BE 5370 Homework Assignment 1 (2022)

See syllabus for due date/time

August 31, 2022

Contents

1 Introduction

The objective of this programming assignment is to help you become familiar with how to represent and process 3D images using Python libraries **NumPy**, **SciPy** and **PyTorch**. The assignment is largely tutorial in nature. After completing it, you should feel comfortable to compete the more complex assignments to come.

This assignment is motivated by the problem of registration of longitudinal images (images of the same individual acquired at the beginning and end of a research study). Changes in images over time can reveal progression of disease or even inform researchers whether a treatment to slow a disease is working. Sometimes, changes expected in a longitudinal study are very subtle. For example, in Alzheimer's disease, even the regions of the brain most affected change at the rate of 2-4% a year. Such changes can easily be missed when comparison between longitudinal images is performed in a careless way. Fox et al., 2011 is a good short paper discussing the need to process longitudinal images with great care.

In this assignment, we will only touch the surface of this problem. In Part I of the assignment, we will set up your Python environment and create some basic functions for loading and displaying images. In Part II, we will examine how different approaches to image interpolation and filtering affect the differences measured between longitudinal images. In Part III, we will take advantage of PyTorch's ability to automatically compute derivatives to implement affine image registration from scratch.

Below are some general guidelines for this assignment:

- *Give yourselves at least 3 weeks to finish this assignment – more if you are new to Python or PyTorch!*
- You are encouraged to work in groups of 2. If so, *each of you is still expected to work on the entire assignment*, as opposed to dividing the work into parts. This is so that of you both learn from the assignment.
- Read the entire assignment before starting. There are many parts, some will take longer than others.

2 Data

The following data will be provided for one participant in a longitudinal imaging study.

1. A T1-weighted MRI scan acquired at the beginning of the study (`baseline.nii`)
2. A T1-weighed MRI scan acquired three years later (`followup.nii`)

3. An image representing the segmentation of three anatomical structures in the baseline image (`seg.nii`).
4. A text file describing the affine transformation between the followup image and the baseline image (`f2b.txt`). This will be described in a later section.

The following Python packages will be used in this assignment:

1. `nibabel`, a package for reading and writing NIFTI format images,
2. `numpy` and `scipy`, standard scientific computing packages,
3. `matplotlib`, a package for generating plots and figures.
4. `torch`, the PyTorch library for tensor math, auto-differentiation, and deep learning.

3 How to Format Your Assignment

- Your assignment should be in the form of a **Jupyter Notebook**, which allows you to combine code, figures, paragraphs with formulas and images in a single document. You can use the free **Google Colab** service or your local Python environment.
- The specific functions that you are asked to write below should be defined using the `def` keyword in your notebook.
- For the final assignment, turn in the PDF of notebook, as well as the `.ipynb` file.
- When generating plots and figures, be sure to give figures titles, axis titles, legends, etc., as relevant. The quality of the figures and plots is part of the grading criteria.
- You don't need to write a lot of text, but include enough so that someone outside of our class could get a basic idea of what the notebook is doing. In other words, don't just include section headings with figures and code boxes, but also a few sentences explaining what you have done in each step. It can be as simple as "This function reads a NIFTI image into a 3D array".

3.1 Grading

Your grade will be graded according to the following criteria:

- Successful completion of all components of the assignment: **80 points**.
  - The number of points for each component is given in the section heading below.
- Overall quality of the writing, plots, figures, and code: **20 points**.

4 Part I: Python Imaging Basics

4.1 Setup Your Python Environment

4.1.1 Option 1: Working Locally on Your Computer

If you don't already have Python and these packages in on your system, I recommend installing them using **miniconda**. After following the instructions to download and install **miniconda**, use the following commands to install the needed packages:

```
conda install -c conda-forge -c pytorch jupyter-lab nibabel numpy scipy matplotlib torch
```

4.1.2 Option 2: Working on Google Colab

The necessary packages should already be installed when you create a new Google Colab notebook. If you need to install some other package, add a cell like this to your notebook:

```
!pip install mypackage
```

The easiest way to access assignment data in Google Colab is to upload the data directory to your Google Drive and then give your notebook permissions to access your Google Drive. In the panel on the left of your notebook, select the folder icon, and then click on the Google Drive icon. After you grant your notebook access to your Google Drive, you will be able to access files using the path `/drive`.

4.1.3 Import and Configure Python Libraries

Place the following code at the top of your Jupyter notebook. Make sure to change `data_dir` to the location of the data directory for this assignment. In the rest of your notebook, use expressions like `os.path.join(data_dir, "baseline.nii")` to access assignment data. This makes it easier for others to run your notebook.

```
# Import required libraries
import os
import numpy as np
import nibabel as nib
import matplotlib.pyplot as plt
from scipy import interpolate, signal

# Configure matplotlib options
import matplotlib
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = [12, 8]

# Path to the data for this experiment
data_dir='/Users/pauly/Documents/penn/BE5370/homework/hw1/data'
```

On Google colab, with Google Drive connected, the last line should be like this:

```
data_dir='/drive/penn/BE5370/homework/hw1/data'
```

4.2 Explore the Data in ITK-SNAP (5 points)

As the first step, simply view the baseline and follow-up images in ITK-SNAP.

- Get the **ITK-SNAP 3.8** binary for your architecture from http://itksnap.org.
- Open the baseline image as the "main" image (`File->Open Main Image...`).
  - Play around with crosshair, zooming and panning tools.
- Use the layer inspector (`Tools->Layer Inspector`) to adjust image contrast and color map.

- Go to the "Info" tab on the layer inspector to view the image dimensions and voxel spacing. Voxel spacing refers to the dimensions of a single voxel in the physical space. It is usually measured in mm. In the images in this assignment, spacing is not the same in the x, y and z dimensions. Thus, if you do not keep track of the spacing, the images will not be displayed correctly, i.e., they will have wrong aspect ratio (look stretched).

- Load the segmentation of the baseline image and use the "Update" button in the bottom left panel to generate a 3D rendering.
- Open the followup image as an additional image layer (`File->Add Another Image...`) in ITK-SNAP. You will notice that the images are not registered.
- Use the registration tool (`Tools->Registration`) to perform registration between images. Try manual and automatic modes. Experiment with different registration settings (e.g., rigid vs. affine, different metrics, masking).

