Distributed Multimedia Systems CS403

Laboratory Experiments

Exercise 1 Matlab Background and Image Basics

Objectives

- Gain workable knowledge of MATLAB.
- Use the MATLAB image-processing environment.
- Understand some of the basic concepts of Image Processing

1. MATLAB Background.

Matlab, *MAT* rix *LAB* oratory, is a powerful, high level language and a technical computing environment which provides core mathematics and advanced graphical tools for data analysis, visualisation, and algorithm and application development. It is intuitive and easy to use since it follows many other programming languages like C, but also, many common functions have already been programmed in the main program or on one of the many toolboxes.

In MATLAB everything is a matrix, (this is the mantra of matlab, *everything is a matrix*) and therefore the programme structure it is a bit different from other programming languages, such as C or JAVA. Matrix operations are programmed so that element-wise operations or linear combinations are more efficient than loops over the array elements. The *for* instruction is not always recommended (but you can still use it any time).

Once you are in MATLAB, many UNIX commands can be used: *pwd, cd, ls,* .... To get help over any command you can type:

```
>> help command
```

For example try:

```
>> help for
>> help sum
>> help fft
>> help help
```

To create a matrix you can type its values directly:

```
>> x = [ 1 2 3 4 5 6 7 8 9 10 ];
```

Which is equivalent to:

```
>> x = 1:10;
```

where only initial and final values are specified. It is possible to define the initial and final value and increment (`lower limit:increment:upper limit`) using the colon operator in the following way:

```
>> z = 0 : 0.1 :20;
```

Both are **1 x 10** matrices. Note that this would be different from:

```
>> y = [1;2;3;4;5;6;7;8;9;10];
```

or

```
>> y=[1:10]';
```

Both are **10 x 1** matrices. The product x*y would yield the *inner product* of the vectors, a single value, y*x would yield the *outer product*, a 10 x 10 matrix, while the products x*x and y*y are not valid because the matrix dimensions do not agree. If element-to-element operations are desired then a dot "." before the operator can be used, e. g. x.*x would multiply the elements of the vectors:

```
>> x*y

ans =

   385

>> x*x
??? Error using ==> *
Inner matrix dimensions must agree.

>> y*x

ans =

1     2     3     4     5     6     7     8     9     10
2     4     6     8     10    12    14    16    18    20
3     6     9     12    15    18    21    24    27    30
4     8     12    16    20    24    28    32    36    40
5     10    15    20    25    30    35    40    45    50
6     12    18    24    30    36    42    48    54    60
7     14    21    28    35    42    49    56    63    70
8     16    24    32    40    48    56    64    72    80
9     18    27    36    45    54    63    72    81    90
10    20    30    40    50    60    70    80    90    100
>> x.*x

ans =

     1     4     9    16    25    36    49    64    81   100
```

The Matrix:

$$mat = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

can be obtained by typing:

```
>> mat = [1 2 3 4;2 3 4 5;3 4 5 6;4 5 6 7];
```

The final semicolon (;) inhibits the echo to the screen. Any individual value of the matrix can be read by typing (without the semicolon):

```
>> mat (2,2)
```

```
ans =

3
```

Mathematical functions can be used over the defined matrices, for example:

```
>> s1 = sin (z);
```

A column or line of a matrix can be obtained from another one:

```
>> s2(1,:) = -s1/2;
>> s2(2,:) = s1;
>> s2(3,:) = s1 * 4;
```

To display a 1D matrix you can use *plot*, and for 2D you can use *mesh*, Figure 1 shows the result of typing:

```
>> plot(s1);
>> mesh(s2);
```

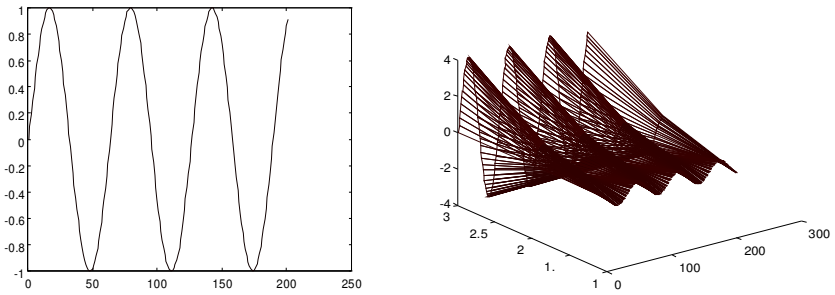


Figure 1

1.1 Use *help* for the following functions.

sign	subplot	abs	imshow	surf	colormap
sum	cumsum	fix	round	subplot	whos
title	for	who	sqrt	conv	floor
det	fft	abs	semilogx	axes	axis
zeros	ones	rand	randn	pi	real

