WK2_Image_Analysis_Task

January 23, 2020

1 Image Analysis Task

In this programming task, we'll get to work with some sample medical imaging data in the DICOM format and we'll learn how to visualise it and how to do some basic medical image analysis tasks, such as smoothing and segmentation.

Important Note: The code for this task is quite complex. You are not expected to fully understand the code below, but just to try your hand at it, so that you get an idea of what it is like to handle medical imaging data. You are welcome to simply run the code without spending time figuring out what it does. Explanations are provided for the curious ones!

1.1 Part 1: Importing packages needed

Pydicom is a Python package for working with DICOM files. It allows us to read and easily manipulate DICOM files with the use of Python.

ITK (Insight Segmentation and Registration Toolkit) is a library that enables image processing, and it is widely used for the development of image segmentation and image registration programs. Here, we'll use *SimpleITK*, as it is a simplified interface to ITK.

The *OS* module in Python provides a way of using operating system dependent functionality, for instance for working with files and folders.

As already discussed in Week 1, *NumPy* is a widely used Python package for working with large, multi-dimensional data (note that here we'll be using it to manipulate pixel data), while *Matplotlib* is a Python library for plotting data.

Now, lets import the packages.

```
In [1]: import pydicom
        import SimpleITK
        import os
        import numpy
        import matplotlib.pyplot as plt
        %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

## Part 2: Loading the data

In this programming task we're going to be working with a dataset from an MR examination of a patient's head. The images are captured as DICOM files and they are stored in a folder called 'MyHead'.

Run the code in the following cell to create a list called 'lstFilesDCM', which contains all DICOM files within 'MyHead'.

Remember that you are not expected to understand all the code in this programming task. (If, however, you are really curious, this is what this code does: It first creates a list called 'lstFilesDCM' that is initially empty. It then uses the *os.walk()* function from the OS module to traverse the MyHead folder. For each file found in this folder, we check whether it is a DICOM file, and if it is, we add its extended name (i.e. the folder and file name) to the lstFilesDCM list.)

```
In [2]: PathDicom = "./readonly/MyHead/" # File path
        lstFilesDCM = []  # create an empty list
        for dirName, subdirList, fileList in os.walk(PathDicom):
            for filename in fileList:
                if ".dcm" in filename.lower():  # check whether the file is DICOM
                    lstFilesDCM.append(os.path.join(dirName,filename))
```

Run the code in the following cell to get the first five elements of the 'lstFilesDCM' list, i.e. the extended names of the first five DICOM files.

```
In [3]: lstFilesDCM[:5] # first five DICOM files
```

```
Out[3]: ['./readonly/MyHead/MR000096.dcm',
         './readonly/MyHead/MR000078.dcm',
         './readonly/MyHead/MR000159.dcm',
         './readonly/MyHead/MR000069.dcm',
         './readonly/MyHead/MR000087.dcm']
```

1.1.1 Reading a DICOM file

Run the code in the following cell to use Pydicom's *read_file()* function to read the first DICOM file within lstFilesDCM (identified as 'lstFilesDCM[0]'). The resulting object is called 'HeadDs'. Note that we could have called it anything we wanted.

```
In [6]: HeadDs = pydicom.read_file(lstFilesDCM[0]) # Pydicom's read_file() function to read fi
```

1.1.2 Getting metadata

We can now get metadata of interest for this image with the use of appropriate Pydicom functions (e.g. *PatientPosition* or *Modality*).

Run the code in the following cell to get the position of the patient relative to the imaging equipment space.

```
In [7]: HeadDs.PatientPosition # Use Pydicom functions to obtain metadata (PatientPosition/Mod
```

```
Out[7]: 'HFS'
```

HFS stands for Head First-Supine. This means that the patient's head was positioned toward the front of the imaging equipment and it was in an upward direction.

Run the code in the following cell to get the date the study started. Note that the date format is YYYYMMDD.

```
In [6]: HeadDs.StudyDate # Date the study started
```

```
Out[6]: '20111110'
```

Run the code in the following cell to get the image modality. Note that 'MR' stands for Magnetic Resonance, 'CT' stands for Computed Tomography and 'PT' stands for Positron Emission Tomography (PET).

```
In [8]: HeadDs.Modality # Image Modelaity = MR (Magnetic Resonance)
```

```
Out[8]: 'MR'
```

## Part 3: Visualisation

Now that we've got an idea of the imaging data that we're going to be working with, we'll visualise the data by creating a two-dimensional plot. But first we'll need to extract some additional information needed for plotting.

Important note: Remember that you don't need to fully understand the code in this programming task. In fact, you could simply reuse the code in Part 3 for a new collection of MR images, given that you've called the list of your DICOM files 'lstFilesDCM' and the first image 'HeadDs'.

1.1.3 Preparing for visualisation

In order to plot the data with Matplotlib, we first need to i) combine the pixel data from all DICOM files (i.e. from all slices) into a 3D dataset, and ii) specify appropriate coordinate axes.

We'll start by calculating the total dimensions of the combined 3D dataset. These should be: (Number of pixel rows in a slice) x (Number of pixel columns in a slice) x (Number of slices) along the x, y, and z cartesian axes. We'll call the calculated dimensions 'CalcPixelDims'.

```
In [10]: # 1) Combine pixel data from all DICOM files into 3D Dataset
         # 2) Specify appropriate coordinate axes.
         CalcPixelDims = (int(HeadDs.Rows), int(HeadDs.Columns), len(lstFilesDCM))
```

Run the code in the following cell to get CalcPixelDims.

```
In [11]: CalcPixelDims
```

```
Out[11]: (256, 256, 176)
```

Now that we've calculated the dimensions, we can create a 3D NumPy array to address the first point discussed above. Note that a NumPy array is understood as a grid of values, all of the same type.

The following code creates a NumPy array called 'HeadImgArray' that has the same size as CalcPixelDims, and which contains the pixel data from all DICOM files in lstFilesDCM.