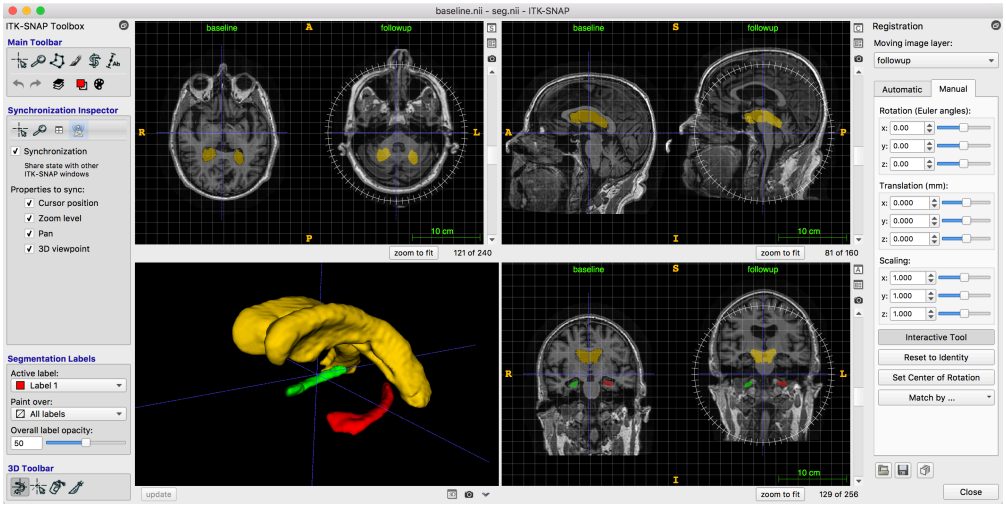

Figure 1: 3D image rendered in ITK-SNAP

4.2.1 Deliverables.

- Include in your Notebook screenshots demonstrating the images loaded and co-registered in ITK-SNAP.
- In a short paragraph describe what automatic registration settings provided the best registration and how well automatic registration did compared to manual registration.

4.3 Loading and Displaying 3D Images with Python (10 points)

4.3.1 Reading and Writing Images

Use the functions below to read and write 3D images from/to NIfTI files on disk.

```
def my_read_nifti(filename):
    """ Read NIfTI image voxels and header

    :param filename: path to the image to read
```

```
    :returns: tuple (img,hdr) consisting of numpy voxel array and nibabel NIfTI header
    """
    img = nib.load(filename)
    return img.get_fdata(), img.header

def my_write_nifti(filename, img, header = None):
    """ Write NIfTI image voxels and header

    :param filename: path to the image to save
    :param img: numpy voxel array
    :param header: nibabel NIfTI header
    """
    if header is not None:
        nifti_img = nib.Nifti1Image(img, affine=header.get_best_affine(), header=header)
    else:
        nifti_img = nib.Nifti1Image(img, affine=np.ones((len(img.shape),1)))
    nib.save(nifti_img, filename)
```

- The function `my_read_nifti` returns a tuple consisting of an n-dimensional voxel array and a 'header' object created by the **nibabel** package. The header contains some of the same information that you saw in the *Info* tab of the ITK-SNAP layer inspector, including voxel spacing.
  - You would call the function like this:
    ```
    I_bl,hdr_bl = my_read_nifti(os.path.join(data_dir, 'baseline.nii'))
    ```
  - Note that Python tuples are a useful mechanism for returning multiple values of different types from a function.
  - To learn more about NIfTI headers, see NIfTI image headers in nibabel.
- The function `my_write_nifti` takes as input the filename where to write the image, the 3D voxel array, and the optional header. When the header is not supplied, a default header is substituted, with 1mm spacing.

4.3.2 Viewing Images

Write a function `my_view` to generate a visualization of a 3D image similar to ITK-SNAP, as shown below. Divide the plot area into four subplots, and in three of them, display orthogonal slices from the 3D image.

The index of the voxel where the three slices intersect is called the *crosshair position*, and should be an optional parameter `xhair` to your visualization function. Just like in ITK-SNAP, the crosshair position should be visualized using two orthogonal line segments in each subplot (blue lines in the figure). When `xhair` is not specified, the crosshairs should be placed at the center of the image.

Other parameters to the function will include the color map (`cmap`) used to render the image (e.g., `'jet'` or `'hot'`, with the default of `'gray'`) and the range of intensities that should be plotted (`crange`), similar to the minimum and maximum intensity values in the ITK-SNAP contrast adjustment window.

