7060 Image Sensors & Processing

Tutorial 1: A bit of a MATLAB refresher

D.Gibbins

Location: CAT Suite IW. B15 (unless otherwise advised)

Summary

This tutorial is intended as a bit of a refresher for those students who do not have much experience of programming in MATLAB as well as looking at how MATLAB represents imagery (which you will need to understand for your upcoming assignments). Here we will look at arrays, image representations and finally construct two functions ***make_bayer.m*** and ***display_bayer.m*** which look at the Bayer colour patterns discussed in class.

You do not need to attend this tutorial so long as you try the exercises in your own time, but those of you with little or no MATLAB experience are strongly encouraged to attend. As you go through the tutorial please complete the exercises / or steps given in boxes.

This is all intended as a bit of practice before you get your first coding assignment in week 3.

Background Materials

For those student's unfamiliar with MATLAB it is highly recommended that they obtain and read through a printed copy of the "**EEESAU MATLAB Primer**" notes. These provide a good beginners guide to how to best use MATLAB for problem solving.

PREPARATION BEFORE ATTENDING / ATTEMPTING THE TUTORIAL:

Before starting this tutorial:

1. Please read through this document (and save yourself time in the tute!)

2. log in and using a web browser, go to the course site on MyUni and copy and unzip the file ***tutorial1_images.zip****. Alternatively, the files 'peppers.gif' and 'peppers.tif' referred to in this tutorial can be created from MATLAB's inbuilt 'peppers.png' files by copying (or cutting as pasting from the PDF file) the following commands:*

```
X=imread('peppers.png');
[Xtmp,Ctmp]=rgb2ind(X,255);
imwrite(Xtmp,Ctmp,'peppers.gif');
imwrite(X(:,:,2),'peppers.tif');
```

(please note the quote (') character used in MATLAB strings is the one found on the [" '] key on your keyboard)

PART A - A reminder of MATLAB Arrays, expressions & Indexing

NOTE: *SKIP THIS SECTION IF YOU ALREADY UNDERSTAND HOW MATLAB REFERENCES ARRAYS AND THE SPECIAL ':' AND 'end' SYNTAXES*

As with other languages MATLAB can assign values to variable names such as A=3; or S='hello';. In addition, MATLAB natively understands vectors, matrices and higher dimensional arrays. For example the expressions B=[1 2 3]; or C = [1 2 3 ; 4 5 6 ; 7 8 9]; create two variables in the workspace called B and C with are a 3 element row vector and a 3x3 array (the ";" here indicates the end of a row in the matrix). Similar to other languages individual values within the array can be accessed via bracketed expressions such as A(1) or B(2,1). If required, the function size() can be used at any time to determine the size of the variable (eg. size(B) → [1 3] , size(C) → [3 3]).

Importantly MATLAB was designed to readily understand 1 or 2 dimensional arrays so that the mathematical operations + - * and / can be used directly on arrays. For example, the expression B+B would add the vector B to itself, and C*C is the result of multiplying matrix C with itself. Other expression symbols such as ' .* ./ and \ are defined for operations such as 'transpose' 'element multiplication' 'element division' and 'linear solution'. For example, A \ (B') means take the transpose of B and find the solution to the linear expression Ax = B. Higher dimensional arrays such as 3 or 4 dimensions can also be readily created in MATLAB however some of these mathematical expressions are not fully defined (eg. the transpose of a 3D array is undefined).

For those of you who have programmed in languages such as C, C++, Java etc this ability to directly perform matrix operations can greatly simplify your code. For example, the code fragment:

```
for r=1:size(C,1)
        for c=1:size(C,2)
                C(r,c)=C(r,c)+5;
        end
end
```

which, in MATLAB, adds 5 to the array C is equivalent to the very simple command

```
C=C+5;
```

A number of shortcuts for indexing arrays also exist in MATLAB which use the colon (:) symbol instead of a specific index value. A colon : by itself is used to indicate all the elements of a given dimension of an array, whilst an expression like 1:3 means the entries of that array between 1 and 3 (ie. 1 2 and 3). Adding a third value to the colon expression can be used to skip entries in an array for example the expression 1:2:10 means the values 1 3 5 7 and 9.

