Department of Bioengineering

Image Processing Practical 1

Image Display

1.1 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 the image reading section, 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 image data, and may be outside the range of values in the *colour palette*. The processed images that we shall be considering in the second practical are those that arise after performing arithmetic or set-based operations on pixels, including those obtained from local spatial operators.

1.2 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. 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.

1. We will first load a "raw" data file image into *Matlab* as an image. The file was exported from an image display application which used file formats that are not supported outside of its environment. A magnetic resonance image of the 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 (16384). The commands in *Matlab* are quite similar to those needed to read the data using C programming, or pretty much any functional programming environment.

   Download the file "ImageDataFiles.zip" from the same place on the VLE 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. This path should be specified using the Windows Universal Naming Convention (UNC) (e.g. \\dataserver.domain.where.uk\LaboratoryData). If the zip file is on your current machine, you should not need the \\dataserver.domain.where.uk part of this, but you may need to specify a drive letter (e.g. "C:\"). Also, if working off the home directories, you may be able to access this by including "H:\" before the rest of the directory path. A test to see whether you have the path right is to use the in-built `dir` command in *Matlab.* For example, the command:

   ```
   >> dir('H:\')
   ```

   will show all directories and files in your Home directory, assuming it is mapped to drive-letter "H:". Applying this command to the string `'<PathContainingFiles>'` should allow you to see the names of the image files mentioned in the practical.

   To open the file in *Matlab*, use the following:

   ```
   >>fid=fopen('<PathContainingFiles>\head.128','r'); % Opens file for reading
   >>[x,npels]=fread(fid,[128,128],'uchar'); % Reads data values
                                             % into matrix x with 128 rows,
                                             % and 128 columns
   >>x=x';  % Matlab reads in arrays with a different index order [Ctd...]
            % to that of 'C' File was created using C, so transpose matrix
   >>fclose(fid); % Close the file handle
   ```

Use the `whos` command to examine the effect of the previous 4 commands at each stage of execution. The variable `npels` will contain the number of pixels in the image (16384). If it does not have this value, you will need to retype the previous sequence of commands. The conjugate transpose command on the 3rd line of the above set is simply used to get around a design feature (hangover from Fortran) in *Matlab*'s handling of matrices. Because the image data is real, the effect of the "conjugate" is irrelevant here; it's really just performing a matrix transpose. You may want to put the above lines into a script or function file.

In order to display the image, use the following command:

```
>> image(x);
```

Carefully note the appearance of the image. You may also relate the intensity of the image to the image values themselves (stored in the array x) by using the command

```
>> colorbar;
```

Having provided you with some basic familiarity with the appearance of the image, we will now look at the details of mapping images, as matrices of pixel values, to computer displays.

1.3 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 stored and 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 some high-end workstations, the three components are generally provided by three coaxial connectors connecting the workstation to its monitor. On PC's, the three signals are within the VGA connector. On digital connectors, the signals between the computer and the screen may be digital, so that the conversion to an analogue signal used to drive the pixels onscreen may 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 is performed by either a hardware structure or with a set of low-level software routines, which is called a colour look-up table or a hardware colour palette. Some stages of the mapping may also be done by very fast and complex software algorithms (such as those available in the X window system) which interact very closely with the available hardware.

A rough conceptual diagram of the mapping performed by a colourmap is shown below. Note that the LUT (colourmap) serves 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.

