# ImagesAndColourmaps_NoSolutions

AAB

January 23, 2022

## Image Processing

### Practical 1: Image Storage and Display

**Version 1.0, 23rd Jan, 2022**

### 1. Introduction

The purpose of this practical is to introduce some basic tools and concepts that are required for working with digital images. The choice of the topics for this first practical has been informed by personal experience gained from working in medical imaging, specifically where one is *close* to the image acquisition process, and to the mapping between some measured value and an image representation that, eventually, leads to a displayed image. The other source of motivation for the topics covered in this practical is, to some extent, in visualising information that represents some function of 2 dimensional position, of the form $f(x, y)$.

The mapping of some 2D function of position to a scalar intensity or vector (colour) value is implicit in the concept of an image. Of course, the images we will deal with are always sampled on a discrete spatial grid instead of functions of continuous spatial variables you might have met in mechanics or fluids analysis. Conventionally, as for the main notes, we denote such images by $f(m, n)$, where it is to be understood that $m$ and $n$ are integer values.

Within this practical, there are two distinct themes that should be seen as learning objectives. These objectives are:

1. To appreciate how images are stored on
   disk: what the nature of the image representation is, and how to get at information within an image file. We will touch on meta-information as well, which is often very important in "unscrambling" an image representation on disk, or otherwise in a sequence of bytes.
2. To appreciate the mapping between the image representation held in a computer's memory (such as in an array within a program) and the visual display of the computer; actually, what really needs to be emphasised is that there is a mapping, and appreciating the nature of this mapping can be very important in correctly displaying images.

The **first** topic we will briefly tackle regards the reading of image data held on disk. In the first part of the practical, you should ask the demonstrators to explain the commands being used to read the "head" image. Once you understand the nature of an image representation as a sequence of byte values, it will be easier to understand why dedicated file formats are needed to represent image data.

The **second** topic deals with the display of images on a computer screen. Since most of you are familiar with the idea of auto-scaling graphical plots (found in Matlab, Excel etc.), the display of image data, in which the dynamic range of pixel values may differ from that of the "displayable", range may present some surprises. This is compounded by fact that in image display, there is often no associated representation of the actual magnitude or signs of the values being displayed. When trying to interpret scientific measurements over 2D space via an image on screen, one therefore has to be a little bit cautious.

So, following Section 2 (on reading an image as a stream of raw bytes), there will be an introduction to colourmaps, also known as colour palettes. Once you appreciate colourmaps, and the way that image display relies on them, you will be in a position to begin working with processed images, where the dynamic range and values that are produced as a result of some stages of image processing may not match those of typical "raw" captured image data: values may be outside the range of the colour palette. The processed images that we shall be considering in Practical 2 are those that arise after performing arithmetic or set-based operations on pixels, including those obtained from local spatial operators.

## 2. An image as a "byte stream"

In developing new imaging systems, or in working with digital cameras, one sometimes encounters images stored in their most unstructured form: as a series of bytes, where each byte encodes the colours or intensities received by a pixel sensor. Indeed, if working with unusual medical imaging equipment, sometimes one is faced with a proprietary file format for storing image data, and one has to read supposedly structured data by treating it as *unstructured*. Whilst this may sound bizarre, it is a fact of life to which many imaging scientists get used to when using rare, expensive or legacy imaging equipment.

For those of you that have met the idea of "serialization", such as in *Python*, it is the idea that data as stored on a hard disk, or as a series of bits to be transmitted in sequence, like a message, are stored in a one-dimensional fashion, even if the values represent a spatial structure, like an image.

We will first load a "raw" data file image into *Python* and organise it into an image array. The file was *exported* from image display software used decades ago in-house, and with a proprietary file format. A magnetic resonance image of a slice of the human head is provided, with pixel dimensions of 128 rows x 128 columns. The image you will work with was exported into a file as a sequence of bytes, of length equal to the number of pixels (16,384). The commands in *Python* are quite similar to those needed to read the data using C programming, or pretty much any functional programming environment, including *Matlab*.

Start by downloading the file "ImageDataFiles.zip" from the same place on BlackBoard where you found this document. Download this, and unzip the files into a directory that you have access to. We will refer to the directory containing the unzipped files as `<PathContainingFiles>`, and so you need to replace this string with the true path for your particular case in what follows below.

If the zip file is on your current machine, you may need to specify a drive letter (e.g. `C:\<PathContainingFiles>`). If the files are on a remote server, and you are on a Windows machine, this path might also be specified using the Windows Universal Naming Convention (UNC) (e.g.`\\dataserver.domain.where.uk\LaboratoryData`); the equivalent is usually easily found for Linux or MacOS.

Also, if working off the networked home directories, you may be able to access this by including `H:\`, or `M:\` before the rest of the directory path, depending on what IMperial's ICT's latest choice of naming convention is. A test to see whether you have the path right is to use the following from either a *Python* interpreter, or execute a cell containing this command:

[1]:
```
import os
import sys

os.path.isfile('IMGS/head.128')
# If you have the path to the file right, you will get 'True'
```

[1]:
```
True
```

We can also grab simple file-system information about this file's statistics

[2]:
```
# Let's look at the file size.....
if (os.path.isfile('IMGS/head.128')):
    FileStats = os.stat('IMGS/head.128')
    print('Size of file is', FileStats.st_size,'bytes')
```

```
Size of file is 16384 bytes
```

[3]:
```
with open('IMGS/head.128', 'rb') as binary_file:
    # `binary_file` is now an object with methods for reading....
    # Read the whole file at once
    # Note that you may not find many examples of this
    # online, as reading pure binary files is seen as rare
    # by many who work with pre-prepared images
    data = binary_file.read()
```

We can check on the size of this chunk of data like so:

[4]:
```
len(data)
```

[4]:
```
16384
```

But as far as *Python* knows, this is not an image - it is just a collection of bytes.

[5]:
```
type(data)
```

[5]:
```
bytes
```

We need to tell the *Python* kernel that this lump of data can be treated as a collection of unsigned integer values, converting each byte to an integer between 0 and 255 (so, an *unsigned* integer). If you do not know what this means, please speak with a GTA/UTA. Luckily, `numpy` has code for doing this, trivial as it is, it is really useful.

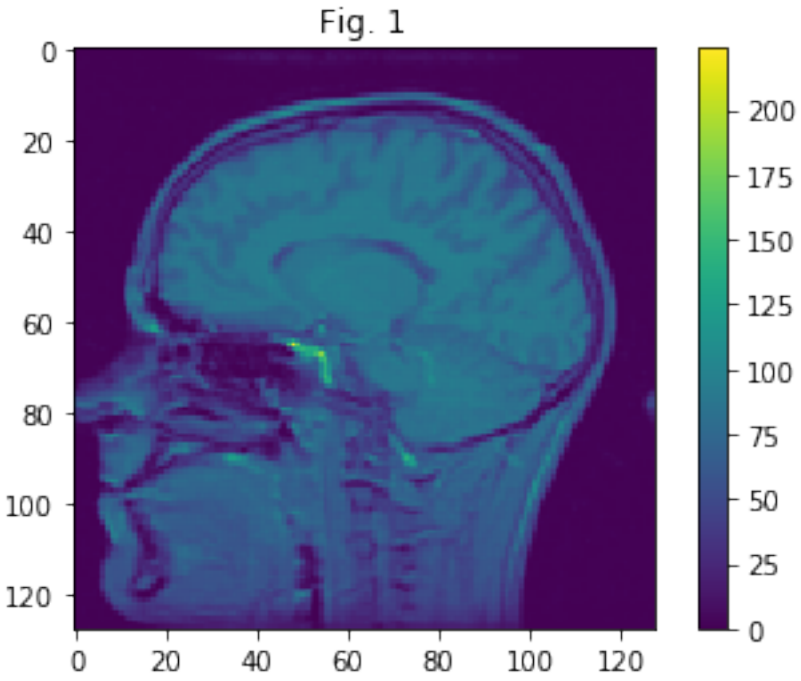
[6]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Line below allows us to treat bytes data as integers...
data = np.frombuffer(data, dtype=np.uint8)