```
In [18]: # Create 3D NumPy array with same size as CalcPixelDims

         HeadImgArray = numpy.zeros(CalcPixelDims, dtype=HeadDs.pixel_array.dtype)

         for filenameDCM in lstFilesDCM:
             ds = pydicom.read_file(filenameDCM)
             HeadImgArray[:, :, lstFilesDCM.index(filenameDCM)] = ds.pixel_array
```

Now that we've addressed the first point, we move on to specify appropriate coordinate axes.

We use Pydicom's *PixelSpacing* and *SliceThickness* functions to calculate the spacing between pixels in the three axes. We call the resulting object 'CalcPixelSpacing'.

```
In [19]: # Use PixelSpacing & SliceThickness to calculate spacing b/w pixels in the three axes

         CalcPixelSpacing = (float(HeadDs.PixelSpacing[0]), float(HeadDs.PixelSpacing[1]), flo
```

Then, we use CalcPixelDims and CalcPixelSpacing to calculate coordinate axes. You don't need to understand what this code does. Just run it to compute x, y and z.

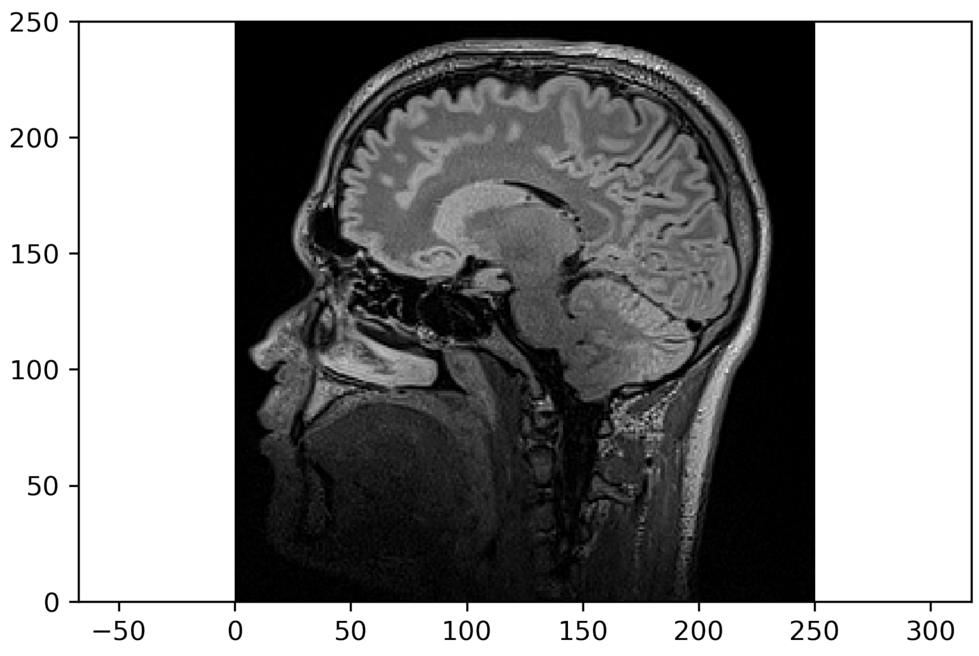
```
In [20]: # Use CalcPixelDims and CalcPixelSpacing to calculate coordinate axes:

         x = numpy.arange(0.0, (CalcPixelDims[0]+1)*CalcPixelSpacing[0], CalcPixelSpacing[0])
         y = numpy.arange(0.0, (CalcPixelDims[1]+1)*CalcPixelSpacing[1], CalcPixelSpacing[1])
         z = numpy.arange(0.0, (CalcPixelDims[2]+1)*CalcPixelSpacing[2], CalcPixelSpacing[2])
```

1.1.4 Visualising

We can now plot the data with the use of the pyplot module in matplotlib.

Again, you don't need to understand in detail what this code does. You can simply run it to get a plot of the head MR image. Note that you can modify the number in the last line of the code to get a different slice of the head.

```
In [21]: # Plot data using pyplot in matplotlib.
         # Will obtain a plot of the head MR image.

         plt.figure(dpi=300)
         plt.axes().set_aspect('equal', 'datalim')
         plt.set_cmap(plt.gray())
         plt.pcolormesh(x, y, numpy.flipud(HeadImgArray[:, :, 125]))

Out[21]: <matplotlib.collections.QuadMesh at 0x7fb3f883fcf8>
```



## Part 4: Segmentation

In this part of the notebook we'll use SimpleITK to segment the white and grey matter in the MR images.

1.1.5 Specifying a helper function

We'll be visualising quite a few 2D SimpleITK images in this part, so here we'll define a function that quickly plots a 2D SimpleITK image with a greyscale colourmap and accompanying axes.

There's no need to understand what this code does - you can simply run it so that you can use the *sitk_show* function later on.

```
In [22]: # Will be visualizing a few 2D SimpleITK images.
         # First, let's define a function to quickly plot a 2D SimpleITK image w/ grayscale co

         def sitk_show(img, title=None, margin=0.05, dpi=40 ):
             nda = SimpleITK.GetArrayFromImage(img)
             spacing = img.GetSpacing()
             figsize = (1 + margin) * nda.shape[0] / dpi, (1 + margin) * nda.shape[1] / dpi
             extent = (0, nda.shape[1]*spacing[1], nda.shape[0]*spacing[0], 0)
             fig = plt.figure(figsize=figsize, dpi=dpi)
             ax = fig.add_axes([margin, margin, 1 - 2*margin, 1 - 2*margin])

             plt.set_cmap("gray")
             ax.imshow(nda,extent=extent,interpolation=None)

             if title:
                 plt.title(title)
plt.show()
```

1.1.6 Loading the data in SimpleITK

We now need to tell SimpleITK to read the DICOM files within the 'MyHead' folder. Run the code in the following cell to extract the contents of the MRI dataset and create the corresponding 3D image called 'img3DOriginal'.

Note that you don't need to understand what this code does. You can simply run it, so as to use img3DOriginal later on. You can also reuse this code for a new collection of MR images, given that you've called the location of your DICOM files 'PathDicom'.

```
In [24]: # Load data inside SimpleITK. Run this code in the cell to extract the contents of MR
         # and create the corresponding 3D image called 'img3DOriginal'.

