# First MATLAB Programming Assignment

CIS 537 / BE 537

September 12, 2017

This assignment is due via Canvas October 9 at 3:00 pm.

## 1 Introduction

The goal of this programming assignment is to help you become familiar with the basic tools of 3D image processing in the MATLAB environment. This assignment is motivated by the problem of longitudinal image registration. Comparison of MRI scans acquired at different time points is a powerful tool for quantifying the effects of age and disease on the brain. Changes in the volume of brain structures as small as one or two percent can be detected by comparing MRI scans. However, such changes can be easily confounded by various methodological biases introduced by careless application of image processing. The paper by Fox et al., 2011, assigned for the *Applications of Image Normalization* lecture is a good source for learning about the pitfalls of longitudinal image processing. In this assignment, we will only touch the surface of this problem. We will examine how different approaches to image interpolation and filtering affect the differences measured between longitudinal images.

**Please give yourselves at least 2 weeks to finish this assignment – more if you are not familar with MATLAB.**

## 2 Data

The following data will be provided for one subject.

1. A T1-weighted MRI scan acquired at the beginning of the longitudinal study (**baseline.nii**)
2. A T1-weighed MRI scan acquired three years later (**followup.nii**)
3. An image encoding 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.

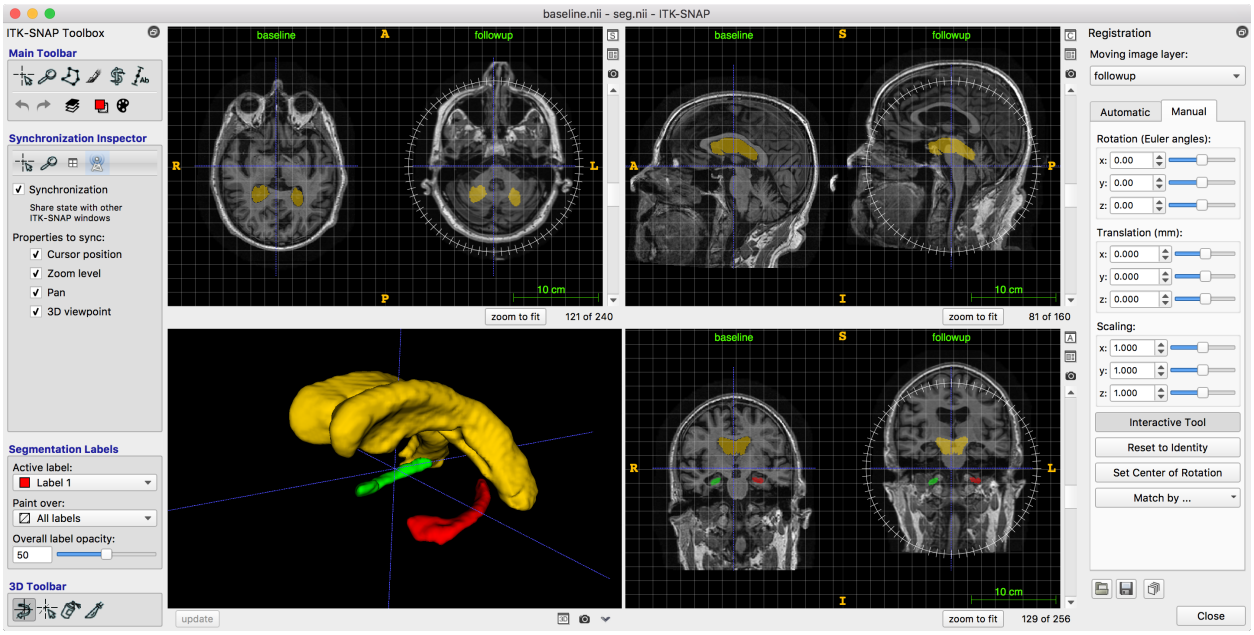


Figure 1: Example of the two images and segmentation viewed in ITK-SNAP.

# 3 Tasks and Deliverables

## 3.1 Explore the Data in ITK-SNAP

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

- Obtain and install the ITK-SNAP 3.6 binary for your architecture from 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[1].
- Load the segmentation of the baseline image and use the "Update Mesh" button 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.

Figure 1 shows an example of these images viewed in ITK-SNAP.

**Deliverables.** There are no deliverables for this step.

[1] 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).