1.2 There are many toolboxes with specialised functions, try:

```
>> help images    Image processing toolbox.
>> help signal    Signal processing toolbox.
>> help stats           Statistics toolbox
>> help nnet            Neural networks toolbox
```

1.3 To find out which toolboxes you have installed type ver:

```
>> ver
-------------------------------------------------
MATLAB Version 6.0.0.88 (R12) on SOL2
MATLAB Server Hostid: 8081cb38
MATLAB License Number: 137398
```

```
------------------------------------------------
MATLAB Toolbox                    Version 6.0   (R12)        06-Oct-2000
Image Processing Toolbox          Version 2.2.2 (R12)        10-Mar-2000
Signal Processing Toolbox         Version 5.0   (R12)        01-Jun-2000
Wavelet Toolbox                   Version 2.0   (R12)        16-Jun-2000
```

2. Image Basics.

If we consider an image as a two-dimensional function $f_a(x,y)$ where, for every co-ordinate position (x,y) exist a function value, then, a digital image $I = f(x,y)$ corresponds to a two-dimensional discrete function. This digital image can be analysed as a two-dimensional matrix in which every element of the matrix is considered as a picture element or *pixel* (sometimes called *pel*). A typical size of an image is a square of 2^n x 2^n pixels: 128 x 128, 256 x 256, 512 x 512 pixels. In the general case an image can be of size m-by-n, m pixels in the vertical direction and n pixels in the horizontal direction. The value of the pixel can represent light intensity, attenuation, depth information, or intensity of radio wave depending on the type of information contained by the image. This intensity level is the essential information to perform any operation like segmentation in most of the techniques.

Colour images contain important information about the perceptual phenomenon of colour related to the different wavelengths of visible electromagnetic spectrum. In many cases, the information is divided into three primary components *Red, Green* and *Blue* or psychological qualities *hue*, *saturation* and *intensity*. Prior knowledge of the objects colour can lead to classification of pixels but this is not always known.

The intensity level describes how bright or dark the pixel at the corresponding position should be coloured. There are two ways to represent the number that represents the brightness of the pixel: The *double class* (or data type). This assigns a floating number between 0 and 1 to each pixel. The value 0 corresponds to black and the value 1 corresponds to white. The other class is called *uint8* which assigns an integer between 0 and 255 to represent the brightness of a pixel. The value 0 corresponds to black and 255 to white. The class *uint8* only requires roughly 1/8 of the storage compared to the class *double*. On the other hand, many mathematical functions can only be applied to the doubles. The following image formats are supported by Matlab: BMP, HDF, JPEG, PCX, TIFF, XWB, PNG.

To read and display images use *imread* and *imshow* (if *imshow* is not working you can try *imagesc*, or *image*, in the worst case scenario that none of these work, use the steps in 2.2):

```
im1 = imread ('flowers.tif');       %Reads and image and puts it in im1
im2 = imread ('moon.tif');
im3 = imread ('tire.tif');
im4 = imread ('mri.tif');
figure(1)
imshow(im1);                        % Displays the image
figure(2)
subplot(2,2,1); imshow(im1);
subplot(2,2,2); imshow(im2);
subplot(2,2,3); imshow(im3);
subplot(2,2,4); imshow(im4);

figure(1)
clf;
subplot(3,1,1)
imshow(im1(:,:,1))                  % Displays Red component of the image
subplot(3,1,2)
imshow(im1(:,:,2))                  % Displays Green component of the image
```

```
subplot(3,1,3)
imshow(im1(:,:,3))                    % Displays Blue component of the image
```

2.1 Can you notice the differences of the colour components? Look at the yellow flower on the left.

The images are located in the directory /package/matlabr12/toolbox/images/imdemos. Notice the differences of size and colour of the images. The images can not be manipulated when their type is *uint8*, they have to be converted to *doubles*:

```
im4 = double(im4);              % Converts image to a matrix of doubles
im5 =randn(128,128);            % Creates a 128 x 128 normally
                                % distributed image (gaussian noise)
```

2.2Now try:

```
surf(im4); shading flat;
colormap(gray);
colormap(jet);
rotate3d on;
```

2.2 Use the mouse to rotate the image and change the view point. This is equivalent to use the *view* command. Change the axis of the image with the *axis* command. Repeat the process for other images.

Once the images are in a double type, the are treated and handled as matrices so that any operations; addition, subtraction, multiplication, inversion, transposition, ... are valid, as long as they follow matrix operations.

