0],cmap='gray')
ax[0][1].imshow(data[:,:,0,10],cmap='gray')
ax[1][0].imshow(data[:,:,0,50],cmap='gray')
ax[1][1].imshow(data[:,:,0,100],cmap='gray')
fig.suptitle("Misma tajada con diferentes direcciones de gradiente",fontsize=20)
```

[4]:
```
Text(0.5, 0.98, 'Misma tajada con diferentes direcciones de gradiente')
```



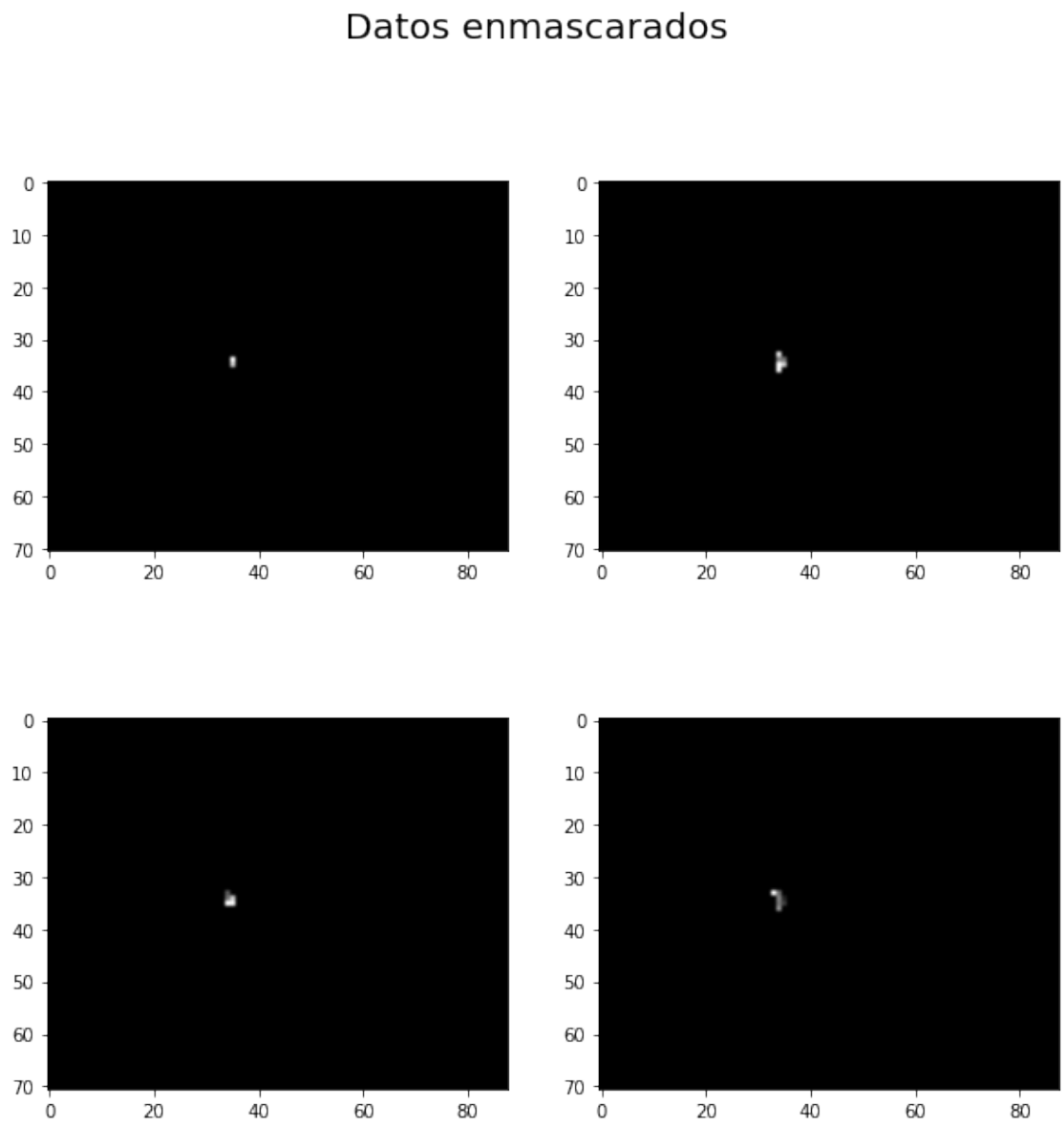
[5]:
```
#Enmascaramos el conjunto de datos para evitar ajustar tensores de difusión al␣
 ↪fondo de la imagen
maskdata, mask = median_otsu(data, vol_idx=range(10, 50),␣
 ↪median_radius=3,numpass=1, autocrop=True, dilate=2)
print('maskdata.shape (%d, %d, %d, %d)' % maskdata.shape)
```