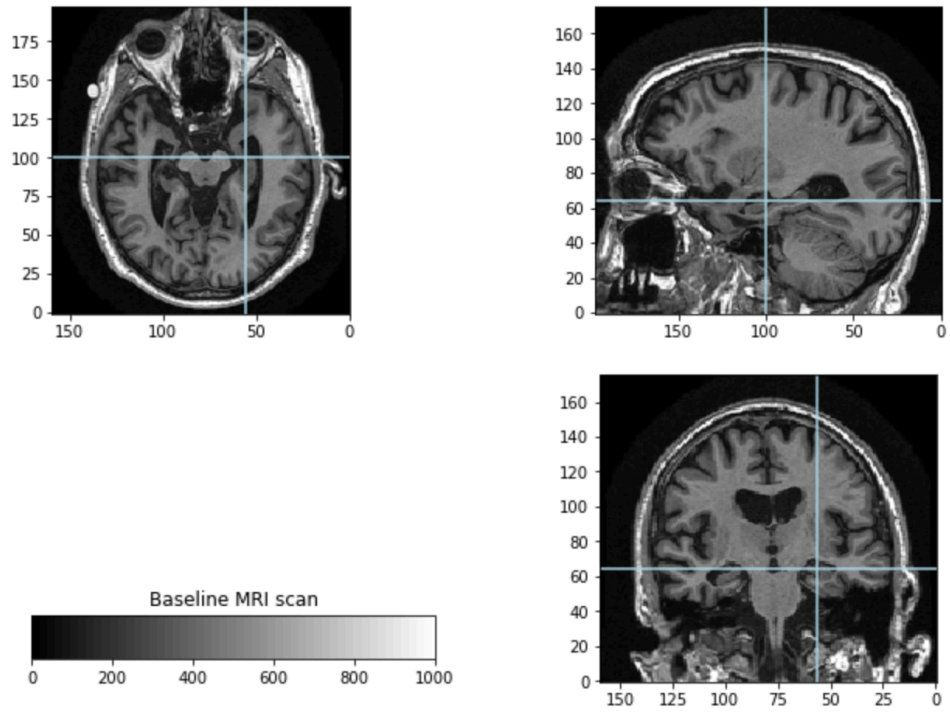

Figure 2: Expected `my_view` output

The header obtained from `my_read_nifti` is also passed in to the function as an optional parameter. This is so that the scaling of the axes in your plot matches voxel spacing. To get full credit for this portion of the assignment you must make sure that axis scaling (also known as aspect ratio) is correct and matches ITK-SNAP.

Your function will have the following signature:

```
def my_view(img, header=None, xhair=None, crange=None, cmap='gray'):
    """Display a 3D image in a layout similar to ITK-SNAP

    :param img: 3D voxel array
    :param header: Image header (returned by my_read_nifti)
    :param xhair: Crosshair position (1D array or tuple)
    :param crange: Intensity range, a tuple with minimum and maximum values
    :param cmap: Colormap (a string, see matplotlib documentation)
    """
```

Here are some hints to help you implement this function:

- Read the NumPy documentation on array indexing and slicing to learn how to extract 2D slices from 3D volumes
- Use **matplotlib** function `subplots` to generate a *figure* object and an array of *axis* objects.
- Use the **matplotlib** function `imshow` to display 2D slices in three of the axes. Use the parameter `aspect` to specify the aspect ratio of the pixels in each 2D plot. Other useful parameters are `cmap`, `vmin` and `vmax`.

- You may need to transpose the 2D arrays you get from slicing to get them into correct orientation. You may also need to flip the direction of the x and y axes to get the orientation right. This can be done using **matplotlib** functions `invert_xaxis()` and `invert_yaxis()`.
- Use functions `axvline` and `axhline` to draw the crosshairs.
- Feel free to customize your function by adding other parameters, like `title` for adding a title to the plot, etc.

To place the colorbar as in the figure above, you can use the following code:

```
fig, axs = plt.subplots(2,2)
...
im1 = imshow(...)
...
axs[1,0].axis('off');
cax = plt.axes([0.175, 0.15, 0.3, 0.05])
plt.colorbar(im1, orientation='horizontal', ax=axs[0,0], cax=cax)
```

4.3.3 Deliverables

1. The code for the function `my_view`.
2. A figure showing the baseline image displayed by `my_view`, with `xhair=(56,100,64)`. Make sure the spacing and orientation match ITK-SNAP. The figure should appear like the one above.

5 Part II: Effects of Low-Pass Filtering on Longitudinal Measures

5.1 Apply Affine Transformations to 3D Image (10 points)

5.1.1 Theory

When doing the ITK-SNAP part of this assignment, you experimented with rigid and affine registration modes. Rigid registration finds the 3D rotation and translation of a "moving" image that optimally aligns it with the "fixed" image. Affine registration additionally allows the "moving" image to undergo scaling and shearing. In this part of the assignment, you will write Python code to read an affine transformation computed by an external tool from a file and "apply" it to a moving image. In this assignment, the baseline image plays the role of the "fixed" image and the followup image plays the role of the "moving" image.

Mathematically, a 3D affine transformation is represented by a function $T : \mathbb{R}^3 \to \mathbb{R}^3$ that has the form

$$T(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$$

where A is an invertible 3×3 matrix and $\mathbf{b}$ is a 3×1 vector. Rotation, translation, and shearing are encoded in the matrix A, and translation is encoded in the vector $\mathbf{b}$. The affine transformation T maps every point in $\mathbb{R}^3$ to some other point in $\mathbb{R}^3$, with the important property that if three points $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ are collinear (i.e., they lie on the same line), then the transformed points $T(\mathbf{x}_1)$, $T(\mathbf{x}_2)$, $T(\mathbf{x}_3)$ are also collinear.

After loading the affine transformation parameters (i.e., the numbers that make up A and $\mathbf{b}$) from an external file, your task will be to apply this transformation to the followup image, bringing it into better alignment with the baseline image. To do this, first, you will need to assign a coordinate system to the baseline image. Voxels in a 3D image are organized along rows, columns, and slices. We will define our coordinate system such that the center of the voxel located in the x-th row, y-th column, and z-th slice has coordinates $[x, y, z]^t$. Indexing is zero-based. This means that the center of the very first voxel in the image has coordinates $[0, 0, 0]^t$ and the corners of the very first voxel have coordinates $[-0.5, -0.5, -0.5]^t, [0.5, 0.5, 0.5]^t$. The coordinates of the center of the very last voxel are $[N_x - 1, N_y - 1, N_z - 1]$ where N_x is the number of rows in the image, N_y is the number of columns, and N_z is the number of slices.

Let $\mathbf{x}_P = [x_P, y_P, z_P]^t$ be the coordinates assigned to the voxel P in the baseline image. The affine transformation T maps these coordinates into new coordinates in the space of the followup image.

$$T(\mathbf{x}_P) = A\mathbf{x}_P + \mathbf{b}$$

Applying a transformation T to the followup image involves sampling the intensity values in the followup image at locations $T(\mathbf{x}_P)$ for all voxels P in the baseline image. The result of this operation is the "resampled image" R, given by

$$R(\mathbf{x}_P) = I_F(T(\mathbf{x}_P))$$

where $I_F(\mathbf{x})$ is a function representing the intensity of the follow-up image at coordinate $\mathbf{x}$. It is important to note that I_F is evaluated at coordinates $T(\mathbf{x}_P)$ that are not necessarily at the centers of the voxels. For example, if $\mathbf{x}_P = [7, 4, 12]$, we may have $T(\mathbf{x}_P) = [8.23, 5.23, 11.2]$. This means that the followup image must be *interpolated* at the coordinates $T(\mathbf{x}_P)$. There are different interpolation approaches available, and interpolation has the effect of low-pass filtering the image, as we discuss in class.

