Landmark-based Registration

September 30, 2024

1 Introduction

This notebook focuses on MRI (Magnetic Resonance Imaging) is a critical tool in medical imaging, providing high-resolution, non-invasive scans of soft tissues like the brain. Different types of MRI scans, such as T1-weighted and T2-weighted, highlight different tissue properties, and accurate analysis of these images is crucial for diagnosing and monitoring various conditions. The notebook covers the following concepts.

- Differentiating between voxel and world coordinates for accurate image interpretation, hence understanding the difference between the coordinate systems.
- Using a viewer to explore 3D MRI volumes.
- Aligning MRI datasets through affine and rigid transformations, where rigid is preferred for brain scans as it preserves size and shape.
- Matching voxel grids between different MRI datasets for comparison.

The goal is to enhance understanding of image alignment and resampling, which are crucial in medical imaging.

[1]:
```
# For the pop up viewer
pip install ipython
```

```
Requirement already satisfied: ipython in /opt/anaconda3/lib/python3.12/site-
packages (8.25.0)
Requirement already satisfied: decorator in /opt/anaconda3/lib/python3.12/site-
packages (from ipython) (5.1.1)
Requirement already satisfied: jedi>=0.16 in /opt/anaconda3/lib/python3.12/site-
packages (from ipython) (0.18.1)
Requirement already satisfied: matplotlib-inline in
/opt/anaconda3/lib/python3.12/site-packages (from ipython) (0.1.6)
Requirement already satisfied: prompt-toolkit<3.1.0,>=3.0.41 in
/opt/anaconda3/lib/python3.12/site-packages (from ipython) (3.0.43)
Requirement already satisfied: pygments>=2.4.0 in
/opt/anaconda3/lib/python3.12/site-packages (from ipython) (2.15.1)
Requirement already satisfied: stack-data in /opt/anaconda3/lib/python3.12/site-
packages (from ipython) (0.2.0)
Requirement already satisfied: traitlets>=5.13.0 in
/opt/anaconda3/lib/python3.12/site-packages (from ipython) (5.14.3)
Requirement already satisfied: pexpect>4.3 in
/opt/anaconda3/lib/python3.12/site-packages (from ipython) (4.8.0)
```

```
Requirement already satisfied: parso<0.9.0,>=0.8.0 in
/opt/anaconda3/lib/python3.12/site-packages (from jedi>=0.16->ipython) (0.8.3)
Requirement already satisfied: ptyprocess>=0.5 in
/opt/anaconda3/lib/python3.12/site-packages (from pexpect>4.3->ipython) (0.7.0)
Requirement already satisfied: wcwidth in /opt/anaconda3/lib/python3.12/site-
packages (from prompt-toolkit<3.1.0,>=3.0.41->ipython) (0.2.5)
Requirement already satisfied: executing in /opt/anaconda3/lib/python3.12/site-
packages (from stack-data->ipython) (0.8.3)
Requirement already satisfied: asttokens in /opt/anaconda3/lib/python3.12/site-
packages (from stack-data->ipython) (2.0.5)
Requirement already satisfied: pure-eval in /opt/anaconda3/lib/python3.12/site-
packages (from stack-data->ipython) (0.2.2)
Requirement already satisfied: six in /opt/anaconda3/lib/python3.12/site-
packages (from asttokens->stack-data->ipython) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

1.1 Input data and code hints

Import Python libraries:

[1]:
```
import matplotlib.pyplot as plt
%matplotlib tk
plt.ion()
import numpy as np
np.set_printoptions( suppress=True )
import nibabel as nib
import scipy
import IPython
IPython.__version__
```

[1]:
```
'8.25.0'
```

Read the two 3D scans you'll be working with in this exercise:

[4]:
```
# Load T1 and T2 data
T1_fileName = 'IXI014-HH-1236-T1.nii.gz'
T2_fileName = 'IXI014-HH-1236-T2_moved.nii.gz'
T1 = nib.load( T1_fileName )
T2 = nib.load( T2_fileName )
T1_data = T1.get_fdata()
T2_data = T2.get_fdata()
```

Below is code to define a simple interactive viewer class that can be used to visualize 2D cross-sections of a 3D array along three orthogonal directions. It takes a 3D volume as input and shows the location a "linked cursor" in all three cross-sections.

[7]:
```
class Viewer:
    def __init__(self, data ):
        self.fig, self.ax = plt.subplots()
        self.data = data
        self.dims = self.data.shape
        self.position = np.round( np.array( self.dims ) / 2 ).astype( int )
        self.draw()
        self.fig.canvas.mpl_connect( 'button_press_event', self )
        self.fig.show()

    def __call__(self, event):
        print( 'button pressed' )
        if event.inaxes is None: return

        x, y = round( event.xdata ), round( event.ydata )