# The next line tells us how to organise it
x = np.reshape(data,(128,128))
```

[7]:
```
# See box below for the exaplnation of this directive
%matplotlib inline
plt.imshow(x)
plt.colorbar()
plt.title('Fig. 1')
```

[7]: `Text(0.5, 1.0, 'Fig. 1')`




It is convenient to display the images directly into the Notebook (this is called *inline* figure graphics). However, you will find that it is often very useful to either be able to zoom into an image, or hover the mouse (cursor) over a particular pixel to interrogate its value. To do this, you will want to use an external dipslay tools, which would normally exist outside of the notebook, providing these extra tools. To access these, what you need to do is to place the following somewhere in your notebook, after you have imported **matplotlib**:

```
%matplotlib
```

You should then re-start the kernel, re-running all commands up to the point of image display. Rather opaquely, this “directlive” tells the Jupyter kernel to invoke an external image viewer/plot

figure, which provides you with some extra functionality for zooming, panning and most importantly, interrogating pixel values directly; so, you can easily use the mouse/cursor to figure out what value exists in the image array at a particular location. This is very useful for really getting to appreciate the nature of colourmaps, and also to probe specific values you may have in an image, particularly after applying some operator.

To go back to having the images displayed directly in the Notebook, without the extra helpful tools, use

```
%matplotlib inline
```

**Interlude 1** Key learning outcomes from this first exercise with the `head.128` image.

If you investigate this image file using `Finder` on MacOS or using the File Browser (or command line commands) on *Windows 10/11*, or using `ls -l` on Linux, you will see that it contains only 16,384 bytes. That is, exactly one byte for every pixel. There is no additional information about how many rows and columns of the image, no information on the colour space to be used, or indeed a colour map. The only way we can load and display this image is because *a priori* the author of this notebook knew what the image contained, and that information is then used to reshape and display the image once the sequence of bytes has been loaded from disk.

In contrast, the other images we will look at are stored in standard file formats, including TIFF and DICOM 3.0. For these image types, not only will there be additional information about the image (such as geometry), but also one or more possible forms of compression will be applied.

### 3. The nature of colourmaps

In order to understand the operation of colourmaps, also known as colour Look-Up Tables (LUT's) or colour palettes, you need to know a little bit about how images are displayed. In particular, it is important to realize that there is a distinct stage in any image display system which consists of mapping the pixel values contained in the image matrix, $x(m, n)$, onto the colour and intensity space "understood" by the screen.

Display systems of many current PC's, and of Linux workstations (including Mac OS X, which is built on top of a Berkeley Unix distribution), have three colour input signals, corresponding to red, green and blue components. On older high-end workstations, the three components are generally provided by three coaxial connectors connecting the workstation to its monitor. On older everyday PC's, the three signals are within the VGA connector. On modern digital connectors, the signals between the computer and the screen are be digital, and *serialized* so that the conversion to an analogue signal used to drive the pixels onscreen will occur in the monitor. So there may be subtle differences in where the conversion from digital representation to a final displayed intensity happens; don't get too hung up on this, but do recognise that some sort of mapping from byte values to intensity takes place.

Any colour which you see on screen (including shades of grey) is made from specific combinations of the three additive primary colours, red, green and blue. Thus, any program or "application" which displays an image on your computer must have some means of mapping the pixel values contained in the image to the appropriate combinations of red, green and blue signals. The required mapping process is performed by using a combination of hardware and a set of low-level software routines.

The nature of the mapping is specified by a data structure that represents a colour look-up table or possibly even a hardware-implemented colour palette.

A rough conceptual diagram of the mapping performed by a colourmap is shown in **Fig. 2**. Note that the colour look up table (LUT) or colourmap really does serve as a kind of reference table, associating particular values of red, green and blue with particular pixel values. Some palettes contain a fourth column, which is known as the alpha channel, which specifies transparency.

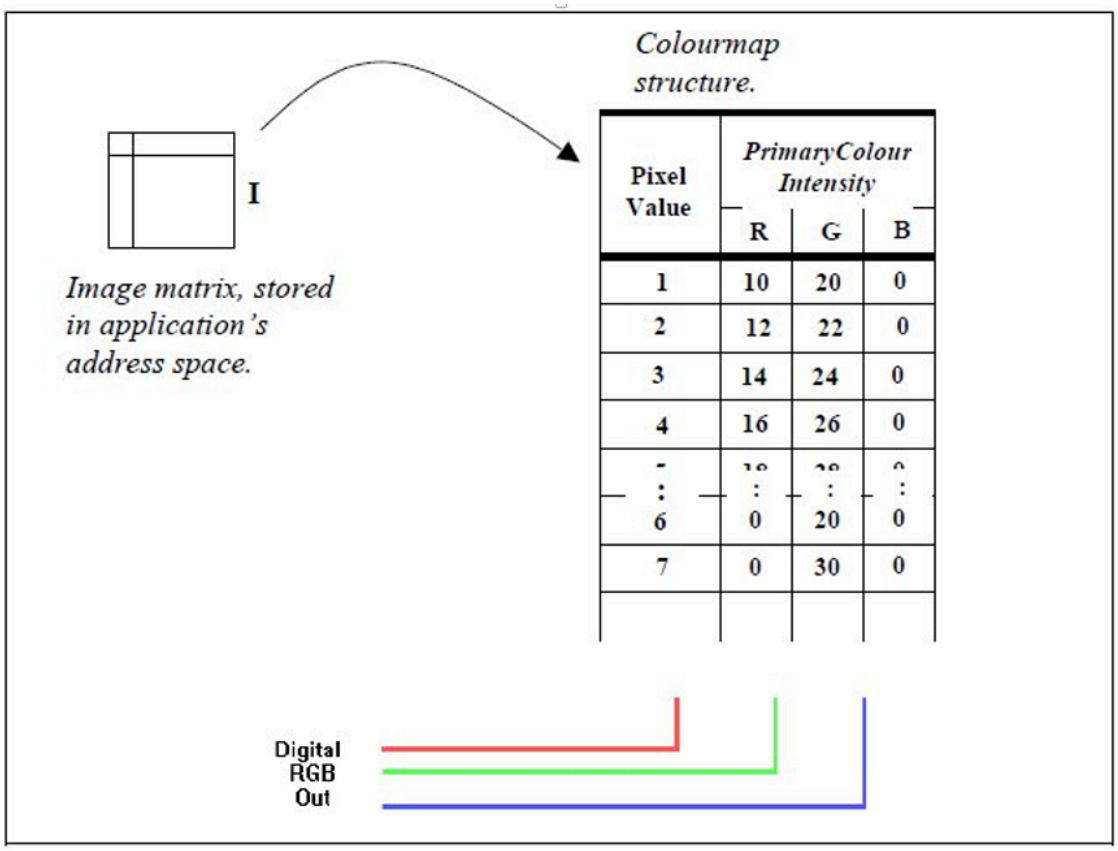


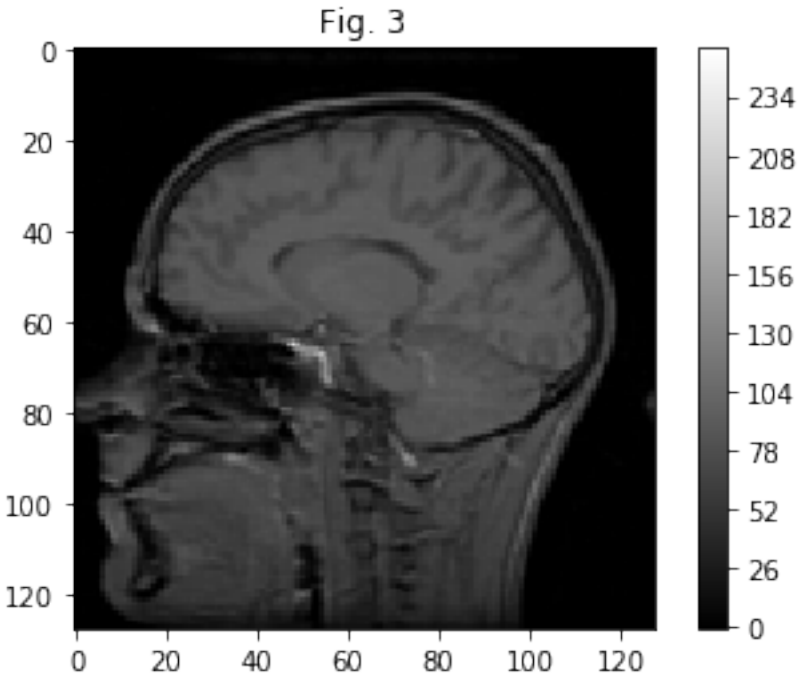

**Fig 2.** Colourmap concept. An image array containing a single byte per pixel does not have a predefined appearance on screen without a colourmap. The colourmap shown here as 4 column table is actually corresponds to a 3 column colourmap, has the one column that only holds the row number is just the index corresponding to the row of the 2D array representing the colourmap. So, this would be stored as an $N\times$ 3 array, where $N$ correesponds to the number of rows (or colourmap "slots" or "entries"). The colourmaps in **matplotlib** contain 4 columns, with the last column corresponding to a so called alpha value. We can be safe in setting this to 1, and that represents a transparency of 0 (so alpha really represents *opacity*, though it is sometimes called transparency).

So, let's apply a grey scale colourmap to the image (that's what most medical images have as default; colours are usually reserved for functional, or activation maps)

[8]:
```
from matplotlib import colors
plt.imshow(x, cmap='gray', norm=colors.NoNorm()) # Integer x values mapped␣
 ↪directly to
```