2.3 The functions *max, min, mean, std, sort* are very useful when you are interested in the statistics of an image. Find out more of them with the command *help* and apply them to the previous images. What is the response for m x 1, or 1 x n matrices? What happens when the size is m x n? *mean2* and *std2* can be useful in those cases.

3. Scripts and Functions

Scripts and functions, also called *m-files,* are widely used and make life easier while using Matlab. M-files group several commands that are stored in a text file with the extension *.m* and can be later used as any normal matlab function. For example, a simple function that displays and image with the command *surf*:

```
function outputImage = surfImage (inputImage)
% function outputImage = surfImage (inputImage)
% this function uses surf to display an image
surf(inputImage);
shading flat;
view(0,-90);
axis tight;
outputImage=inputImage*2;           %for example,a different output image
```

This function should be saved as a file with the same name, *surfImage.m*. In order to use the function an input image should be provided, the output is not necessary though. Try:

```
>> surfImage(im4);
```

When a set of instructions are to be repeated a number of times, but no input and output parameters are to be specified, these can be saved into an m-file, but without the header, that is, the first line of the previous example. This would be called a *script*.

3.1 Use *help* to find out more about scripts and functions.

4. Sampling and Quantising

When we want to manipulate an image in a computer it is necessary that this image is digitised both in its spatial and intensity dimensions. Digitisation in the spatial coordinates *(x,y)* is called *image sampling* and amplitude digitisation is called *grey-level quantisation*. Let the original digital image $I= f(x,y)$ have dimensions for rows and columns N_r x N_c. Let L_c =1,2,...,N_c and L_r=1,2,...,N_r be the horizontal and vertical spatial domains, and G=1,2,...,N_g the set of grey tones. The image I can be represented then as a function that assigns a grey tone to each pair of co-ordinates:

$$L_r \times L_c ; I : L_r \times L_c \rightarrow G$$

In many cases the dimensions of the image are integer powers of two:

$$N_r = 2^{nr},\ N_c = 2^{nc},\ N_g = 2^{ng}$$

The number of bits required to store the image become:

$$b = N_r \text{ x } N_c \text{ x } ng$$

Since $f(x,y)$ is already an approximation of the original image $f_a(x,y)$, the resolution required for a "good" representation can vary according to the image itself and the quality desired (which defines what "good" means for every application). To sample an image in matlab, you can use the index of the corresponding matrix to select a reduced number of entries of the matrix. First convert image into a matrix of doubles:

```
>>im2=double(im2);            %transform into doubles to manipulate
```

In the same way that the matrices were defined with the colon operator (*lower limit : increment : upper limit*) the elements can be selected. Try:

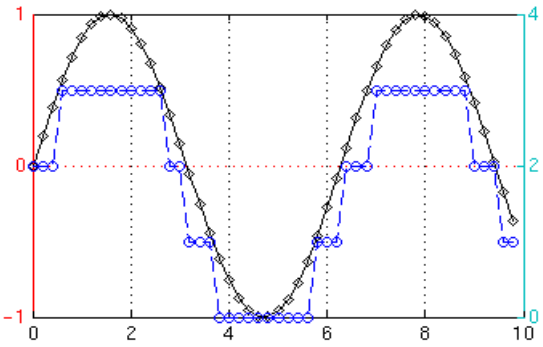
```
>> surfImage(im2(1:1:end,1:1:end))
>> surfImage(im2(1:2:end,1:2:end))
>> surfImage(im2(1:3:end,1:3:end))
>> surfImage(im2(1:4:end,1:4:end))
```

4.1 The previous instructions should display the moon image with different resolutions. Try this on other images.

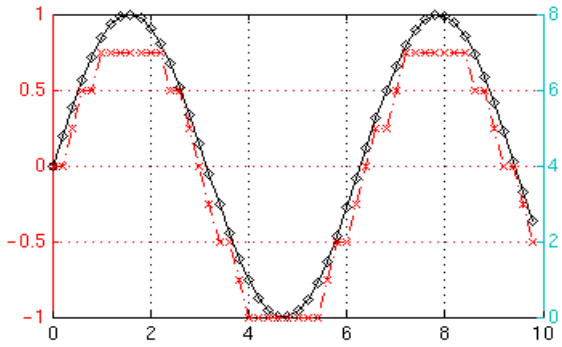
In all cases the number of grey levels is not modified. To quantise any signal, the *uencode* function can be used (if uencode is not working, in my web page there is a simple quantising function called *quant.m*). The following example quantises a sine function. Figure 2 shows the sine curve and its quantised version with (a) 2 bit = 4 levels and (b) 3 bits = 8 levels.

```
>> t=0:0.2:9.9;
>> x=sin(t);
>> y2=uencode(x,2);
```

```
>> y3=uencode(x,3);
>> [ax,h1,h2]=plotyy(t,x,t,y2);
>> [ax,h3,h4]=plotyy(t,x,t,y3);
```



(a)



(b)

Figure 2

4.2 Find out more of the functions *uencode* and *plotyy*. The function *plotyy* returns the handles of the two axes created in *ax*, and the handles of the graphics objects from each plot in *h1* and *h2*. *ax*(1) is the left axes and *ax*(2) is the right axes.

