18. Application: DICOM

DICOM (Digital Imaging and Communications in Medicine) is the international standard to transmit, store, retrieve, print, process, and display medical imaging information. It is in particular widely used to store volumetric data from methods such as CT, MR, Ultrasound, etc.

This kind of specific image format is typically not supported by general packages such as scikit-image. However in most cases, independent dedicated packages exist. A simple Google search leads us to the pydicom (https://pydicom.github.io/pydicom/stable/getting_started.html) package.

In [1]:
```
import os
import matplotlib.pyplot as plt
plt.gray()
import pydicom
import numpy as np
import skimage
import ipyvolume as ipv
```

We will use an MRI dataset of a head available on the data sharing platform Zenodo. In this course, most data have been made directly available. To show the full procedure, we will here include the download step.

Install the missing package:

In [2]:
```
!pip install --user pydicom
```

```
Requirement already satisfied: pydicom in /usr/local/lib/python3.5/dist-p
ackages (1.4.1)
You are using pip version 19.0.3, however version 20.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' comman
d.
```

In [3]:
```
import pydicom
```

18.1. Download

The donwload address on Zenodo is:

In [4]:
```
data_address= 'https://zenodo.org/record/16956/files/DICOM.zip?download=
1'
```

Create a folder where to put the data:

In [5]:
```
#os.makedirs('MyData')
```

We can use the `urllib` native package to proceed with download which provides us with a zip file:

In [6]:
```
import urllib

urllib.request.urlretrieve(data_address, 'MyData/mri.zip')
```

Out[6]: ('MyData/mri.zip', <http.client.HTTPMessage at 0x7fb257782400>)

To automate the process we now also automatically unzip the file using the `zipfile` module:

In [7]:
```
import zipfile
```

In [8]:
```
with zipfile.ZipFile('MyData/mri.zip', 'r') as zip_ref:
    zip_ref.extractall('MyData/mri/')
```

18.2. Importing one slice

We define the general path to the folder containing slices:

In [9]:
```
path = 'MyData/mri/DICOM/ST000000/SE000002/'
```

Now we use the `pydicom` package to import a single slice using the `dcmread()` function:

In [10]:
```
single_slice = pydicom.dcmread(path+'MR000000')
```

A DICOM file does not just contain image data but a very extensive set of metadata. You can see these metadata by just printing the variable:

In [11]:
```
single_slice;
```

All that information is also available as attributes of the variable. For example you can get the patient's name:

In [12]:
```
single_slice.PatientName
```

Out[12]: 'LIONHEART^WILLIAM'

But also numerical values such as pixel spacing or position of slice in the stack:

In [13]:
```
single_slice.PixelSpacing
```

Out[13]: [0.8984375, 0.8984375]

In [14]:
```
single_slice.SliceLocation
```

Out[14]: "0.0"

18.3. Loading the complete stack

As we have already done previously, we have first to parse the folder content to gather the files belonging to the stack. Here we simply list the folder content:

In [15]:
```
file_list = os.listdir(path)
```

In [16]:
```
#file_list
```

We can now load each slice using a comprehension list. From the file sorting, we already see that we'll later have to reorder the slices.

In [17]:
```
slices = [pydicom.dcmread(path+x) for x in os.listdir(path)]
```

In principle we could reorder the file by names but this is going to depend on file name formatting. A more general solution is to reorganize based on the location of the file in the stack. Let's recover that position:

In [18]:
```
positions = [int(x.SliceLocation) for x in slices]
```

In [19]:
```
#positions
```

We then use `np.argsort()` function to get the indices of the ordered list:

In [20]:
```
import numpy as np
index_ordered = np.argsort(positions)
```

In [21]:
```
index_ordered
```

Out[21]:
```
array([21,  2,  1, 20,  3, 11, 13,  9, 29, 28, 22, 26, 18,  5, 23, 16, 31,
       15, 12, 10,  0, 19,  6,  4, 24, 14, 17,  8, 30,  7, 27, 25])
```

And finally use that ordered list to reorder the slices themselves:

In [22]:
```
reordered = []
slices_ordered = [slices[x] for x in index_ordered]
```

18.4. Visualization

Finally we can visualize our volume. First let's create an actual volume by stacking the planes:

In [23]:
```
volume = np.stack([x.pixel_array for x in slices_ordered])
```

In [24]:
```
volume.shape
```

Out[24]:
```
(32, 256, 256)
```

For the rendering, we'll see here two different solutions. The first one is ipyvolume, a leight-weight volume viewer purely based on browser technology. The syntax is very similar to matplotlib.

In [25]:
```
#import ipyvolume as ipv
```

In [26]:
```
ipv.figure()
ipv.volshow(volume)
ipv.show()
```

```
/usr/local/lib/python3.5/dist-packages/ipyvolume/serialize.py:81: Runtime
Warning: invalid value encountered in true_divide
  gradient = gradient / np.sqrt(gradient[0]**2 + gradient[1]**2 + gradien
t[2]**2)
```

As ipyvolume is fully browser-based, it's very easy to save an image as a web page. For example we can just type:

In [27]:
```
ipv.save('interactive_view.html')
```

```
/usr/local/lib/python3.5/dist-packages/ipyvolume/serialize.py:81: Runtime
Warning: invalid value encountered in true_divide
  gradient = gradient / np.sqrt(gradient[0]**2 + gradient[1]**2 + gradien
t[2]**2)
```

And this saves for us a full interactive version of the figure above. This can therefore be very useful for demonstration purposed e.g. to insert an image on a web-page.

Note that customizing the aspect of the view requires some work and that this package is not as mature as others.

An alternative solution is to use the ITK (Insight Toolkit), a very popular image processing tool suite in medical imaging (an interesting but more challenging alternative to scikit-image). ITK in particular offers a volume viewer compatible with Python and Jupyter:

In [29]:
```
import itkwidgets as itkw
import itk
```

We can just call the `view()` function:

In [30]:
```
itkw.view(volume)
```

We see that the head looks compressed because the acquisition is anisotropic (large depth dimension that width/height). Above we simply passed a Numpy array to the viewer. However we can also create a native ITK format to adjust parameters more easily:

In [31]:
```
image_from_array = itk.image_from_array(volume)
```

This object has now several new attributes and methods such as:

In [32]:
```
image_from_array.GetSpacing()
```

Out[32]:
```
itkVectorD3 ([1, 1, 1])
```

We can try to guess and adjust the spacing:

In [33]:
```
image_from_array.SetSpacing((1,1,10))
```

Or we can use the itk package to read the native spacing:

In [34]:
```
itk_slice = itk.imread(path+'MR000001')
spacing = itk_slice.GetSpacing()
spacing
```

Out[34]: itkVectorD3 ([0.898438, 0.898438, 6])

In [35]:
```
image_from_array.SetSpacing(spacing)
```

In [36]:
```
itkw.view(image_from_array)
```

18.5. Image processing

Finally, we can do the same image processing operations as we did before, just in 3D. For example a thresholding:

In [37]:
```
import skimage.filters
```

In [38]:
```
vol_thresh = volume>200
```

In [39]:
```
itkw.view(vol_thresh.astype(np.uint8))
```