## 3.2 Image IO and Display in ITK-SNAP

1. Write a MATLAB function that loads an image into a 3D array and returns the image and the voxel spacing of the image. The signature for this function should be

$$[\texttt{image}, \texttt{spacing}] = \texttt{myReadNifti}(\texttt{filename})$$

where **spacing** is a $3 \times 1$ vector and **image** is a 3D array. This function will just be a wrapper around the **load_nii** function in the provided NIFTI IO library[2]. *Note: to avoid problems down the line, make sure to convert the image volume to double precision using the* ***double*** *command.*

2. Write a function that saves the image, given the spacing and image volume. This function will have the signature

$$\texttt{myWriteNifti}(\texttt{filename}, \texttt{image}, \texttt{spacing})$$

and it will be a wrapper around the **make_nii** and **save_nii** functions. Use the 32 bit floating point datatype when saving the image (see help for **save_nii**).

3. Write a MATLAB function to open an image in ITK-SNAP by first saving it to a temporary file, and then launching ITK-SNAP. The function should have the signature

$$\texttt{myViewInSNAP}(\texttt{image}, \texttt{spacing}).$$

Use MATLAB command **sprintf** to generate the command line to execute; **tempname** to generate a temporary filename (need to add extension '.nii' in the end); **system** to launch SNAP; and **delete** to delete the temporary file. To tell ITK-SNAP to open an image, you will need to call the executable with the **-g** option. Call it with the **-h** option to see the list of all command line options. On Windows, the ITK-SNAP executable will be located in

$$\texttt{c:}\backslash\texttt{Program Files}\backslash\texttt{ITK} - \texttt{SNAP}$$

and on the Mac, it is in

$$/\texttt{Applications}/\texttt{ITK} - \texttt{SNAP.app/Contents/MacOS/InsightSNAP}$$

**Deliverables:**

1. The code for the functions described above.

## 3.3 Three-Slice Image Display in MATLAB

Write a function to generate a visualization of a 3D image similar to ITK-SNAP, as shown in Figure 2. This function will divide the plot window into four subplots, and in three of the subplots, 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 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). Your function will have the following signature:

[2]The output of **load_nii** is a MATLAB structure (**struct**). To extract spacing from it, use the field "hdr.dime.pixdim(2:4)"

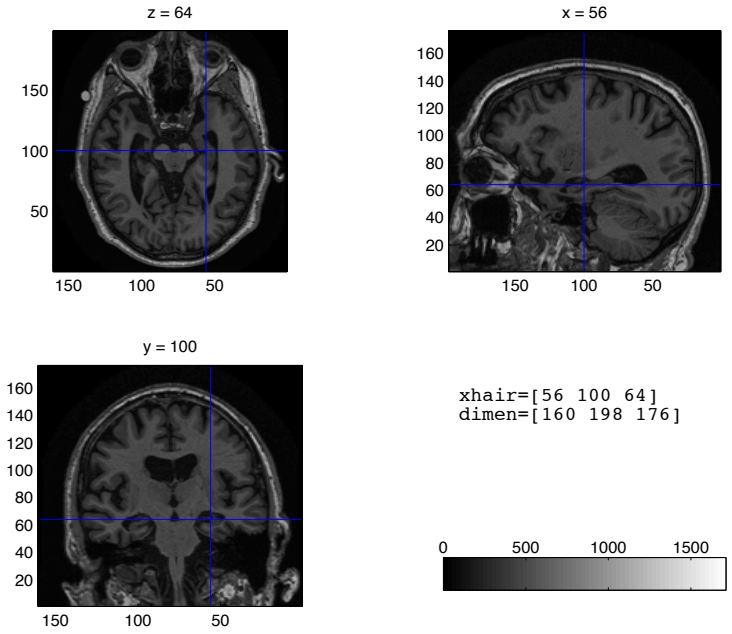


Figure 2: Example of a correct implementation of the myView command.

```
myView: Display orthogonal slices through a 3D image
usage:
    myView(image,spacing,crosshair,crange,cmap)
required parameters:
    image                   3D image volume
    spacing                 3x1 vector of voxel spacings
optional parameters:
    crosshair               3x1 vector giving the crosshair position.
                            Defaults to the center of the image.
    crange                  1x2 vector giving the range of intensity to
                            be displayed. Defaults to [imin imax], where
                            imin and imax are the minimum and maximum
                            intensity in the image.
    cmap                    String, giving the name of the color map
                            to use (see colormap). Defaults to 'gray'.
```

Make sure that the three views are oriented in the same way as when the images are displayed in ITK-SNAP using **myViewInSNAP** command. Also make sure that the plotted slices are correctly stretched, i.e., that the aspect ratio of each subplot is proportional to the voxel spacing.

Here are a few hints to make this work:

- Call **doc nargin** for help on functions with optional parameters.
- Read MATLAB help on multidimensional arrays to learn how to extract 2D slices from 3D arrays. In particular, you will need the **squeeze** command.
- Use the **imagesc** command to display images. Use **caxis** to set the color axis limits, and **colormap** to select the colormap.

