Propuesta 2

June 14, 2021

1 Propuesta práctica 2

En esta práctica se debe implementar un corregistro entre una imagen DICOM que representa la resonancia magnética del cerebro de un paciente anonimizado y una imagen DICOM que representa la media ponderada de 152 resonancias magnéticas de cerebros.

A continuación se importan las librerías necesarias

[1]:
```
import glob
import pydicom

from matplotlib import pyplot as plt

import numpy as np
from skimage import measure
import math

from scipy.optimize import least_squares
from scipy import ndimage
```

A continuación se definen una serie de funciones para facilitar la programación del corregistro. Estas funciones han sido proporcionadas por el profesor durante el curso.

[2]:
```
def traslacion(punto, vector_traslacion):
    x, y, z = punto
    t_1, t_2, t_3 = vector_traslacion
    punto_transformado = (x+t_1, y+t_2, z+t_3)
    return punto_transformado


def rotacion_axial(punto, angulo_en_radianes, eje_traslacion):
    x, y, z = punto
    v_1, v_2, v_3 = eje_traslacion
    #   Vamos a normalizarlo para evitar introducir restricciones en el␣
 ↪optimizador
    v_norm = math.sqrt(sum([coord ** 2 for coord in [v_1, v_2, v_3]]))
    v_1, v_2, v_3 = v_1 / v_norm, v_2 / v_norm, v_3 / v_norm
    #   Calcula cuaternión del punto
    p = (0, x, y, z)
```

```
    #   Calcula cuaternión de la rotación
    cos, sin = math.cos(angulo_en_radianes / 2), math.sin(angulo_en_radianes /␣
↪2)
    q = (cos, sin * v_1, sin * v_2, sin * v_3)
    #   Calcula el conjugado
    q_conjugado = (cos, -sin * v_1, -sin * v_2, -sin * v_3)
    #   Calcula el cuaternión correspondiente al punto rotado
    p_prima = multiplicar_quaterniones(q, multiplicar_quaterniones(p,␣
↪q_conjugado))
    # Devuelve el punto rotado
    punto_transformado = p_prima[1], p_prima[2], p_prima[3]
    return punto_transformado


def transformacion_rigida_3D(punto, parametros):
    x, y, z = punto
    t_11, t_12, t_13, alpha_in_rad, v_1, v_2, v_3, t_21, t_22, t_23 = parametros
    #   Aplicar una primera traslación
    x, y, z = traslacion(punto=(x, y, z), vector_traslacion=(t_11, t_12, t_13))
    #   Aplicar una rotación axial traslación
    x, y, z = rotacion_axial(punto=(x, y, z), angulo_en_radianes=alpha_in_rad,␣
↪eje_traslacion=(v_1, v_2, v_3))
    #   Aplicar una segunda traslación
    x, y, z = traslacion(punto=(x, y, z), vector_traslacion=(t_21, t_22, t_23))
    punto_transformado = np.array((x, y, z))
    return punto_transformado


def multiplicar_quaterniones(q1, q2):
    """Multiplica cuaterniones expresados como (1, i, j, k)."""
    return (
        q1[0] * q2[0] - q1[1] * q2[1] - q1[2] * q2[2] - q1[3] * q2[3],
        q1[0] * q2[1] + q1[1] * q2[0] + q1[2] * q2[3] - q1[3] * q2[2],
        q1[0] * q2[2] - q1[1] * q2[3] + q1[2] * q2[0] + q1[3] * q2[1],
        q1[0] * q2[3] + q1[1] * q2[2] - q1[2] * q2[1] + q1[3] * q2[0]
    )


def cuaternion_conjugado(q):
    """Conjuga un cuaternión expresado como (1, i, j, k)."""
    return (
        q[0], -q[1], -q[2], -q[3]
    )
```

En cambio, las siguiente funciones se han desarrolado durante la práctica

[45]:
```
def load_dcm_from_folder(path):
    dcm_files = []
    instance_number = []
    dcm_images = []
    for file in glob.glob(path + '/*.dcm'):
        dcm_file = pydicom.dcmread(file)

        dcm_files.append(dcm_file)
        instance_number.append(dcm_file.InstanceNumber)
        dcm_images.append(dcm_file.pixel_array)

    if len(dcm_files) == 1:
        dcm_images = dcm_file.pixel_array
    elif len(dcm_files) > 1:
        sorted_index = np.argsort(np.array(instance_number))
        dcm_images = np.array(dcm_images)[sorted_index]

    return dcm_files, dcm_images

def position_list(image3d):
    d, h, w = image3d.shape
    positions = []
    for z in range(d):
        for y in range(h):
            for x in range(w):
                positions.append(np.array((z, y, x)))
    return positions

def my_mse (img1, img2):
    diff = img2-img1
    return np.linalg.norm(diff)
```