        #
        if ( x > (self.dims[0]-1) ) and ( y <= (self.dims[1]-1) ): return # lower-right quadrant

        #
        if x < self.dims[0]:
          self.position[ 0 ] = x
        else:
          self.position[ 1 ] = x - self.dims[0]

        if y < self.dims[1]:
          self.position[ 1 ] = y
        else:
          self.position[ 2 ] = y -self.dims[1]

        print( f"  voxel index: {self.position}" )
        print( f"  intensity: {self.data[ self.position[0], self.position[1], self.position[2] ]}" )

        self.draw()

    def draw( self ):
        #
        # Layout on screen is like this:
        #
        #     ^           ^
        #  Z  |        Z  |
        #     |           |
        #     ----->       ---->
        #       X            Y
        #     ^
```

```
        #  Y  |
        #     |
        #     ----->
        #       X
        #
        dims = self.dims
        position = self.position

        xySlice = self.data[ :, :, position[ 2 ] ]
        xzSlice = self.data[ :, position[ 1 ], : ]
        yzSlice = self.data[ position[ 0 ], :, : ]

        kwargs = dict( vmin=self.data.min(), vmax=self.data.max(),
                       origin='lower',
                       cmap='gray',
                       picker=True )

        self.ax.clear()

        self.ax.imshow( xySlice.T,
                        extent=( 0, dims[0]-1,
                                 0, dims[1]-1 ),
                        **kwargs )
        self.ax.imshow( xzSlice.T,
                        extent=( 0, dims[0]-1,
                                 dims[1], dims[1]+dims[2]-1 ),
                        **kwargs )
        self.ax.imshow( yzSlice.T, extent=( dims[0], dims[0]+dims[1]-1,
                                            dims[1], dims[1]+dims[2]-1 ),
                        **kwargs )

        color = 'g'
        self.ax.plot( (0, dims[0]-1), (position[1], position[1]), color )
        self.ax.plot( (0, dims[0]+dims[1]-1), (dims[1]+position[2],␣
↪dims[1]+position[2]), color )
        self.ax.plot( (position[0], position[0]), (0, dims[1]+dims[2]-1), color␣
↪)
        self.ax.plot( (dims[0]+position[1], dims[0]+position[1]), (dims[1]+1,␣
↪dims[1]+dims[2]-1), color )

        self.ax.set( xlim=(1, dims[0]+dims[1]), ylim=(0, dims[1]+dims[2]) )

        self.ax.text( dims[0] + dims[1]/2, dims[1]/2,
                      f"voxel index: {position}",
                      horizontalalignment='center', verticalalignment='center' )

        self.ax.axis( False )
```

```
        self.fig.canvas.draw()
```

The code below shows how to visualize the T1-weigthed and the T2-weighted volumes with this viewer class. The initial location of the cursor is in the middle of the volume in each case. It can be changed by clicking on one of the cross-sections. The viewer also displays the voxel index **v** of the cursor. Play around and try to understand what the Viewer() class does.

[28]:
```
T1_viewer = Viewer( T1_data )
```

[30]:
```
T2_viewer = Viewer( T2_data )
```

---

2 Task 1: Coordinate Systems

Familiarize yourself with the concept of coordinate systems. In your report, explain why we need to differentiate between "voxel coordinates/indices" **v** and "world coordinates" **x**. Why does the T2-weighted volume look so compressed in the viewer? For the enthusiastic student: calculate the voxel size in each dataset.

The world coordinate system used in both the T1-weighted and the T2-weighted scan follows the RAS convention. Equipped with this information, determine the voxel index **v** of the center of the left eye of the patient in the T1-weighted scan. Do the same for the T2-weighted scan.

Hints:

- The affine voxel-to-world matrix of the T1-weighted scan is given by

  `T1.affine`

- In nibabel, the RAS convention is used (see bottom of https://nipy.org/nibabel/coordinate_systems.html)

The voxel coordinates/indices reffer to the 3 different numbers, usually i,j,k; they are the grid indices of individual volume elements. The world coordinates represents the actual topology of the patient. We could think that both coordinates always have the same ratio, but actually it does not. The MRI scan is not always isotropic, the spacing between voxels differs across the three dimensions.

T2 looks compressed in the viewer because the spacing between voxels in the slice plane (e.g., x and y dimensions) is often much smaller than the slice thickness (z-dimension), resulting in the appearance of compression or stretching along the z-axis when visualized (it is an MRI scan).

[16]:
```
# Voxel size
T1_affine = T1.affine
# print (T1_affine)
T2_affine = T2.affine