         # NOTE: You do not need to understand what this code does.
         # Can simply run it, so as to use img3DOriginal later on.
         # Can reuse this code for a new collection of MR images, given the location of your D

         reader = SimpleITK.ImageSeriesReader()
         filenamesDICOM = reader.GetGDCMSeriesFileNames(PathDicom) # Location of DICOM files
         reader.SetFileNames(filenamesDICOM)
         img3DOriginal = reader.Execute() # 3D image
```

In order to keep things simple, we'll segment a 2D slice of the 3D image (rather than the entire 3D image). Run the code below to specify that we want Z-slice 50 of the 3D image. We'll call this slice 'imgOriginal'.

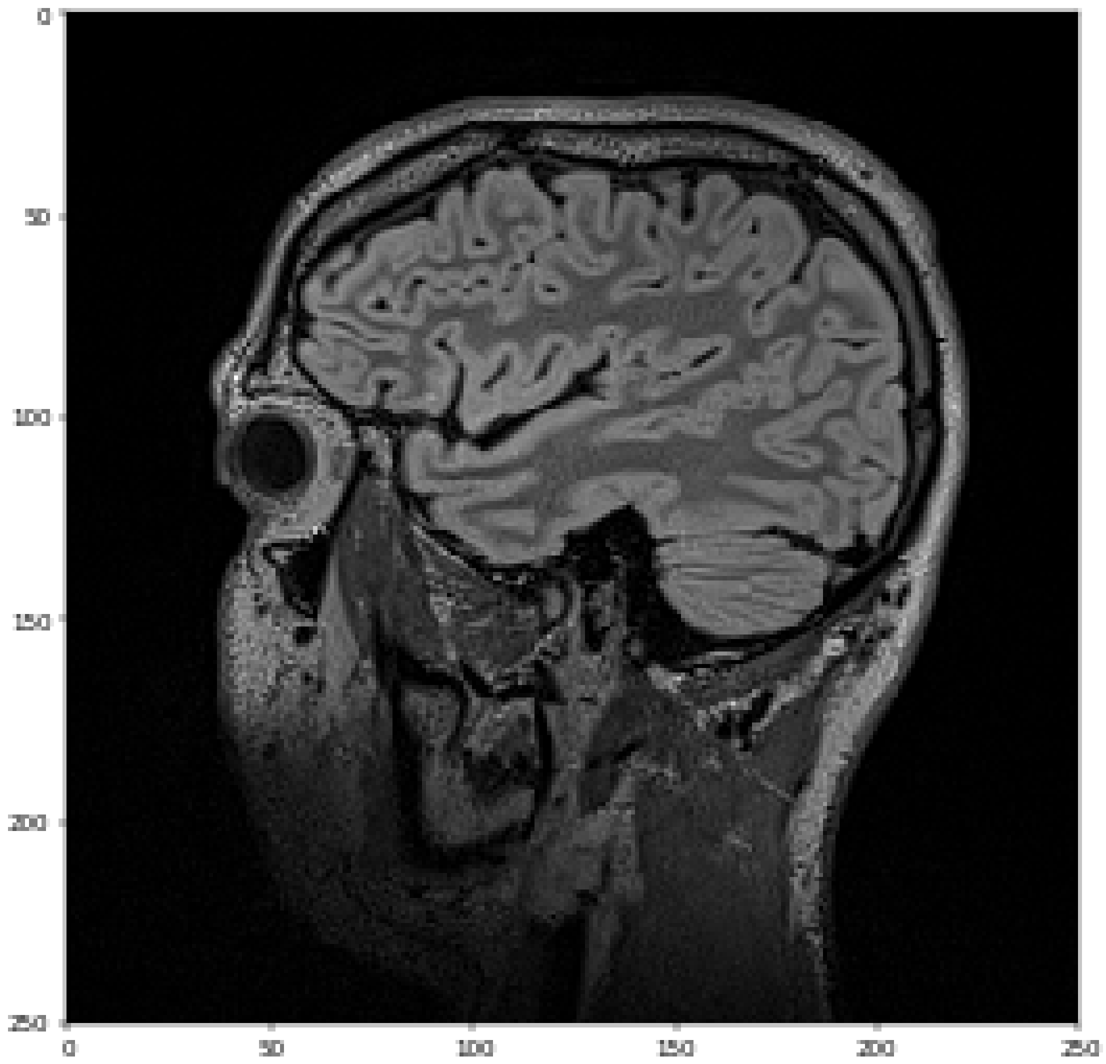
Note that you could ask for a different Z-slice, if you want.

```
In [26]: # Segment a 2D slice of the 3D image (vs. entire 3D image).
         # WANT: Z-slice 50 of the 3D image.
         imgOriginal = img3DOriginal[:,:,50]
```

1.1.7 Visualising the original data

We'll now call our *sitk_show* function to visualise the original data.