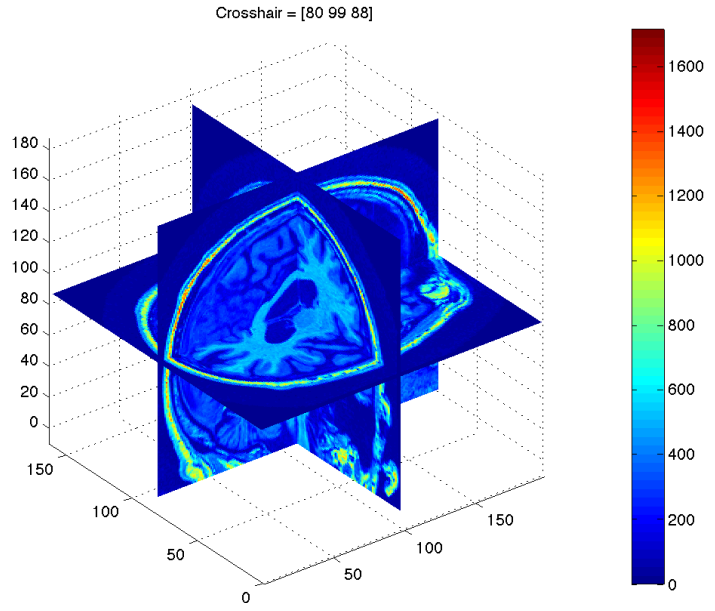


Figure 3: Example of a correct implementation of the myView3D command.

- Use command **set(gca,'XDir','reverse')** and **set(gca,'YDir','normal')** to reverse the direction of axes in the plots. This will help you orient the slice display similarly to ITK-SNAP.
- Use the **subplot** command to generate multiple plots in one view.
- Use the **daspect** command to control the aspect ratio of each plot.
- Use **line** to draw the crosshairs.
- Use **text** and **colorbar** for the right-bottom subplot.

**Deliverables:**

1. The code for the function **myView**.
2. A figure showing side by side the baseline image displayed in ITK-SNAP and the same image displayed by **myView**, with the crosshair in the same location.

## 3.4 3D Visualization of Orthogonal Slices

Generate a function to view the three orthogonal slice planes in a single 3D MATLAB plot, with slices shown as crossing planes. See Figure 3 for the example. This function will use the MATLAB **slice** command. The signature will be similar to **myView**:

`myView3D(image, spacing, crosshair, crange, cmap)`

with the last three parameters being optional.

Here are some hints to make this work:

- Be sure to read the help for the **slice** command carefully. Note that the volume is expected to be a $N_y \times N_x \times N_z$ array, not a $N_x \times N_y \times N_z$ array. So you will need to permute the order of the parameters to the **slice** command accordingly.
- The **slice** command will crash if the image is not in double precision format. See the instructions for loading images above.
- The **slice** command tries to draw edges around every pixel, which makes the image very hard to see. Use the following syntax to hide these edges:

$$\texttt{h} = \texttt{slice}(...);$$

$$\texttt{set}(\texttt{h},'\texttt{EdgeColor}','\texttt{none}');$$

- Similar to section 3.3, use **caxis** to set the color axis limits, **colormap** to select the colormap, **colorbar** to show colorbar and **daspect** to control the aspect ratio.

**Deliverables:**

1. The code for the function **myView3D**.
2. A figure showing the baseline image displayed by **myView3D**.

## 3.5 Apply Affine Transformations to 3D Image

### 3.5.1 Theory

The term "3D affine transformation" refers to a mapping $T : \mathbb{R}^3 \to \mathbb{R}^3$ that is linear. Geometrically, the affine transformation has the property that it preserves relative distances between points. For example if points A, B and C lie on the same line, and the distance from $A$ to $B$ is twice the distance from $B$ to $C$, then after applying the affine transform $T$ to the three points, the distance from $T(A)$ to $T(B)$ will still be twice the distance from $T(B)$ to $T(C)$. In image analysis, affine registration refers to a registration algorithm that finds the affine transform $T$ that optimally aligns a pair of images.

Algebraically, the 3D affine transformation is described by a 4x4 matrix, which encodes the translation, rotation, scaling and shearing between the two images. The matrix is of the form

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \mathbf{b}_1 \\ A_{21} & A_{22} & A_{23} & \mathbf{b}_2 \\ A_{31} & A_{32} & A_{33} & \mathbf{b}_3 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $A$ is a 3 by 3 matrix of rotation, scaling and shearing, and $\mathbf{b}$ is a translation vector. In our case, the affine transformation between the baseline image and the followup image is already provided to us, having been computed by an image registration program FLIRT. Our task will be to apply this transformation to the followup image.

First, we must associate each voxel $P$ in the image with a set of coordinates. Each voxel is a 3D box. Voxels are organized along rows, columns, and slices in the image. 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 $[x, y, z]^t$. This

means that the center of the very first voxel in the image has coordinates $[1,1,1]^t$ while the corners of the very first voxel have coordinates $[0.5,0.5,0.5]^t, [1.5,0.5,0.5]^t$, and so on.

Let $\mathbf{x}_P = [x_P, y_P, z_P]^t$ be the coordinates of a voxel P in the baseline image. Then the coordinates of the corresponding position in the followup image are

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

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

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

It is important to note that the followup image $I_F$ 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 that are available, and interpolation has the effect of low-pass filtering the image, as we will discuss in class.

### 3.5.2 Implementation

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:

$$\texttt{transform} = \texttt{myTransformImage}(\texttt{fixed}, \texttt{moving}, \texttt{A}, \texttt{b}, \texttt{method}),$$

where **fixed**, **moving**, and **transform** are all 3D arrays, **A** is a $3 \times 3$ matrix, **b** is a $3 \times 1$ vector, and **method** is a an optional string parameter specifying which interpolation scheme to use (choices are 'linear', 'nearest', 'cubic'; see the help for **interpn** command).

Note that to receive full credit, your implementation of this function should not use **for**/**while** loops!

Here are some hints for implementing this function:

1. Use the command **data=load('-ascii','filename.txt')** to load the matrix $M$ from the file.
2. Use the command **ndgrid** to generate the arrays of $x$, $y$ and $z$ coordinates for each voxel in the fixed image.
3. Use the command **interpn** to interpolate the moving image at a list of $x$, $y$ and $z$ coordinates. To assign 0 intensity to voxels that map outside of the moving image domain, you can pass "0" to **interpn** after "method" argument.
4. Use the syntax **Z(:)** to extract all elements in a 3D array **Z** as a 1D flat array, and use the command **reshape** to reshape a 1D array back to a 3D volume.
5. If you encounter out of memory errors, clean up memory with the **clear** command and try to rewrite your code to use as little memory as possible (e.g., perform operations in-place).

**Deliverables:**

1. The code for the function **myTransformImage**.
2. A figure showing the resampled followup image side by side with the baseline image, with crosshairs in the same location. Both should be visualized using the **myView** command.

## 3.6 Examine the Effects of Low Pass Filtering on Difference Image Computation

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

### 3.6.1 Implement Gaussian Low-Pass Filtering for 3D Volumes

As we discussed 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 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

$$\texttt{filtered} = \texttt{myGaussianLPF}(\texttt{image}, \texttt{sigma})$$

where $\sigma$ 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:

$$G_\sigma(x, y, z) = G_\sigma(x) \cdot G_\sigma(y) \cdot G_\sigma(z)\,, \tag{1}$$

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.

**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 $t > 3\sigma$, 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 $2 * \lceil 3 * \sigma \rceil + 1$. To deal with non-integer $\sigma$, you may find command **ceil** useful.

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, i.e., Equation (1) . Hint: you can take advantage of the **ndgrid** command to implement this function efficiently. Do not use built-in MATLAB functions to compute the Gaussian filter.

**Convolution with the Gaussian Filter** In theory, you could compute the convolution with the Gaussian filter using the **convn** function in MATLAB. However, because of the size of the images, this would be very expensive computationally. Instead, we will perform filtering using the Fourier transform, i.e., taking advantage of the fact that, given a signal $f$ and a kernel $g$,

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

The computational cost of performing the Fourier transform 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.

Since we have not focused on Fourier transforms in class, I have provided you with a function **imfilter3d**, which performs convolution via the Fourier transform. The output of **imfilter3d(image,kernel)** should be the same as the output of **convn(image,kernel,'same')**. Your function should call **imfilter3d**.

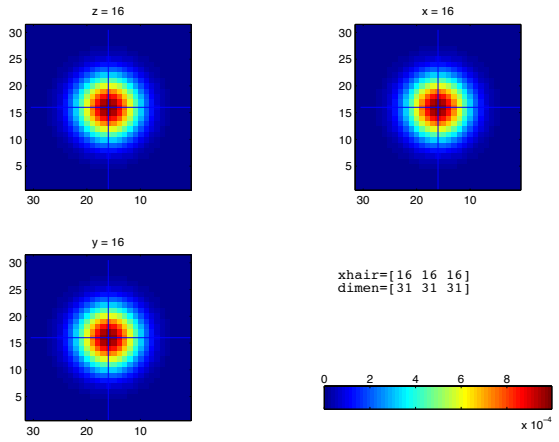


Figure 4: Visualizing the Gaussian filter.