So for example (if variables A,B and C are as shown earlier):

C(:,:) → [1 2 3 ; 4 5 6 ; 7 8 9]

C(:,1) → [1 ; 4 ; 7] ← ie first column

C(:,1:2) → [1 2 ; 4 5 ; 7 8] ← ie first two columns

C(A,:) → [7 8 9]

C(2,B) → [4 5 6]

Note that MATLAB uses (row,column) ordering of the array indexes. This can sometimes cause confusion when thinking of arrays in terms of (x,y) coordinates as we will to later on in the course.

In addition, the key word 'end' can be used in an array index to refer to the last element in that dimension, for example:

C(2:end,:) → [4 5 6 ; 7 8 9]

C(end,end) → 9

One special feature of MATLAB arrays is that they can also be accessed as if they were a simple one-dimensional vector of values. This treats the array as a column vector with each column stacked under one another. For example:

C(2) → 4 ← ie 2nd element in first column

C(:) → [1 ; 4 ; 7 ; 2 ; 5 ; 8 ; 3 ; 6 ; 9] ← this will print out as a column of numbers

This 'feature' can be useful for example in calculating values for 2 or 3 dimensional arrays such as the sum or mean (we will use this trick in one of the assignments later on in the course).

Exercise: *Assign the variables A, B and C as shown above and try out the array indexing for yourself. Once you understand how indexing works continue to the next section.*

Some useful MATLAB array functions:

- **size()** - size of an array (eg. size([1 2 3; 4 5 6]) → [2 3])
- **numel()** - number of elements in array (eg. numel([1 2 3; 4 5 6]) → 6)
- **zeros()**, **ones()** - create an array of zeros or ones (eg. zeros(2,3) → [0 0 0 ; 0 0 0])
- **inv()** - inverse of given square matrix.
- **sum()** - sum columns of matrix (note: sum(X,n) will sum along the n-th dimension of x eg. sum([1 2 3; 4 5 6],2) → [6 ; 15])
- **mean(), median(),max(),min()** - mean, median, max or min operations. Syntax is same as for sum() above.

PART B - Images in MATLAB

How MATLAB Represents Images

Images in MATLAB are typically represented as NxM or NxMx3 arrays of numbers (where N and M is the number of rows and columns of the image respectively). An NxM array of numbers typically contain monochrome data (or data with as associated colormap - see later), whilst by default an NxMx3 array is assumed to contain red, green and blue colour information (if say the variable **X** contained an NxMx3 image then**, X(:,:,1), X(:,:,2)** and **X(:,:,3)** would be the **red, green** and **blue** arrays of 2D data).

The type of the array in MATLAB is also important.

Typically, MATLAB uses arrays of real numbers (doubles). By convention values between 0.0 and 1.0 represent the minimum (black) and maximum value of the data although this is not an enforced rule. NOTE: Values outside this range may be clipped by some MATLAB operations such as the image display functions (they may not be displayed properly, or cause an "out of range" error).

Alternatively, images are represented by arrays of unsigned 8-bit integers where 0 and 255 represent the minimum (black) and maximum (white) values. Images read in from file are usually in this format.

Care must be taken when using MATLAB as some of the image processing, display and related functions frequently assume the image data is in a given format (eg. double range 0.0..1.0 or uint8 0..255).

How to load and display Images

The simplest mechanism for loading images into MATLAB is the **imread** command. This MATLAB function is given a string (single forward quotes ‘) which is the name of the file to read in and returns an array of image data (a second parameter may also be returned, this is the colourmap of the image data – if it has one).

For example:

```
X = imread(‘fred.jpg‘);
```

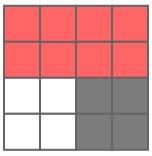
or

```
[X,cmap] = imread(‘fred.gif‘);
```

Would attempt to read in the contents of a file called ‘fred.jpg’ (the jpg denotes the file format, in this case JPEG) and assigns the result to variable **X.** The second form of the command is for some image types where the images are represented by an array of colour indexes (X) and a colourmap (cmap) (refer lecture 2b and see brief explanation below).