```
maskdata.shape (71, 88, 62, 160)
```

[6]:
```
fig,ax=plt.subplots(2,2,figsize=(10,10))
ax[0][0].imshow(maskdata[:,:,0,0],cmap='gray')
ax[0][1].imshow(maskdata[:,:,0,10],cmap='gray')
```

```
ax[1][0].imshow(maskdata[:,:,0,50],cmap='gray')
ax[1][1].imshow(maskdata[:,:,0,100],cmap='gray')
fig.suptitle("Datos enmascarados",fontsize=20)
#Nótese que el ajuste por tensores de difusión se dice que se hace solamente en␣
 ↪aquellos
#boxeles de materia blanca. Esta máscara no funciona sobre materia gris
```

[6]: Text(0.5, 0.98, 'Datos enmascarados')



[7]:
```
#Ajuste de los tensores de difusión
tenmodel = dti.TensorModel(gtab)
```

```
tenfit = tenmodel.fit(maskdata)
```

[8]:
```
#Algunas dimensiones importantes
print(gtab.gradients.shape) #La magnitud de los gradientes a lo largo de las␣
↪dimensiones correspondientes
print(tenfit.shape) #Información para todos los boxeles del conjunto de datos
```

```
(160, 3)
(71, 88, 62)
```

[9]:
```
print('Computing anisotropy measures (FA, MD, RGB)')
FA = fractional_anisotropy(tenfit.evals)
```