```
In [27]: # Visualize original data
         sitk_show(imgOriginal)
```



1.1.8 Smoothing

Smoothing is the process of reducing noise within an image or producing a less pixelated image. The result is an image with sharp edges or boundaries preserved and smoothing occurring only within a region.

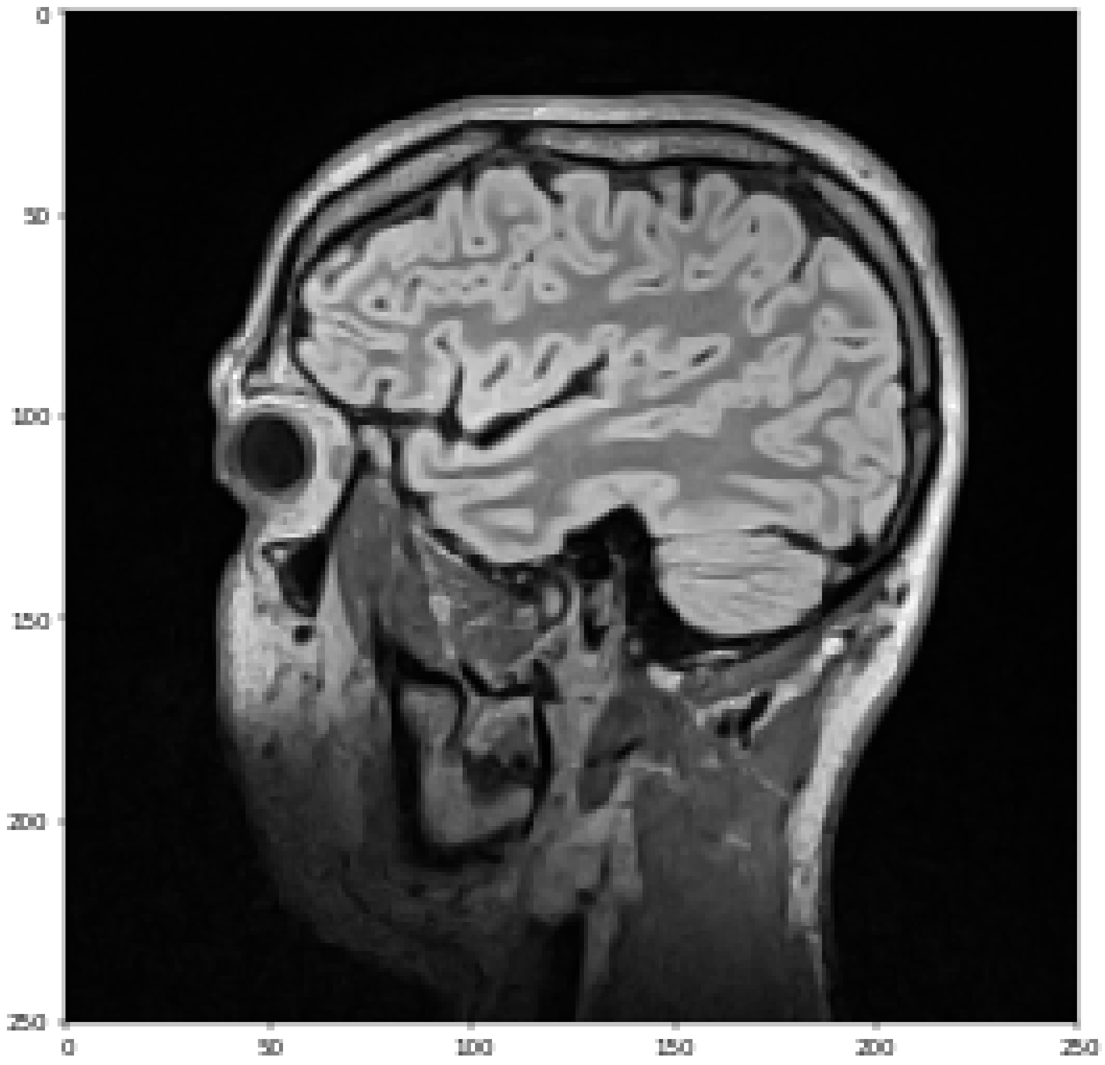
In our case, we can see that the original image data exhibits quite a bit of noise, which is rather typical of MRI datasets. We will reduce the noise, so as to ease the process of segmentation later on.

Run the code in the following cell to apply a Curvature Flow Image Filter to smoothen *imgOriginal*. We'll call the resulting image 'imgSmooth'.

```
In [28]: # Smoothing:
         # - The process of reducing noise within an image or producing a less pixelated image
         # - The result is an image w/ sharp edges or boundaries preserved and smoothing occur
         # Original image data - results a bit of noise (typical of MRI datasets).
         # We will reduce noise, to ease process of segmentation.

         # CODE: Apply a Curvature Flow Image Filter to smoothen imgOriginal.
         imgSmooth = SimpleITK.CurvatureFlow(image1=imgOriginal,
                                             timeStep=0.125,
                                             numberOfIterations=5)
```

Now, lets see the results of the smoothened image.

```
In [29]: # Results of smoothened image:
         sitk_show(imgSmooth)
```



1.1.9 Segmentation with the ConnectedThreshold filter

We will now apply region growing techniques to segment the white and grey matter of the smoothened image.

Initial segmentation of the white matter Run the code in the following cell to apply SimpleITK's *ConnectedThreshold* filter function to imgSmooth. The image of the resulting segmented white matter is called 'imgWhiteMatter'.

As with other parts of this programming task, you don't need to fully understand what this code does. You can simply run it to get the initial segmentation of the white matter.

If, however, you're really curious, here's a brief explanation: This filter starts from a seed point in the image that we know is white matter, and looks at all connected pixels. If their values lie within a range of interest, then they are labelled as white matter. In our case, we've set the seed to be point (150, 75), as by inspecting the previous image, it looked like that point was indeed white matter. We've also set the range of values to be between 130 and 190, as these were roughly the values that the white matter pixels exhibited (this was inspected with the use of a separate DICOM viewing software).

```
In [30]: # Apply region growing techniques to segment white and grey matter of smoothened imag

         # INITIAL SEGMENTATION OF WHITE MATTER:
         # CODE:
         # - Will apply SimpleITK's ConnectedThreshold filter functoin to imgSmooth.

         # EXPLANATION:
         # - The filter starts from a seed point in the image (known as white matter), and loo
         # - If their values lie within a range of interest, then they are labelled white matt


         lstSeeds = [(150,75)] # Set Seed point (b/c looked like the point was white matter)