> ***A QUICK EXPLANATION OF IMAGE COLORMAPS****: We will cover colormaps properly in the first half of the week 2 lectures. However, in short, colormaps are a method of representing an image composed of a small set of colours. Here the colour image is defined in terms of a colormap which is a list of defined colours with which to represent the image data, and an index array which defines the colour of each pixel in terms of an index into this list of colours. For example if colormap for a given image is [1 0 0; 0 0 0; 1 1 1] which defines the three colours red, black and white,*

and the index array [0 0 0 0 ; 0 0 0 0; 2 2 1 1; 2 2 1 1] this defines a simple 4x4 image comprising the following a red rectangle and a black and white square:



Three main functions exist in MATLAB for displaying image data. These are **image, imagesc** and **imshow**. Each of these functions takes an array if image data and attempts to display it on the screen in the current figure window. If no figure exists, the window will be created.

Just to make things confusing each of these 3 functions differs in the way they display the image. The function **image** renders the image data based on the size and type of the array (ie. double or uint8) and fits the image to the size of the figure window. Note that this function assumes the values are in the ranges 0..1 (double arrays) or 0..255 (uint8 arrays).

The **imagesc** command displays the image in a similar way. However the range of values inside the image array is rescaled so that the maximum and minimum are within 0..1. In this way image data that does not have the assumed range of values can be displayed.

Finally, **imshow** displays the image data, but in this case it attempts to resize the image window so that it is the same size as the input image. As with **image** it does not rescale the range of colour values so clipping of the monochrome or colour info can, and will, occur.

Thus to display the array **X** as an image one might use the commands:

```
image(X);
```

or

```
imagesc(X);
```

or

```
imshow(X);
```

One final point to note is how MATLAB displays monochrome image data. Unlike what you might expect, a monochrome image is not displayed by default in monochrome (ie shades of grey). Instead MATLAB uses the current colourmap for the figure window (usually a rather garish range colours from black thru red, to green yellow and so on). To render monochrome images correctly you need to assign a new colourmap to the figure window using the **colormap** command. For a true 'grayscale' image use the command:

```
colormap(gray);
```

after the **image, imagesc** commands (imshow should to this automatically). This instructs matlab to display the image using a colormap comprised of around 64 shades of Grey). If required a colormap of 256 shades of grey can be obtained by using ' colormap(gray(256));'.

(note the American spelling of 'color' and 'gray' as used in MATLAB)

***A QUICK COMMENT ON FIGURE COLORMAPS**: Just to make things a little more confusing the image() and imagesc() functions handle the colormap data a little differently. imagesc() will always map the min and max image values to the first and last colormap entries, whilst image() will map 0.0 and 1.0 to the first and last*

entries if the image data is type double or to values 0 and N-1 for type uint8 (where N is the number of entries in the colormap). This can cause all sorts of problems when displaying 8-bit data as matlab by default uses colormaps with 64 entries. Personally, I tend to use either imshow() or imagesc() and avoid using image() unless I have to.

If the image data is, in fact, represented by an array of colour indexes and a colourmap then you should use this colour map in place of the 'gray' instruction to display the image with the correct colors. For example:

```
colormap(cmap);
```

where **cmap** is the variable containing the images colour information (recall the *[X,cmap]=imread('fred.gif');* command described previously).

Exercise: *Using this information as a guide, read in the three images 'peppers.png', 'peppers.gif' and 'peppers.tif' into variables* ***I1, I2*** *and* ***I3*** *and display their contents. The three images are shown as they should appear when correctly displayed on the next page.*

Examine the data structures created by the ***imread*** *command for each image type. What is the difference between each of these representations of the image data? In order to display 'peppers.gif' correctly was anything else needed other than* ***I2*** *?*

Figure 1: The images peppers.png, peppers.gif and peppers.tif. The PNG and GIF versions are colour and should look very similar to one another.