```
                                                    # corresponding row of the␣
↪colourmap
plt.colorbar()
plt.title('Fig. 3')
```

[8]: Text(0.5, 1.0, 'Fig. 3')

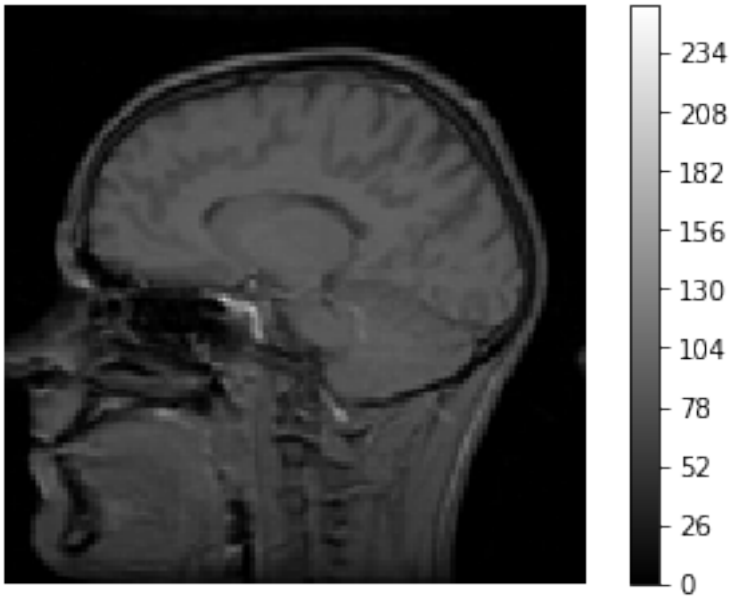



We can also turn off the axis if we don't need to inspect pixel locations like this:

[9]:
```
plt.imshow(x, cmap='gray', norm=colors.NoNorm())
plt.colorbar()
plt.axis('off')
plt.title('Fig. 4')
```

[9]: Text(0.5, 1.0, 'Fig. 4')

Fig. 4




Now, we are going to investigate the colourmap structures.

[10]:
```
import matplotlib.cm as cm
import matplotlib.colors as colors
cmap = cm.get_cmap('gray')
```

[11]:
```
type(cmap)
```

[11]: `matplotlib.colors.LinearSegmentedColormap`

If we want to see a bit more about what *methods* and attributes are available to us, we can use the following syntax:

`dir(cmap)`

The number of entries in the colourmap is provided by `cmap.N`, and the colour look-up-table itself that represents the gray scale map is held in `cmap._lut`.

So, if we type:

`cmap.N`

you will find the (quite common) colourmap size of 256 entries. The colour table iteself has some surprises; let's look at this step by step. First, ty this:

[12]:
```
np.shape(cmap._lut) # Should give us the shape of the colourmap
```

[12]: `(259, 4)`

So, there are two odd observations. First, rather than having 3 columns (for R, G and B) respectively, this colourmap has 4 columns! Secondly, rather than having 256 rows (as suggested by `cmap.N`, there are an extra 3 entries in the look-up-table. First, let us examine the values in the rightmost column:

[13]:
```
A = cmap._lut[:,3]
```

So, we can see that the rightmost values are pretty much all 1, apart from the very last value. The values in this column are actually opacity values, and explain how this colour should be blended with a background colour. We'll come back to this later; for now, let's look at the R,G and B values, which are the leftmost, second and third columns respectively from the left:

[14]:
```
RGBA = cmap._lut
RGBA[:,0]
```

[14]:
```
array([0.        , 0.00392157, 0.00784314, 0.01176471, 0.01568627,
       0.01960784, 0.02352941, 0.02745098, 0.03137255, 0.03529412,
       0.03921569, 0.04313725, 0.04705882, 0.05098039, 0.05490196,
       0.05882353, 0.0627451 , 0.06666667, 0.07058824, 0.0745098 ,
       0.07843137, 0.08235294, 0.08627451, 0.09019608, 0.09411765,
       0.09803922, 0.10196078, 0.10588235, 0.10980392, 0.11372549,
       0.11764706, 0.12156863, 0.1254902 , 0.12941176, 0.13333333,
       0.1372549 , 0.14117647, 0.14509804, 0.14901961, 0.15294118,
       0.15686275, 0.16078431, 0.16470588, 0.16862745, 0.17254902,
       0.17647059, 0.18039216, 0.18431373, 0.18823529, 0.19215686,
       0.19607843, 0.2       , 0.20392157, 0.20784314, 0.21176471,
       0.21568627, 0.21960784, 0.22352941, 0.22745098, 0.23137255,
       0.23529412, 0.23921569, 0.24313725, 0.24705882, 0.25098039,
       0.25490196, 0.25882353, 0.2627451 , 0.26666667, 0.27058824,
       0.2745098 , 0.27843137, 0.28235294, 0.28627451, 0.29019608,
       0.29411765, 0.29803922, 0.30196078, 0.30588235, 0.30980392,
       0.31372549, 0.31764706, 0.32156863, 0.3254902 , 0.32941176,
       0.33333333, 0.3372549 , 0.34117647, 0.34509804, 0.34901961,
       0.35294118, 0.35686275, 0.36078431, 0.36470588, 0.36862745,
       0.37254902, 0.37647059, 0.38039216, 0.38431373, 0.38823529,
       0.39215686, 0.39607843, 0.4       , 0.40392157, 0.40784314,
       0.41176471, 0.41568627, 0.41960784, 0.42352941, 0.42745098,
       0.43137255, 0.43529412, 0.43921569, 0.44313725, 0.44705882,
       0.45098039, 0.45490196, 0.45882353, 0.4627451 , 0.46666667,
       0.47058824, 0.4745098 , 0.47843137, 0.48235294, 0.48627451,
       0.49019608, 0.49411765, 0.49803922, 0.50196078, 0.50588235,
       0.50980392, 0.51372549, 0.51764706, 0.52156863, 0.5254902 ,
       0.52941176, 0.53333333, 0.5372549 , 0.54117647, 0.54509804,
       0.54901961, 0.55294118, 0.55686275, 0.56078431, 0.56470588,
       0.56862745, 0.57254902, 0.57647059, 0.58039216, 0.58431373,
       0.58823529, 0.59215686, 0.59607843, 0.6       , 0.60392157,
       0.60784314, 0.61176471, 0.61568627, 0.61960784, 0.62352941,
       0.62745098, 0.63137255, 0.63529412, 0.63921569, 0.64313725,
```

