Advanced Topics in Computer Vision and Deep Learning (EEEM071)

Week 1 - Colour Image Manipulation in MATLAB

Dr Xiatian Zhu

Introduction

In this lab you will use learn how to use Matlab to do basic image processing and explore different colour representations for images.

If you are unfamiliar with Matlab you should first work through the optional Introduction to Matlab worksheet. This is available on SurreyLearn under the ‘Getting Started’ section for this module. If you took module EEE3032 (computer vision and pattern recognition) then you may have already used this resource to get up to speed on Matlab. In which case you are sufficiently covered.

Ex1. Loading an Image

Locate the image of the Surrey Stag ‘surrey.png’ available on SurreyLearn in the lab sheet section. Alternatively you can download an image from the Internet for this lab sheet.

```
>> img = imread(‘surrey.png’);
```

Now type:

```
>> imshow (img);
```

This will display the colour image ‘surrey.png’. If you would like to the coordinates of the pixels in the image, then type:

```
>> axis on
```

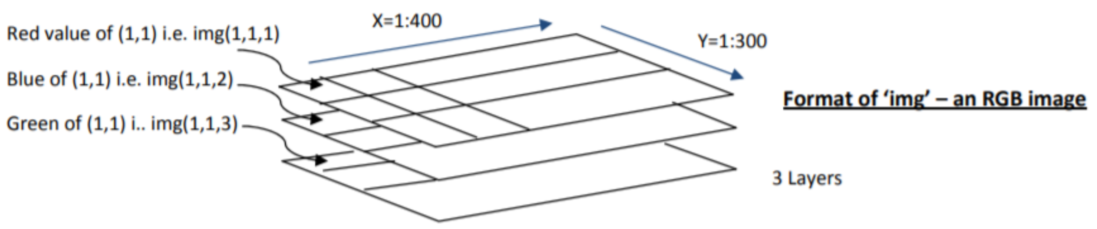
This will add axis ticks indicating the pixel coordinates. Remember Matlab addresses matrices from one, not zero, so the coordinates of the top-left pixel are (1,1) and the bottom-right (400,300).

Ex2. Colour image representation

Variable ‘img’ contains the image, which is 400 pixels wide, by 300 pixels high. Run this command to check:

```
>> size(img)
```

Matlab expresses the ‘y’ coordinate (rows) first, then the ‘x’ coordinate (columns). In this case you will see that the ‘img’ variable is a 3 dimensional matrix. You can think of this as three 300x400 matrices stacked on top of one another. This is because the image is colour (RGB). The 3 layers in the stack store the red, green, and blue values.


Red value of (1,1) i.e. img(1,1,1)
Blue of (1,1) i.e. img(1,1,2)
Green of (1,1) i.. img(1,1,3)
X=1:400
Y=1:300
Format of 'img' – an RGB image
3 Layers

Suppose we wanted to read the RGB colour of pixel coordinates (2,3), into a variable p. The following works:

```
>> p = img(3,2,:)
```

Here we have swapped 3 and 2 because Matlab requires the y coordinate first, then the x. The use of ':' for the third coordinate is shorthand for "all of this dimension". So we are extracting a "vertical" slice through the 3D matrix. To extract the red value of pixel (1,2) we would use

```
>> p = img(2,1,1)
```

And so on. The values in an image range from 0-255, in order words 8 bits are used for each colour channel. Thinking back to the first lecture, what kind of frame buffer is this? How many colours could such an image represent?

Ex3. Working with ranges of pixels

We address more than one pixel at once, using the colon operator. Suppose we wanted to set the R, G and B values to 255 for a range of pixels – i.e. set many pixels to be 'white'. The range we are interested in is the rectangle with top-left coordinates (100,200) and bottom left coordinates (150,300). The command would be:

```
>> img(200:300,100:150,:)=255;
```

Check it with:

```
>> imshow (img);
```

We can use the same syntax to 'crop' part of the image within a rectangle. Reload the image (see Ex.1) and then try:

```
>> subimg = img(200:300,100:150,:);

>> imshow (subimg);
```

Ex4. Saving an image

Try saving your cropped image using the following command. Load the result in a viewer e.g. eog ~/out.jpg to check

```
>> imwrite (subimg,'~/out.jpg');
```

You will likely be familiar with '.jpg', so called "JPEG files". JPEG is a preferred image format for many applications (e.g. digital photography) because of its ability to compress large images to small file sizes. However JPEG compression is “lossy”; some information is thrown away when you compress. This is can be problematic for Computer Vision work. Suppose you had an intermediate image processing result you wanted to save, and then load later to continue working. Due to lossy compression, the file you load will not be exactly the same as the file you saved. Some of the pixel values will change slightly. For this reason it is better to use an uncompressed image format such as Windows Bitmap (.bmp), or a compressed format that does not use lossy compression (such as TIFF - .tif, or PNG - .png).

Ex5. Normalised images

As seen in Ex.3, the ‘imread’ function will load an image into a matrix where pixels values are in range [0,255].