5.1.2 Implementation

First, write a function read the matrix A and vector $\mathbf{b}$ from file `f2b.txt` using NumPy function `loadtxt`. The file actually contains a 4x4 matrix in the form below, i.e., the top left corner holds the matrix A and the top of the last column holds the vector $\mathbf{b}$.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & b_1 \\ A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Write a function with the following signature that will return a tuple containing the matrix A and vector $\mathbf{b}$.

```
def my_read_transform(filename):
"""
Read Greedy-style 3D transform (4x4 matrix) from file

:param filename: File name containing transform file
```

```
:returns: tuple (A,b) where A is the 3x3 affine matrix, b is the translation vector
"""
```

Next, write a function that will apply an affine transformation to the followup image, resulting in a resampled image. This function will have the following signature:

```
def my_transform_image(I_ref, I_mov, A, b, method='linear', fill_value=0):
    """
    Transform a moving image into the space of the fixed image

    :param I_ref: 3D voxel array of the fixed (reference) image
    :param I_mov: 3D voxel array of the moving image
    :param A: 3x3 affine transformation matrix
    :param b: 3x1 translation vector
    :param method: Interpolation method (e.g., 'linear', 'nearest')
    :param fill_value: Value with which to replace missing values (e.g., 0)
    :returns: 3D voxel array of the affine-transformed moving image
    """
```

To implement your function, you will need to study and use NumPy command `meshgrid` to generate the coordinates of all the voxels in the fixed image space, i.e., $\mathbf{x}_P$ and the SciPy command `interpolate.interpn` to interpolate the moving image at positions $T(\mathbf{x}_P)$.

Note that to receive full credit, your implementation of this function should not use **for/while** loops to iterate over voxels in the image!

Hints:

- In my code, I had to set `indexing='ij'` in the call to `meshgrid`
- You will need to set `bounds_error=False` in the call to `interpn`
- The parameters `fill_value` and `method` should be passed on to `interpn`

5.1.3 Deliverables

1. The code for the function **my_transform_image**.
2. A figure showing the resampled followup image (shown below), followed by a figure showing the baseline image. Both should be visualized using the `my_view` command with `xhair=(56,100,64)`.

5.2 Effects of Low Pass Filtering on Difference Image Computation (15 points)

In this part of the assignment, you will examine how low pass filtering affects computation of differences between the baseline image and the affine-transformed followup image.

5.2.1 Implement Gaussian Low-Pass Filtering for 3D Volumes

As we discuss in class, Gaussian low-pass filtering has multiple desirable properties, including the property that the complexity of the image (in terms of local extrema) decreases when filtering with a Gaussian. Your task is to implement low-pass filtering with an isotropic 3D Gaussian filter (an

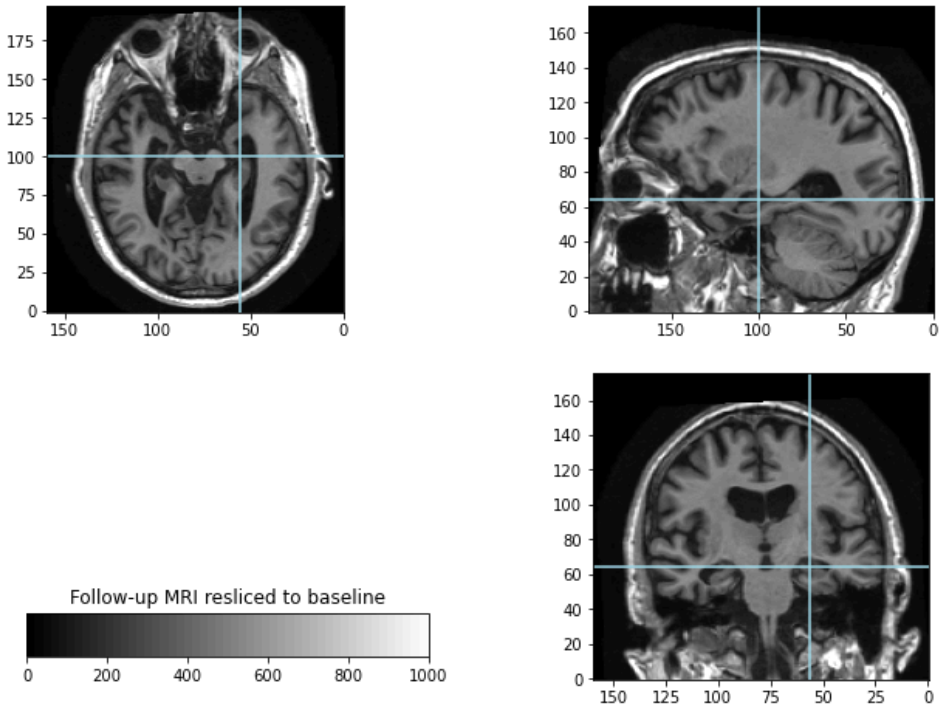

Figure 3: Resliced image plotted with `my_view`

isotropic Gaussian filter is spherically symmetric, i.e., it applies the same amount of smoothing in all directions). The signature for this function will be

```
def my_gaussian_lpf(image, sigma):
    """
    Apply 3D Gaussian low-pass filtering to an image
    :param image: 3D voxel array of the input image
    :param sigma: Standard deviation of the Gaussian kernel, in voxel units
    :returns: 3D voxel array of the filtered iamge
    """
```

where σ is the standard deviation of the Gaussian filter, in units of voxels. The 3D Gaussian filter can be computed as the product of one-dimensional Gaussian filters (decomposition property):

$$G_\sigma(x, y, z) = G_\sigma(x) \cdot G_\sigma(y) \cdot G_\sigma(z)\,,$$

and the one-dimensional Gaussian filter is given by

$$G_\sigma(t) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{t^2}{2\sigma^2}\right).$$

Thus, your task is to (1) generate the Gaussian filter; (2) perform convolution between the image and the Gaussian filter.