```
Computing anisotropy measures (FA, MD, RGB)
```

[10]:
```
FA[np.isnan(FA)] = 0 #Aquellos valores del background con resultado NAN
```

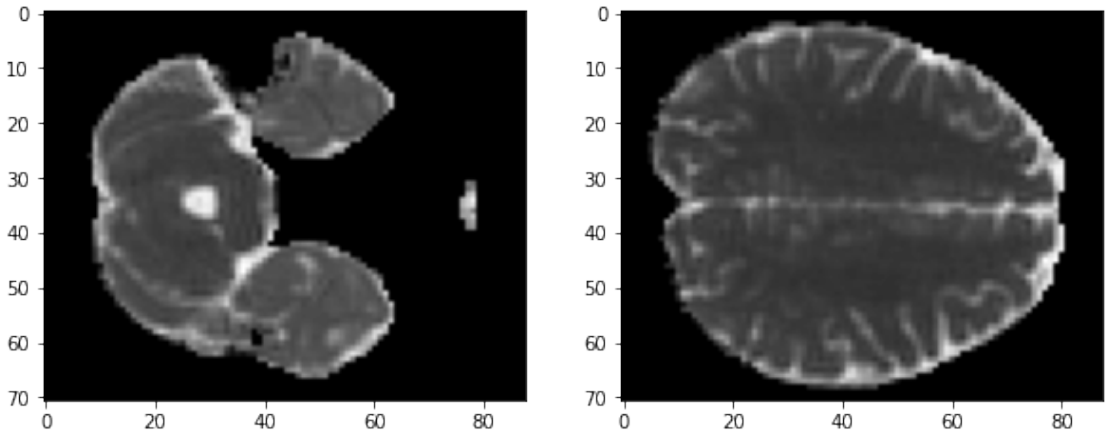
Procedemos a una visualización de los datos con los que calculamos de los tensores de difusión

[11]:
```
MD1 = dti.mean_diffusivity(tenfit.evals)
FA = np.clip(FA, 0, 1)
RGB = color_fa(FA, tenfit.evecs)
```

2 Algunas visualizaciones de difusión media

[12]:
```
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(MD1[:,:,10],cmap='gray')
ax[1].imshow(MD1[:,:,40],cmap='gray')
```

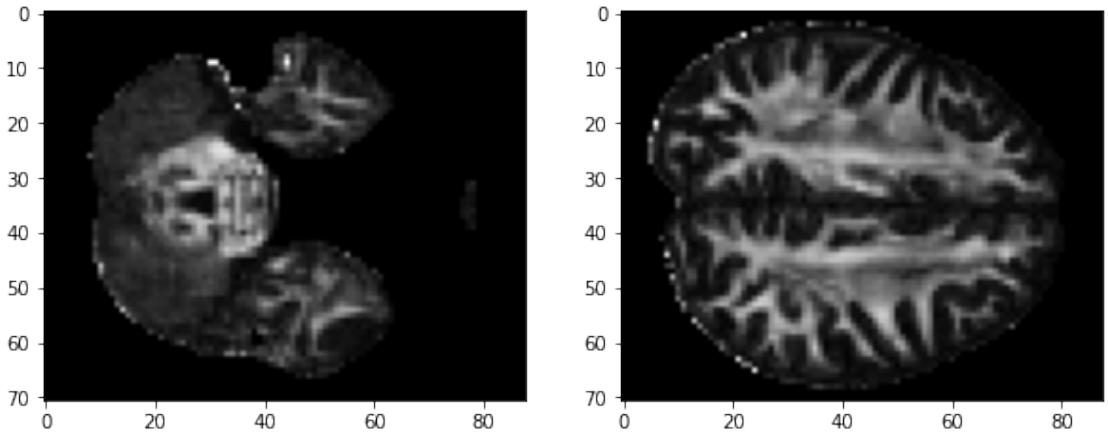
[12]:
```
<matplotlib.image.AxesImage at 0x7f6f5444fe20>
```



3 Visualización con Fractional Anisotropy

[13]:
```
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(FA[:,:,10],cmap='gray')
ax[1].imshow(FA[:,:,40],cmap='gray')
```

[13]:
```
<matplotlib.image.AxesImage at 0x7f6f543c2e50>
```



[14]:
```
execute=False
if execute:
    sphere = get_sphere('repulsion724')
    # Enables/disables interactive visualization
    interactive = False
    ren = window.Scene()
    evals = tenfit.evals[13:43, 44:74, 28:29]
    evecs = tenfit.evecs[13:43, 44:74, 28:29]
    cfa = RGB[13:43, 44:74, 28:29]
    cfa /= cfa.max()
    ren.add(actor.tensor_slicer(evals, evecs, scalar_colors=cfa,␣
 ↪sphere=sphere,scale=0.3))
    print('Saving illustration as tensor_ellipsoids.png')
    window.record(ren, n_frames=1, out_path='tensor_ellipsoids.png',size=(600,␣
 ↪600))
    ren.clear()
    tensor_odfs = tenmodel.fit(data[20:50, 55:85, 38:39]).odf(sphere)
    odf_actor = actor.odf_slicer(tensor_odfs, sphere=sphere, scale=0.
 ↪5,colormap=None)
    ren.add(odf_actor)
    print('Saving illustration as tensor_odfs.png')
    window.record(ren, n_frames=1, out_path='tensor_odfs.png', size=(600, 600))
```

4 Parte 2: Adaptar el tutorial de dipy a los datos de humano proporcionados en la tarea

[15]:
```
#Lectura de los valores para los gradeintes del archivo scheme
info=[]
with open('NODDI_DWI.scheme','rb') as f:
    data=f.readlines()
    f.close()
for i in range(1,len(data)):
    info.append(np.asarray(data[i].split(),dtype=np.float32))
info=np.stack(info)
```

[16]:
```
def compute_bval(G:float,DELTA:float,delta:float):
    gmr = 2*math.pi*42.576*1e6
    bval = gmr**2*G**2* delta**2 * (DELTA-delta/3)
    return round(bval*1e-6,0)
def compute_bvals_bvecs(informacion:array,threshold:Union[float,None]):
    bvecs=np.zeros((informacion.shape[0],3))
    bvals=np.zeros(informacion.shape[0])
    indices=None
    for j in range(informacion.shape[0]):
        ␣
 ↪bvals[j]=compute_bval(informacion[j][3],informacion[j][4],informacion[j][5])
        bvecs[j]=informacion[j,:3]
    if threshold is not None:
        indices=np.where((bvals<=threshold+10))
        bvals=bvals[indices]
        bvecs=bvecs[indices]
        indices=indices
    return bvecs,bvals,indices
```

[17]:
```
bvecs,bvals,indices=compute_bvals_bvecs(info,threshold=700)
```

[18]:
```
#Los bvalores se ven bien y como se dice en la tarea solo tomamos los de␣
 ↪aproximadamente 700
bvals
```

[18]:
```
array([  0., 706., 706., 706., 706., 706., 706., 706., 706.,   0., 706.,
       706., 706., 706., 706., 706., 706., 706.,   0., 706., 706., 706.,
       706., 706., 706., 706., 706.,   0.,   0.,   0.,   0.,   0.,   0.])
```

[19]:
```
#creación de la tabla de gradiente
Gtab=gradient_table(bvals,bvecs)
```

[20]:
```
#Lectura de los datos de humano dados
datos,affine=load_nifti("NODDI_DWI.nii")
```

```
mascara=load_nifti_data("BRAIN_MASK.nii")
```

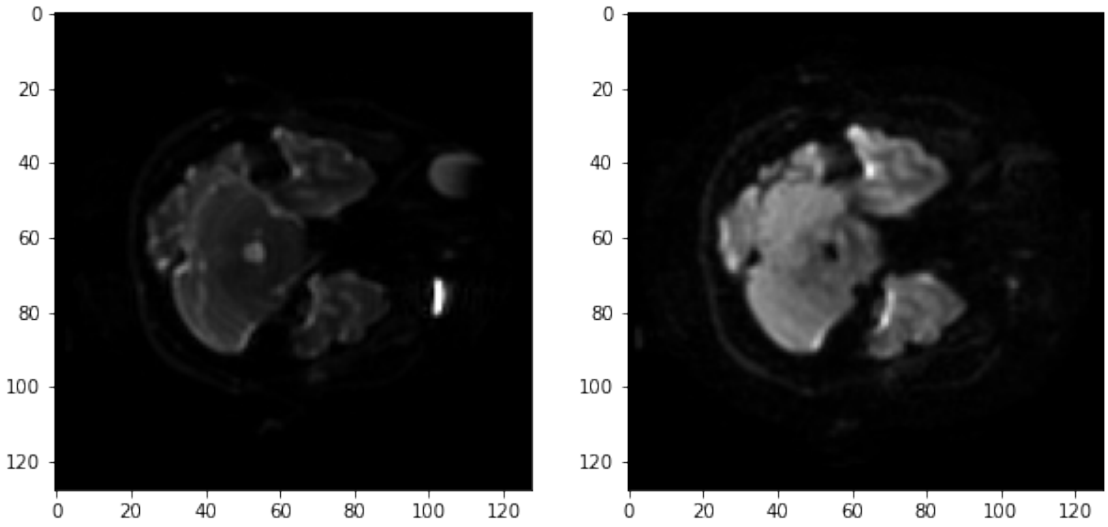
[21]:
```
datos=datos[:,:,:,:,indices].squeeze(3)
```

[22]:
```
datos=datos.squeeze(3)
print(datos.shape)
```

```
(128, 128, 50, 33)
```

[23]:
```
#Una breve visualización de los datos
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(datos[:,:,10,0],cmap='gray')
ax[1].imshow(datos[:,:,10,10],cmap='gray')
```

[23]:
```
<matplotlib.image.AxesImage at 0x7f6f64a1b400>
```



[24]:
```
#En el tutorial utilizan el filtro de Otsu que se vió en la primera parte del␣
 ↪curso para enmascarar los datos,
#usaremos la misma forma de enmascarar que en el tutorial
maskdata, mask = median_otsu(datos, vol_idx=range(0, 33),␣
 ↪median_radius=3,numpass=1, autocrop=True, dilate=2)
print('maskdata.shape (%d, %d, %d, %d)' % maskdata.shape)
```

```
maskdata.shape (74, 102, 50, 33)
```

[25]:
```
tenmodel = dti.TensorModel(Gtab)
tenfit = tenmodel.fit(maskdata)
```

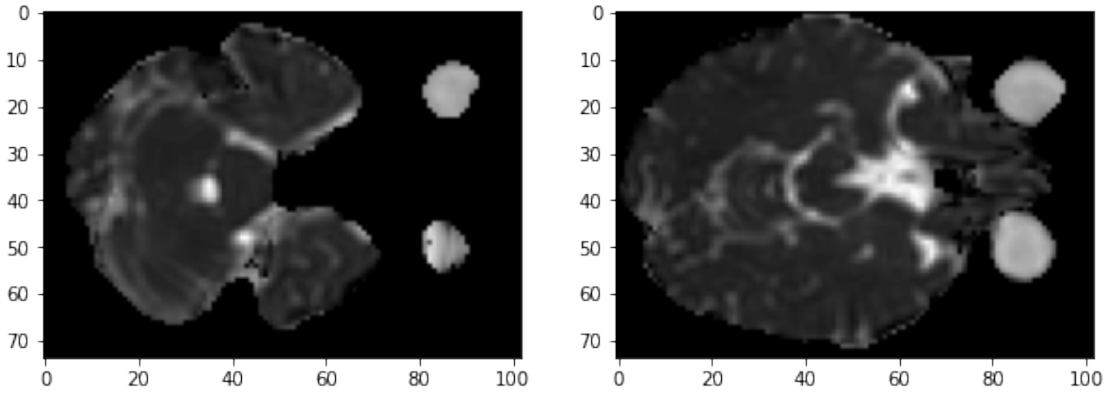
[26]:
```
print('Computing anisotropy measures (FA, MD, RGB)')
FA = fractional_anisotropy(tenfit.evals)
FA[np.isnan(FA)] = 0 #Aquellos valores del background con resultado NAN
MD1 = dti.mean_diffusivity(tenfit.evals)
FA = np.clip(FA, 0, 1)
RGB = color_fa(FA, tenfit.evecs)
```

```
Computing anisotropy measures (FA, MD, RGB)
```

5 Visualización de difusión media en este conjunto de datos

[27]:
```
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(MD1[:,:,10],cmap='gray')
ax[1].imshow(MD1[:,:,15],cmap='gray')
```

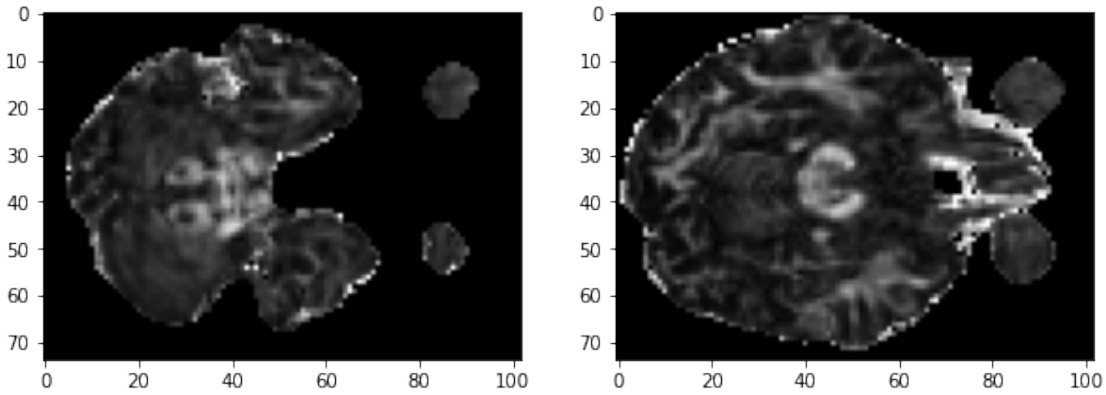
[27]:
```
<matplotlib.image.AxesImage at 0x7f6f649c81c0>
```



6 Visaulización de las imágenes con FA

[28]:
```
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(FA[:,:,10],cmap='gray')
ax[1].imshow(FA[:,:,15],cmap='gray')
```

[28]:
```
<matplotlib.image.AxesImage at 0x7f6f631859a0>
```



[29]:
```
execute=False
if execute:
    sphere = get_sphere('repulsion724')
    # Enables/disables interactive visualization
    interactive = False
    ren = window.Scene()
    evals = tenfit.evals[13:60, 20:80, 15:16]
    evecs = tenfit.evecs[13:60, 20:80, 15:16]
    cfa = RGB[13:60, 20:80, 15:16]
    cfa /= cfa.max()
    ren.add(actor.tensor_slicer(evals, evecs, scalar_colors=cfa,␣
 ↪sphere=sphere,scale=0.3))
    window.record(ren, n_frames=1, out_path='tensor_ellipsoids_custom_data.
 ↪png',size=(600, 600))
    ren.clear()
    tensor_odfs = tenmodel.fit(datos[13:60, 20:80, 15:16]).odf(sphere)
    odf_actor = actor.odf_slicer(tensor_odfs, sphere=sphere, scale=0.
 ↪5,colormap=None)
    ren.add(odf_actor)
    window.record(ren, n_frames=1, out_path='tensor_odfs_custom_data.png',␣
 ↪size=(600, 600))
```

7 Parte 3. Construir el solver para poder ajustar el tensor de difusión por mínimos cuadrados

[67]:
```
class TensorInformation(object):
    def __init__(self,evals,evecs):
        self.evals=evals
        self.evecs=evecs
class DTEstimator:
    '''
```

```
    Clase para estimar el tensor de difusión mediante mínimos cuadrados usando␣
↪la fórmula dada
    en clase
    '''
    def __init__(self,bvals,bvecs,masked):
        '''
        Vamos a inicializar la clase de solución mediante los valores de b y␣
↪los vectores b, en
        las direcciones del gradiente
        '''
        self.gtab=gradient_table(bvals,bvecs) #Inicializamos la tabla de␣
↪gradeintes magnéticos
        self.masked_data=masked
        self.G,self.ind=self.compute_G_matrix()
        self.SquaredG=np.linalg.inv(np.dot(self.G.T,self.G))
        #La matriz G es una misma para todas las observaciones ya que solo␣
↪depende de los gradientes
        #Y esos gradientes no cambian de voxel en voxel
        self.tensors=np.ndarray(self.masked_data.shape[:3],dtype=np.object)
        self.FA=np.ndarray(self.masked_data.shape[:3],dtype='float32')
        self.MD=np.ndarray(self.masked_data.shape[:3],dtype='float32')
        self.fit_data()
    def compute_G_matrix(self):
        #Esta tabla es compartida con los cálculos
        indices=np.nonzero(self.gtab.bvals)
        G=np.zeros((indices[0].shape[0],6),dtype='float32')
        for i in range(G.shape[0]):
            #Por todo el espacio de renglones calculamos las entradas de la␣
↪matriz G
            if i in indices[0].tolist():
                g1,g2,g3=self.gtab.bvecs[i]
                G[i][0]=g1**2
                G[i][1]=2*g1*g2
                G[i][2]=2*g1*g3
                G[i][3]=g2**2
                G[i][4]=g2*g3
                G[i][5]=g3**2
        return G,indices[0]

    def estimate_dt_in_boxel(self,x:int,y:int,z:int):
        #Algun boxel en alguna de las tajadas que se tienen del cerebro
        S0=self.masked_data[x,y,z,0]
        #print(S0)
        #Estimamos vectores de Si de la siguiente forma
        #Solo se ajusta a los vectores no cero de la máscara cabe aclarar
        #print(S0)
        #print(self.gtab.bvals[self.ind])
```

```
        Si_hat=np.log(self.masked_data[x,y,z,self.ind]/S0)/(-1*self.gtab.
↪bvals[self.ind])
        SiSquared=np.dot(self.G.T,Si_hat)
        coefs=np.dot(self.SquaredG,SiSquared)
        d1,d2,d3,d4,d5,d6=coefs
        MR=np.zeros((3,3))
        MR[0][0]=d1
        MR[0][1]=MR[1][0]=d2
        MR[0][2]=MR[2][0]=d3
        MR[1][1]=d4
        MR[1][2]=MR[2][1]=d5
        MR[2][2]=d6
        vals,vecs=np.linalg.eig(MR)
        a=zip(vals,vecs)
        a=sorted(a,key=lambda a:a[0],reverse=True)
        vals,vecs=zip(*a)
        vals=np.array(vals)
        return vals,vecs
    def fit_data(self):
        corrupted=0
        for i in range(self.masked_data.shape[0]):
            for j in range(self.masked_data.shape[1]):
                for k in range(self.masked_data.shape[2]):
                    if self.masked_data[i,j,k,0]!=0:
                        try:
                            v,vec=self.estimate_dt_in_boxel(i,j,k)
                            self.tensors[i,j,k]=TensorInformation(v,vec)
                        except:
                            v=np.zeros(3)
                            vec=np.zeros((3,3))
                            self.tensors[i,j,k]=TensorInformation(v,vec)
                            corrupted+=1
                    else:
                        v=np.zeros(3)
                        vec=np.zeros((3,3))
                        self.tensors[i,j,k]=TensorInformation(v,vec)
                    self.FA[i,j,k]=fractional_anisotropy(self.tensors[i,j,k].
↪evals)
                    self.MD[i,j,k]=dti.mean_diffusivity(self.tensors[i,j,k].
↪evals)
        print("Corrupted Boxels: {}".format(corrupted))
```

[68]:
```
estimator=DTEstimator(bvals,bvecs,maskdata)
```

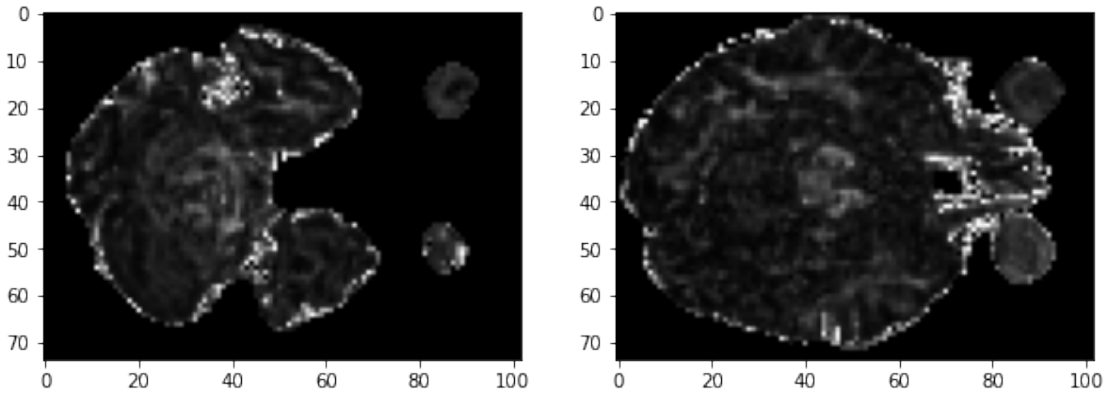
```
/tmp/ipykernel_4750/75207163.py:49: RuntimeWarning: divide by zero encountered
```

```
in log
  Si_hat=np.log(self.masked_data[x,y,z,self.ind]/S0)/(-1*self.gtab.bvals[self.in
d])

Corrupted Boxels: 6
```

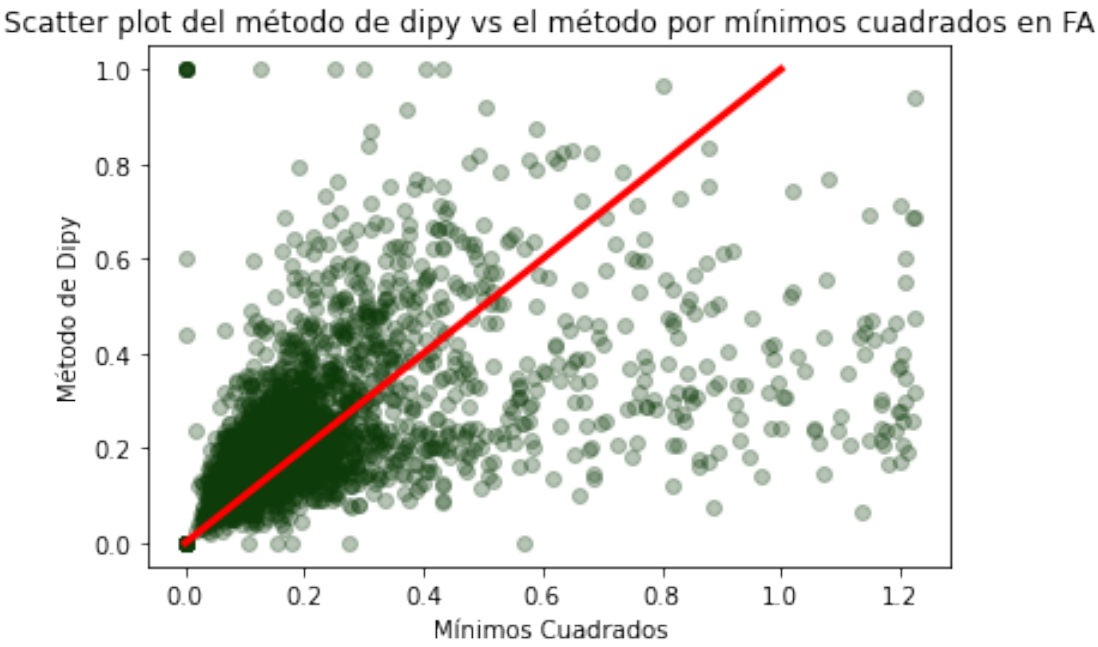
[69]:
```
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(estimator.FA[:,:,10],cmap='gray')
ax[1].imshow(estimator.FA[:,:,15],cmap='gray')
```

[69]: <matplotlib.image.AxesImage at 0x7f6f38379850>



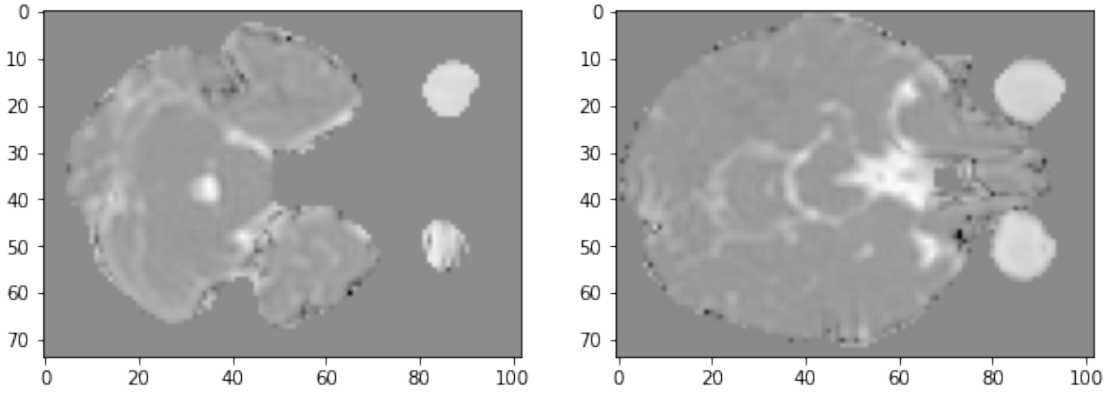
[70]:
```
x=estimator.FA[:,:,10].reshape(-1,)
y=FA[:,:,10].reshape(-1,)
plt.scatter(x,y,alpha=0.3,color='#0E3B0A')
plt.plot(np.linspace(0,1,num=1000),np.
 ↪linspace(0,1,num=1000),color='red',linewidth=3)
plt.title("Scatter plot del método de dipy vs el método por mínimos cuadrados␣
 ↪en FA")
plt.xlabel('Mínimos Cuadrados')
plt.ylabel('Método de Dipy')
```

[70]: Text(0, 0.5, 'Método de Dipy')



[71]:
```
fig,ax=plt.subplots(1,2,figsize=(10,10))
ax[0].imshow(estimator.MD[:,:,10],cmap='gray')
ax[1].imshow(estimator.MD[:,:,15],cmap='gray')
```

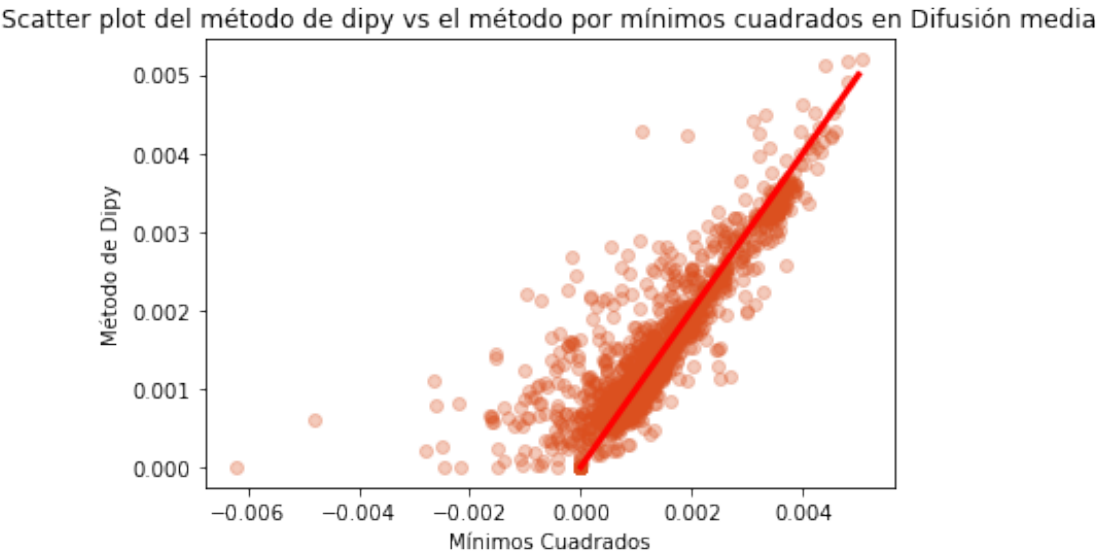
[71]:
```
<matplotlib.image.AxesImage at 0x7f6f42bc1610>
```



[72]:
```
#Revisemos ahora la difusión media
xm=estimator.MD[:,:,10].reshape(-1,)
ym=MD1[:,:,10].reshape(-1,)
plt.scatter(xm,ym,alpha=0.3,color='#DB501F')
```