4.3 Use *help* to study about handles and their properties, they are quite useful!

4.4 Use *uencode* to quantise any of the previous images, change the number of quantising bits and display the quantised images. What can you observe?

4.5*False contouring* is a common phenomenon that appears when an image is quantised and there is an insufficient number of grey levels for smooth areas, have you perceived it? Quantise several images and try to compare this phenomenon.

4.6 Use one of the references provided below to study about *Nonuniform* sampling and quantising.

5. Histograms

The grey level histogram for the image I ranges from 1 to N_g, being 1 black and N_g white, and is defined as:

$$h(g)=\frac{\#\{(k,l)\in(L_r\times L_c): I(k,l)=g\}}{\#\{L_r\times L_c\}},\ g\in G$$

where # denotes the number of elements in the set. To visualise the histograms of images 4 and 5 try:

```
>> hist(im4(:),50);
>> hist(im5(:),50);
```

5.1 Notice the colon operator, what function is it performing?

The results are shown in figure 2.

5.2 Analyse the histograms. Were you expecting something simmilar?

5.3 We can consider image 5 as noise. Try adding image 4 and image 5 multiplied by a certain constant, e.g. a certain level of noise. Observe the image and the histogram.

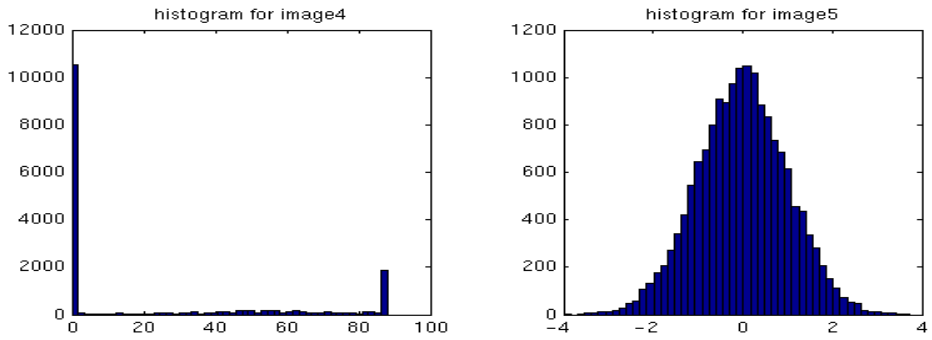


Figure 3

One simple and popular segmentation technique is *grey level thresholding*, which can be either based on global (all the image) or local information. In each scheme a single or multiple thresholds for the grey levels can be assigned. The philosophy is that pixels with a grey level $\leqslant x$ belong to one region and the remaining pixels to other region. In any case the idea is to partition into regions, *object/background*, or $object_a$ / $object_b$ /... *background*. The thresholding methods rely on the assumption that the objects to segment are distinctive in their grey levels and use the histogram information, thus ignoring spatial arrangement of pixels. Although in many cases good results can be obtained, in some particular cases the intensities of certain structures are often not uniform and therefore simple thresholding can divide a single structure into different regions. Another matter to consider is the noise intrinsic to the images that can lead to a misclassification. In many cases the optimal selection of the threshold is not a trivial matter.

Try segmenting some regions of the images. Two examples of the tire image are shown in figure 3.

```
>> subplot(1,3,1); hist(im3(:),50);
>> subplot(1,3,2); imshow(im3>60);
>> subplot(1,3,3); imshow(im3<60);
```

