EEE3032 – Computer Vision and Pattern Recognition

Lab Worksheet 1 – Image Processing in Matlab

Miroslaw Bober (m.bober@surrey.ac.uk)

Task: In today's lab you will learn how to do basic image processing in Matlab. The experiments focus on loading and saving images, and applying linear filtering operations via convolution you learned in lectures this morning.

If you are unfamiliar with Matlab you should first work through the optional Introduction to Matlab worksheet.

Resources: Download the Matlab code for the labs and unzip the code into a folder in your home area. Ensure the code folder is added to the Matlab search path (use File -> Set Path.. -> Add with subfolders). If you are unsure how to do this check Ex.5 of the above-mentioned "introduction to Matlab" worksheet or failing that ask the tutors.

All lab sheets and code are available from SurreyLearn

Ex1. Loading an Image

Locate the image 'sphinx.jpg' bundled with the lab code. It will be within folder 'testimages' within the folder you unzipped the code to. Modify the path on the following line accordingly:

```
>> img = imread('~/cvprlab/testimages/sphinx.jpg');
```

Now type:

```
>> imgshow (img);
```

This will display the colour image with axis marks 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).

If the imgshow function is not found by Matlab you did not set up the Matlab search path properly (see above)

It is possible that Matlab may distort the image to fit it onto the screen. You can ensure that the x and y axis are both of equal scale using the following command (this also works for graph plots etc.)

```
>> axis equal;
```

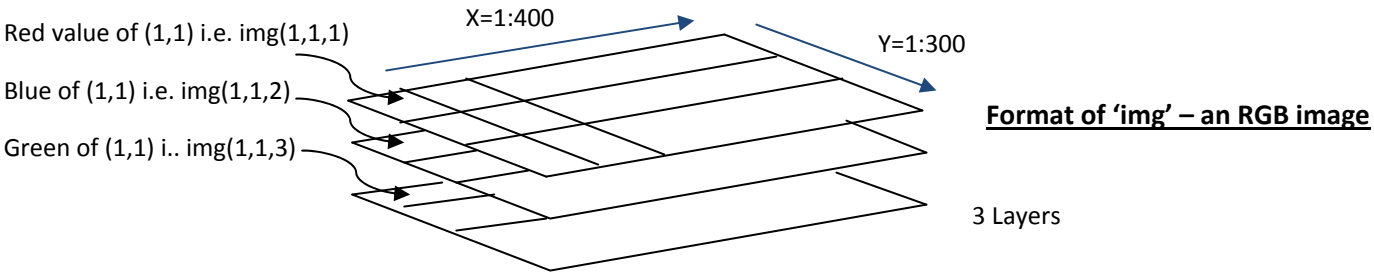
Ex2. Colour image representation

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.



Format of 'img' – an RGB image

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, and are of class 'uint8' (unsigned 8 bit integer) which is different from the default type of 'double'. This presents us with some inconveniences discussed later (Ex.5).

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:

```
>> imgshow(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,:);
>> image (subimg);
>> axis equal;
```

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 of type 'uint8' (check this with the class command). Matlab greatly restricts the operations one can perform on 'uint8' matrices. For example, multiplication and division are not allowed. This is an inconvenience when performing image processing work.

A more convenient format for images is 'double'. More convenient still, is 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.

```
>> imgshow(normimg);
```

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

Ex6. Greyscale conversion

It is finally time to do some image processing! Our first task will be to convert the colour sphinx 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. Filter for blur (low pass filter)

As discussed in this morning's lecture, we can blur an image using a low-pass filter. Although several forms exist, in this exercise we will use the box / mean blur. Create a matrix containing the kernel of the 3x3 box filter:

```
>> K=ones(3,3) ./ 9
```

Now use Matlab's 2D convolution function to convolve the normalised greyscale image of the sphinx with the box blur filter.

```
>> blurred= conv2(greyimg,K,'same');
```

The third parameter 'same' ensures the output image is the same size as the input. Otherwise conv2 would crop a 1 pixel border from the image, corresponding to the invalid zone for a 3x3 filter (i.e. where, when positioning K on the image during convolution, the kernel would partly lie outside of the image). View the result with imgshow.

Try different sized box filters for K. Ensure your elements in K sum to 1, or you may end up with pixel values >1.

Try a Gaussian filter in K. Do you feel the visual quality has improved?

If you have not done so already, create your own image blurring functions for the box and Gaussian filter.

Ex8. Filter for edge detection

Create matrices containing the 'x' and 'y' kernels of the Sobel filter.

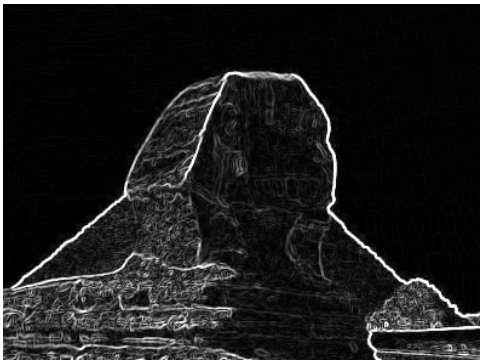
```
>> Kx = [1 2 1  ; 0 0 0 ; -1 -2 -1]./4;
>> Ky = Kx';
```

Convolve the normalised greyscale image with the filters to get the smoothed 1st derivatives with respect to x and y.

```
>> dx = conv2(greyimg,Kx,'same');
>> dy = conv2(greyimg,Ky,'same');
```

Compute the magnitude of the gradient.

```
>> mag = sqrt (dx.^2 + dy.^2);
>> imgshow(mag)
```



Please note there is a bug in Matlab with figure display. Sometimes if a figure window has been closed abnormally, later commands fail to display a figure. If this is happening type "close all" to reset the windows.

Ex.9 Thresholding an image

As seen in Ex.8, the Sobel edge magnitude lies in range 0-1.

We might interpret this as the probability that a pixel is part of an edge (apologies to statisticians!.

Usually at some stage in an Computer Vision application we need to decide whether an pixel is, or isn't part of an edge.

A simple way to do this is by "thresholding" the image. We pick a threshold value, say 0.15, and decide that all pixels > 0.15 are edge, otherwise they are not.

Thresholding can be performed very quickly in Matlab.

```
>> thresholded = (mag >0.15);
```

The result is an image with 0 for non-edge, and 1 for edge pixels. This is called a binary image, or mask. Check the output by displaying 'thresholded' and experimenting with other threshold values.

```
>> imgshow(thresholded);
```

Is it possible to pick a threshold value (e.g. an alternative to 0.15) so that all parts of the image manifest clear, intuitively correct edges?