With ***I3*** *(ie 'peppers.tif') try using different colormaps such as* ***hot*** *and* ***cool*** *(a full list of inbuilt colormaps can be found using the 'help graph3d' command).*

One other useful feature in MATLAB is the magnifying glass on the toolbar of each window. This can be used to scrutinise part of the displayed image. Take a close look at the 'peppers.gif' image. What do you notice about the way the image data is represented (as compared to 'peppers.png') ?

Try reading in and displaying any of the other supplied example images. Several of these images will be used in assignment 1.

PART C - Creating MATLAB scripts & functions

Whilst you can enter a long list of commands at the MATLAB prompt, it is often useful to place a list of instructions in a text file which can then be executed from the command line.

There are two basic forms. In the first case the text file (eg say called 'example.m' where the 'm denotes that it's a matlab script or function file) simply contains a list of commands as they might be typed in at the console. For example:

```
I = imread('peppers.png');
figure(99);
imagesc(I);
```

which will load a pre-existing matlab image called 'peppers.png' and then display it on the screen. This form is known as a script file. Here the *figure* command creates a new figure window (numbered 99) for the image to be displayed in.

Exercise: *In MATLAB, open the editor (eg. type 'edit' at the matlab prompt). Open a new m-file and type in the above 3 lines of text (or type 'edit load_image.m'). Once you have done this save the file under the name 'load_image.m'.* *(continued overleaf)*

Exercise (cont'd)

In MATLAB run this script by typing in the command load_image at the command prompt. This should run the above commands and the peppers picture should appear on the screen.

Okay, now let us change this script to a function with one parameter which can be run from the command line. Go into the MATLAB editor and change the script file to the following (remember you cut and paste this text from the PDF document):

```
function I = load_image(image_filename)
I = imread(image_filename);
figure(99);
imagesc(I);
return
```

Save this file and run the following command at the prompt:

```
img = load_image('peppers.png');
```

this should produce the same result as before, except that we can change the supplied filename to another image name and this function will load and display the given image and assign the image data to the variable 'img'. Note that the variables 'I' and 'image_filename' are only defined within the function whilst it is being evaluated.

A more involved example (make_bayer.m)

In the course notes, we have looked at how a CCD camera typically captures a colour image as a 2x2 repeating Bayer pattern of red, green and blue values. Here we will write a MATLAB function which given a 3D array (MxNx3) will produce an NxM array of values mimicking a Bayer pattern sensor.

Action: *Open the matlab editor, create a new file and enter the following text (remember you cut and paste this text from the PDF document to save time):*

```
function Ibayer = make_bayer(I)
if (nargin<1)
   error('this function requires an image as input');
end
Ibayer = zeros(size(I,1),size(I,2));
Ibayer(1:2:end,1:2:end) = I(1:2:end,1:2:end,1);
Ibayer(2:2:end,2:2:end) =  I(2:2:end,2:2:end,3);
Ibayer(1:2:end,2:2:end) = I(1:2:end,2:2:end,2);
Ibayer(2:2:end,1:2:end) = I(2:2:end,1:2:end,2);
return
```

check your typing and save the file under the name 'make_bayer.m'.

Assuming the variable 'img' used earlier still holds the pepper image, the command

```
img2 = make_bayer(img);
```

should create a variable img2 containing a Bayer pattern representation of the original 'peppers' image.

In this function the variable *nargin* is an inbuilt MATLAB variable which tells you how many parameters were actually passed to the function. So for example, running the function ***make_bayer*** without a parameter should produce the error message "this function requires an image as input". The *size* and *zeros* functions in above the code are used to create a new array of values (initially 0) of the required size.

Action: *Check this by running the above command and using the image or imagesc commands to display the result. Do you understand what each line of the function does? (if not put up your hand and ask) Once you have this working move on to the next steps.*

Note, as highlighted earlier if you are uncomfortable with using '**:**' indexing, all of the above expressions can also be constructed using MATLAB '**for**' loops for example:

```
for r=1:2:size(I,1)
        for c=1:2:size(I,2)
                Ibayer(r,c) = I(r,c,1);
        end
end
```

is equivalent to the line:

```
Ibayer(1:2:end,1:2:end) = I(1:2:end,1:2:end,1);
```

in the 'make_bayer.m' function (or any other code you write). If you are more comfortable with using for loops than ':' then by all means use them in your code. The main downside of such loops is that MATLAB can take much, much longer to compute the desired results.

And finally, a little coding exercise for you…

As a final exercise, use what you have learned in the above examples to create a new function to be called **display_bayer.m**. This function is given a Bayer pattern image as created by **make_bayer.m**, and attempts to display the Bayer pattern in colour. That is the red channel is displayed as red, the green channels as green and so on.

To achieve this, you will need to do the following:

1. Create an array of size NxMx3 where NxM is the size of the bayer image (the commands '*zeros*' and '*size*' can be used to do this).
2. Copy and modify the instructions used in **make_bayer.m** so that they copy the values back into the appropriate channels of the NxMx3 array. You may use either '*for*' loops or the ':' indexing shown earlier.
3. Display the result using either the commands ***image*** or ***imagesc***. Note that you will probably have to divide the result by 255 in order for MATLAB to display it correctly (recall that MATLAB assumes an array of doubles representing an image contains values on the range 0..1 whilst the original image is uint8).

Assuming all goes well, running the commands:

```
I=imread('peppers.png');
I2=make_bayer(I);
display_bayer(I2);
```

should display the original image and a colour checkerboard image of the peppers image in a form similar to that given in lecture 1.

Exercise: *Given the above hints, attempt to write the function '**display_bayer.m**' and use the image 'peppers' or another colour image to test your function. If you run out of time during the tutorial please try to complete this in your own time*

NOTE: *Please ask for assistance from the tutor if you are having difficulty getting this function to work.*

Additional Exercise (Time Permitting)

Exercise: *Extend the function load_image so that it can also read in GIF images (such as peppers.gif) and returns an NxMx3 array of RGB values.*

Hint: *The second return parameter to the matlab **imread** function (eg. [I,C]=imread('peppers.gif') is not empty (ie isempty(C) is false) if the image is represented as a colourmap and an array of colourmap indexes. In this case the function ind2rgb can be used to convert the image data to a full colour NxMx3 array.*

END OF THE TUTORIAL

Some Useful Commands (* denotes function is in the image processing toolbox):

- **imread / imwrite** – read an image or write to a file (eg. I=imread('fred.tif'); imwrite(I,'fred_copy.tif');)
- **image, imagesc, imshow*** – image display commands (eg. imshow(X);)
- **colourmap** – se tthe colourmap used by a figure (eg. colourmap(hot))
- **caxis** – set the range of colorvalues displayed (eg. caxis([0 255]))
- **colorbar** – add a colorbar to the side of a display plot
- **double / uint8** – convert array to given type (eg. Y=double(X);)
- **ind2gray*** – convert an color indexed image to grayscale (eg. Y=ind2gray(X,cmap);)
- **ind2rgb*** – convert a color indexed image to full rgb color (eg. Y=ind2rgb(X,cmap))
- **rgb2ind*** – convert full colour image to an indexed image with a limited number of colours (eg. Xind = rgb2ind(X,128);)
- **rgb2hsv* / hsv2rgb*** – convert RGB image to/from HSV colorspace
- **rgb2gray*** – convert color image to grayscale
- **histeq*** – histogram equalisation (Y=histeq(X);)
- **medfilt2*** – 2D median filter (Y=medfilt2(X,[1 1 1; 1 1 1; 1 1 1]);
- **conv2** – 2D convolution filter (see also imfilter, eg. Y=conv2(X,[1 1; 1 1]);)
- **fspecial** – function for constructing several common filters (op=fspecial 'disk',2);)
- **imnoise*** – add noise to a given image (eg. Inoisy=imnoise(I, 'gaussian');)
- **sum, mean, max, min, std** – the sum, mean, max, min or standard deviation of a row of data.
- **sin, cos, tan, log, sqrt** …etc.. – All of the usual mathematical expressions handle matices.
- **size, isempty –** get the size of an array or test to see if a given variable is empty.
- **ones, zeros –** create an array of 1's or 0's. (eg. X = zeros(5,5) creates a 5x5 array of 0's)