```
0.64705882, 0.65098039, 0.65490196, 0.65882353, 0.6627451 ,
0.66666667, 0.67058824, 0.6745098 , 0.67843137, 0.68235294,
0.68627451, 0.69019608, 0.69411765, 0.69803922, 0.70196078,
0.70588235, 0.70980392, 0.71372549, 0.71764706, 0.72156863,
0.7254902 , 0.72941176, 0.73333333, 0.7372549 , 0.74117647,
0.74509804, 0.74901961, 0.75294118, 0.75686275, 0.76078431,
0.76470588, 0.76862745, 0.77254902, 0.77647059, 0.78039216,
0.78431373, 0.78823529, 0.79215686, 0.79607843, 0.8       ,
0.80392157, 0.80784314, 0.81176471, 0.81568627, 0.81960784,
0.82352941, 0.82745098, 0.83137255, 0.83529412, 0.83921569,
0.84313725, 0.84705882, 0.85098039, 0.85490196, 0.85882353,
0.8627451 , 0.86666667, 0.87058824, 0.8745098 , 0.87843137,
0.88235294, 0.88627451, 0.89019608, 0.89411765, 0.89803922,
0.90196078, 0.90588235, 0.90980392, 0.91372549, 0.91764706,
0.92156863, 0.9254902 , 0.92941176, 0.93333333, 0.9372549 ,
0.94117647, 0.94509804, 0.94901961, 0.95294118, 0.95686275,
0.96078431, 0.96470588, 0.96862745, 0.97254902, 0.97647059,
0.98039216, 0.98431373, 0.98823529, 0.99215686, 0.99607843,
1.        , 0.        , 1.        , 0.        ])
```

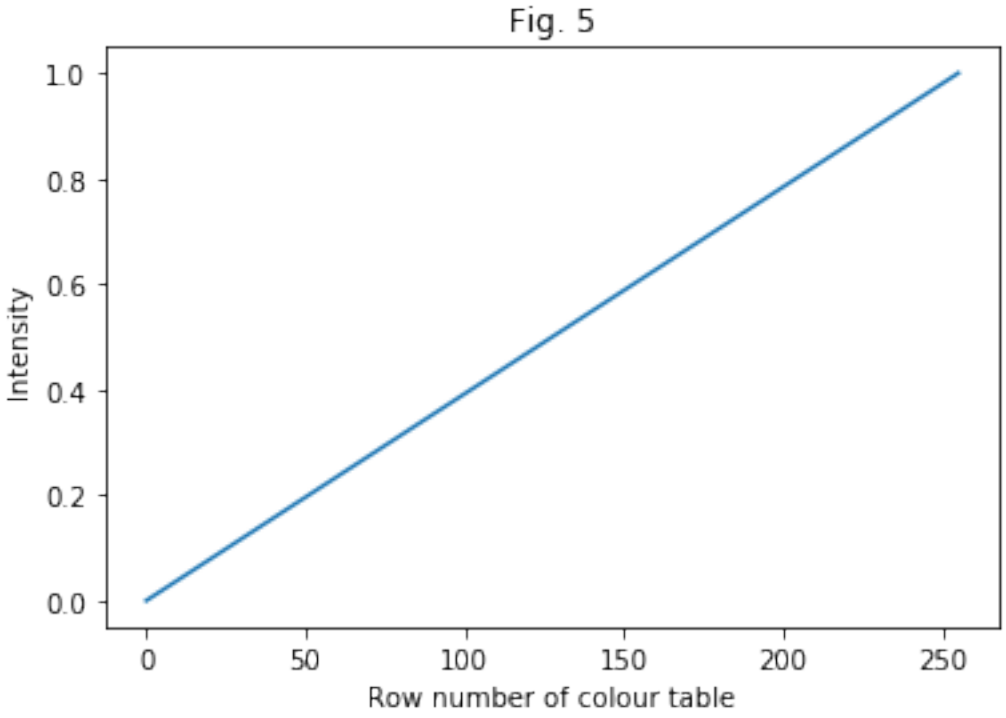
So, this is worth a plot, I think:

```
plt.plot(RGBA[:,0])
```

You can see that the values in this first colunm increase ina linear fashion until near the end. Indeed, if we take only the first 256 values of this array:

[15]:
```
plt.plot(RGBA[0:256,0])
plt.xlabel('Row number of colour table')
plt.ylabel('Intensity')
plt.title('Fig. 5')
```

[15]:
```
Text(0.5, 1.0, 'Fig. 5')
```




**Quiz:**

1. What is the pattern of values in this column?
2. What values are in the other 2 columns i.e. column index 1 and column index 2? Why are the values like this?
3. Why are there more entries in the table than the colourmap seems to suggest (`cmap.N`)?

Now, image display based on `plt.imshow` is not *fully* correctly implemented in **matplotlib**. We can see this if we compare what the **matplotlib** documentation says *should* happen with what actually happens. We can get confirmation of these inconsistencies by comparing with outputs in *Matlab*, where the documentation is consistent with the behaviour of equivalent code. The difference and inconsistencies are almost certainly because of the user-contributed nature of **matplotlib**, but it is something one can live with!

Let's return to look at the command we have used for display; the option `norm=colors.NoNorm())` sets `imshow` to map colours directly through the colour palette; so the value in the image array `x` is used to identify a specific row of the color palette, and the color specified in that row is used for the corresponding pixel. This gives us precise control over the mapping of values in image array `x` to screen appearance.

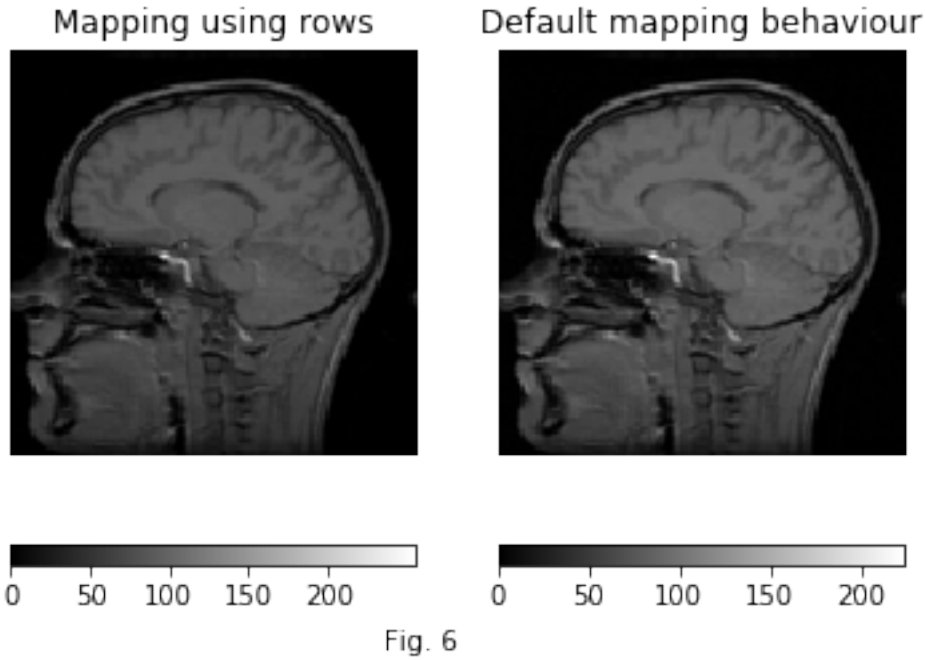
However, the *default* behaviour of `imshow()` (without forcing the option of no normalisation) is to rescale values between the smallest and largest values in the image array, `x`, so that they lie in the range 0 to 1 in a floating point representation. Using this mapping is convenient, but loses you the intuition of how colourmaps work in a more mechanistic sense. In short, you need to imagine that the first row of the table corresponds to floating point 0, and the last row corresponds to floating

point 1. Intermediate rows of the colour lookup table will approximate to some floating point value. This is quite misleading, because the colourmap entries are always indexed (e.g. correspond to rows of a data structure) and this fact is being "hidden" in the default behaviour of `imshow` that relies on mapping both the image pixel values (when the image is a scalar) and the colourmap size (in terms of numbers of rows) into the same $[0, 1]$ floating point range.

Indeed, in many lower-level colourmap implementations (and particularly at hardware level), you will find that the colourmap has a finite number of slots (typically 256, or 512, or perhaps in the thousands).

[16]:
```
fig = plt.figure()
fig.text(0.45,0.1, 'Fig. 6')
ax = fig.add_subplot(1, 2, 1)
imgplot = plt.imshow(x,cmap='gray', norm=colors.NoNorm())
ax.set_title('Mapping using rows')
plt.colorbar(ticks=[0, 50, 100, 150, 200], orientation='horizontal')
plt.axis('off')
ax = fig.add_subplot(1, 2, 2)
imgplot = plt.imshow(x, cmap='gray')
ax.set_title('Default mapping behaviour')
plt.colorbar(ticks=[0, 50, 100, 150, 200], orientation='horizontal')
plt.axis('off')
```

[16]:
```
(-0.5, 127.5, 127.5, -0.5)
```




So, although the image on the right might look slightly brighter, because the largest and smallest

values in the image array, `x` are being mapped (this case, stretched) to the brightest and darkest values in the colourmap, this can sometimes cause unexpected behaviour when the colourmap is non-linear, or contains different bands of intensity and colour values.

Nevertheless, for most of the practicals we shall meet, the defualt behaviour will do fine. We will look at a case where the default behaviour of `imshow()` is *not* right, in **Section 4**!

**Exercise 1** Try changing the entries in the colourmap created above, and re-render the `head` image. ■

### 4. A More Complicated Colourmap

[17]:
```
from PIL import Image
imageObj = Image.open('IMGS/trees.tif')
cmapdata = imageObj.getpalette()
imdata = imageObj.getdata()
```

There are other many image reading libraries in *Python*; some of them (like the `opencv` library) are real dogs to install, and can lead to all sorts of incompatabilities. I strongly advise the use of the Python Image Library; PIL works well for many of the problems you are likely to meet.

Let us now look at the shape of these two items of data: the image array, `x`, and the colormap data, held in variable `cmap`:

[18]:
```
print('Shape: ',np.shape(imdata), 'of type', type(x))
```