         # Set range of values to be b/w 130 and 190
         imgWhiteMatter = SimpleITK.ConnectedThreshold(image1=imgSmooth,
                                                       seedList=lstSeeds,
                                                       lower=130,
                                                       upper=190,
                                                       replaceValue=1)
```

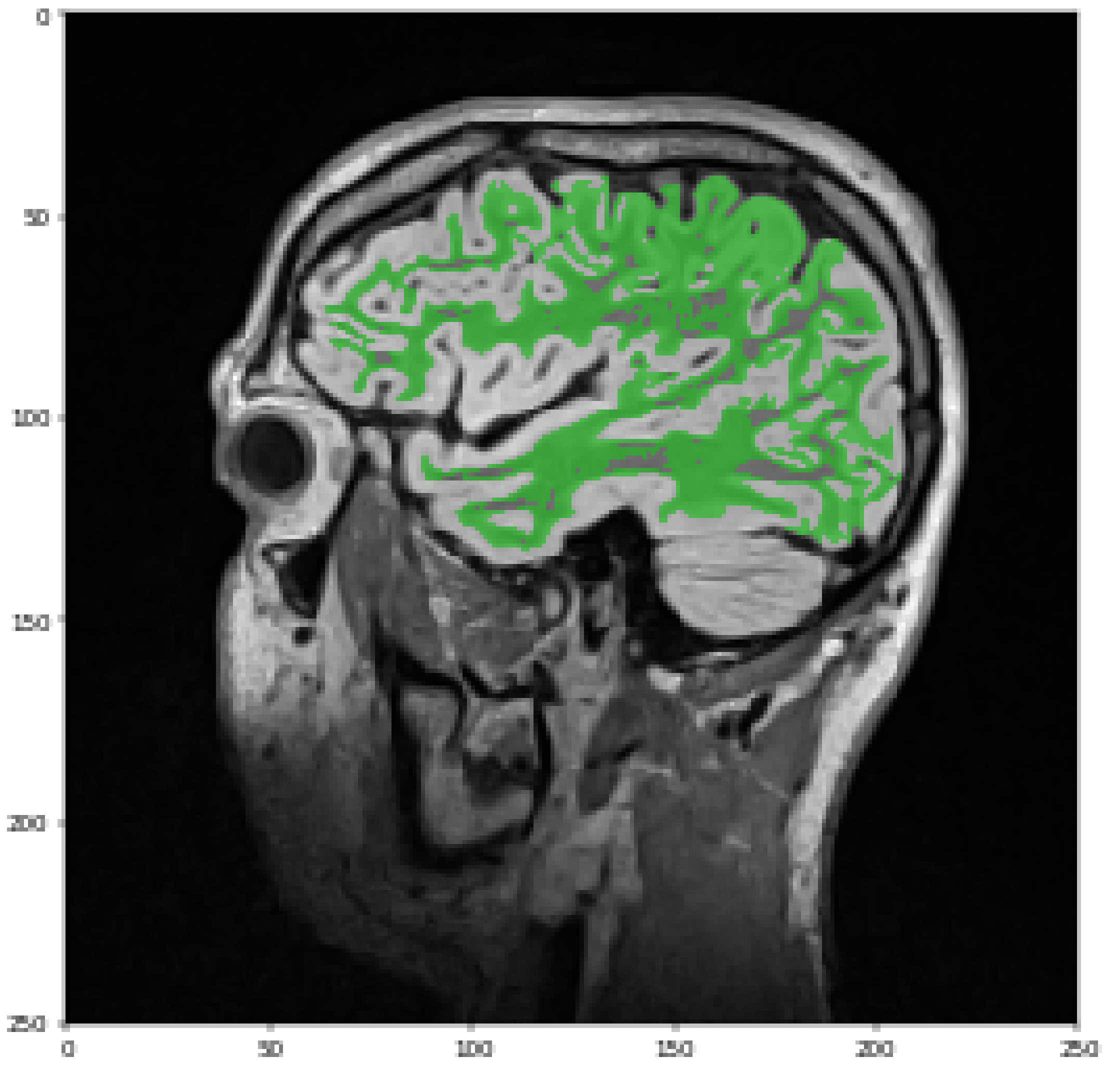
Next, we want to view the result of the segmentation. If we tried to visualise imgWhiteMatter, then all we would see would be a white-colour label in a black backdrop, which doesn't give us much insight. However, if we use a label overlay, then we can see the result of the segmentation of white matter in a basic RGB (red, green, blue) colour. In our case, it will show as green.

Note that before overlaying imgSmooth and imgWhiteMatter, we first need to manipulate imgSmooth so we can successfully mix the two images. This is what the following code does. The resulting image is called 'imgSmoothInt'.

```
In [31]: # View result of segmentation.
         # Use label overlay to see the result of the segmentation of white matter in a basic
         # IN OUR CASE, will show as GREEN!
         # NOTE:
         # - Before overlaying imgSmooth and imgWhiteMatter,
         # must first manipulate imgSmooth so we can successfully mix the two images.

         # Resulting image:
         imgSmoothInt = SimpleITK.Cast(SimpleITK.RescaleIntensity(imgSmooth), imgWhiteMatter.Ge
```

Now let's use SimpleITK's *LabelOverlay* function to overlay 'imgSmoothInt' and 'imgWhiteMatter', and let's visualise the result.

```
In [32]: # Use SimpleITK's LabelOverlay function to overlay imgSmoothInt and imgWhiteMatter an