```
plt.plot(np.linspace(0,0.005,num=1000),np.linspace(0,0.
↪005,num=1000),color='red',linewidth=3)
plt.title("Scatter plot del método de dipy vs el método por mínimos cuadrados␣
↪en Difusión media")
plt.xlabel('Mínimos Cuadrados')
plt.ylabel('Método de Dipy')
```

[72]:
```
Text(0, 0.5, 'Método de Dipy')
```



Es interesante notar en esta gráfica que los puntos no forman como tal una linea recta sobre la recta identidad, esto debido al factor de ruido ϵ que se le agrega a cada una de las señales en los boxeles.

[73]:
```
#Error cuadrático medio de las observaciones
from sklearn.metrics import mean_squared_error

print("Error cuadrático medio en Fractional Anisotropy: {}".
↪format(mean_squared_error(x,y)))
```

```
Error cuadrático medio en Fractional Anisotropy: 0.014114638675963354
```

Conclusiones y comentarios finales

De los experimentos realizados en el presente trabajo es posible concluir que el método por mínimos cuadrados es un buen método para estimar el tensor de difusión de un boxel utilizando varias dimensiones de gradiente. No obstante como es posible observar en la última figura, es posible ver que aunque los puntos se encuentran cercanos a la recta identidad, la presencia de ruido impide que dichos puntos formen una linea recta que es lo que se esperaría de los dos métodos.

7.1 Anexo 1: Algunas imágenes de los elipsoides

[37]:
```
from PIL import Image

p1=Image.open("tensor_ellipsoids_custom_data.png")
p2=Image.open("tensor_odfs_custom_data.png")
```

[42]:
```
f,ax=plt.subplots(1,2,figsize=(15,15))
ax[0].imshow(p1)
ax[1].imshow(p2)
```

[42]:
```
<matplotlib.image.AxesImage at 0x7f4fd78b6820>
```