voxel_size_T1= np.abs(np.diag(T1_affine)[:3])
print (f"Voxel size of T1: {voxel_size_T1} mm")
```

```
voxel_size_T2= np.abs(np.diag(T2_affine)[:3])
print (f"Voxel size of T2: {voxel_size_T2} mm")

# Voxel index v of the center of the left eye of the patient
# x= Av +t where x is the world coordinate, v is the voxel index, A is the affine matrix and t is the translation vector
# A^-1(x -t) = v  where A^-1 is the inverse of the affine matrix

A = np.linalg.inv(T1.affine[:3,:3])
t= T1_affine[:3,3]
# print(t)
voxel_indx =[64, 120, 103,1] # Left eye of the patient
# print (voxel_indx)
x = T1_affine @ voxel_indx
print ("World coordinates of T1 for the left patient eye will be: ",x[:3] ,"mm")

A = np.linalg.inv(T2.affine[:3,:3])
t= T2_affine[:3,3]
# print(t)
voxel_indx =[141, 216, 15,1] # Left eye of the patient
# print (voxel_indx)
x = T2_affine @ voxel_indx
print ("World coordinates of T2 for the left patient eye will be: ",x[:3] ,"mm")
```

```
Voxel size of T1: [0.01506851 0.12376414 0.01646696] mm
Voxel size of T2: [0.83986628 0.86080861 4.79742765] mm
World coordinates of T1 for the left patient eye will be:  [32.75329093
75.34755456 -0.436423  ] mm
World coordinates of T2 for the left patient eye will be:  [-19.56145275
116.24901803   3.76028705] mm
```

For this task it is very important to know the difference between world coordinates and voxel coordinates. Voxel coordinates refer to the position of a voxel in a 3D grid (so it has i, j, k) that makes up the MRI volume. World coordinates, on the other hand, represent the actual position of a voxel in the patient's body and are expressed in millimeters. To transform voxel coordinates to world coordinates we use `x = Av + t` as stated on the slides. The affine matrix has important information regarding scaling, rotation and translation. Being the diagonal of the matrix (except the element in position 4,4 which is a 1) the voxel size. We then got the world coordinates from the voxel coordinates that we measured in the left eye by using the same formula but with the inverse of the affine matrix.

---

3 Task 2: Resample the T2-weighted scan to the image grid of the T1-weighted scan

In this task you should resample the T2-weighted scan to the image grid of the T1-weighted scan, i.e., create a new 3D volume that has the same size as the T1-weighted volume, but that contains interpolated T2-weighted intensities instead. In particular, for each voxel index $\mathbf{v}_{T1}$ in the T1-weighted image grid, you should compute the corresponding voxel index $\mathbf{v}_{T2}$ in the T2-weighted volume as follows (see section 2.1 in the book):

$$\begin{pmatrix} \mathbf{v_{T2}} \\ 1 \end{pmatrix} = \mathbf{M}_{T2}^{-1} \cdot \mathbf{M}_{T1} \cdot \begin{pmatrix} \mathbf{v_{T1}} \\ 1 \end{pmatrix}.$$

At the location $\mathbf{v}_{T2}$, you should then use cubic B-spline interpolation to determine the intensity in the T2-weighted scan, and store it at index $\mathbf{v}_{T1}$ in the newly created image.

Once you have created a new volume like this, visualize it overlaid on the T1-weighted volume as follows:

```
Viewer( T2_data_resampled / T2_data_resampled.max() + T1_data / T1_data.max() )
```

Hints: - you can create a coordinate grid in 3D with the function

```
V1,V2,V3 = np.meshgrid( np.arange( T1_data.shape[0] ),
                        np.arange( T1_data.shape[1] ),
                        np.arange( T1_data.shape[2] ), indexing='ij' )
```

- the following SciPy function interpolates the T2-weighted volume at voxel coordinates $(1.1, 2.2, 3.3)^T$ and $(6.6, 7.7, 8.8)^T$ using cubic interpolation:

```
scipy.ndimage.map_coordinates( T2_data, np.array( [ [1.1,2.2,3.3], [6.6,7.7,8.8] ] )
```

[26]:
```
V1,V2,V3 = np.meshgrid( np.arange( T1_data.shape[0] ),
                        np.arange( T1_data.shape[1] ),
                        np.arange( T1_data.shape[2] ), indexing='ij' )
T1_voxel_coords = np.vstack([V1.ravel(), V2.ravel(), V3.ravel(), np.ones(V1.
 ↪size)])

T2_voxel_coords_n = np.linalg.inv(T2_affine) @ T1_affine @T1_voxel_coords
T2_voxel_coords = T2_voxel_coords_n[:3,:]

T2_data_resampled = scipy.ndimage.map_coordinates(T2_data, T2_voxel_coords,␣
 ↪order=3).reshape(T1_data.shape)
T2_viewer_r = Viewer( T2_data_resampled / T2_data_resampled.max() + T1_data /␣
 ↪T1_data.max() )
```

In summary we used MT1 and MT2, the affine matrices for the T1 and T2 scans and used the formula given to find the location in the T2 scan for each voxel in the T1 scan. We first created the meshgrid that represent the voxel indices in T1. We then stack them into a matrix after using ravel which transforms a 3D array into a 1D array and we also addeed an extra row of 1s. We the create a matrix where each column has a 3D voxel coordinate in T2 and we take the first 3 rows to have te actual T2 voxel coordinates without the ones. Then we did a cubic B spline interpolation of order 3 to know the values of the voxels that are not integers and then reshaped into T1 dimensions.

Finally we normalized by dividing by their max value so we get intensities that are between 0 and 1 and overlaid the two volumes so we can actually see the T2-weitghted scan onto the T1.

---

4 Task 3: Collect corresponding landmarks

Using the viewer class, record the voxel coordinates $\mathbf{v_{T1}}$ and $\mathbf{v_{T2}}$ of at least five corresponding landmarks in the T1- and the T2-weighted volumes, respectively. List them in your report, and explain why you picked them.

> ***Hint:*** - Avoid picking landmarks that are very close to each other or that all lie approximately in the same 2D plane. - You can double-check which landmarks you've selected as follows: T1_viewer = Viewer(T1_data) T1_viewer.position = (20, 30, 40) T1_viewer.draw()

[ ]:
```
# t1
nose = [20, 85, 75]
nuca_alta = [233, 124, 52]
nuca_baja = [187, 66, 106]
barbilla = [55, 10, 82]
frente = [52, 166, 86]

# t2
nose = [120, 221, 8]
nuca_alta = [132, 35, 22]
nuca_baja = [132, 34, 1]
barbilla = [126, 223, 2]
frente = [121, 204, 23]

#Another set of landmarks:
# t1
ojo_dcho = [61, 114, 107]
nuca_alta = [218, 163, 46]
nuca_baja = [210, 52, 65]
oreja_dcha = [153, 96, 21]
frente = [51, 174, 78]

# t2
ojo_dcho =[18, 114, 106]
nuca_alta = [185, 147, 95]
nuca_baja = [172, 23, 109]
oreja_dcha = [126, 223, 2]
frente = [86, 83, 35]
```

For this part we just had to select landmarks using the viewer taking into acount that they had to be evenly distributed, anatomically recognizable and in different planes to provide a good 3D view.

---

5 Task 4: Perform affine landmark-based registration

Using the landmarks you recorded in the previous task, compute the parameters of the 3D affine transformation that brings the landmarks in the T1-weighted image closest to the corresponding ones in the T2-weighted image. For this purpose, use Equation (2.8) in the book.

Once you've determined the affine transformation, register to two images by resampling the T2-weighted image to the image grid of the T1-weighted image, and overlay the two images as in Task 2. To map voxel coordinates $\mathbf{v}_T1$ to $\mathbf{v}_T2$, you'll have to use

$$\begin{pmatrix} \mathbf{v_{T2}} \\ 1 \end{pmatrix} = \mathbf{M}_{T2}^{-1} \cdot \mathbf{M} \cdot \mathbf{M}_{T1} \cdot \begin{pmatrix} \mathbf{v_{T1}} \\ 1 \end{pmatrix},$$

where $\mathbf{M}$ is your 4×4 affine matrix (see book).

What happens when you increase/decrease the number of corresponding landmarks that are used in the computations? Comment.

> ***Hint:*** - remember that the affine matrix works in *world* coordinates, so you'll have to map your landmarks to world coordinates first. - you can use
>
> `np.hstack()`
>
> to append a column of ones to an existing matrix (e.g., to construct the $\mathbf{X}$ matrix in Equation (2.8) in the book).

[37]:
```
# Step 1: Create a matrix X & Y, that holds the landmark in world coordinate␣
 ↪chosen before
# Step 2: Calculate M = YX^T(XX^t)^-1
# Step 3: Calculate the provided equation

# T1 and T2 voxel coordinates from the landmarks you've provided
T1_landmarks_vox = np.array([
    [20, 85, 75],  # Nose
    [233, 124, 52],  # High nape
    [187, 66, 106],  # Low nape
    [55, 10, 82],  # Chin
    [52, 166, 86]  # Forehead
])

T2_landmarks_vox = np.array([
    [120, 221, 8],  # Nose
    [132, 35, 22],  # High nape
    [132, 34, 1],  # Low nape
    [126, 223, 2],  # Chin
    [121, 204, 23]  # Forehead
])

# Convert voxel coordinates to world coordinates using affine matrices
T1_landmarks_world = (T1_affine @ np.hstack((T1_landmarks_vox, np.
 ↪ones((T1_landmarks_vox.shape[0], 1))))).T).T[:, :3]
```

```
T2_landmarks_world = (T2_affine @ np.hstack((T2_landmarks_vox, np.
 ↪ones((T2_landmarks_vox.shape[0], 1))))).T).T[:, :3]

# Construct the X and Y matrices for the affine transformation computation
X = np.hstack([T1_landmarks_world, np.ones((T1_landmarks_world.shape[0], 1))]).
 ↪T  # Shape (4, N)
Y = np.hstack([T2_landmarks_world, np.ones((T2_landmarks_world.shape[0], 1))]).
 ↪T  # Shape (4, N)

# Compute the affine matrix M using Equation (2.8)
M = Y @ X.T @ np.linalg.inv(X @ X.T)

# Apply the affine transformation to resample the T2 image onto the T1 grid
# Using the equation: v_T2 = M_T2^-1 * M * M_T1 * v_T1

# Create the coordinate grid for T1
V1, V2, V3 = np.meshgrid(np.arange(T1_data.shape[0]),
                         np.arange(T1_data.shape[1]),
                         np.arange(T1_data.shape[2]), indexing='ij')
T1_voxel_coords = np.vstack([V1.ravel(), V2.ravel(), V3.ravel(), np.ones(V1.
 ↪size)])

# Compute the transformation matrix from T1 to T2 using the affine matrix M
T1_to_T2_affine = np.linalg.inv(T2_affine) @ M @ T1_affine

# Transform the T1 voxel coordinates into T2 voxel coordinates
T2_voxel_coords = T1_to_T2_affine @ T1_voxel_coords
T2_voxel_coords = T2_voxel_coords[:3]  # Discard the homogeneous coordinate␣
 ↪(the 4th row)

# Interpolate the T2 data at these T2 voxel coordinates using cubic␣
 ↪interpolation
T2_data_resampled = scipy.ndimage.map_coordinates(T2_data, T2_voxel_coords,␣
 ↪order=3).reshape(T1_data.shape)

# Normalize both volumes to be in [0, 1] range
T2_data_resampled_normalized = T2_data_resampled / T2_data_resampled.max()
T1_data_normalized = T1_data / T1_data.max()

# Visualize the overlay
Viewer(T2_data_resampled_normalized + T1_data_normalized)
```

[37]: <__main__.Viewer at 0x1667306e0>

The code comments present every step of the process. By increasing the number of landmarks it would improve the accuracy of the transformation (this is only if the new landmarks are spread across the entire volume taking into acount the measures in task 3). On the other hand decreasing

the number of said landmarks makes the registrarion less reliable as it is giving more weight to each of the few points we selected. It might cause some misalignment.

---

6 Task 5: Perform rigid landmark-based registration

Repeat Task 4, but this time using a *rigid* transformation model. Vary the number of landmarks that are used again, and comment. Which transformation model (affine or rigid) is more appropriate to use in this specific application?

Hint: - a singular value decomposition can be computed using

`np.linalg.svd()`

- the determinant of a matrix can be computed using

`np.linalg.det()`

[ ]:
```
# Landmarks for T1 (source) and T2 (target) in world coordinates (convert if needed)
# We could have used the already defined ones, but just to make sure

T1_landmarks_world = (T1_affine @ np.hstack((T1_landmarks_vox, np.ones((T1_landmarks_vox.shape[0], 1))))).T).T[:, :3]
T2_landmarks_world = (T2_affine @ np.hstack((T2_landmarks_vox, np.ones((T2_landmarks_vox.shape[0], 1))))).T).T[:, :3]

# Compute the mean of both sets of landmarks
T1_mean = np.mean(T1_landmarks_world, axis=0)
T2_mean = np.mean(T2_landmarks_world, axis=0)

# Center the landmarks by subtracting the centroids
T1_centered = T1_landmarks_world - T1_mean
T2_centered = T2_landmarks_world - T2_mean

# Compute the cross-covariance matrix H
H = T1_centered.T @ T2_centered

# Perform SVD on the cross-covariance matrix
U, S, Vt = np.linalg.svd(H)

# Compute the rotation matrix R
R = Vt.T @ U.T

# Ensure a right-handed coordinate system (determinant of R must be +1)
if np.linalg.det(R) < 0:
    Vt[-1, :] *= -1  # Flip the last column of Vt to ensure a positive determinant
```

```
    R = Vt.T @ U.T

# Compute the translation vector t
t = T2_mean - R @ T1_mean

# Now, we have the rigid transformation: Rotation (R) and Translation (t)

# Transform the T1 voxel grid to T2 using the rigid transformation

# Create a voxel grid for T1 (as done in earlier examples)
V1, V2, V3 = np.meshgrid(
    np.arange(T1_data.shape[0]),
    np.arange(T1_data.shape[1]),
    np.arange(T1_data.shape[2]),
    indexing='ij'
)

# Stack voxel coordinates into homogeneous coordinates (N, 4) format
T1_voxel_coords = np.vstack([V1.ravel(), V2.ravel(), V3.ravel(), np.ones(V1.
 ↪size)])

# Convert T1 voxel coordinates to world coordinates
T1_world_coords = T1_affine @ T1_voxel_coords

# Apply the rigid transformation (Rotation + Translation)
T1_transformed_world_coords = R @ T1_world_coords[:3, :] + t[:, np.newaxis]

# Convert the transformed world coordinates back to T2 voxel coordinates
T2_voxel_coords = np.linalg.inv(T2_affine) @ np.
 ↪vstack([T1_transformed_world_coords, np.ones(T1_transformed_world_coords.
 ↪shape[1])])

# Interpolate the T2 image at the transformed coordinates
T2_voxel_coords = T2_voxel_coords[:3, :]
T2_voxel_coords = T2_voxel_coords.reshape((3,) + T1_data.shape)  # Reshape to␣
 ↪match the T1 volume shape

# Use cubic interpolation to sample T2 data at the transformed coordinates
T2_data_resampled = scipy.ndimage.map_coordinates(T2_data, T2_voxel_coords,␣
 ↪order=3)

# Visualize the result
# Normalize both images to the range [0, 1] for visualization
T1_data_normalized = T1_data / T1_data.max()
T2_data_resampled_normalized = T2_data_resampled / T2_data_resampled.max()

# Combine the images for visualization
```

```
overlay = T1_data_normalized + T2_data_resampled_normalized

Rigid_image = Viewer (overlay)
```

The code comments present every step of the process. For this task we now used a rigid transformation to align the T1 weighted to the T2 weighted. Rigid transformation focuses on changing the position and orientation of the brain without changing its size or shape. In brain MRI, it makes more sense to use rigid transformation as the brain tipically does not change size or shape in a short period of time, this means that the brain structure stays the same so by using rigid transformation we make sure the brain images are actually aligned.

7 Conclusion

The exercise made us help understand coordinate systems, transforming form voxel to world coordinate in task 1. We also learned how to resample MRI images and select siginificant landmarks which is vital for our future as biomedical engineers. We then performed both affine and rigid transformation and realized rigid transformation is better to use for brain MRI because it mantains the brain structure better. Overall we gained a comprehensive understanding on analyzing MRI data which is essential for accurately interpreting brain scans in medical imaging applications.