         sitk_show(SimpleITK.LabelOverlay(imgSmoothInt, imgWhiteMatter))
```



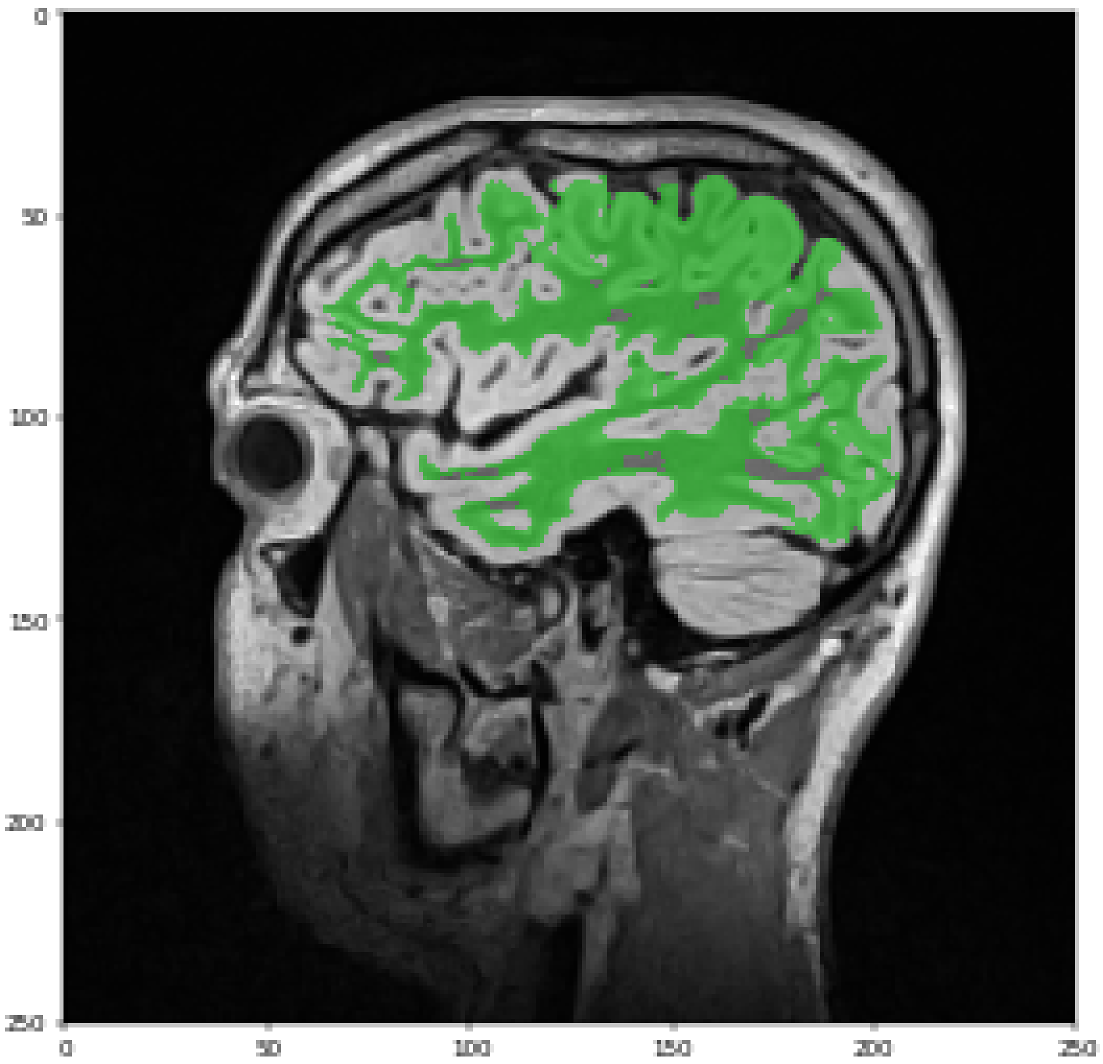
Hole-filling of the segmented white matter As you can see in the figure above, there are several holes in the segmented white matter. We'll rectify this by applying hole-filling.

Run the code in the following cell to use SimpleITK's *VotingBinaryHoleFilling* filter to fill in holes in *imgWhiteMatter*. The resulting image is called 'imgWhiteMatterNoHoles'.

```
In [33]: # NOTE:
         # - There are several holes in the segmented white matter.
         # - We will rectify this by applying hole-filling.

         # Run the code in the following cell to use SimpleITK's VotingBinaryHoleFilling later
         # - To fill in holes in imgWhiteMatter

         imgWhiteMatterNoHoles = SimpleITK.VotingBinaryHoleFilling(image1=imgWhiteMatter,
                                                                   radius=[2]*3,
                                                                   majorityThreshold=1,
                                                                   backgroundValue=0,
                                                                   foregroundValue=1)
```

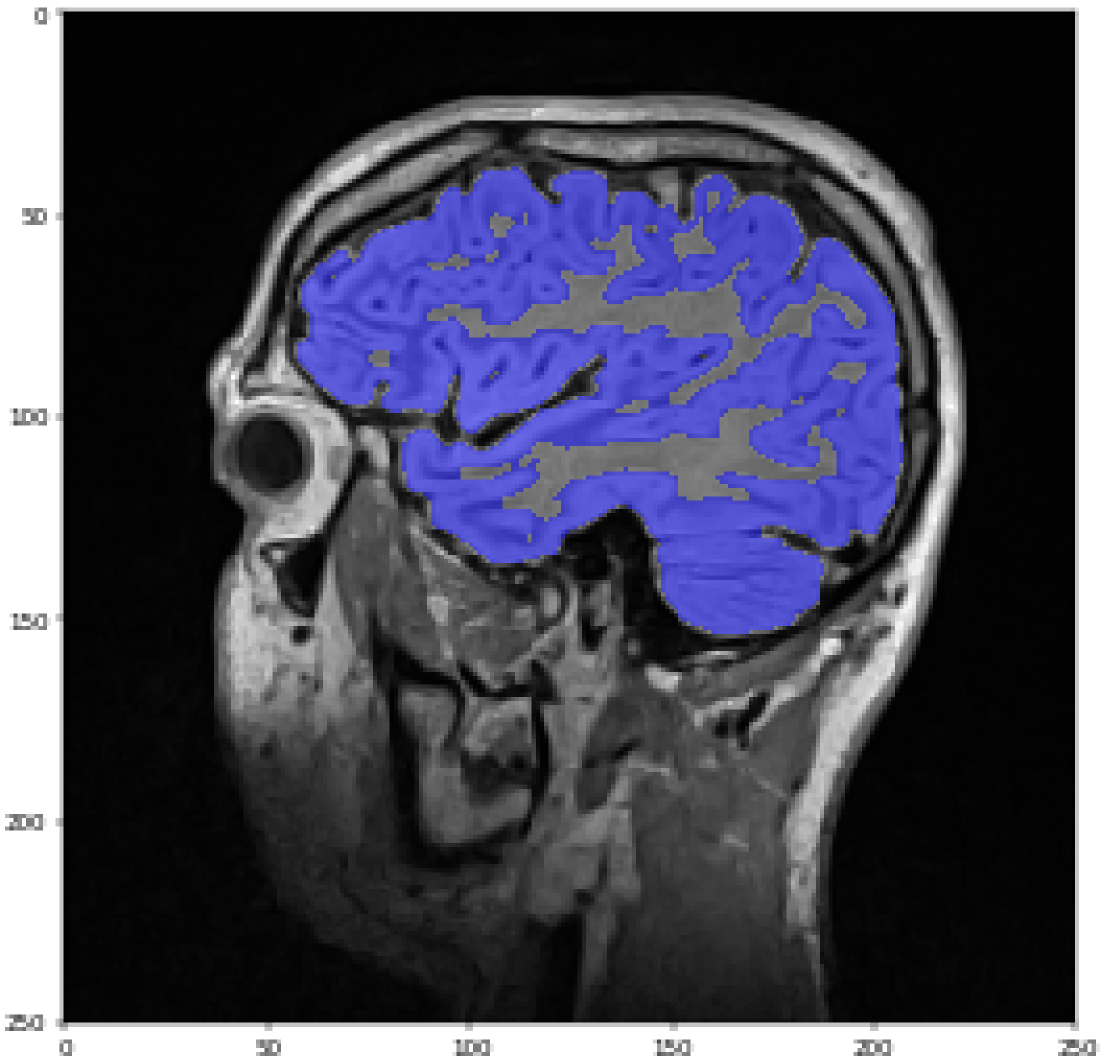
Now let's use SimpleITK's *LabelOverlay* function to overlay 'imgSmoothInt' and the newly obtained 'imgWhiteMatterNoHoles', and let's visualise the result.

```
In [34]: # LabelOverlay to overlay imgSmoothInt and imgWhiteMatterNoHoles and create visual.

         sitk_show(SimpleITK.LabelOverlay(imgSmoothInt, imgWhiteMatterNoHoles))
```



Segmentation and hole-filling of grey matter We'll now repeat the process for grey matter. In other words, we'll do some preliminary segmentation and then we'll perform hole-filling of the segmented grey matter. We'll then overlay the 'imgSmoothInt' and the 'imgGreyMatterNoHoles' images and we'll visualise the result.

```
In [35]: # Repeat process for grey matter.
         # - Will do some preliminary segmentation and then perform hole-filling of segmented

         lstSeeds = [(119, 83), (198, 80), (185, 102), (164, 43)]

         imgGreyMatter = SimpleITK.ConnectedThreshold(image1=imgSmooth,
                                                       seedList=lstSeeds,
                                                       lower=150,
                                                       upper=270,
                                                   replaceValue=2)

imgGreyMatterNoHoles = SimpleITK.VotingBinaryHoleFilling(image1=imgGreyMatter,
                                                          radius=[2]*3,
                                                          majorityThreshold=1,
                                                          backgroundValue=0,
                                                          foregroundValue=2) # labelGr

sitk_show(SimpleITK.LabelOverlay(imgSmoothInt, imgGreyMatterNoHoles))
```

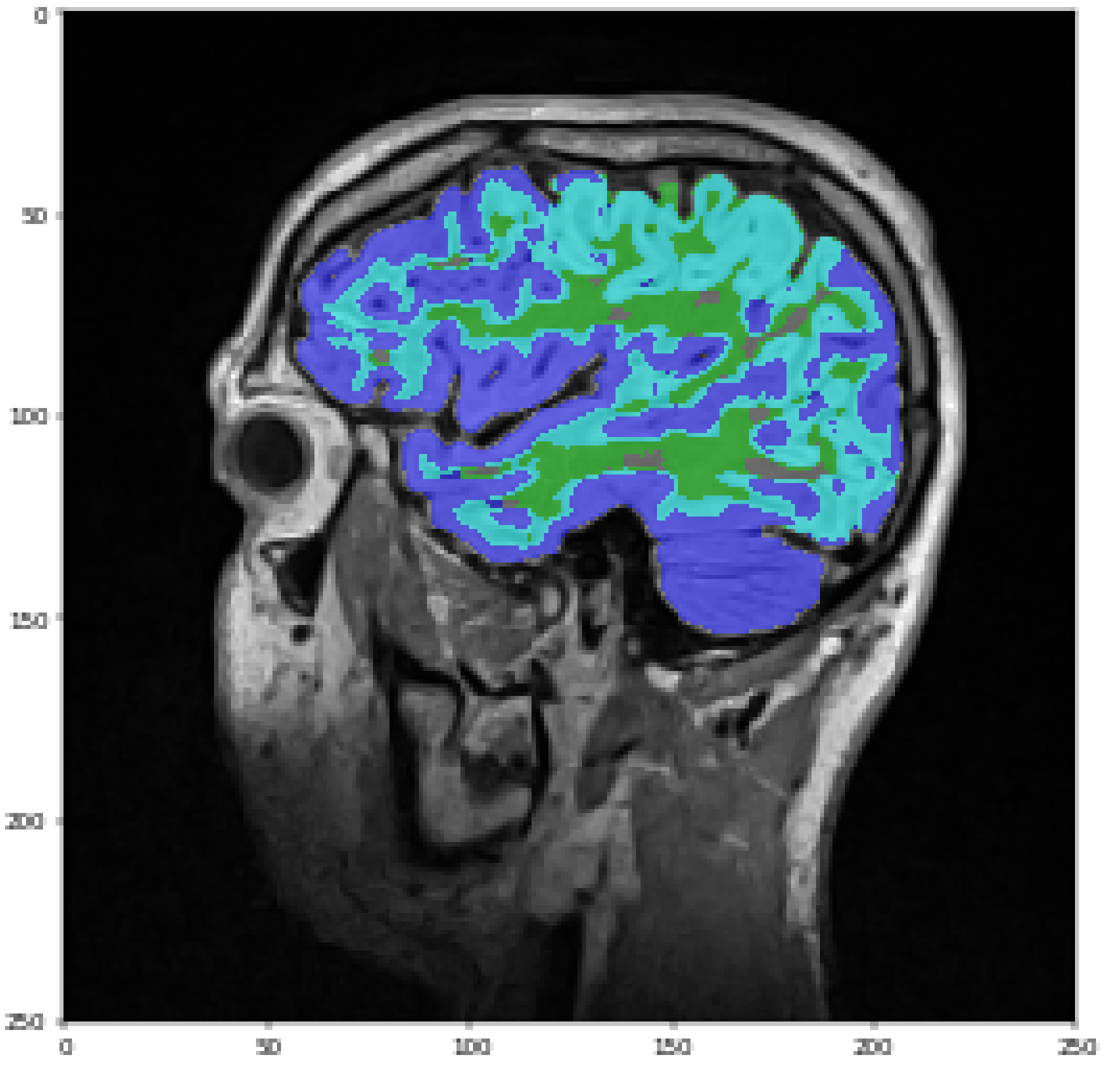


Combining the white and grey matter Lastly, we want to combine the two label-fields, in other words, the white and grey matter. We do this by running the code below, with the results being stored under 'imgLabels'.