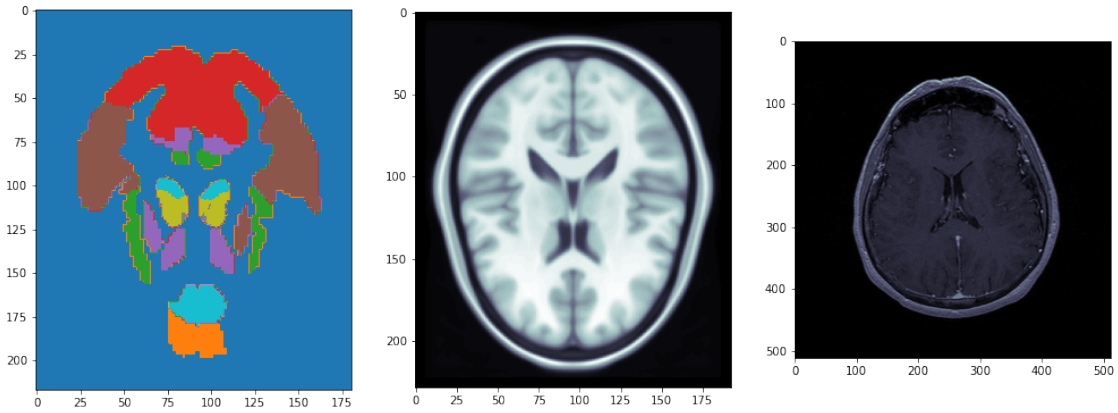
[4]:
```
dcm_AAL3, image_AAL3 = load_dcm_from_folder('P2 - DICOM/AAL3_1mm')
dcm_icbm, image_icbm = load_dcm_from_folder('P2 - DICOM/icbm_avg')
dcm_RM_Brain, image_RM_Brain = load_dcm_from_folder('P2 - DICOM/
 ↪RM_Brain_3D-SPGR')
```

[10]:
```
print(image_AAL3.shape)
print(image_icbm.shape)
print(image_RM_Brain.shape)
```

```
(181, 217, 181)
(193, 229, 193)
(212, 512, 512)
```

Seguidamente, se muestra un corte de cada una de las imágenes DICOM. Los cortes, de izquieda a derecha, se corresponden al atlas, paciente medio y paciente anonimizado.

[22]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(17,7))
ax1.imshow(image_AAL3[84,:,:], cmap='tab10')
ax2.imshow(image_icbm[90,:,:], cmap=plt.cm.bone)
ax3.imshow(image_RM_Brain[117,:,:], cmap=plt.cm.bone)
plt.show()
```

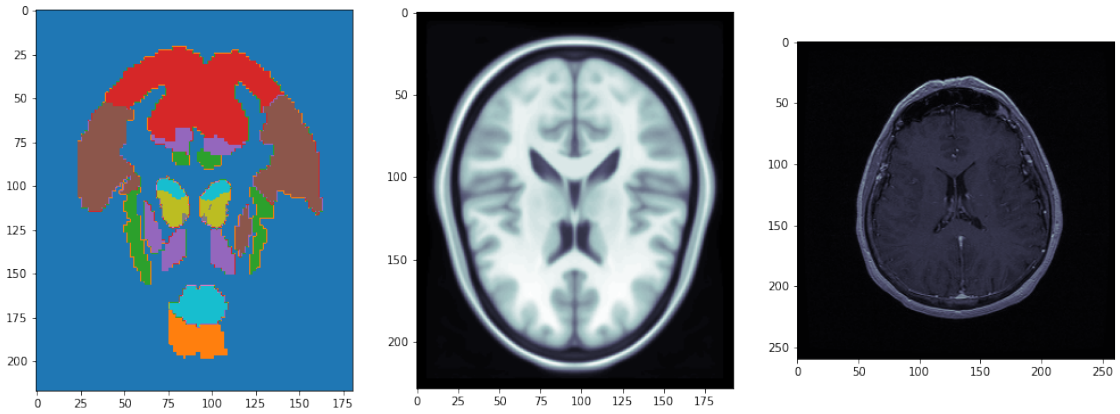


Después de verificar las cabeceras DICOM, se comprueba que el Slice thickness y el Pixel spacing de la imagen del paciente anónimo es de 2mm y 0.5078x0.5078 mm respectivamente. Esto significa que el ancho y alto de los píxeles representan 0.5078 y cada corte son 2mm. En cambio, el Pixel spacing tiene unas dimensiones de 1x1 mm (el tag de Slice thickness no aparece, por lo tanto no se va a redimensionar en la profundidad).

A continuación se redimensiona la imagen del paciente anónimo para que concuerda con las dimensiones de píxel del paciente medio.

[57]:
```
image_RM_Brain_0 = ndimage.zoom(image_RM_Brain, (1, 0.5078, 0.5078))
```

[58]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(17,7))
ax1.imshow(image_AAL3[84,:,:], cmap='tab10')
ax2.imshow(image_icbm[90,:,:], cmap=plt.cm.bone)
ax3.imshow(image_RM_Brain_0[117,:,:], cmap=plt.cm.bone)
plt.show()
```

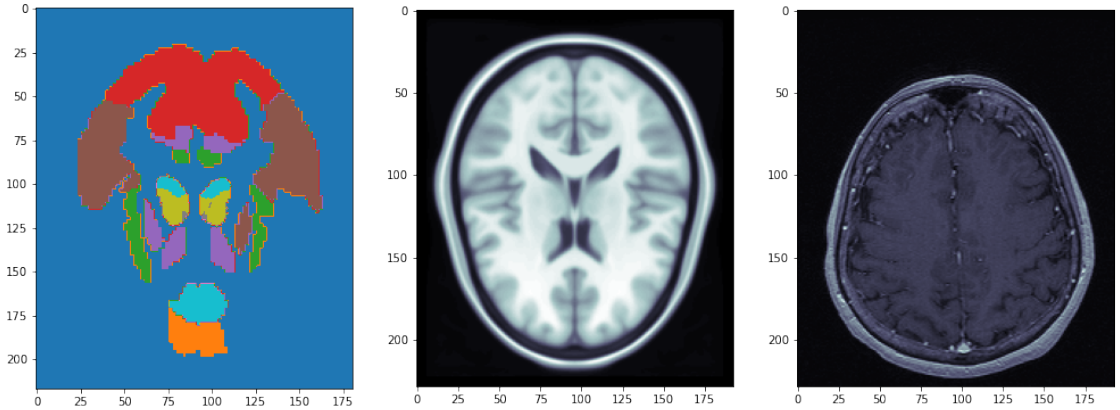


Después de la redimensión, se ha pasado de un tamaño de 512x512 píxeles a unos 260x260 píxeles aproximadamente. Sin embargo, aún no se tiene el mismo tamaño que el paciente medio. Si se observa la imagen del paciente medio, se puede ver que tiene bastante imagen vacía a los lados de la imagen. Se puede recortar esta parte para reducir el tamaño y acercarse más al tamaño del paciente medio. Además los primeros cortes no proporcionan ninguna información sobre el cerebro, también se puede ajustar en esta dimensión.
NOTA: la visualización se ha realizad utilizando la interfaz programada en el práctica 1 con ciertas modificaciones.

Por lo tanto, se puede realizar el siguiente recorte y se obtendrán las mismas dimensiones que la imagen del paciente medio

[64]:
```
image_RM_Brain_0 = image_RM_Brain_0[19:,:229, 34:227]
```

[65]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(17,7))
ax1.imshow(image_AAL3[84,:,:], cmap='tab10')
ax2.imshow(image_icbm[90,:,:], cmap=plt.cm.bone)
ax3.imshow(image_RM_Brain_0[117,:,:], cmap=plt.cm.bone)
plt.show()
```



Seguidamente, se va a realizar una prueba de rotación para comprobar que efectivamente las funciones realizan el comportamiento esperado.

Se definen los parámetros para la rotación. En este caso se va a rotar 45º respecto el eje z (el que apunta hacia fuera de la pantalla).

[66]:
```
parametros_rotacion = [0, 0, 0,
                       np.pi/4, 1, 0, 0,
                       0, 0, 0]
```

Se obtienen una lista con la posición inicial de los vóxeles.

[67]:
```
init_pos = position_list(image_RM_Brain_0)
```

Se calcula la posición final de cada vóxel dada la rotación definida

[68]:
```
final_pos = [transformacion_rigida_3D(voxel, parametros_rotacion) for voxel in␣
 ↪init_pos]
```

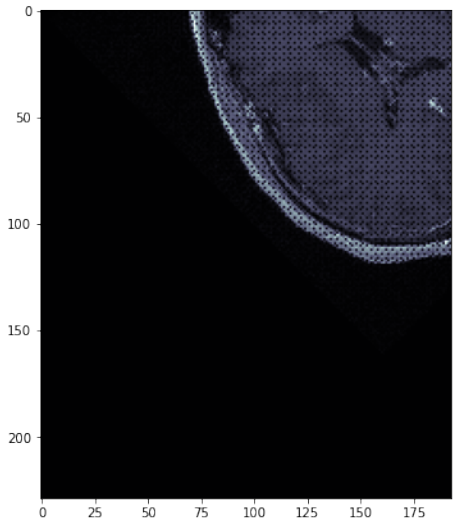
Una vez se tiene la posición final, hay que asignarle el nivel de gris correspondiente

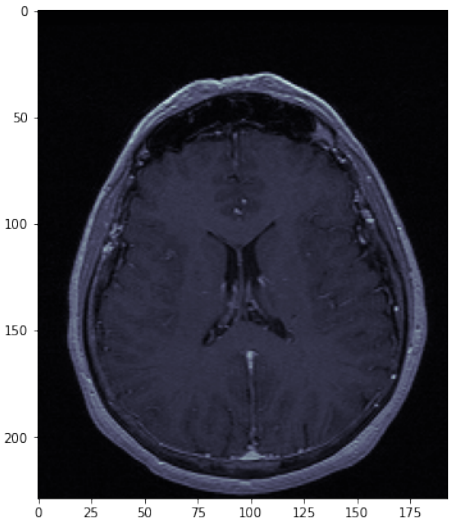
[69]:
```
final_pos = np.array(final_pos).astype(int)
final_pos = np.round(final_pos)

new_image = np.zeros(image_RM_Brain_0.shape)
d, h, w = image_RM_Brain_0.shape

for i, voxel_pos in enumerate(final_pos):
    new_d, new_h, new_w = voxel_pos
    if new_d >= 0 and new_d < d and new_h >= 0 and new_h < h and new_w >= 0 and␣
 ↪new_w < w:
        index = init_pos[i]
        value_gray = image_RM_Brain_0[tuple(index)]
        new_image[tuple(voxel_pos)] = value_gray
```

[73]:
```
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(17,7))
ax1.imshow(new_image[98,:,:], cmap=plt.cm.bone)
ax2.imshow(image_RM_Brain_0[98,:,:], cmap=plt.cm.bone)
plt.show()
```





Como se puede ver, se ha realizado una rotación de 45º. Sin embargo, como no se realiza ninguna interpolación durante la asignación de valores sino que se redondea la posición del nuevo vóxel, se pueden observar puntos vacíos dentro de la región de la imagen.

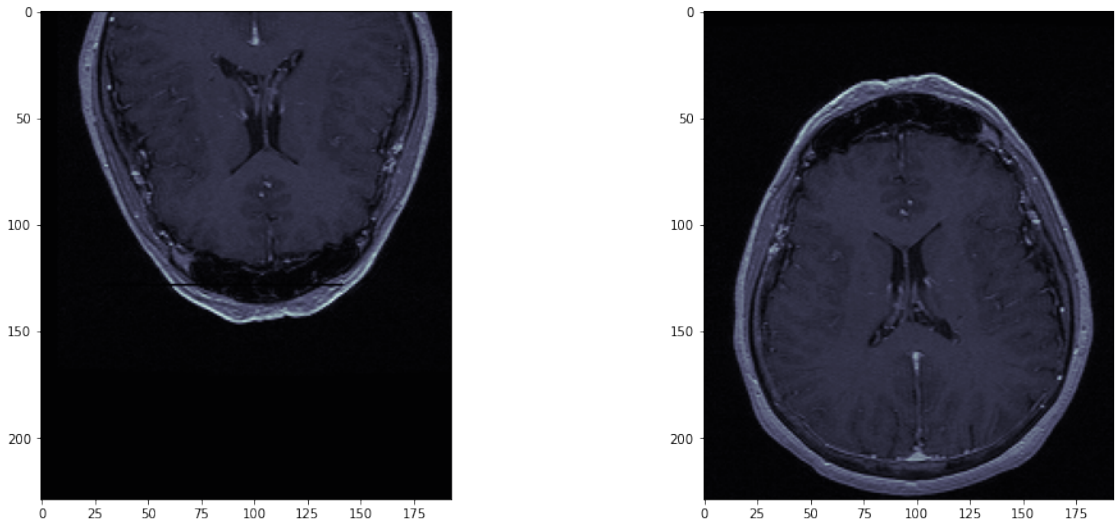
Antes de realizar el corregistros, se deben establecer los parámetros iniciales para facilitar al algoritmo encontrar la solución. Visualizando las imágenes y también las cabeceras DICOM se observa que las imágenes tienen aproximadamente una diferencia de 180º entre ellas. Por lo tanto, a continuación se establecen y comprueban estos parámetros iniciales.

[76]:
```
parametros_test = [0, 0, 0,
                   np.pi, 1, 0, 0,
                   0, 175, 200]
init_pos = position_list(image_RM_Brain_0)
final_pos = [transformacion_rigida_3D(voxel, parametros_test) for voxel in␣
 ↪init_pos]
final_pos = np.array(final_pos).astype(int)
final_pos = np.round(final_pos)