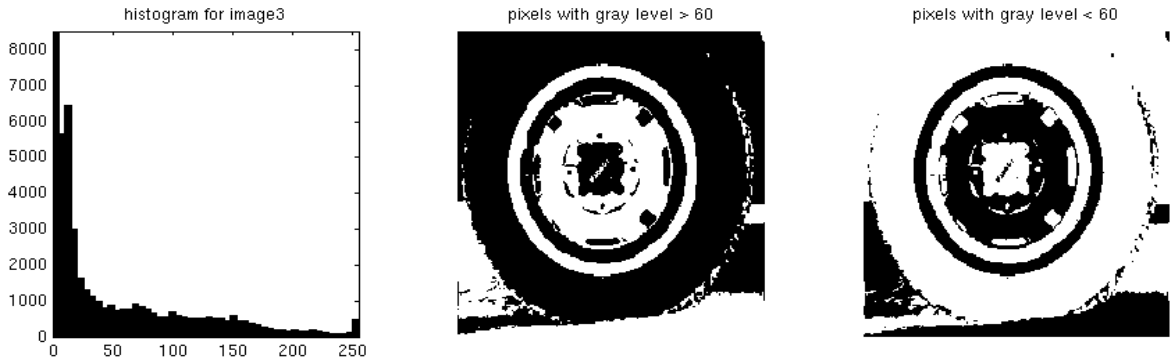


Figure 4

In some cases where noise is present, the boundaries of certain regions can be lost. An example of that follows. Generate a matrix with two regions of noise with different mean and variance.

```
>> image6(1:128,1:64)=20+10*randn(128,64);
>> image6(1:128,65:128)=40+8*randn(128,64);
```

5.4 The histogram and image are presented in figure 4. Even that the two regions seem distinctive, they cannot be properly segmented just by a thresholding operation. Try to find a proper threshold and view the segmentation.

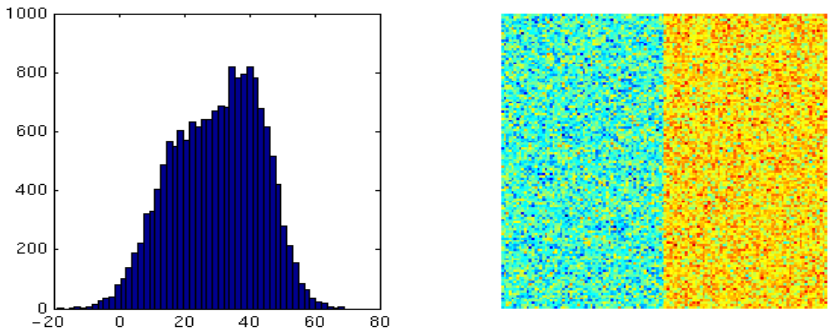


Figure 5

Now try a different approach. Since the images are formed by noise, then an average of the neighbouring pixels should tend to the mean of each region. This is called a pyramid or tree reduction of the image which has some filtering properties. The successive reduced images are considered as the stages of a pyramid.

Let the original image *I* be considered as the bottom level of a pyramid g_0. The first level, g_1 is obtained through a low pass filter or reduction of g_0. For every level, the superior or *parent* will be obtained by:

$$g_k = REDUCE\left(g_{k-1}\right)$$

Each node *i,j* of the higher level is a weighted average of the values of the lower level within a (for example) *2* x *2* window *w(m,n)*:

$$g_k(i,j) = \sum_{m=0}^{1} \sum_{n=0}^{1} w(m,n)\, g_{k-1}(2i+m, 2j+n)$$

If *w(m,n)* = 1 $\forall$ *m,n*, then the *parent* value is just the arithmetic mean of the *children*.

5.5 Write a function that performs the *reduce* operation, can you do this without using any *for-loops*?
5.6 Using *tic; operation; toc* you can measure the time elapsed by the operation, compare the time of a function with and without loops in it.
5.7 Once you have the function, construct a pyramid with the reduced versions of the image6. Display their histograms. What can you observe? Repeat the experiment with other images.

Sometimes is interesting and useful to change the image's intensity values from its present range to a new range, this technique is called Intensity Adjustment and is easily done in matlab with the *imadjust* function. The syntax of the function is

```
J = imadjust(I,[low high],[bottom top])
```

where *low* and *high* are the intensity range in the input image to be kept, which are mapped then to *bottom* and *top* in the output image. The values should be between o and 1. In this way, contrast can be increased or decreased in the whole image or in certain regions depending on the values of the vectors. For example try:

```
im7 = imread('trees.tif');
im8 = imadjust(im7,[0 0.3],[0.5 1]);
subplot(1,2,1); imshow(im7)
subplot(1,2,2); imshow(im8)
```

5.5 What happens with the image? Notice the detail revealed. Experiment with other values in both vectors.

5.6 Use *help* to study more about the histogram functions *imadjust* and *histeq*. Try to equalise the histogram of some images.

6. References

Sonka, Milan, Hlavac,Vaclav, Boyle, Roger, *Image processing Analysis and Machine Vision*, PWS Publishing, 2nd ed, 1998.

Gonzalez, Rafael C., Woods, Richard E., *Digital Image Processing*, Addison-Wesley, 1993.