It is more convenient to normalise the pixel values from 0-255 to the range 0-1. The following command will normalise an image in this way:

```
>> normimg = double(img)./255;
```

The advantage of this form is that the various maths operations we will perform on images are much easier.

```
>> imshow(normimg);
```

Get into the habit of normalising your images when loading them.

You may wonder what the function ‘double’ does, above. When MATLAB loaded up the image, it was stored in matrix img in the uint8 data type. This means the values of each element in the matrix could not adopt values outside the range [0,255]. By changing the data type to floating point (‘double’) we avoid this limitation which is helpful later when performing image processing.

Ex6. Greyscale conversion

It is finally time to do some image processing! Our first task will be to convert the Surrey Stag image into a greyscale image.

Recall that RGB images are effectively a 3D matrix; three matrices “stacked” into 3 layers. Greyscale images have 1 layer rather than 3; i.e. they are simply a 2D matrix.

The value for each pixel represents the intensity (0=black, 255=white – although here we are going to deal with normalised images so 0=black and 1=white).

Recall from Lecture 1, the equation to convert colour (RGB) images into greyscale:

*Intensity = 0.30 * red + 0.59 * green + 0.11 * blue*

We can create the greyscale image by multiplying the red, green and blue channels (layers) of the image by these proportions, and summing them to create a new image:

```
>> greyimg = normimg(:,:,1)*0.30 + normimg(:,:,2)*0.59 +
normimg(:,:,3)*0.11;
```

Ex7. Conversion of RGB to HSV and back again

Matlab has built in functions to convert images between colour spaces. By default images are loaded in the RGB colour space.

Our next task will be to convert the Surrey Stag image from an RGB image into a HSV image.

```
>> hsvimg=rgb2hsv(normimg);
```

The size of the image is exactly the same; it is a 3 channel image, but now the channels mean different things. The first is the Hue, the second the Saturation, the third the Value. Check the size via:

```
>> size(hsvimg)
```

Now let's manipulate the colours of the image. If we set the Hue i.e. first channel of the image to all zeros, it will take on a red appearance.

```
>> hsvimg(:,:,1) = 0
```

To view the image we will convert it back into RGB space and use the imshow command as before

```
>> rgbimg = hsv2rgb(hsvimg);

>> imshow (rgbimg)
```

Note that the silver/grey/white/black parts of the Surrey Stag remain the same but the foliage in the background has experienced colour shift. That is because regardless of the Hue, very small or very large V values cause black or white colours in RGB so they have not changed.

Ex8. Interpolation in different colour spaces

In this final exercise we will explore the effects of colour space on interpolation. Interpolation is the process of calculating a value in between two values.

If I ask you to give me a value halfway between 3 and 4 you would likely say 3.5. Or if I ask for a number one quarter of the way between 3 and 4 you would likely say 3.25. You have performed an interpolation between a pair of numbers, specifically a linear interpolation.

We will now perform a linear interpolation on two colours, which are 3 dimensional vectors in either RGB or HSV space.

Let's start by making two images, one red and one green. We will consider these images to be in the RGB space.

```
>> black = zeros (100,100,3);

>> red = black;

>> red (:,:,1)=1;

>> green = black;
>> green (:,:,2) = 1;
```

Notice how for the red image, all the pixels in the first channel are set to maximum (1) and in the green image the same has been done to the second channel. (Recall the diagram at the top of the lab sheet if this is unclear).

Satisfy yourself that you have created a red and a green image using the imshow command e.g.

```
>> imshow (red);
```

```
>> imshow (green);
```

Now let's linearly interpolate (mix) those images in equal parts – creating an image that has pixel values half way between red and green.

```
>> rgbinterp = (red + green ) /2;
```

Use imshow to see the result

```
>> imshow(rgbinterp)
```

Have a look at one of the pixel values e.g. the pixel at (1,1) – all the pixels have the same value

```
>> rgbinterp(1,1,:)
```

You should confirm that the pixel has red channel of 0.5, green channel of 0.5 and blue channel of 0 and looks murky green.

Now let's try the same interpolation of the images, but in HSV instead of RGB space. First convert the two images to HSV space.

```
>> redhsv=rgb2hsv(red);
```

```
>> greenhsv=rgb2hsv(green);
```

```
>> hsvinterp = (redhsv + greenhsv ) /2;
```

Let's convert the resulting image in HSV space back to RGB space and look at the image

```
>> out = hsv2rgb(hsvinterp);
```

```
>> imshow(out)
```

You should see a bright yellow image

Now take a look at a pixel in the output image e.g. at location (1,1). You should see that the pixel value is different – it has red channel of 1.0, green channel of 1.0 and blue channel of 0.

The takeaway message is that the colour space you are working in impacts the outcomes of operations you perform on the image e.g. interpolation and mixing of colours in the images.

Learning Summary

You have now learned to load and save images in MATLAB and how to address pixels in those images in order to read and write to a pixel or range of pixels. You have learned how to crop images using similar notation. You have learned how to transform images into a different colour space to create a simple visual effect. You have learned the importance of colour space choice in terms of the impact it has when interpolating/blending colours.