new_image = np.zeros(image_RM_Brain_0.shape)
d, h, w = image_RM_Brain_0.shape

for i, voxel_pos in enumerate(final_pos):
    new_d, new_h, new_w = voxel_pos
    if new_d >= 0 and new_d < d and new_h >= 0 and new_h < h and new_w >= 0 and␣
 ↪new_w < w:
        index = init_pos[i]
        value_gray = image_RM_Brain_0[tuple(index)]
        new_image[tuple(voxel_pos)] = value_gray
```

[77]:
```
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(17,7))
ax1.imshow(new_image[98,:,:], cmap=plt.cm.bone)
ax2.imshow(image_RM_Brain_0[98,:,:], cmap=plt.cm.bone)
plt.show()
```



Como se puede observar en la imagen anterior, la rotación de 180º y luego el desplazamiento de 175 y 200 píxeles consigue colocar la imagen más próxima a la posición que se requiere. Sin embargo, se puede desplazar la imagen un poco más hacia abajo (valores y más grandes).

[79]:
```
parametros_iniciales = [0, 0, 0,
                        np.pi, 1, 0, 0,  # Inicializamos el eje de la rotacion␣
↪a un vector unitario
                        0, 215, 200]
```

[84]:
```
def funcion_a_minimizar(parametros):
    init_pos = position_list(image_icbm)
    final_pos = [transformacion_rigida_3D(voxel, parametros) for voxel in␣
↪init_pos]
    final_pos = np.array(final_pos).astype(int)
    final_pos = np.round(final_pos)

    new_image = np.zeros(image_RM_Brain_0.shape)
    d, h, w = image_RM_Brain_0.shape

    for i, voxel_pos in enumerate(final_pos):
        new_d, new_h, new_w = voxel_pos
        if new_d >= 0 and new_d < d and new_h >= 0 and new_h < h and new_w >= 0␣
↪and new_w < w:
            index = init_pos[i]
```

```
            value_gray = image_RM_Brain_0[tuple(index)]
            new_image[tuple(voxel_pos)] = value_gray

    return my_mse(image_icbm, image_RM_Brain_0)
```

[85]:
```
resultado = least_squares(funcion_a_minimizar,
                          x0=parametros_iniciales,
                          verbose=1)
```

```
`gtol` termination condition is satisfied.
Function evaluations 1, initial cost 1.5563e+10, final cost 1.5563e+10, first-
order optimality 0.00e+00.
```

Parece ser que el algoritmo no realiza más de una evaluación. Esto puede ser provocado porque el algoritmo de optimización no es el correcto o porque se redondea la posición final del vóxel después de la transformación y por ello quedan vóxeles vacíos.