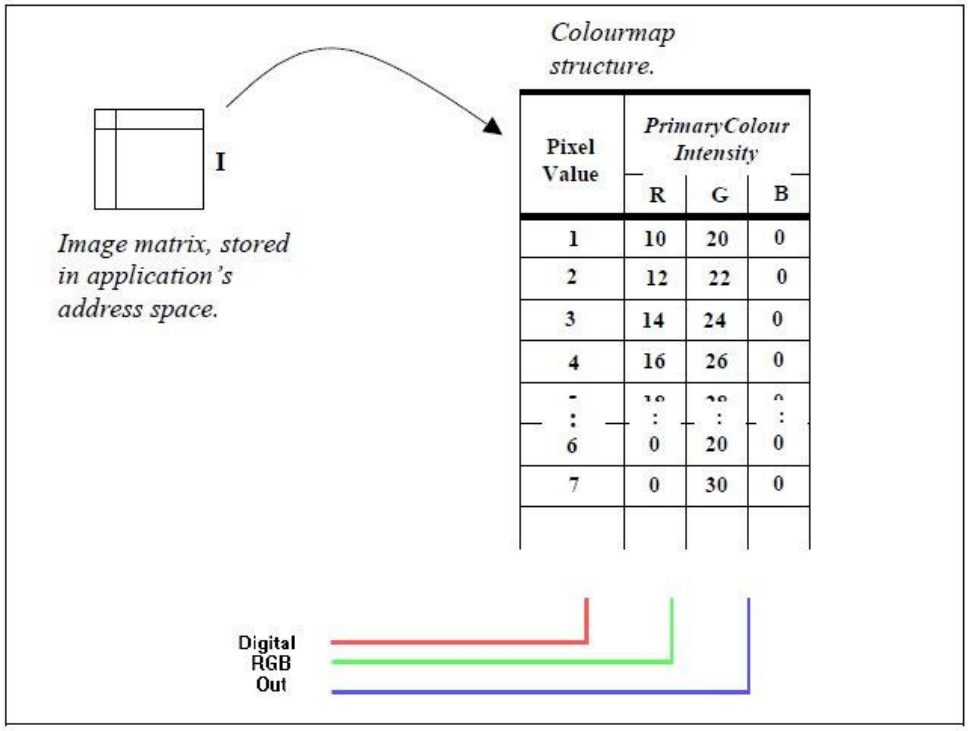

Figure 1 *Illustration of a Colourmap Structure*

Note that the values used in the R,G,B columns are shown as taking integer values. However, many colourmaps will have R,G,B values that range from 0 to 1.

Question: Given an image array *x*(*m*,*n*), containing values in the range 0 to 255, how should a palette be set up in order to display *x*(*m*,*n*) as a grey-scale image ? You should assume that the colour values themselves – the intensities of *R*,*G* and *B* values – range from 0 to 1.

Matlab has a number of fixed colourmaps which may be called up at will, including a 64 element grey scale. We shall start our experiments by using the default grey-scale colourmap, then altering it to suit our purposes. Redisplay the image using the following sequence of commands (you can turn this into a function):

```
>>image(x);colormap(gray(64));colorbar;
```

Note the appearance of the image, and note from the colour bar to the right of the displayed image the colourmap being used. The command of the form `colormap(<thecolormap>)` used above simply sets the active hardware palette on the computer to use the mapping as specified by the Nx3 array in the argument `<thecolormap>`. In order to confirm that the result returned by the command `gray` really is an *N*x3 array, use the command :

```
>> map = gray(64); % Assign the output of the colourmap command to
                   % a variable called "map"
```

then use the `whos` command to examine the dimensions of dummy. Note that `gray()` itself does not appear with the `whos` command because it is a function. What is *N* ?

1.4 Your Own Colourmap

A 64 level greyscale colourmap is inadequate for many medical images. It allows only 64 possible grey levels, and hence produces quantization error in the displayed image. Because the structure of a colourmap is simply an *N*x3 vector, we can create our own colourmap and fill it with the appropriate values.

1. Create a colourmap providing 256 possible shades of gray. You should define this colourmap as a *Matlab* variable. Call it `mymap`. What values need to be used in the Red, Green and Blue (first, second and third columns) of each colourmap entry (row) ?

2. Redisplay the image stored in array `x` and note the difference.

3. Rewrite your display function to take an argument which specifies the colourmap to be used in the image display.

   N.B. Since *Matlab* version 5.1, the built-in command `gray()` accepts an argument, which is the number of gray-levels that you wish to have. In everyday usage, you therefore do not need to go through the process of constructing the colourmap yourself for grey-scale images; however, for the purpose of meeting the learning objectives of this practical, you should generate your own colourmaps, as this process will help cement your understanding of how colourmaps work.

4. Try creating your own "snazzy" colourmaps.

1.5 Colourmaps for colour images

Matlab provides some test images in JPEG (Joint Photographic Experts Group) and TIFF (Tagged Image File Format). Try loading in the file `trees.tif' (again from the LaboratoryData directory) with the commands:

```
>> clear; close all; % Close all windows, and clear all variables
>>[X,map] = imread('trees.tif');
```

Using the `whos` command, note the effect of the second command. Now use the commands:

```
>> image(X); % Display the image…
>> colormap(map); % with its colormap (as read  from the TIFF file)
```

Display the colorbar for this colormap. What do you notice? Now, look at the RGB components of this colourmap. Are you surprised ? (You should be!). There is a very important lesson that you need to learn 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 often done by an ad-hoc algorithm that tries to allocate colours as needed by an application, but depending on 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.