5.2.1.1 Generating the Gaussian Filter The Gaussian filter will be a 3D array, the center of which corresponds to $(x, y, z) = 0$. The ideal size of the array is not obvious to determine. The Gaussian function has infinite support, i.e., $G_\sigma(t) > 0$ for all t. However, for $t > 3.5\sigma$, the value of $G_\sigma(t)$ is very small, so it is common to set the size of the array in each dimension to be $2 * \lceil 3.5 * \sigma \rceil + 1$.

Your implementation should not use loops to compute the Gaussian filter. It should also be as efficient as possible, taking advantage of the decomposition property of the 3D Gaussian noted above. Do not use high-level Python functions that compute the Gaussian filter for you.

5.2.1.2 Convolution with the Gaussian Filter You could compute the convolution with the Gaussian filter directly. However, because of the size of the images, this would be very expensive computationally. Instead, we will perform filtering using the fast Fourier transform, i.e., taking advantage of the fact that, given a signal f and a kernel g,

$$f \circ g = F^{-1}\left[F[f] \cdot F[g]\right].$$

The computational cost of performing the fast Fourier transform (FFT) of an image is $O(n \log n)$, where n is the number of voxels in an image, and the computational cost of a convolution is $O(nk)$, where k is the number of voxels in the kernel. Thus, unless our kernels are very small, it is more efficient to use the Fourier transform to perform filtering.

Using FFT to perform convolution in Python is easy using the SciPy function `signal.fftconvolve`

5.2.1.3 Testing your Gaussian Filter To test if your function `my_gaussian_lpf` works, create an image equivalent of a delta function: an image of zeros with a single 1 in the center. Apply your Gaussian LPF function to this image, and plot the output using `my_view`. Include in your report the result for $\sigma = 4$, with the delta image of the size $31 \times 31 \times 31$. Your result should look like this:

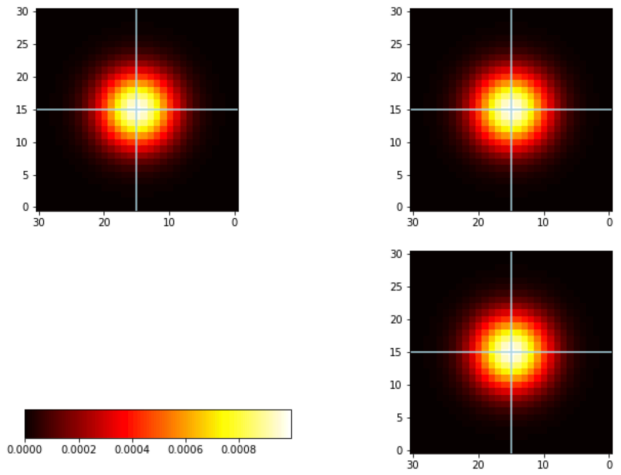

Figure 4: Gaussian kernel in `my_view`

5.2.2 Implement Mean Filtering

This is easy! Implement the function

```
def my_mean_lpf(image, radius):
    """Apply 3D mean filtering to an image"""
```

to perform low pass filtering with the mean filter. The mean filter is just a numeric representation of the *rect* function, i.e., a constant 3D array that adds up to one. The parameter `radius` gives the size of the mean filter, i.e., if radius is 1, the filter is a $3 \times 3 \times 3$ array, if radius is 2, the filter is a $5 \times 5 \times 5$, etc.

5.2.3 Compute and Show Difference Images

The objective of this section is to examine how the difference images between the baseline image and the resampled followup image are affected by different low-pass filtering options applied to the baseline and followup images. More specifically, you will compute the difference image

$$D(\mathbf{x}) = R\left[(I_F \circ K_F)\right](\mathbf{x}) - (I_B \circ K_B)(\mathbf{x})$$

where I_B and I_F are the baseline and follow-up images; K_B and K_F denote low-pass filtering kernels which are then applied to the baseline and follow-up images to obtain the filtered images $(I_B \circ K_B)$ and $(I_F \circ K_F)$ respectively; and R denotes the resampling operation, i.e. applying the function `my_transform_image` to an image.

Compute difference images for the following combination of low-pass filtering and interpolation options.

K_B	K_F	Interpolation
None	None	Nearest Neighbor
None	None	Linear
Gaussian, $\sigma = 2$	Gaussian, $\sigma = 2$	Linear
Mean, radius=2	Mean, radius=2	Linear

Display the difference images with `my_view`, as illustrated below. Please use the same intensity range (`crange`) for the four figures so that the color maps are consistent.

5.2.4 Deliverables

1. The code for the functions defined above.
2. Figure of the Gaussian filter, as described above under "Testing your Gaussian Filter."
3. Figure of the baseline image filtered with the Gaussian filter at $\sigma = 2$ and filtered with the mean filter with radius 2.
4. Figure with the four difference images from the table above.

5.3 Quantify Intensity Difference over Regions of Interest (10 points)

Next, we will quantify the root mean square (RMS) intensity difference between the baseline image and the transformed followup image over anatomical regions of interest (ROIs) defined in the

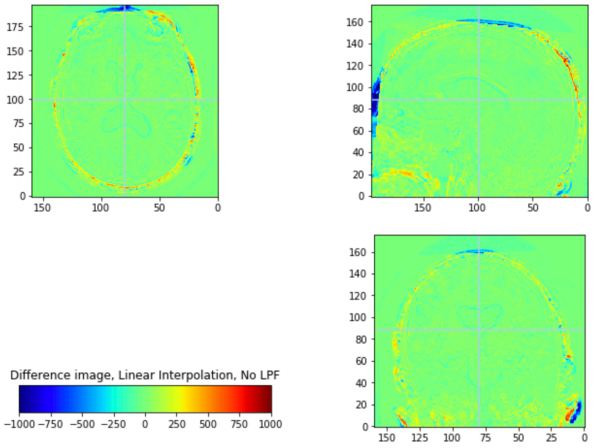

Figure 5: Difference image

baseline image. These ROIs are encoded in the segmentation image **seg.nii**. This image assigns a different label to every voxel, as follows:

Label	Anatomical ROI
1	Left Hippocampus
2	Right Hippocampus
3	Lateral Ventricles

An anatomical ROI is the set of all voxels assigned a particular label. The RMS intensity difference over an ROI is computed simply as

$$\mathrm{RMS}[\mathrm{ROI}] = \left[\frac{1}{|\mathrm{ROI}|} \sum_{\mathbf{x}_i \in \mathrm{ROI}} D(\mathbf{x}_i)^2 \right]^{\frac{1}{2}}$$

where $|\mathrm{ROI}|$ denotes the size of the ROI.

Write a function to calculate the above expression, with the following signature:

```
def my_rms_over_roi(image, seg, label):
    """Compute RMS of a difference image over a label in the segmentation"""
```

Note that to receive full credit, your implementation of this function should not use **for/while** loops to iterate over voxels. Instead, you may find the function `np.where` useful for extracting voxels corresponding to each label.

Finally, generate plots of RMS intensity difference as a function of the parameters of low-pass filtering kernels. For Gaussian filtering, use the range between 0.5 and 5.0 with increment 0.5 for the parameter σ and apply low pass filtering with each σ to both baseline and followup images; transform the filtered followup image; and compute the RMS intensity difference. For the mean filter, use the range from 1 to 10 with increment 1 for the radius parameter.

Think about why the plots you obtain look like they do. Write a short explanatory paragraph.

5.3.1 Deliverables

1. The code for the function defined above and the code for making the curves.
2. Two plots of RMS vs. low pass filtering parameter for Gaussian and Mean LPF, respectively. Use different color curves on the same plot for different anatomical labels. Include a legend.
3. A short explanatory paragraph

6 Part III: Affine Registration with PyTorch

This is the last and most advanced part of the assignment. We will implement some of the functionality above in PyTorch, to prepare us for the next assignment. First, we will code the functionality to apply affine transformations to 3D images using PyTorch routines `affine_grid` and `grid_sample`. Then, almost for free, we will implement an automated affine image registration algorithm!

Before starting this assignment you should complete the PyTorch "Learn the Basics" tutorials

To use PyTorch, add the following lines to the top cell of your notebook:

```
import torch
import torch.nn.functional as tfun
```

Note that if you experience a crash when calling `affine_grid` later in the assignment, add this line to the top cell as well.

```
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

6.1 Loading 3D images into PyTorch tensors (5 points)

Use the following code to convert our 3D NumPy voxel arrays into PyTorch tensors:

```
T_bl = torch.from_numpy(I_bl).unsqueeze(0).unsqueeze(0)
```

- This code will produce a 5-D tensor with dimensions $[1, 1, N_x, N_y, N_z]$ where N_x, N_y, N_z are the dimensions of the NumPy array.
- The first two dimension are the "minibatch" and "channel" dimensions.
  - When using PyTorch for deep learning, we would be processing multiple mini-batches of data in parallel (e.g., multiple images at once). In this assignment, the minibatch will be of size 1.
  - When working with color images, the red, green, and blue channels of the image would be stored together as parts of a single tensor. Here we are not using color images, so the channel dimension is also 1.
  - The reason we add the "minibatch" and "channel" dimensions is that PyTorch routine `grid_sample` expects a 5-D tensor when working with 3D images (see documentation).

Conversely, to view one of these 5-D tensors using our existing function `my_view`, we would write:

```
my_view(T_bl.squeeze().detach().cpu().numpy(), hdr_bl, cmap='gray')
```

- The code above works even if `T_bl` is part of a PyTorch computational graph (data structure used for automatic differentiation) and is loaded into the GPU.
  - `detach()` (see documentation) returns a new tensor that is detached from the computational graph and not involved in auto-differentiation

- `cpu()` copies the tensor into CPU memory (unless it is already on the CPU)
- `numpy()` converts the tensor into a NumPy array

6.1.1 Deliverables

- Demonstrate the ability to load the baseline image into a tensor and visualize with `my_view`

6.2 Applying Affine Transformations in PyTorch (15 points)

6.2.1 Different Coordinate Systems

In the NumPy part of the assignment, we used a coordinate system where the 3D coordinate of the **center** of the voxel with index a, b, c was just (a, b, c). The center of the voxel at one corner of the image had the coordinates $(0, 0, 0)$ and the center of the voxel at the opposite corner of the image had coordinates $(S_x - 1, S_y - 1, S_z - 1)$, where (S_x, S_y, S_z) are the image dimensions. In PyTorch, functions `affine_grid` and `grid_sample` use a different coordinate system that is *normalized* to the range $[-1, 1]$. One **corner** of the image has the coordinate $(-1, -1, -1)$ and the opposite corner has the coordinate $(1, 1, 1)$. The difference between these coordinate systems is illustrated below for an image of size $4 \times 4 \times 4$.

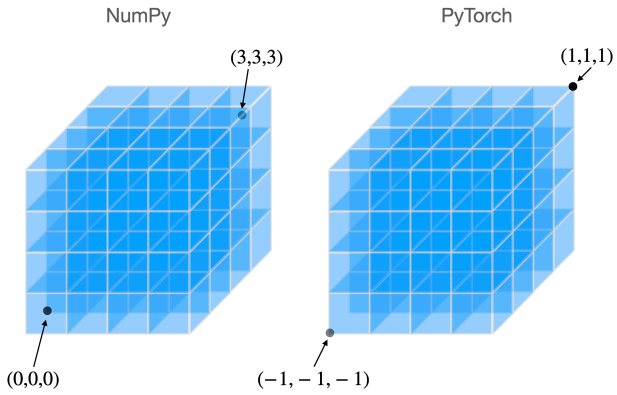

Figure 6: NumPy and PyTorch coordinate systems

Additionally, the order of coordinates is reversed compared to NumPy. So, for example, in an $100 \times 100 \times 100$ voxel image, the pixel with the index $[5, 25, 50]$ will have the normalized PyTorch coordinates $(1.0, 0.5, 0.1)$.

The change of coordinates between NumPy and PyTorch can be represented as an affine transformation

$$\mathbf{x}' = W\mathbf{x} + \mathbf{z}$$

where

$$W = \begin{bmatrix} 0 & 0 & \frac{2}{S_z} \\ 0 & \frac{2}{S_y} & 0 \\ \frac{2}{S_x} & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{z} = \begin{bmatrix} \frac{1}{S_z} - 1 \\ \frac{1}{S_y} - 1 \\ \frac{1}{S_x} - 1 \end{bmatrix}.$$

The affine transformation $A, \mathbf{b}$ that you loaded from the text file `f2b.txt` is defined in the NumPy style coordinate system. It does not make sense for the normalized PyTorch coordinates. For example, the z-component of $\mathbf{b}$ is 23.4575 which means tranlation by 23 voxels in NumPy coordinates, but in PyTorch coordinates it means translation by almost 12 image extents. We need to modify $A, \mathbf{b}$ into a corresponding affine transformation $A', \mathbf{b}'$ in PyTorch coordinates. We can do so by solving the following equation for A' and $\mathbf{b}'$:

$$W(A\mathbf{x} + \mathbf{b}) + \mathbf{z} = A'(W\mathbf{x} + \mathbf{z}) + \mathbf{b}' \quad \forall \mathbf{x} \in \mathbb{R}^3$$

Solve this equation, and include the derivation in your notebook. *Hint:* if you have difficulty solving this equation, try solving the 1D version of it first. In one dimension, $A, W, \mathbf{b}, \mathbf{z}$ are just scalars. Also, make use of the fact that the equation holds for every value of $\mathbf{x}$.

Next, write a pair of functions, with signatures below, that convert affine transformations between the NumPy and PyTorch coordinate systems:

```
def my_numpy_affine_to_pytorch_affine(A, b, img_size):
    """
    Convert affine transform (A,b) from NumPy to PyTorch coordinates

    :param A: affine matrix, represented as a shape (3,3) NumPy array
    :param b: translation vector, represented as a shape (3) NumPy array
    :returns: tuple of NumPy arrays (A',b') holding affine transform in PyTorch coords
    """


def my_pytorch_affine_to_numpy_affine(A, b, img_size):
    """
    Convert affine transform (A,b) from PyTorch to NumPy coordinates

    :param A: affine matrix, represented as a shape (3,3) NumPy array
    :param b: translation vector, represented as a shape (3)NumPy array
    :returns: tuple of NumPy arrays (A',b') holding affine transform in NumPy coords
    """
```

If your code is correct and `A` and `b` refer to the affine transformation loaded from `f2b.txt`, then the following code

```
A_prime, b_prime = my_numpy_affine_to_pytorch_affine(A, b, I_bl.shape)
A_prime, b_prime
```

will produce the following output

```
(array([[ 0.9939    , -0.120375  ,  0.04881818],
        [ 0.08791111,  1.001     ,  0.03361616],
        [-0.04499   , -0.0314325 ,  1.0003    ]]),
 array([ 0.18924318,  0.06270253, -0.00460125]))
```

and this code

```
A_test, b_test = my_pytorch_affine_to_numpy_affine(A_prime, b_prime, I_bl.shape)
A_test, b_test
```

will produce the output

```
(array([[ 1.0003, -0.0254, -0.0409],
        [ 0.0416,  1.001 ,  0.0989],
        [ 0.0537, -0.107 ,  0.9939]]),
 array([ 5.6887, -5.8519, 23.4575]))
```

Include the code above in your notebook to show that your functions work correctly.

6.2.2 Funciton to Apply Affine Transformation

Now, write a function with the following signature that will apply an affine transformation to a moving image, but using PyTorch routines `affine_grid` and `grid_sample`.

```
def my_transform_image_pytorch(T_ref, T_mov, T_A, T_b,
                               mode='bilinear', padding_mode='zeros'):
    """
    Apply an affine transform to 3D images represented as PyTorch tensors

    :param T_ref: Fixed (reference) image, represented as a 5D tensor
    :param T_mov: Moving image, represented as a 5D tensor
    :param T_A: affine matrix in PyTorch coordinate space, represented as a shape (3,3) tensor
    :param T_b: translation vector in PyTorch coordinate space, represented as a shape (3) tensor
    :param mode: Interpolation mode, see grid_sample
    :param padding_mode: Padding mode, see grid_sample
    :returns: Transformed moving image, represented as a 5D tensor
    """
```

When implementing this function, note:

- The logic of the function will be similar to `my_transform_image`
- The function `affine_grid` can be called by providing the affine matrix as an input. However, to keep the logic similar to `my_transform_image`, I recommend calling it with an identity matrix as input (use the code `torch.eye(3,4).unsqueeze(0)` to generate it) and then manually applying the transformation $\mathbf{y} = A'\mathbf{x} + \mathbf{b}'$ to the components of the grid tensor.
- When calling `affine_grid` and `sample_grid`, set the option `align_corners = False`
- Notice that up to now, we represented affine matrices and translation vectors as NumPy arrays. In this function, we expect them to be input as PyTorch tensors. This is so that we can use PyTorch auto-differentiation features to compute derivatives with respect to the affine transform parameters. To convert a NumPy array to a PyTorch tensor, use the `torch.tensor` function

This function should generate **approximately the same output** as `my_transform_image` for the same inputs. To test if this is indeed the case, include the following test code in your assignment:

```
# Convert the affine transform (A,b) to a PyTorch affine transform
A_prime, b_prime = my_numpy_affine_to_pytorch_affine(A, b, I_bl.shape)
T_A, T_b = torch.tensor(A_prime), torch.tensor(b_prime)

# Convert the baseline and follow-up images to PyTorch tensors
```

```
T_bl = torch.from_numpy(I_bl).unsqueeze(0).unsqueeze(0)
T_fu = torch.from_numpy(I_fu).unsqueeze(0).unsqueeze(0)

# Apply the transform to the follow-up image using PyTorch
T_fu_reslice = my_transform_image_pytorch(T_bl, T_fu, T_A, T_b)

# Apply the transform to the follow-up image using NumPy
I_fu_reslice = my_transform_image(I_bl, I_fu, A, b)

# Compute the difference between two ways of resampling
I_diff = I_fu_reslice - T_fu_reslice.squeeze().detach().cpu().numpy()

# Compute RMS difference between interpolated images over our labels
for label in (1,2,3):
    print('RMS over label %d is %f' % (label, my_rms_over_roi(I_diff, I_seg, label)))

# Visualize the difference image
my_view(I_diff, xhair=(56,100,64), crange=[-50,50], cmap='jet')
plt.title('Difference between images resliced with NumPy and PyTorch');
```

You should observe that the image `I_diff` is very close to zero (as shown below, there will be some non-zero differences near the edges due to slightly different ways in which PyTorch and SciPy implement interpolation routines) and the RMS values printed out in the `print` statement should be below 0.001.

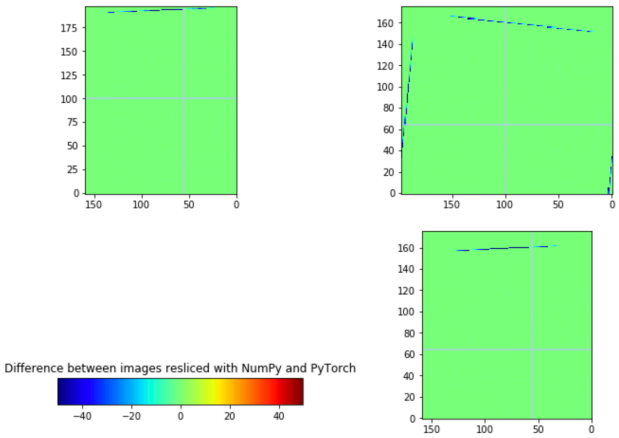

Figure 7: Difference image `I_diff`

6.2.3 Deliverables

- Derivation of A' and $\mathbf{b}'$
- Code for the functions `my_numpy_affine_to_pytorch`, `my_pytorch_affine_to_numpy` and `my_transform_image_pytorch`.
- Results of running the code above showing that these functions work correctly

6.3 Affine Registration using LBFGS Optimizer (10 points)

Finally we will witness the power of PyTorch to compute derivatives automatically and perform numerical optimization.

6.3.1 Objective Function for Affine Registration

Write a function with the following signature, which will compute the RMS difference between a reference image and the transformed moving image. This function will serve as the *objective function* for numerical optimization. By minimizing the difference between the fixed image and moving image as a function of the registration parameters A and $\mathbf{b}$, we will perform affine registration. Most of the work for this function will be done by calling `my_transform_image_pytorch`

```
def my_affine_objective_fn(T_ref, T_mov, T_A, T_b):
    """
    Compute the affine registration objective function

    :param T_ref: Fixed (reference) image, represented as a 5D tensor
    :param T_mov: Moving image, represented as a 5D tensor
    :param T_A: affine matrix in PyTorch coordinate space, represented as a shape (3,3) tensor
    :param T_b: translation vector in PyTorch coordinate space, represented as a shape (3) tensor
    :returns: RMS difference between the reference image and transformed moving image
    """
```

Calling this function using the variables `T_bl, T_fu, T_A, T_b` defined above should return

```
tensor(87.0636, dtype=torch.float64, grad_fn=<SqrtBackward>)
```

Moreover, we can **automatically** compute the partial derivatives of the objective function with respect to A' and $\mathbf{b}'$ like this:

```
# Create tensors T_A and T_b and track their partial derivatives
T_A = torch.tensor(A_prime, requires_grad=True)
T_b = torch.tensor(b_prime, requires_grad=True)

# Compute the objective function (forward pass)
obj = my_affine_objective_fn(T_bl, T_fu, T_A, T_b)

# Compute the partial derivatives of the objective function with respect to
# elements of T_A and T_b automatically (backward pass)
obj.backward()

# Print the objective function value and partial derivatives
obj, T_A.grad, T_b.grad
```

which will output something like

```
(tensor(87.0636, dtype=torch.float64, grad_fn=<SqrtBackward>),
 tensor([[-14.9921, -78.2015,  29.2921],
         [ 53.0794, 164.1188,   3.7942],
         [-37.4370,  42.8359, -68.0925]], dtype=torch.float64),
 tensor([ 105.6620, -576.1331,  329.4097], dtype=torch.float64))
```

6.3.2 Minimizing the Objective Function

Numerical optimization repeatedly evaluates the objective function and its partial derivatives to find parameter values that minimize the objective function. Figuring out the options for calling the LBFGS optimizer in PyTorch is not trivial, so just use the code below to do the optimization:

```
# Starting point for optimization - identity affine transform. Note that
# the LBFGS implementation in PyTorch requires all the parameters (i.e.
# variables that we are optimizing over) to be contained in a single
# tensor, which we call T_opt
T_opt = torch.tensor(np.eye(4,4), requires_grad=True)

# Objective function for optimization, a wrapper around my_affine_objective_fn
f_opt = lambda : my_affine_objective_fn(T_bl, T_fu, T_opt[0:3,0:3], T_opt[0:3,3])

# Initialize the LBFGS optimizer with a line search routine
optimizer = torch.optim.LBFGS([T_opt],
                              history_size=10,
                              max_iter=4,
                              line_search_fn="strong_wolfe")

# Keep track of the objective function values over the course of optimization
opt_history = []

# Run for a few iterations
for i in range(50):
    optimizer.zero_grad()
    objective = f_opt()
    objective.backward()
    optimizer.step(f_opt)
    opt_history.append(objective.item())
    print('Iter %03d   Obj %8.4f' % (i, objective.item()))
```

- Use this optimization code and visualize the resliced moving image after optimization using `my_view`. Also plot the change in objective function over iteration of the optimization algorithm (contained in array `opt_history`).
- Convert the matrix A' and vector $\mathbf{b}'$ computed by the optimization to a NumPy space matrix and compare to the affine transform loaded earlier from `f2b.txt`. They should be similar, since the latter was obtained by performing affine registration in ITK-SNAP.
- Examine whether applying Gaussian smoothing to the baseline and moving images before doing optimization impacts affine registration. You can use a single value of σ for this experment, e.g., $\sigma = 5$. Does the registration converge faster? Does it produce an affine transform that is very different?

6.3.3 Deliverables

- Plot of `opt_history` showing the change in objective function over iteration of the optimization algorithm

- Plot of the resliced moving image after registration
- Optimal affine transform converted to the NumPy coordinate system
- Results of the experiment examining Gaussian smoothing effects on registration