**Testing your Gaussian Filter**

To test if your function **myGaussianLPF** 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 **myView**. 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 Figure 4 above.

### 3.6.2 Implement Mean Filtering

This is easy! Implement the function

$$\texttt{filtered} = \texttt{myMeanLPF}(\texttt{image}, \texttt{radius})$$

to perform low pass filtering with the mean filter. The mean filter is just a numeric representation of the rect function, i.e., a 3D array of constant values that add up to one. The parameter radius gives the size of the mean filter, i.e., if radius is 1, the filter is a 3x3x3 array, if radius is 2, the filter is a 5x5x5 array and so on.

### 3.6.3 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

$$D(\mathbf{x}) = (B \circ K_B)(\mathbf{x}) - \mathcal{R}\left[(F \circ K_F)\right](\mathbf{x})$$

where $B$ and $F$ are the baseline and follow-up images; $K_B$ and $K_F$ denote low-pass filtering kernels applied to the baseline and followup images; and $\mathcal{R}$ denotes the resampling operation, i.e. applying the function **myTransformImage** to an image. Compute difference images for the following combination of low-pass filtering and interpolation options, and display them with **myView**. Figure 5 shows an example of what the result should look like.

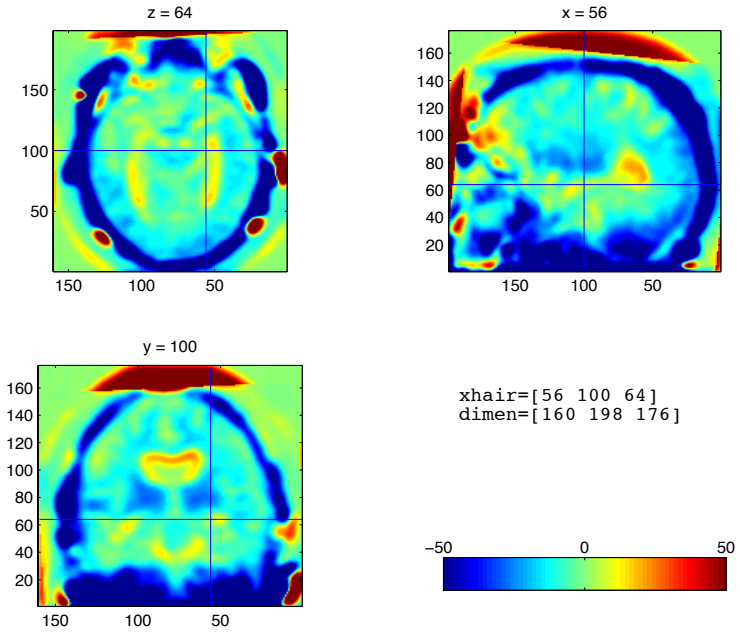


Figure 5: A difference image obtained with Gaussian low pass filtering of baseline and followup images with $\sigma = 3$.

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

**Deliverables:**

1. The code for the functions defined above and the code for making the difference images.
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 (please adjust the range of intensity to be displayed for better visualization).

### 3.6.4 Quantify Intensity Difference over Regions of Interest

Last, we will measure 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 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 |ROI| denotes the size of the ROI.

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

$$\texttt{value} = \texttt{myRMSOverROI}(\texttt{image}, \texttt{seg}, \texttt{label})$$

Note that to receive full credit, your implementation of this function should not use **for**/**while** loops!

Finally, generate plots of RMS intensity difference as a function of the parameters of low-pass filtering kernels. For Gaussian filtering, use the range [0:0.5:5] for the parameter $\sigma$ and apply low pass filtering with each $\sigma$ to both baseline and followup images; transform the filtered followup image; and compute the RMS intensity difference. For the mean filter, use the range [0:10] for the radius parameter.

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

**Deliverables:**

1. The code for the function defined above and the code for making the curves.
2. Six plots of RMS vs. low pass filtering parameter (for the six possible combinations of anatomical ROI and type of low pass filter).
3. A short explanatory paragraph

## 3.7 What to Submit

1. All of your MATLAB source code and figures. Please put them in a zip file. Make sure that it is documented, so one can follow what you are doing.
2. A short report in PDF format (please don‘t put it into the zip file, upload it seperatly), including the figures and the explanatory paragraph in section 3.6.4.

## 3.8 How to submit

Please upload your assignment to Canvas. You don‘t need to print out your report and hand it in.