1.6 True Colour (RGB) Images

Now, try loading the image of a couple of flowers:

```
>> clear; close all; % Close all windows, and clear all variables
>> [X,map] = imread('lily.tif');
```

Before displaying the image, use the `whos` command to examine the variables created with the `imread` command. What do you notice ? This type of image is described as containing three *image planes*, one for each of the red, blue and green components in the image. To visualise each of these three components, explicitly (and separately) you can do the following:

```
>> column_ramp = [0:1/255:1]';% Create column of values running from 0 to 1
% The next 3 lines create three colourmaps, intended for displaying the
% red green and blue components separately.
>> redmap = [column_ramp,zeros(256,1),zeros(256,1)];
>> greenmap = [zeros(256,1),column_ramp,zeros(256,1)];
>> bluemap = [zeros(256,1),zeros(256,1),column_ramp];

>> % Below, display the three image planes in 3 separate windows
>> figure(1); colormap(redmap);image(X(:,:,1)); colorbar;
>> figure(2); colormap(greenmap);image(X(:,:,2)); colorbar;
>> figure(3); colormap(bluemap);image(X(:,:,3)); colorbar
```

1. Examine the images carefully. Look at the pinkish-red spots on the petals of the lily. Can you see them easily on the red component image ? In order to confirm it is not an effect of the colourmap itself, display the *red* channel using a grey-scale colour map. Think carefully about the result, when compared against the correct display of the full colour RGB image. Can you suggest why the pink dots are not visible on the red channel ? Is this intuitive ?

2. Now, consult your lecture notes on image processing, and look for the section on trichromatic coefficients. Repeat the exercises above, <u>but instead of using the R, G and B components, use the trichromatic values</u>, being extremely careful with the calculations (use `./` and not `/` for matrix division) ; use the same scheme for colourmaps that you have used immediately above. For this purpose, you will, perhaps, find it useful to use the following pattern to extract the image components and turn them into floating-point arrays (suitable for arithmetic operations):

   ```
   >> R=double(X(:,:,1)); G=double(X(:,:,2)); B = double(X(:,:,3));
   ```

   Only after doing this conversion will you be able to perform the calculation needed to get the trichromatic values (also called normalised RGB values, or nRGB values).

   Also, you will need to ensure that the values in the trichromatic image array are rescaled to be valid colourmap indices before trying to display them. Think <u>very</u> carefully about what this means. If you are sloppy in thinking this through, you will not get the right results. Do NOT ignore this paragraph in bold – if in doubt, ask a GTA to explain what this means.

   Display **each** of the trichromatic coefficients with a grey-scale colour map. Compare the results of displaying the RGB fields as grey-scale images against each of the three corresponding trichromatic values. Comment on the results to a demonstrator, who will give you feedback.

   Important ! Please also note that the behaviour of the `image` command <u>depends on the dimensions of the argument passed to it; if you get this wrong, or simply don't check</u>, then you may get either nonsense results or a blank image:

- if the argument is a 2D array, the command expects to use a colourmap, and the values in the array are treated as indices into the colourmap: they MUST be integers or unsigned character values.
- if the array contains a third dimension, with 3 "slices" in the third dimension, the `image` command assumes that the third dimension contains values that correspond to R,G, and B values of RGB colour space (respectively), and requires no colourmap. In this case, the array should be either
  - floating point with values between and between 0 and 1.
  - integer, with values between 0 and 255

Also note that the command `imshow` behaves differently and is categorically not recommended for anything other than displaying images that have been captured with digital cameras. It is almost useless for displaying the results of calculations performed on images. Definitely **not recommended** for this course. Finally, there is a command called `imagesc`. Though we will use this later, this practical is designed so that you **don't** use it until you understand fully what it does!

1.7 DICOM 3.0 Images

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

1. Use the following command to obtain the header information from the file (you'll find it !)

   ```
   >> info = dicominfo('US-PAL-8-10x-echo.dcm');
   ```

   See if you can interpret some of the information in this structure. Next, use the following sequence of commands:

   ```
   >> [X, map] = dicomread('US-PAL-8-10x-echo.dcm');
   >> montage(X, map);
   ```

   The second command above creates what is known as an image montage, which is formed by displaying a sequence of images side by side. This is the "standard" way of creating a hard-copy of multi-image data, be it a short sequence or a volumetric data set.

   Use the `whos` command to display variable names and their sizes. Create a colourbar, and examine the colormap entries carefully. Are you surprised ? Discuss with a demonstrator to see if your understanding is correct.

2. In fact, the data that you have loaded above is movie data. To display this as a moving sequence, use the following commands:

   ```
   >> M=immovie(X,map);
   >> movie(M,20,10);
   ```

   Play around with the parameters of the movie command. Have some fun !

1.8 Extra "Stretch" challenge

See if you can figure out how to create a logical image array, the same size as the movie sequence, where the values in the array are 1 where the image values contain grey-scale values (including black) and 0 for the pixels that correspond to the colour flow mapping overlay.

Warning: This is a bit more difficult than it looks! How can you use this mask to identify and count the proportion of the ultrasound scan that contains moving blood ?