```
In [36]: # Combine two label-fields: white & grey matter

         imgLabels = imgWhiteMatterNoHoles | imgGreyMatterNoHoles
```

Now let's use SimpleITK's *LabelOverlay* function to overlay 'imgSmoothInt' and the newly obtained 'imgLabels', and let's visualise the result.

```
In [37]: # LabelOverlay
         sitk_show(SimpleITK.LabelOverlay(imgSmoothInt, imgLabels))
```



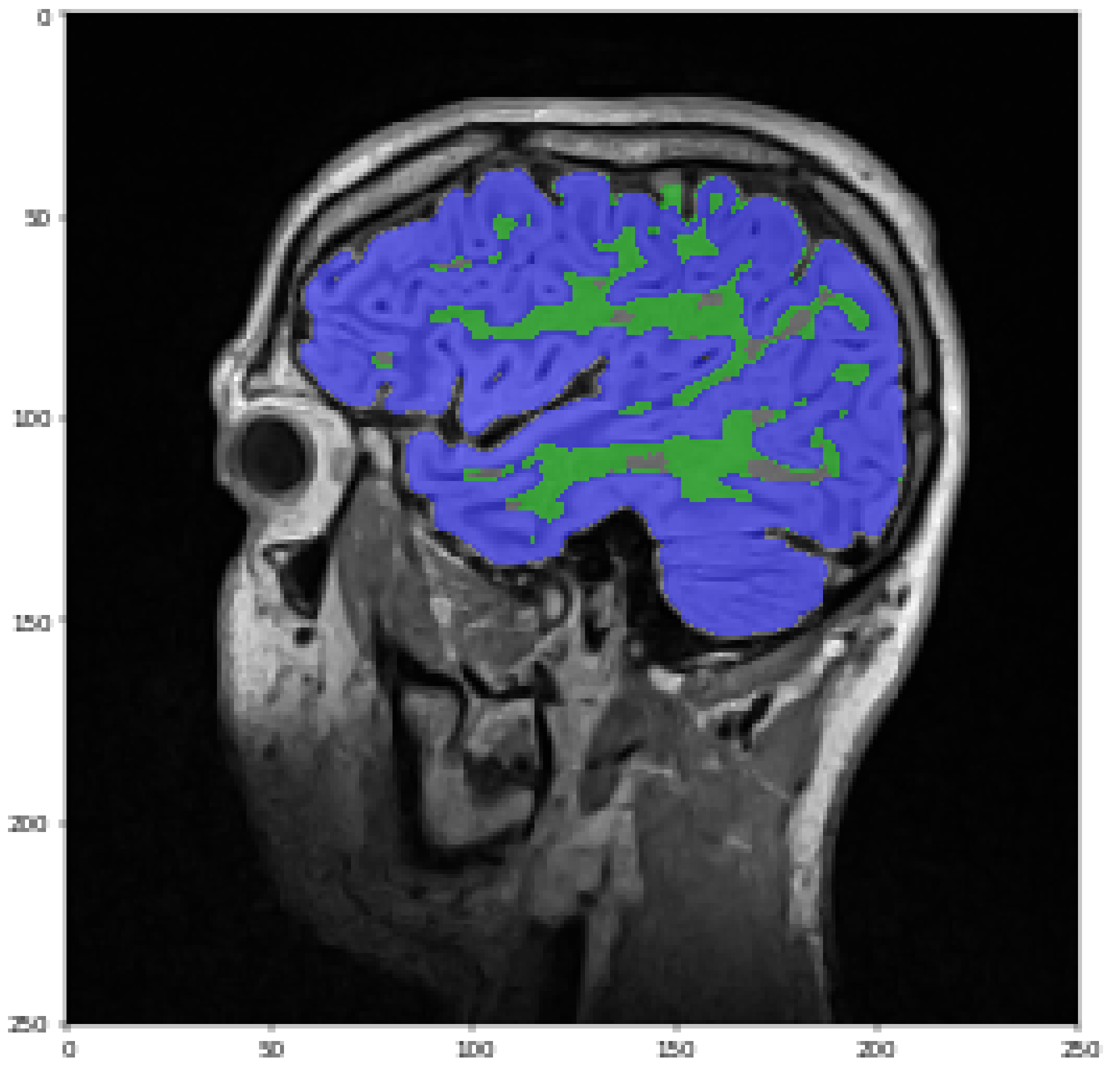
Note that the cyan-coloured label are regions where both the white matter and grey matter overlap from the initial segmentation process above.

The majority of those regions should actually be part of the grey matter. Run the code in the following cell to fix this, with the results being stored under 'imgLabels2'. Note that you don't need to understand in detail what the code does. All you need to know is that it assigns any overlapping regions to the grey matter.

```
In [39]: # We have cyan-coloured label where white/grey matter overlap, but we should have mos

         # CODE:
         # - Should assign any overlapping regions to the grey matter.
         imgMask = (imgWhiteMatterNoHoles/1) * (imgGreyMatterNoHoles/2)
         imgMask2 = SimpleITK.Cast(imgMask, imgWhiteMatterNoHoles.GetPixelIDValue())
         imgWhiteMatterNoHoles = imgWhiteMatterNoHoles - (imgMask2*1)
         imgLabels2 = imgWhiteMatterNoHoles + imgGreyMatterNoHoles
```

Now let's use SimpleITK's *LabelOverlay* function to overlay 'imgSmoothInt' and the newly obtained 'imgLabels2', and let's visualise the result.

```
In [40]: sitk_show(SimpleITK.LabelOverlay(imgSmoothInt, imgLabels2))
```



And that's it! You've now had a taste of medical image processing!

You're welcome to play around with this code. If you have any questions or comments, please post them in the forums.

```
In [41]: # SUMMARY:
         # - We had just done medical image processing, to segment grey matter from white matt
         # while working with a dataset from an MR examination of a patient's head.
```