```
Shape:  (90300,) of type <class 'numpy.ndarray'>
```

So, as in the first example, the imageObj contains a pointer to a location in memory that holds 90,300 items in the format of a `numpy` array. But these should probably be pixels, so let's ignore the type for the moment. But the key question is: what dimensions should the image be?

The `imageObj` is an instantiated class which contains a couple of attributes, which are `height` and `width` (you can see this if you do `dir(imageObj)`.

[19]:
```
print('Dimensions of image (WxH):', imageObj.width, 'x', imageObj.height,␣
↪'pixels')
```

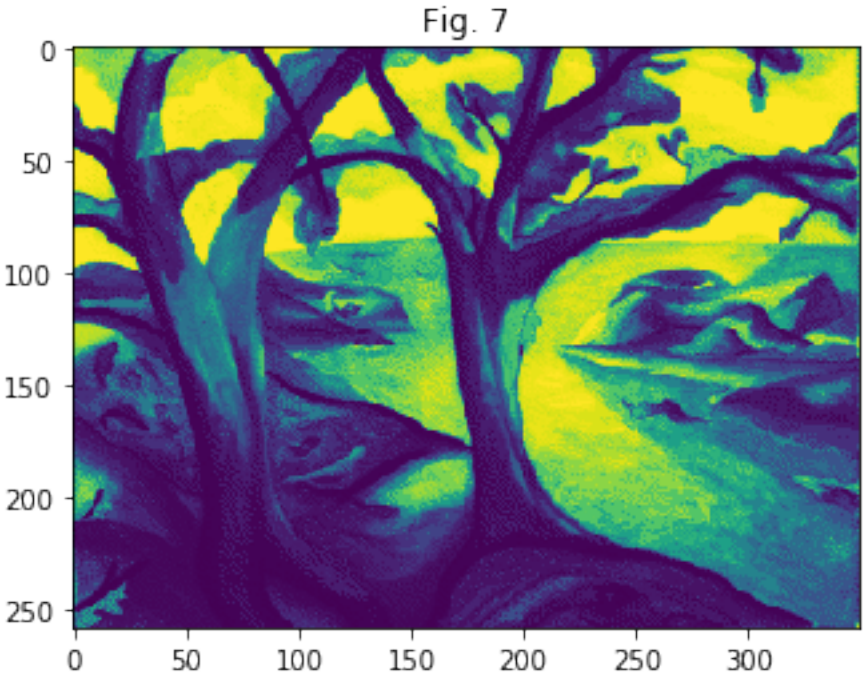
```
Dimensions of image (WxH): 350 x 258 pixels
```

…and we can confirm that $350 \times 258 = 90,300$, a useful sanity check!

So, if we do this:

[20]:
```
x = np.reshape(imdata, (imageObj.height, imageObj.width))
x = np.asarray(x, dtype=np.uint8)
plt.imshow(x)
plt.title('Fig. 7')
```

[20]:
```
Text(0.5, 1.0, 'Fig. 7')
```




Now, we need to work on the colourmap.

[21]:
```
np.shape(cmapdata)
```

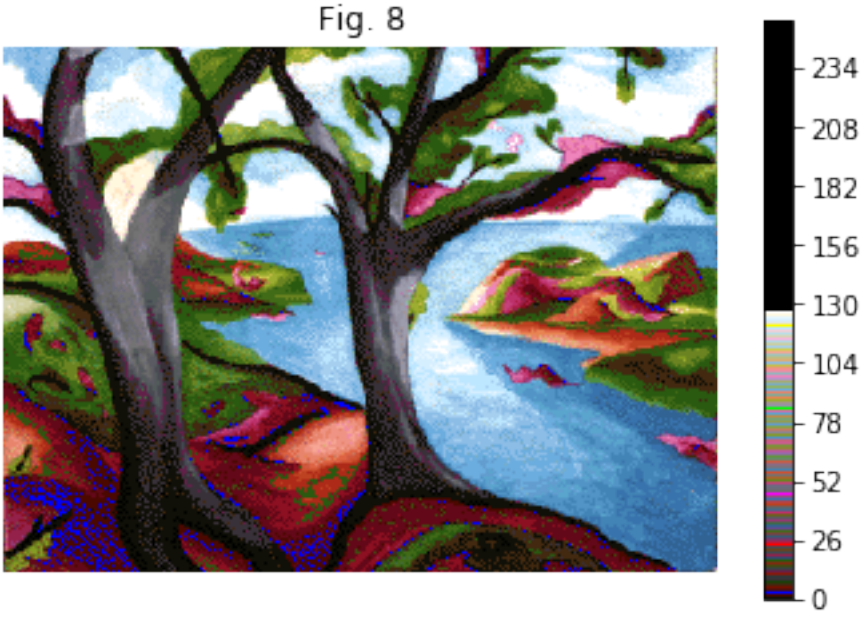
[21]:
```
(768,)
```

Ok, so we do not have something of the expected 3 or 4 column table. The number of values is divisible by both 3 and 4; however, we can see that if we divide 768 by 4 we get 192, and if we divide 768 by 3, we get 256. The latter is a quite common size for a colourmap, so let's assume and use that option.

[22]:
```
cmapNx3 = np.reshape(cmapdata,(256,3), order='C')
cmapNx3n = np.asarray(cmapNx3, dtype=float)/255.0
cmap = colors.ListedColormap(cmapNx3n, name='trees_colormap')
plt.imshow(x, cmap=cmap, norm=colors.NoNorm())
plt.colorbar()
plt.axis('Off')
plt.title('Fig. 8')
```

[22]:
```
Text(0.5, 1.0, 'Fig. 8')
```

Fig. 8




Well, I am afraid the image is not quite right.... But it is difficult to tell what is going wrong here, so you'll be forgiven for being puzzled! First, note that the colourmap entries appear to be rather disorganised. Now, look at the RGB components of this colourmap. Are you surprised? (You should be!). There is a very important lesson to be learned from this example – be very careful about how you interpret the values in the image matrix of a paletted image (or an image using a colourmap). It is not necessarily the case that the largest entries in the matrix correspond to the brightest intensities in the image. This is only the case for a monotonic, linear colourmap (usually, but not always, true for a grey-scale colourmap).

The allocation of colours (RGB components) to colourmap locations is sometimes done by an *ad-hoc* algorithm that tries to allocate colours as needed by an application, but whilst also trying to optimise available resources in the colourmap, operating system or device; it also depends on whether other applications have also requested certain colours to be available in the palette, so allocation of colours can be a sort of negotiation, particularly on resource-constrained systems. Always check the colourmap when working with palette images, where there is one value per pixel stored in the image matrix.

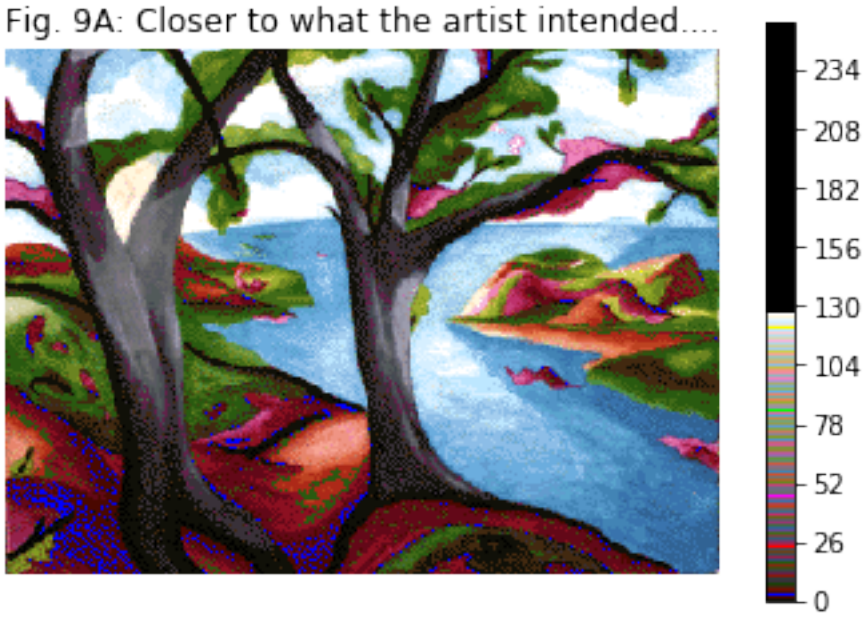
But the real issue with the display of this particular image has to do with the way that `imshow()` works within Jupyter notebooks. By default, the algorithm tries to scale the image up (in size) so that it occupies a reasonable portion of screen area. In doing so, it has to map pixels of the image onto pixels of your notebook (indeed, of the laptop or monitor screen). By definition, this usually requires some sort of spatial scaling operation: if neighbouring pixels in the image array are mapped exactly to neighbouring pixels on Jupyter notebook, there would be no issue. But by default, `imshow` maps to a sort of intermediate size that is neither too large nor too small for convenience. This requires it (in essence) to zoom into or out of the image. It's default way of doing this is to use *bilinear* interpolation. However, bilinear interpolation will fail dramtically if the colour map is not linear (so, if colours and intensities do not smoothly change from one row to

the other).

We can circumvent this by requesting that `imshow()` uses nearest-neighbour interpolation. To put this simply, if `imshow()` zooms up or down to display the image, it simply uses the nearest pixel from the original image after performing the coordinate transformation (i.e. zoom in or out), rather that trying to guess intermediate pixel values according to a linear assumption. So, turning off linear interpolation:

[23]:
```
plt.imshow(x, cmap=cmap, interpolation='nearest', norm=colors.NoNorm())
plt.axis('Off')
plt.title('Fig. 9A: Closer to what the artist intended....')
plt.colorbar()
```

[23]:
```
<matplotlib.colorbar.Colorbar at 0x1146096d0>
```

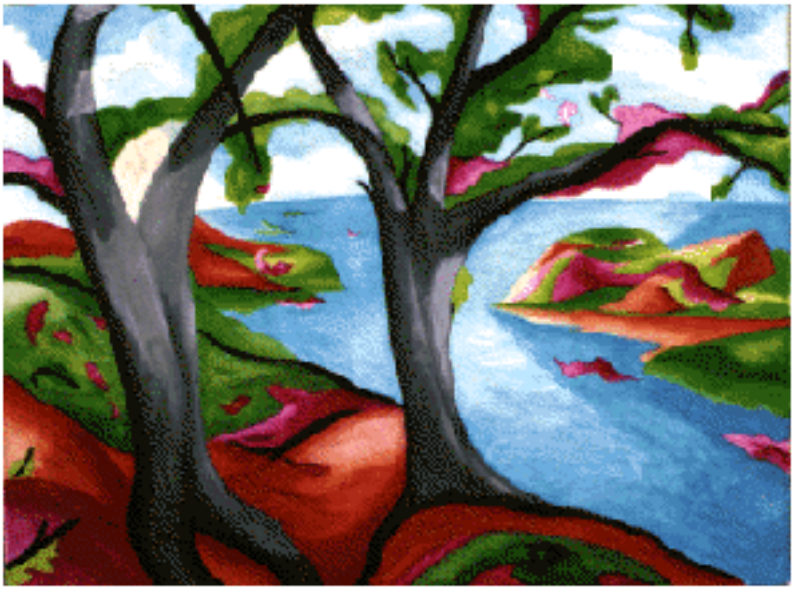



Now that we have wrangled our way to getting this result, and discovered the rather surprising nature of this colour palette, I can reveal (!) that *Python* does have methods that will correctly display this image, already built in. If we look at information held in `imageObj`, we will see that it contains quite a few pieces of information. These are used by `imshow` to correctly apply the intended colour map to the image:

[24]:
```
plt.imshow(imageObj)
plt.title('Fig 9B: Definitely what the artist intended....')
plt.axis('Off')
```

[24]:
```
(-0.5, 349.5, 257.5, -0.5)
```

Fig 9B: Definitely what the artist intended....



**Quiz** Can you see the differences between Figures A and B above? What do you believe is the source of these differences, and how might the image be fixed using the individual commands we have used above?

There is one more surprise in store for us in that `trees.tif` image. Try this:

```
imageObj.n_frames
```

What do you think this means? If you have time, explore, but this is **not** a priority to solve during this lab session....

**Take Home Message** An image array that contains a scalar value for each pixel is very common in image processing, medical imaging and computer vision. But such an image is uninterpretable unless you either i) know *for certain* that the image should be interpreted using a linear colour map, or ii) you have and make use of the colourmap supplied with the image by the software that encoded it.

### 5. True Colour Images

These are decidedly different from the paletted or colourmap images we have looked at so far. In this case, for each pixel, we store red green and blue component values. We will now look at such an image.

[25]:
```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
```

[26]:
```
imageObj = Image.open('IMGS/lily.tif')
cmapdata = imageObj.getpalette()
imdata = imageObj.getdata()
```

First, we are going to try to read the image ourselves from the raw pixel data. Remember that `imshow` is designed to read the relevant attributes in the object and display this for us, but this would 'hide' what is going on in this routine, so we'll do this the hard way first, to make sure you understand what goes on under the hood....

We start by getting the dimensions of the image (sometimes called the *geometry*):

[27]:
```
print('Dimensions of image (WxH):', imageObj.width, 'x', imageObj.height,␣
↪'pixels')
```

```
Dimensions of image (WxH): 230 x 186 pixels
```

Now, let's look at the shape of the `imdata`....

[28]:
```
np.shape(imdata)
```

[28]:
```
(42780, 3)
```

Hmmm. So, what is going on here? *Hint* See the little calculation we did in the previous section for the `trees.tif` file.

Once you have figured out where the `42780` value comes from, you can guess that the three channels (R,G,B) are being stored explicitly for each pixel. A little dive into the documentation of the `PIL Image` class here shows that you can specifiy the channels to be read individually, or all together. I will do this one channel at a time, then reshape each channel to match the geometry.

[29]:
```
W = imageObj.width
H = imageObj.height
imdata = np.zeros((H,W,3), dtype=np.uint8)
imdata[:,:,0] = np.reshape(imageObj.getdata(band=0), (H,W))
imdata[:,:,1] = np.reshape(imageObj.getdata(band=1), (H,W))
imdata[:,:,2] = np.reshape(imageObj.getdata(band=2), (H,W))
```

[30]:
```
plt.imshow(imdata)
plt.axis('Off')
plt.title('Fig. 11')
```

[30]:
```
Text(0.5, 1.0, 'Fig. 11')
```

Fig. 11

And just to confirm that we've spent this time doing nothing partiuclarly useful, we see what happens when we through the image object directly into `plt.imshow()`:

[31]:
```
plt.imshow(imageObj)
plt.axis('Off')
```

[31]:
```
(-0.5, 229.5, 185.5, -0.5)
```

As a final check that we grasp all of the key points about the difference between paletted and true colour images, we check the size of the colourmap associated with htis file:

[32]:
```
np.shape(cmapdata)
```

[32]:
```
()
```

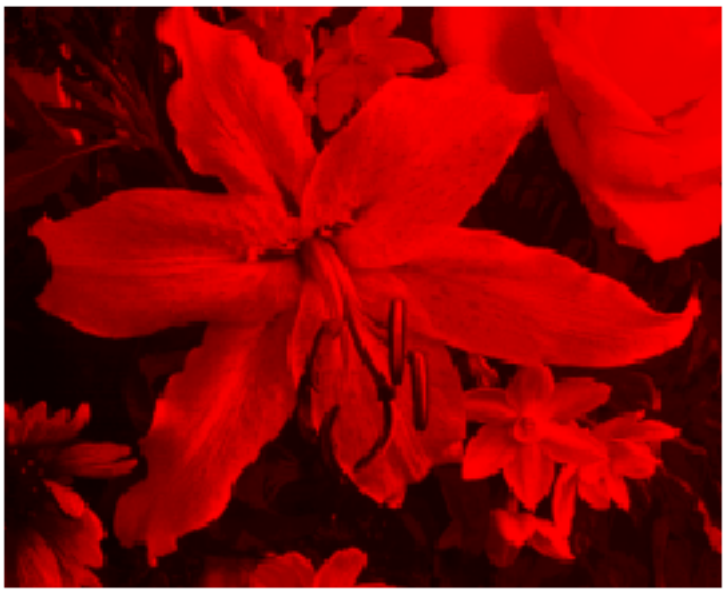
So, as you might hopefully have guessed, the colourmap is empty: because each of the three values is specified explcitly in the file for each pixel, a colourmap makes no sense in this context.

**4.1 Visualising the content of individual colour channels** Remember that the image contains each of the image planes corresponding to red, green and blue channels. We can look at these channels individually, or interpret them as grey-scale intensities. Let's do this both ways to understand what information this gives us. First: the slightly more intuitive thing: look at individual colour channels. We can do this in two ways: we can isolate individual channels, one at a time, and display them as scalar images using a colour map. Or, we can zero the channels other than the ones we are interested in. For example:

[33]:
```
ROnly = np.zeros((H,W,3), dtype=np.uint8)
ROnly[:,:,0] = imdata[:,:,0]
```

[34]:
```
plt.imshow(ROnly)
plt.axis('Off')
```

[34]:
```
(-0.5, 229.5, 185.5, -0.5)
```



**Exercise 2** Repeat the visualisation for the other two channels and note the results. In particular, you should pay attention to the appearance of the pink dots on the flowers (I think of them as the equivalent of "freckles", but that's probably just plain weird!).

What do you notice about the appearance of these dots on the *red* channel in particular. Is this surprising? ■

**Exercise 3** Repeat the visualisation of each of the three channels using a grey-scale colormap. This requires that you extract each of the three channels as a single scalar image field in turn, and use a grey-scale colourmap to display each of the three channels on three separate images. If you do not understand what this means, speak with a UTA/GTA. Make sure you understand this exercise, which is slightly non-intuitive, but serves a purpose in embedding the difference between paletted and true colour images, in addition to explicitly using a linear colour map to interpret/visualise the intensity of a colour channel.

Once you have done this, pay attention to the appearance of the pink dots on the flowers in the grey-scale visualisations of each of the three channels.■

**Exercise 4** In your lecture notes, there is a little section on trichromatic coefficients, or normalised RGB channels. Calculate these normalised (sometimes called *nRGB* values) at each pixel of the `lily` image. You will need to convert each of the three channels to floating point before performing the calculation; you can do that using this syntax:

```
f_imdata = np.asarray(imdata, dtype=float)
```

I also suggest that you create (using `np.zeros`) an array of the same size as your desired image output to hold the (floating point) results.

Once you have calculated the three trichromatic (nRGB) channels, visualise each of them separately using a grey-scale colourmap, and note **in particular** the behaviour of the normalised red and normalised green channels.

Finally, because you will be mapping a floating point image through a colourmap structure, you will want to remove the option that suppresses this behaviour (i.e. do **not** use `norm=colors.NoNorm()`). ■

### 5. DICOM Images

DICOM 3.0 is a standard for the encoding of medical image data, and also for network transmission of those images between scanners. The *Python* library **pydicom** has some routines for reading DICOM 3.0 files from disk, and this is what we'll be doing in this section.

First, follow the guidelines on the **pydicom** intallation page to install this package. Once you have got through this step, you can proceed. Please note that there is a conda instruction further down the installation page, which I recommend using.

First, look at the methods of the `pydicom` module:

[35]:
```
import pydicom as dcm
dir(dcm)
```

```
      ␣
↪---------------------------------------------------------------------------

       ModuleNotFoundError                       Traceback (most recent call␣
↪last)

       <ipython-input-35-e94a6be0ae85> in <module>
   ----> 1 import pydicom as dcm
         2 dir(dcm)


       ModuleNotFoundError: No module named 'pydicom'
```

We can see that there is a `dicomread` method; so let's apply it to the example DICOM 3.0 image `US-PAL-8-10x-echo.dcm` (this is from the *Matlab* (The Mathworks) distribution). Use the following syntax to from the `IMGS/` directory:

[36]:
```
# If the line below does not work, you may be in the wrong directory...
ds = dcm.dcmread('IMGS/US-PAL-8-10x-echo.dcm')
print(ds)
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-36-80481a8403d5> in <module>
         1 # If the line below does not work, you may be in the wrong directory…
   ----> 2 ds = dcm.dcmread('IMGS/US-PAL-8-10x-echo.dcm')
         3 print(ds)


       NameError: name 'dcm' is not defined
```

**Quiz**

1. How many rows do you think are in the colour palette?
2. Do you think that this is a grey-scale palette?

Let's try to get at the pixel data. We can see that there are 10 frames that have been stored in the file (`Number of Frames`).

Let us see what methods are available to us in the `ds` object. We can do that with this command:

[37]:
```
dir(ds)
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-37-d0c51d4da438> in <module>
----> 1 dir(ds)


NameError: name 'ds' is not defined
```

Having done that, we can investigate the types of some of the methods/attributes associated with this class.

For example:

[38]:
```
type(ds.PixelData)
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-38-a6125dd5aa51> in <module>
----> 1 type(ds.PixelData)


NameError: name 'ds' is not defined
```

…which looks like it is treated as a buffer (see the earlier exercise on the `head.128` image. And we can also check out the next promising looking name for the pixel information:

[39]:
```
type(ds.pixel_array)
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)
```

```
    <ipython-input-39-dd6c241ef3fa> in <module>
----> 1 type(ds.pixel_array)

NameError: name 'ds' is not defined
```

Ha! Let's see what the shape of this is:

[40]:
```
np.shape(ds.pixel_array)
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-40-b6619cadcb13> in <module>
----> 1 np.shape(ds.pixel_array)

NameError: name 'ds' is not defined
```

Even better. Note the match in the first dimension of this pixel array with the "Number of Frames" reported by `print(ds)`.

**Exercise 5** Start by extracting and the displaying the information in the pixel array using the default colourmap, without extracting the colourmap from the `ds` object (we will get to this later).

You may want to loop over the frames of the image and display each one in sequence; the command `plt.pause(1)` will place a one second delay after showing each image, so you can see each image (even though the colourmap will not yet be right, until you load it from the `ds` object and apply it). Unless you use such a pause, or force a display refresh in some way, this, the image that is shown will only be updated at the *end* of the loop, and so you will only see the last frame. ■

Of course, displaying the pixel values without the colourmap is somewhat nonsensical, but at least we can see some spatial structure in the images, which means we have at least got the image geometry correct. If we hadn't, we would just see squiggles or zig-zagging lines….

**Exercise 6** Extract the colourmap from the `ds` object. Here are two hints:

a) You will see that there are DICOM elements that correspond to the red, green and blue columns of a look up table; they are stored spearately, but each has the same number of elements. Although they are separate, this is still NOT an RGB image, it is paletted.

b) The values in these tables can be treated as buffers containing raw byte streams in memory, and the earlier example of using `np.buffers` is quite applicable to convert them into appropriate *Python* types (but be careful, as these intensities are specified using 2 bytes, rather than 1, as we have met so far).

*Hint*: You can follow (with some adaptation) the pattern we used for the `trees.tif` image in Section **??** ■

There is a slightly easier way to also produce a valid image using methods attached to the `pydicom` module; specifically, there are data handlers, one of which will directly map the image values through the colourmap to produce an RGB image, which can be displayed as usual using `plt.imshow()`. Because the image representation is in RGB space, interpolators will work fine, because the interpolation will be applied directly to the R, G and B colour spaces, rather than the possibly discontinuous paletted image representation.

You may investigate this yourself, but here is an example below:

```
rgbarr = dcm.pixel_data_handlers.apply_color_lut(im, ds)
rgbarr = np.asarray(arr, dtype=float)/
            float(maximum_channel_value)
plt.imshow(rgbarr)
```

**Appendix 1: Advanced Topics**

We can investigate the implmentation of some *Python* modules - those that have not been converted into C code, for example, using the `inspect` module;here is an example of using this to look at the `colors.Normalize()` method within **matplotlib**.

```
import inspect
print(inspect.getsource(colors.Normalize))
```

[ ]: