In this post I will demonstrate SimpleITK, an abstraction layer over the ITK library, to segment/label the white and gray matter from an MRI dataset. I will start with an intro on what SimpleITK is, what it can do, and how to install it. The tutorial will include loading a DICOM file-series, image smoothing/denoising, region-growing image filters, binary hole filling, as well as visualization tricks.

Introduction

I think that by this point you may've had enough of VTK and its obscure, bordering on the occult, inner-workings. I admit that the topics I covered so far were not that much about visualization but mostly about 'secondary' functionality of VTK. However, information and examples on the former can be found with relative ease and I've posted a few such links in this early post about IPython & VTK.

Today I'll be branching off and talking about its cousin, the Insight Segmentation and Registration Toolkit (ITK), a library created by the same fine folk as VTK, i.e., Kitware, but which focuses on image processing.

Background

Insight Toolkit (ITK)

ITK was originally built to support the 'Visible Human' project, which we used in this past post about surface extraction. It includes a whole bunch of goodies including routines for the segmentation, registration, and interpolation of multi-dimensional image data.

Just like VTK, ITK exhibits the same mind-boggling design paradigms and near-

inexistent documentation (apart from this one book and some little tidbits like the Doxygen docs, a couple presentations, and webinars).

Nonetheless, just like VTK, ITK offers some amazing functionality one just can't overlook in good conscience. Its no coincidence ITK is being heavily employed in image processing software like ParaView, MeVisLab and 3DSlicer. It just works!

SimpleITK

Today, however, we won't be dealing with ITK, but SimpleITK instead! SimpleITK is, as the name implies, a simplified layer/wrapper build on top of ITK, exposing the vast majority of ITK functionality through bindings in a variety of languages, greatly simplifying its usage.

While the usage of ITK would require incessant usage of templates and result in code like this:

```
// Setup image types.
typedef float InputPixelType;
typedef float OutputPixelType;
typedef itk::Image<InputPixelType, 2> InputImageType;
typedef itk::Image<OutputPixelType, 2> OutputImageType;
// Filter type
typedef itk::DiscreteGaussianImageFilter<InputImageType, OutputImageType> FilterType;

// Create a filter
FilterType::Pointer filter = FilterType::New();

// Create the pipeline
filter->SetInput(reader->GetOutput());
filter->SetVariance(1.0);
filter->SetMaximumKernelWidth(5);
filter->Update();
OutputImageType::Pointer blurred = filter->GetOutput();
```

SimpleITK hides all that characteristically un-pythonic-code and yields something like

this:

```
import SimpleITK
imgInput = SimpleITK.ReadImage(filename)
imgOutput = SimpleITK.DiscreteGaussianFilter(imgInput, 1.0, 5)
```

However, the power of SimpleITK, doesn't end in allowing for succinct ITK calls. Some very notable features are:
- Super-simple IO of multi-dimensional image data supporting most image file formats (including DICOM through the infamous Grassroots DICOM library (GDCM), more on that later).
- A fantastic `SimpleITK.Image` class which handles all image data and which includes overloads for all basic arithmetic (`+` `-` `*` `/` `//` `**`) and binary (`&` `|` `^` `~`) operators. These operators just wrap the corresponding ITK filter and operate on a pixel-by-pixel basis thus allowing you to work directly on the image data without long function calls and filter configuration.
- Slicing capability akin to that seen in `numpy.ndarray` objects allowing you to extract parts of the image, crop it, tile it, flip it, etc etc. While slicing in SimpleITK is not as powerful as NumPy, its still pretty darn impressive (not to mention invaluable)!
- Built-in support for two-way conversion between `SimpleITK.Image` object and `numpy.ndarray` (there's a catch though, more on that later as well).
- No pipeline! Those who read my previous posts on VTK, will be aware of that pipeline monstrosity VTK is built upon. Well, SimpleITK does away with that and all operations are immediate (a much more pythonic paradigm).

You'll find a fair number of links to SimpleITK material at the end of this post.

Installation

For better or for worse, SimpleITK isn't written in pure Python but rather C++ which translates to you needing a compiled version of the library. Unfortunately, SimpleITK doesn't come pre-compiled with any of the major alternative Python distros I know so if

you're using Anaconda Python, Enthought Python, Enthought Canopy, or even the OSX MacPorts or Homebrew Python, installing it is a lil' hairier than normal.

Vanilla Python

If you're using a vanilla Python interpreter, i.e., a Python distro downloaded straight from python.org, or the 'system Python' that comes pre-installed with most Linux and OSX distros, then you're in luck! You can simply install the SimpleITK package hosted on PyPI through `pip` as such:

```
pip install SimpleITK
```

Alternatively, you can install it by using `easy_install` and one of the Python eggs (`.egg`), or `pip` and one of the Python wheels (`.whl`) provided at the SimpleITK download page as such:

```
easy_install <egg filename>
```

or

```
pip install --use-wheel <whl filename>
```

> In case you've missed the news, Python wheels (`.whl`) are meant to replace Python eggs (`.egg`) in the distribution of binary Python packages. Installing them is performed through `pip install --use-wheel <whl filename>` but you must have first installed the `wheel` package through `pip install wheel`.

Anaconda Python

Unfortunately, here's where the trouble starts. Alternative Python distros like the Anaconda Python, Enthought Python, Enthought Canopy, or even the OSX MacPorts and Homebrew Python, have their own version of the Python interpreters and dynamic libraries.

However, the aforementioned SimpleITK wheels and eggs are all compiled and linked against vanilla Python interpreters, and should you make the very common mistake of installing one of those in a non-vanilla environment, then upon importing that package you'll most likely get the following infamous error:

```
Fatal Python error: PyThreadState_Get: no current thread
```

This issue, however, isn't exclusive to SimpleITK. Any package containing Python-bound C/C++ code compiled against a vanilla Python will most likely result in an error if used in a different interpreter. Solution? To my knowledge you have one of three options:

1. Compile the code yourself against the Python interpreter and dynamic library you're using.

   That of course includes checking out the code, ensuring you already have (or even worse compile from source) whatever dependencies that package has, and build the whole thing yourself. If you're lucky, the source-code will come with CMake's Superbuild and the process will just take a couple hours of tinkering. If not, you can spend days trying to get the thing to compile without errors, repeating the same bloody procedure over and over and harassing people online for answers.

   In the case of SimpleITK there are instructions on how to do so under the 'Building using SuperBuild' on their Getting Started page. However, a process like this typically assumes a working knowledge of Git, CMake, and the structure of your non-vanilla Python distribution.

2. Wait patiently till the devs of your non-vanilla Python distro build the package for you and give you a convenient way of installing it.

Companies like Continuum Analytics (creators of Anaconda Python) and Enthought often do the work for you. That's exactly why you have packages like VTK all built and ready with those distros. However, in the case of less-popular packages, SimpleITK being one, you might be waiting for some time till enough people request it and the devs take time to do so.

3. Depend on the kindness of strangers. Often enough, other users of non-vanilla Python distros will do the work described in (1) and, should they feel like it, distribute the built package for other users of the same distro.

This last one is exactly the case today. I compiled the latest SimpleITK release (v0.8.0) against an x64 Anaconda Python with Python 2.7 under Mac OSX 10.9.5, Windows 8.1, and Linux Mint 17 and I'm gonna give you the .egg files you need to install the package. You can get the `.egg` files here (hosted on the blog's BitBucket repo):

- SimpleITK .egg for Anaconda Python 2.7 on Windows x64
- SimpleITK .egg for Anaconda Python 2.7 on Linux x64
- SimpleITK .egg for Anaconda Python 2.7 on Mac OSX x64

> Here I should note that the above, i.e., people distributing their own builds to save other people the trouble of doing so themselves, is not that uncommon. Its actually the primary reason behind the creation of Binstar, a package distribution system by the creators of Anaconda Python, principally targeting that distro, meant to allow users to redistribute binary builds of packages and permitting them to be installed through `conda`. There you will find many custom-built packages such as OpenCV, PETSc, etc but I'll get back to Binstar at a later post. However, I didn't find the time to package SimpleITK on Binstar hence .egg files it is for now :).

If you're using Anaconda Python the you simply need to download the appropriate `.egg` and install it through `easy_install <egg filename>`. Uninstalling it is as easy as `pip uninstall simpleitk`.

> If you're using an Anaconda environment, e.g. named `py27` environment as instructed in this past post, then you need to activate that environment prior to installing the package through `activate py27` (Windows) or `source activate py27`

> (Linux/OSX). Also, make sure that environment contains `pip` and `setuptools` before installing the `.egg`. Otherwise, `pip` will be called through the `root` Anaconda environment and be installed in that environment instead.

Summary

The purpose of today's post was to introduce you to SimpleITK, show you how to install it, and give you a taste of its image-processing prowess.

Personally, when I started this blog I intended to only address challenging yet IMHO interesting topics that I once found hard to tackle, and powerful tools that either suffered from poor/insufficient documentation or were not as prominent in the community as they deserve to be.

SimpleITK falls under the latter category. I find the package to be as easy and Pythonic as a Python-bound C++ package can be. In addition, there's a whole treasure-trove of functionality one just can't find elsewhere. However, I find the existing documentation lacking and perhaps its due to the fact that people don't know of its existence.

Hence, today I'll do a little demonstration of some of SimpleITK's functionality, and use it to semi-automatically segment the brain-matter (white and gray) off an MRI dataset of my own head (of which I first spoke in this past post about DICOM in Python). You can download that dataset here and you should extract its contents alongside today's notebook.

The process will include loading the series of DICOM files into a single `SimpleITK.Image` object, smoothing that image to reduce noise, segmenting the tissues using region-growing techniques, filling holes in the resulting tissue-labels, while I'll also show you a few visualization tricks that come with SimpleITK.

However, keep in mind that the presented functionality is the mere tip of a massive iceberg and that SimpleITK offers a lot more. To prove that point, I repeated the process in today's notebook in an 'alternative notebook' where I used different techniques to achieve similar results (feel free to take a look cause I won't be going over this

'alternative notebook' today). In addition, and as I mentioned in the intro, SimpleITK comes with a lot of classes tailored to image registration, interpolation, etc etc. I may demonstrate things like that in later posts.

Image Segmentation

Imports

As always we'll start with a few imports. Nothing special but if this goes through that means that you're installation of SimpleITK probably worked :).

```
import os
import numpy
import SimpleITK
import matplotlib.pyplot as plt
%pylab inline
```

Helper-Functions

There's only going to be a single 'helper-function' today:

- `sitk_show(img, title=None, margin=0.05, dpi=40)` : This function uses `matplotlib.pyplot` to quickly visualize a 2D `SimpleITK.Image` object under the `img` parameter. The code is pretty much entirely `matplotlib` but there's one point you should pay attention to so here's the code:

```
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()
```

As you can see in the first line of the function we convert the `SimpleITK.Image` object to a `numpy.ndarray` through the `GetArrayFromImage` function which resides directly under the `SimpleITK` module. The opposite can be done through the `GetImageFromArray` function which just takes a `numpy.ndarray` and returns a `SimpleITK.Image` object.

However!, there's a serious catch to this conversion! Be careful! If you were to have a `SimpleITK.Image` object with a size/shape of `200x100x50`, then upon conversion to a `numpy.ndarray` that object would exhibit a `shape` of `50x100x200`, i.e., the axes would be backwards.

The reason behind this is briefly outlined in this SimpleITK notebook by the SimpleITK author. It turns out that the `SimpleITK.Image` class doesn't exactly have a bracket (`[` and `]`) operator but instead uses the `GetPixel` method which takes in a pixel index in a `(x, y, z)` order, i.e., the internal array is stored in an 'x-fastest' fashion. However, `numpy` internally stores its arrays in a 'z-fastest' fashion and takes an index in a `(z, y, x)` order. As a result you get the axes backwards!

Now, what does that mean for you? Well for one it means you need to be careful with your indices depending on whether you're addressing the `SimpleITK.Image` or a `numpy.ndarray` derivative. It also means that the result of the `sitk_show` helper-

function, which uses the `GetArrayFromImage` method we're discussing, shows you the `numpy` view of the array and needs to be taken with a grain of salt. Lastly, when you feed point indices to SimpleITK for different algorithms, as we'll see later, these indices need to be in the SimpleITK order and not the NumPy order.

> I should state here that one can easily use `numpy.transpose` to bring the axes in the derivative `numpy.ndarray` in their 'proper' order. However, the array within SimpleITK is what it is and unless you just want to use SimpleITK to load data and then play around with NumPy there's little point to it.

Options

Once again, we'll define a few options to keep the rest of the notebook 'clean' and allow you to make direct changes without perusing/amending the entire notebook.

```
# Directory where the DICOM files are being stored (in this
# case the 'MyHead' folder).
pathDicom = "./MyHead/"

# Z slice of the DICOM files to process. In the interest of
# simplicity, segmentation will be limited to a single 2D
# image but all processes are entirely applicable to the 3D image
idxSlice = 50

# int labels to assign to the segmented white and gray matter.
# These need to be different integers but their values themselves
# don't matter
labelWhiteMatter = 1
labelGrayMatter = 2
```

First of all we define the path where the `.dcm` files are under, i.e., `pathDicom`. Once more, you can get the MRI dataset of my head under here whose contents you should extract next to today's notebook.

I want to emphasize the 2nd option. As I explain in the comments, I'm going to be limiting the segmentation to a single 2D slice of the DICOM dataset. The reason for this is not because the process would become too computationally expensive (clearly slower but not by much) but rather for the sake of clarity.

When it comes to image segmentation, and especially when using algorithms based on region-growing and pixel-connectivity, application to the full 3D image might yield non-intuitive results. For examples, regions might seem entirely disconnected when viewed on one cross-section but end up being connected further down the slices through some small structure. Therefore, and since I'll only be using 2D visualization, I wanted to keep things clear.

I should stress that the code I'll be presenting is entirely applicable to the full 3D image but the parameters used in the function calls, e.g. seed-point indices, won't be. Image segmentation is mostly about trial-n-error so try away.

Lastly, note `labelWhiteMatter` and `labelGrayMatter`. These are simply two integer values, which will act as label indices in the segmentation as we want the different tissues to be characterized by a different index. Its as simple as that.

Loading the DICOM files

Now let's move onto reading the DICOM files. Remember you have to extract the contents of the MRI dataset, i.e., my head, alongside today's notebook.

```
reader = SimpleITK.ImageSeriesReader()
filenamesDICOM = reader.GetGDCMSeriesFileNames(pathDicom)
reader.SetFileNames(filenamesDICOM)
imgOriginal = reader.Execute()
```

Reading the entirety of the DICOM file series goes as follows: We start by creating a new `ImageSeriesReader` object under the name `reader`. We then use the `GetGDCMSeriesFileNames` static method of the `ImageSeriesReader` to retrieve a list of all

`.dcm` filenames which we store under `filenamesDICOM` and which we then pass back to the `reader` through the `SetFileNames` method. Finally, by calling `Execute` we retrieve the entire 3D image under `imgOriginal`.

> If you get a dead kernel here then please do check the console output. It might well be that you forgot to extract the DICOM files out of their .zip or that you've placed them under a different directory to the one under `pathDicom`.

A **very** important point I want to stress here is that SimpleITK uses the Grassroots DICOM library (GDCM) to load DICOM files. As a result, those pesky JPEG compressed DICOM files, e.g., the ones found in the Osirix Datasets page, which we couldn't load with either PyDICOM or VTK in the past post about Python and DICOM, are no longer a deterrence. GDCM is actually the most comprehensive DICOM library I know of so you should be able to handle pretty much any DICOM file :).

Then, as I mentioned in the *Options* section, we limit ourselves to a single 2D slice of the 3D volume. We do so by using the basic slicing offered by the `SimpleITK.Image` class:

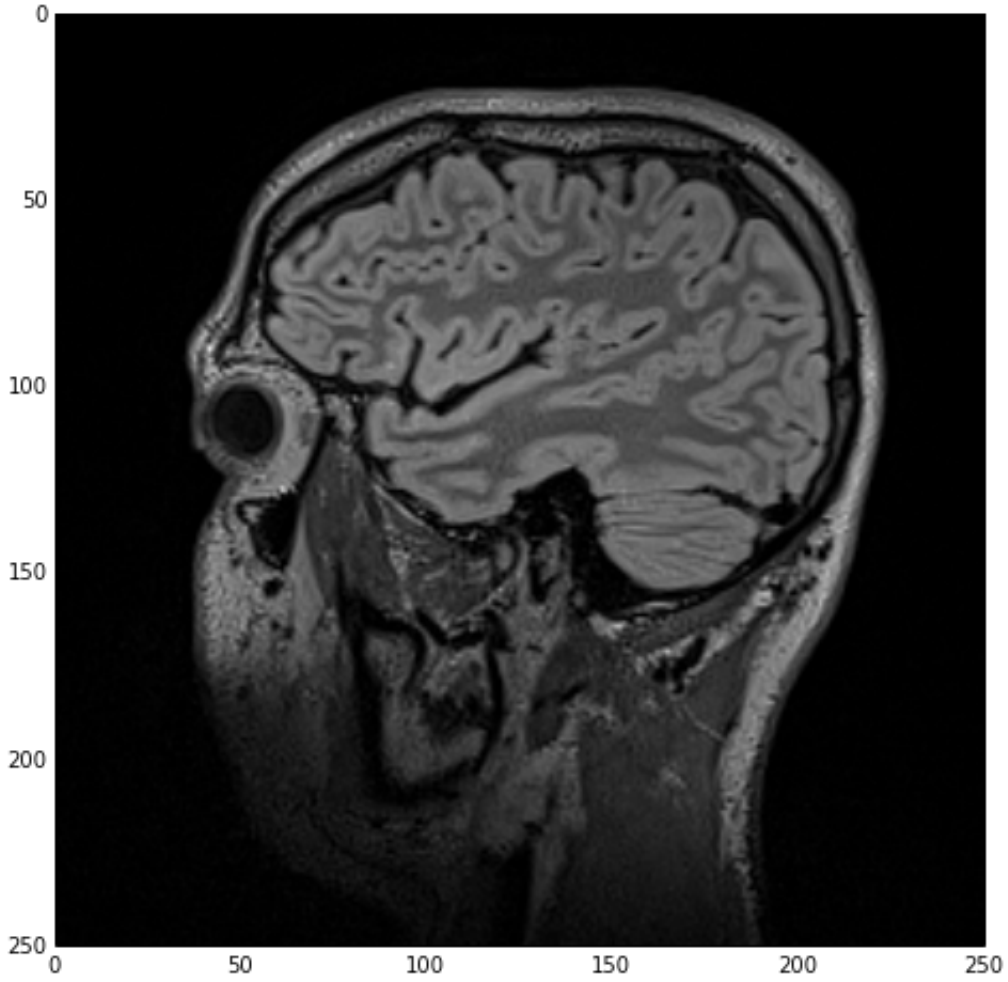
```
imgOriginal = imgOriginal[:,:,idxSlice]
```

Finally, we use the `sitk_show` helper-function to visualize that 2D image which we will be segmenting:

```
sitk_show(imgOriginal)
```

which yields the next figure.

> Note that, ironically, the white-matter (inner structure) appears as gray in the MR image while the gray-matter (outer structure) appears as white. Don't let that throw you off :).



Sagittal cross-section of the original MRI dataset clearly showing the white and gray matter of the brain among other tissues.

Smoothing/Denoising

As you can see from the above figure, the original image data exhibits quite a bit of 'noise' which is very typical of MRI datasets. However, since we'll be applying region-growing and thresholding segmentation algorithms we need a smoother, more homogeneous pixel distribution. To that end, before we start the segmentation, we smoothen the image with the `CurvatureFlowImageFilter`. Here's how we do that:

```
imgSmooth = SimpleITK.CurvatureFlow(image1=imgOriginal,
                                    timeStep=0.125,
                                    numberOfIterations=5)

# blurFilter = SimpleITK.CurvatureFlowImageFilter()
# blurFilter.SetNumberOfIterations(5)
# blurFilter.SetTimeStep(0.125)
# imgSmooth = blurFilter.Execute(imgOriginal)

sitk_show(imgSmooth)
```

The `CurvatureFlowImageFilter` class *"implements a curvature driven image denoising algorithm"*. The math behind this filter are based on a finite-differences algorithm and are quite convoluted. Should you feel like it, you can read more about the algorithm in the class' docs.

A point I'd like to make here is that all of the image filters in SimpleITK can be called in one of two ways. The first way is to directly call a function, `CurvatureFlow` in this case, which nicely wraps all required filter parameters as function arguments (while also exposing optional arguments). This is the way we're calling it above.

However, as you can see in the commented code, an alternative would be to create a filter object, `CurvatureFlowImageFilter` in this case, and set the required parameters one-by-one through the corresponding methods before calling the `Execute` method.

The above holds for all image filters included in SimpleITK. The filter class itself typically has an `ImageFilter` suffix while the corresponding wrapper function maintains the same name minus that suffix. Both approaches, however, wrap the same filters and which one to use is a matter of preference.

> Henceforth I will be using the direct function calls in the code but I'll be referencing the filter class as there's no docs for the former.

Regardless of the calling-paradigm, we end up with a `imgSmooth` image which contains the results of the smoothing. Using `sitk_show` we get the following figure:

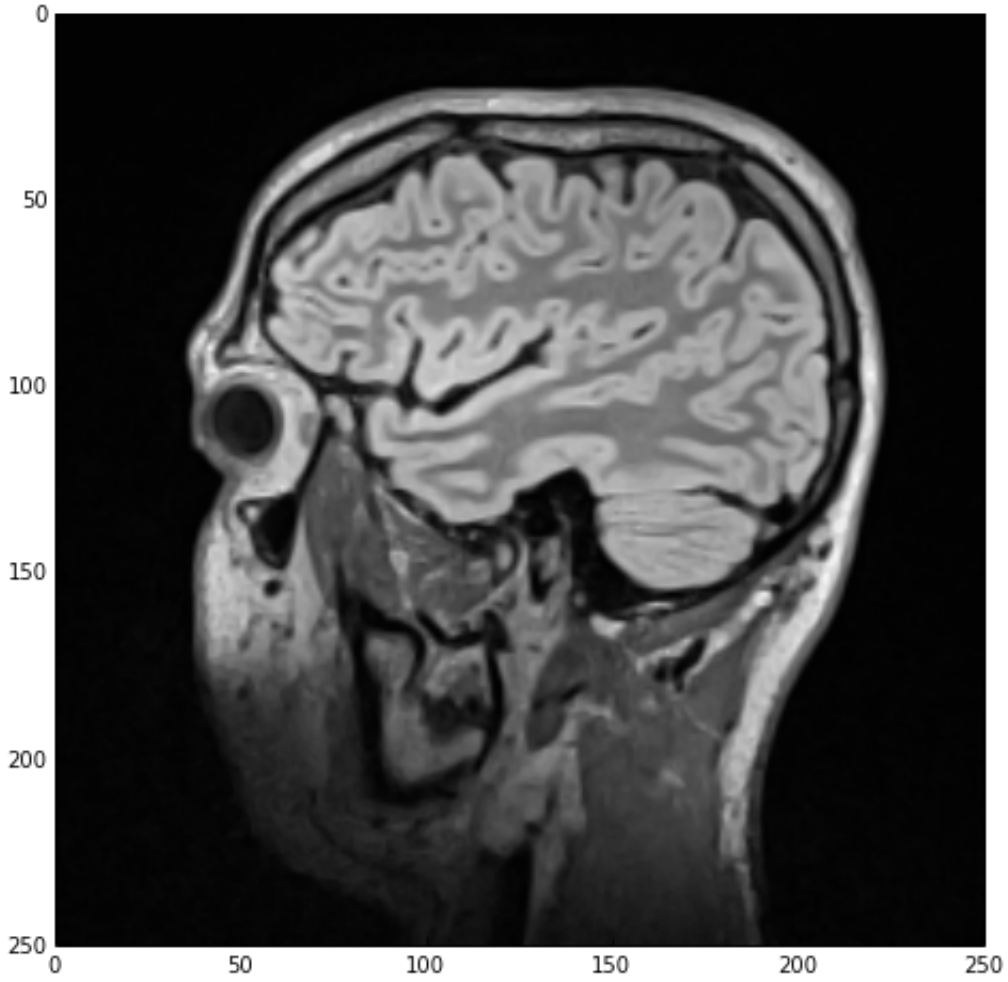


Image after smoothing/denoising with a `CurvatureFlow` filter.

> I should mention that image-smoothing is a very typical first step in the medical image data segmentation process, 'required' by the majority of segmentation algorithms.

Segmentation with the `ConnectedThreshold` filter

Initial segmentation of the white-matter

For the purposes of this post we'll be segmenting the white and gray matter through the `ConnectedThresholdImageFilter` class, or rather the `ConnectedThreshold` function which

wraps the former.

What this filter does is simply *"label pixels that are connected to a seed and lie within a range of values"*. Essentially, this filter operates on the input image starting from a series of given 'seed points'. It then starts 'growing' a region around those points and keeps adding the neighboring points as long as their values fall within given thresholds. Let's see how it's done:

```
lstSeeds = [(150,75)]

imgWhiteMatter = SimpleITK.ConnectedThreshold(image1=imgSmooth,
                                              seedList=lstSeeds,
                                              lower=130,
                                              upper=190,
                                              replaceValue=labelWhiteMatter)
```

So, we start by create a `list` of `tuple` objects with the 2D indices of the seed points. As I mentioned in the *Helper-Functions* section, these indices need to abide by the order expected by SimpleITK i.e., `(x, y, z)`. Through the previous figure we spot a good seed-point in the white-matter (inner structure of the brain) at an `x` index of `150` and a `y` index of `75`. Also, upon inspection of the slice, an inspection I performed while viewing the data in Osirix, I could see that the white-matter pixels exhibited values, roughly, between `lower=130` and `upper=190` which we'll be setting as the threshold limits. Lastly, we configure the filter to set a value of `labelWhiteMatter` to all pixels belonging to the white-matter tissue (while the rest of the image will be set to a value of `0`). As a result of all this we get the `imgWhiteMatter` image.

> Note that we operate on `imgSmooth` and not `imgOriginal`. That was the whole point of de-noising `imgOriginal` anyway right?

Next, the reasonable thing to do would be to inspect the result of the segmentation. We could simply use `sitk_show`, passing `imgWhiteMatter`, but all we would see would be a white-color label in a black backdrop which doesn't exactly give us much insight. Therefore, we'll instead view this newly acquired label overlaid with `imgSmooth`, i.e., the

input-image we used for the segmentation, using the `SimpleITK.LabelOverlayImageFilter` class:

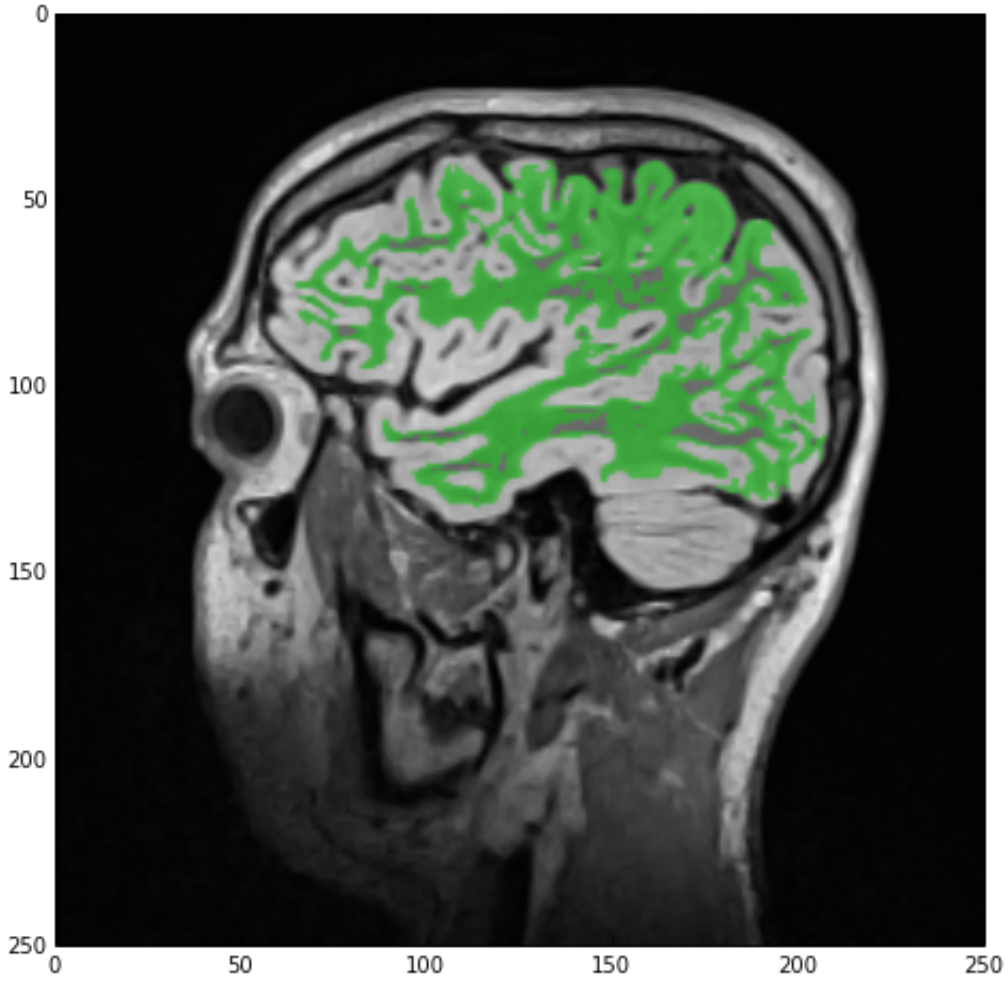
```
# Rescale 'imgSmooth' and cast it to an integer type to match that of 'imgWhiteMatter'
imgSmoothInt = SimpleITK.Cast(SimpleITK.RescaleIntensity(imgSmooth), imgWhiteMatter.Ge

# Use 'LabelOverlay' to overlay 'imgSmooth' and 'imgWhiteMatter'
sitk_show(SimpleITK.LabelOverlay(imgSmoothInt, imgWhiteMatter))
```

Here I want to stress the following point: the result of the de-noising, i.e., `imgSmooth`, comprises pixels with `float` values while the result of the segmentation is of `int` type. Mixing the two won't work, or will at least have unforeseen results. Therefore, we need to first 'rescale' `imgSmooth` to the same integer range and then cast it so that the types of the two images, i.e., `imgSmooth` and `imgWhiteMatter`, match.

We do that by first using the `RescaleIntensityImageFilter` with its range to the default values of `0` and `255`. Subsequently, we 'cast' the image to the same type as `imgWhiteMatter` by using the `CastImageFilter` and the `GetPixelID` method of the `SimpleITK.Image` class. As a result we get an integer version of `imgSmooth` dubbed `imgSmoothInt`.

> We'll be using `imgSmoothInt` for all subsequent label overlays but won't repeat the rescaling/recasting. Keep the name in mind.

Finally, we overlay `imgSmoothInt` and `imgWhiteMatter` through the `SimpleITK.LabelOverlayImageFilter` class which creates a nice basic-color RGB depiction of the otherwise monochrome 2nd image. Using the `sitk_show` helper function we get the next figure.



Label overlay showing the results of the white-matter's initial segmentation over the denoised MR image.

Hole-filling

As you can see from the above figure, our initial segmentation is subpar at best. Not only did we 'eat into' the gray matter, with which we'll deal later, but in addition there are numerous 'holes' in the white-matter label. Let's start by rectifying this.

Hole-filling is a very standard procedure in image segmentation, especially when employing region-growing algorithms. The reason? Well often enough regions of the tissue exhibit pixel values outside the defined thresholds either due to excessive noise in the image or the nature of the tissue itself in the given region. Thankfully, SimpleITK

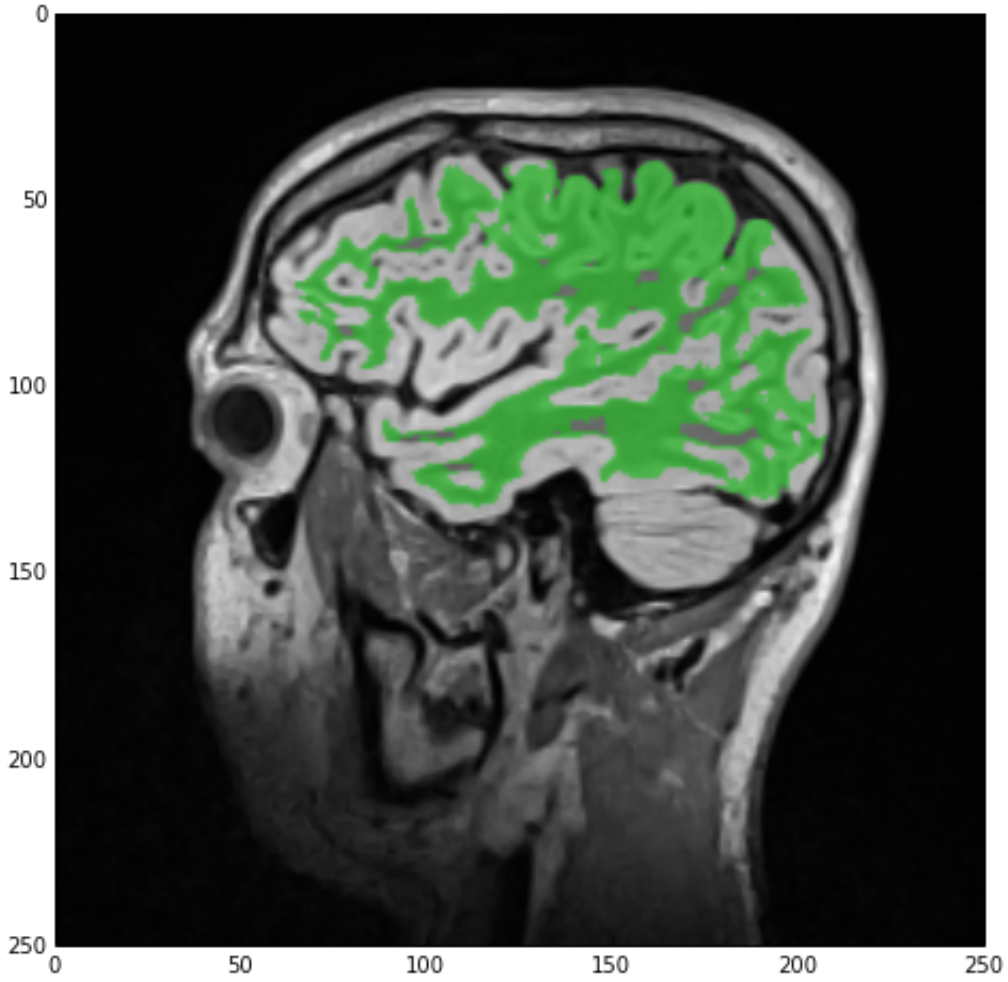
provides us with an arsenal of filters to ameliorate such issues.

One such filter is the `VotingBinaryHoleFillingImageFilter` which in a nutshell *"Fills in holes and cavities by applying a voting operation on each pixel"*. I suggest you check the filter docs for details on the actual implementation but in layman's terms what this filter does is check every 'off' pixel, i.e., a pixel with a `backgroundValue`, and sets it to a `foregroundValue` if the majority of the pixels around it also have a `foregroundValue`. Here's how that's done:

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

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

As you can see we once more skip the whole 'initialize filter-configure filter-execute filter' process and run the wrapper straight away. If my crummy explanation above wasn't sufficient then off to the `VotingBinaryHoleFillingImageFilter` docs. Let me just say that the `radius` defines the pixel area/volume around a hole that is examined and the `majorityThreshold` is the number of pixels over `50%` that need to be 'on' for that hole pixel to be turned 'on'.

The result of the hole-filling operation is stored under `imgWhiteMatterNoHoles` and we once more overlay this new label with `imgSmoothInt` getting the next figure. As you can see the operation wasn't wildly successful but many of the smaller holes were indeed filled. All this means is that we should've imposed looser criteria, e.g., a larger `radius`.



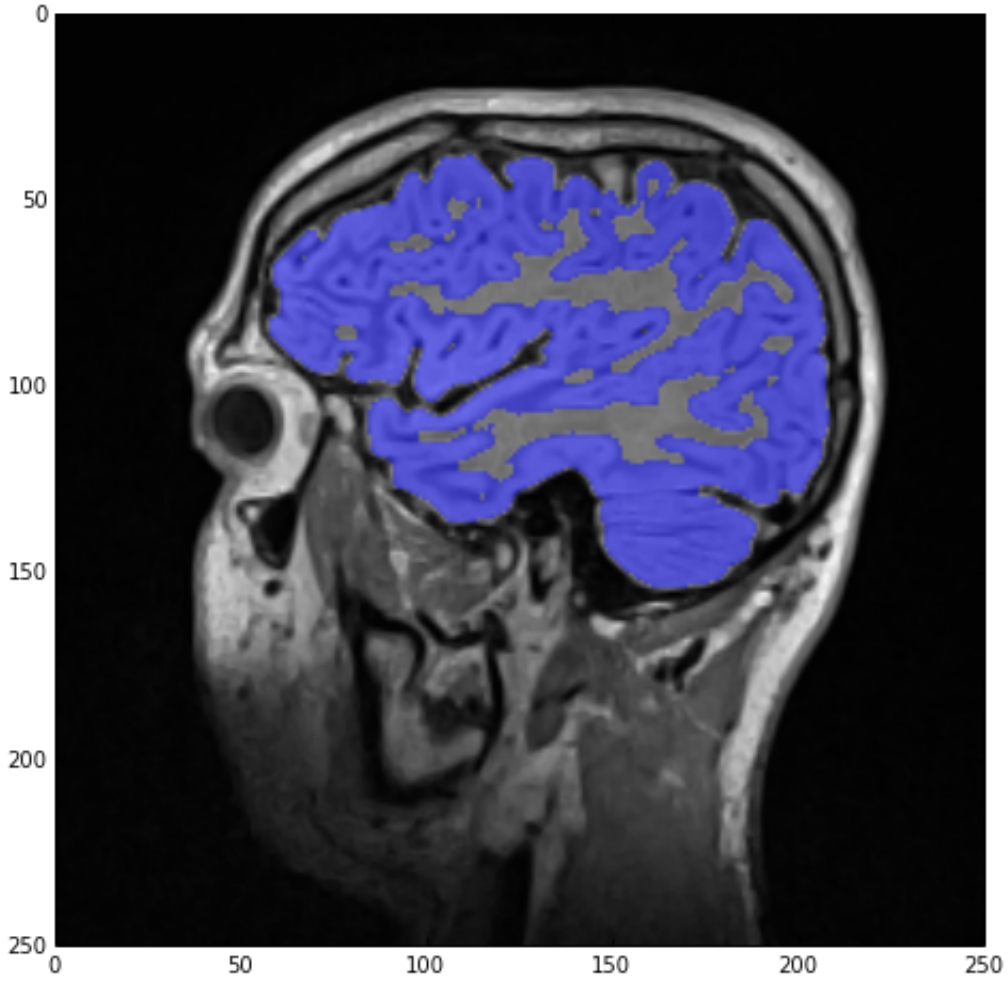
Label overlay showing the results of the hole-filling operation on the white-matter.

Segmentation of the gray-matter

Next we'll repeat the process we just saw but this time for the gray matter. Here's the code:

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

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

imgGrayMatterNoHoles = SimpleITK.VotingBinaryHoleFilling(image1=imgGrayMatter,
                                                         radius=[2]*3,
                                                         majorityThreshold=1,
                                                         backgroundValue=0,
                                                         foregroundValue=labelGrayMatt

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

As you can see, apart from defining four different seeds and changing the thresholds and label index, the process is entirely the same, i.e., segmentation and hole-filling. The result of the label overlay with `sitk_show` can be seen below.



Label overlay showing the results of the gray-matter's segmentation and hole-filling.

Label-field mathematics

Lastly, we want to combine the two label-fields, i.e., the white and gray matter. As I mentioned in the *Introduction*, the `SimpleITK.Image` class overloads and supports all binary and arithmetic operators. Hence, combining the two label-fields is as easy as a simple OR operator (`|`):