Medical Image Processing (SEE120)

Matlab Tutorial - 1

Start Matlab.

Remember those useful commands: - to close all figures:

```
>> close all;
```

to clear all variables from your workspace:

```
>> clear all;
```

to get in-program information on a Matlab function:

```
>> help function_name
```

to get more extensive online help

```
>> doc  function_name
```

- to look for Matlab functions by keywords:

```
>> lookfor keyword
```

Where say keyword could be something like ‘histogram’

1. Variables

In Matlab we can create variables associated to an object (a scalar, a vector, a matrix...) by simply declaring its value, like:

```
>> myvariable1 = 25
```

Where the >> is the MATLAB cursor prompt. In the Command Window, type:

```
>> myvariable1
```

To check the value of this variables. You can manipulate this variable in many ways. Let's do some simple arithmetic:

```
>> myvariable2 = 5

>> myvariable1 + myvariable2
```

```
>> myvariable1 - myvariable2

>> myvariable1,* myvariable2

>> myvariable1./ myvariable2 ...
```

Here all the variables are single numbers but in general the can be matrices so it is good to always use '.*' and './' with dots mean elementwise multiplication and division between matrices of the same size (otherwise MATLAB will try to do matrix multiplication or division).

If you write ';' at the end of a command line, the command will be executed silently, without writing the result to the Matlab Command Window.

```
>> myvariable3 = myvariable1 + myvariable2;
```

Check the value of myvariable3:

```
>> myvariable3
```

2. Create an image

An image is a 2D matrix. Vectors and matrices are built using [ ].
Let's create an image of a rectangular spiral with 20 gray levels increasing constantly from black to white and call it myimage:

```
>> myimage = [ 0 1 2 3 4 ;...

            13 14 15 16 5 ;...

             12 19 18 17 6 ;...

              11 10 9 8 7 ];
```

The ';' inside the [ ] containing your matrix is used to announce a new row. The '...' are there to allow you to a command over several lines Remember, the ';' at the end of a command line is there to execute the command silently.
Check that your matrix looks as expected:

```
myimage
```

You don't have to use .. as above, typing the following will give you exactly the same result:

```
>> myimage = [0 1 2 3 4 ; 13 14 15 16 5 ; 12 19 18 17 6 ; 11 10 9 8 7];
```

Check this:

```
myimage
```

3. Display an image

imshow & imtool: to display the image type imshow(myimage);

It is bit too small... there's only 20 pixels in our image.

imtool can as well be used to display images and has a few interesting options for navigating within images.

>> imtool(myimage);

The default magnification is 100%,; to get a bigger image you can see select in that box "fit to window". It doesn't look like expected...
imshow and imtool, if not otherwise specified, display intensities comprised between 0 (black) and 1 (white). Everything <= 0 is considered black, everything >= 1 is considered white.

So we have two options:

i/ We rescale our image between 0 and 1
For that we divide each element by the max intensity using the function max() (>> help max)

>>myimage_rescaled_0_1 = myimage / max(max(myimage));

Here we need to use: max(max(myimage)) because myimage is a 2-D array and max(myimage) gives a row vector containing the max element from each column.

Applying the max() function a second time on that row vector gives the largest element in that vector. Let's check this:

>> max(myimage) max(max(myimage))

Let's display the result:

>> imtool(myimage_rescaled_0_1);

ii/ We tell imtool to consider the full dynamic range of our image by giving '[]' as a second parameter:

>> imtool(myimage, []);
We can use the same trick with imshow!

4. Load an image

We can load images in many different formats with imread:

```
>> cameraman = imread('cameraman.tif');
>> spine = imread('spine.tif');
```

ImageBrowser(fullfile(matlabroot, 'toolbox/images/imdata')) displays the inbuilt images within MALAB

Alternatively typing

```
>> folder = fileparts(which('spine.tif'))
```

i.e. without ':' at end gives the directory containing the spine.tif file, if copy-paste that directory with your cursor can >>cd to that directory then you can do >ls to see a list of many in built matlab images.

Properties of all the variables you have in the workspace you can inspect using

```
>>whos
```

You can use imshow to display them for instance: imshow(cameraman); imshow(spine);

The second image overwrite the first one in the figure window. To avoid that, open a new empty figure before displaying the second image by giving the command: figure

```
>> close all; imshow(cameraman); figure; imshow(spine);
```

The spine image looks bad because by deault imshow only displays between 0 and 1 with all value of 0 or less being black and all greater or equal to 1 are shown as white. But we know how to improve that:

```
>> figure; imshow(spine, []);
```

To get an image displayed over its full range

You can get information about the image files with the command imfinfo i.e.

```
>> imfinfo spine.tif;
```

5. Classes and Class conversion

Matlab supports various classes for representing pixel values (i.e. - double, single, uint8, uint16, logical ..) – Single and Double are single precision (8 bit) and double precision (16 bit) floating point numbers and uint8 are unsigned 8 bit integers, i.e. integers from 0 to 255 and 16 bit integers. Logical are either 0 or 1.

Let's see what is the class of the image we created (myimage) using the function whos or class:

>> whos myimage

>> class(myimage)

What about the class of the .tif images we loaded:

>> whos cameramam

>> whos spine

Both are of type uint8.

Classes uint8 (unsigned 8 bits) and logical (1 or 0) are often used in image processing and are the usual classes when reading TIFF or JPEG images.

Beware – that some MATLAB commands what is assumed for display range etc depends on the pixel format.

Be beware that uint8 format image don't support negative numbers, so if for instance you convolve a unint8 image with a kernel with negatives the output pixel value less tnan zero become zero. For this reason often good to convert uint8 to single or double precision format.

Classes double and single are often used for computationally intensive operations such as Fourier transform.

For all the above reasons it is important to know how to convert between classes. Matlab offers a certain number of functions for conversion such as: im2uint8, im2uint16, im2double, im2single, im2bw, mat2gray

Convert myimage from class double to class uint8. im2uint8 needs its input to be scaled between 0 and 1. All values in the input that are less than 0 are set to 0 and all values that are more than1 are set to 255. To do that we can use the class conversion function mat2gray:

>>myimage_gray = mat2gray(myimage); myimage_gray

which is exactly the same as myimage_rescaled_0_1!!!
Now we can convert myimage_gray from class double to class uint8:

>> myimage_uint8 = im2uint8(myimage_gray);

Check the class of myimage_uint8: whos myimage_uint8

Compare myimage_rescaled_0_1, myimage_gray and myimage_uint8:
myimage_rescaled_0_1

>>myimage_gray myimage_uint8

Now play a bit with other class conversion functions and see what happens (use the help!). Ex:

```
>> im2foub= im2double(myimage_uint8)

>> myimage_bw = im2bw(myimage_gray)

>> imtool(myimage_bw);

>> whos myimage_bw

>> myimage_bw_75 = im2bw(myimage_gray, 0.75)

>> imtool(myimage_bw_75);
>> whos myimage_bw_75

>> myimage_bw_75_double = im2double(myimage_bw_75)

>> whos myimage_bw_75_double
```

Converting a logical image into double keeps the pixel values 0 or 1!

It would have been possible to obtain myimage_bw_75_double in only one command line. The output of one function can be passed directly as the input to another:

```
>> myimage_bw_75_double = im2double(im2bw(mat2gray(myimage),0.75))
```

It is possible to convert to another class without any rescaling, by using the functions double() or uint8() for example. Compare:

```
>> myimage_uint8_to_double = im2double(myimage_uint8)

>> whos myimage_uint8_to_double
```

6. Saving an image

We can save our images in the current directory using imwrite but we have to be a bit careful. If the array is of class double, imwrite assumes the dynamic range is [0,1] (>> help imwrite). myimage is of class double, but it's dynamic range isn't [0,1] so it won't work. Let's use myimage_gray instead, also of type double and with correct dynamic range:

```
>> imwrite(myimage_gray, 'myimage.tif');
```

>> imwrite rescales the image between 0 and 255 and saves it as uint8.

We can display the resulting image directly to check if everything is alright:

```
>> imtool('myimage.tif');
```

We could also have used myimage_uint8:

```
>> imwrite(myimage_uint8, 'myimage2.tif');
```

Those two methods produce exactly the same results.

7. Array indexing

a/ Indexing Vectors

It is possible to use different indexing schemes in order to refer to single or multiple elements in a vector or a matrix, and to improve the efficiency of programs.

Let's create a row vector:

```
>> myvector1 = [1 3 5 7 9 11 13]; myvector1
```

It is possible to get the transpose of myvector1, by using the transpose operator ('):

```
>> myvector2 = myvector1'
```

```
>> myvector2
```

```
>> myimage_uint8_in_double = double(myimage_uint8)
```

```
>> whos myimage_uint8_in_double
```

We can access a single element in those vectors by refering to its position (in Matlab, the first element is at position 1 (not 0)):

```
>> myvector1(3)
```

```
>>  myvector2(4)
```

We can access blocks of elements using the notation: myvector(*start_element:end_element*): For example, the 3 first elements:

```
>> myvector1(1:3)
```

the following 3 elements:

```
>> myvector1(2:4)
```

all the elements from the third element:

```
>> myvector1(3:end)
```

To access non contiguous elements, we use the following notation:

start_element : step : end_element

For example, to refer to every third elements from the first one:

```
>> myvector1(1:3:end)
```

What does the following do?

```
>> myvector1(end:-2:3)
```

We can refer to any desired elements using a vector as index. For example, to access the 2nd, the 4th and the 5th element:

```
>> myvector1([2 4 5])
```

b/ Indexing Matrices

We can as well index matrices using the coordinates of their elements separated by a comma. First the row, then the column. For example, if we want to extract the element in the second row, fourth column of the matrix myimage, we write:

```
>> myimage(2, 4)
```

Try:

As for vectors, we can select a submatrix of contiguous or non-contiguous elements using the colon (:) symbol:

```
>> myimage(2:end, 2:4)
```

```
>> myimage(3, end-1:-2:1)
```

Or using vectors as indexes, we can select any desired submatrix of not necessarily contiguous elements:

```
>> myimage([4 2], [1 4 3])
```

```
myimage(end, 3) myimage(end, end) myimage(2, end-2)
```

It is possible to select an entire row or column by using the colon symbol alone as row or column coordinate.
To select the entire 2nd row:

```
>> myimage(2, :)
```

To select the entire 3rd column:

```
>> myimage(:, 3)
```

See what happends if you use a single colon as an index into your matrix:

```
>> myimage(:)
```
Using that technique we do not have to call the max() function (or sum(), etc) twice when looking for the max element in the entire matrix:

```
>> max(myimage(:))
```

is the same as:

```
>> max(max(myimage))
```
since myimage(:) is a vector.

Any of those forms of indexing can be used to modify part of the array. Let's create a new matrix A:

```
>> A = [1 2 3; 4 5 6; 7 8 9] B = A;
```

See what the following commands do on the matrix B == A:

```
>> B(2, 1) = 40
>> B(:, 3) = 0
>> B([2 3], [1 3]) = 100
```

c/ Logical Indexing

It is possible to select elements in an array A using another array D of the same size containing a logical 1 at the position of the desired elements and a logical 0 otherwise. So let's create a logical array D with the size of array A.
First let's check the size of array A using the function size():

```
>> size(A)
>> D = logical([1 0 0; 0 0 1; 0 0 0])
```

logical() forces the array to be of class logical. Now, let's use D as an index array to A:

```
A(D)
```

Let's see how this can be used to modify the array A:

```
A(D) = [10 60]
```

```
A(D) = 100
```

8. Good to know: some important standard arrays

```
>> A=zeros(3,4)   % 3 rows x 4 columns all zero matrix
```

```
>> B=ones(5).          % 5x5 all one matrix
>> C=3 * ones(3,2)   % 3x2 all 3 matrix
```

```
>> D=true(5,2)   % 5x2 logical matrix of 1s
```

>> E=eye(4) % 4x4 identity matris i.e. 1's along diagonal, 0 otherwise

>> F=rand(2,4) % 2x4 uniformly distributed random numbers in the interval [0,1]

>> G=randn(2,4) % 2x4 Gausian probability random numbers with variance 1

Check also false(), magic(), See what the following does (check help round):

>> round(10 * rand(5))

9. Operators

a/ Arithmetic operators

Let's define:

>> A = [1 2 3; 4 5 6] B = [9 8 7; 6 5 4] C = [1 2; 3 4]
a =2

b = 4+2*j

Play a bit with:

a +A A +B a *A A * B A * B.' A .* B A ./ B C^a C.^a -C -C^a (-C)^a b

b'
A+b (A+b)' (A+b).'

% complex number

% What happens? Why?

% complex conjugate transpose % transpose

b/ Relational operators

You can use those to compare pair of elements in arrays of equal dimensions. < : less than

<= : less than or equal to
> : greater than
>= : greater than or equal to == : equal to

~= : not equal to

See what these do, and try to understand:

A = [1 2 3; 4 5 6; 7 8 9] B = [0 2 4; 3 5 6; 3 4 9] a =4

A == B A< B A <= B A >B A >= B A ~= B B == a A >a a <A

For instance, instead of using the function im2bw() in order to get the array myimage_bw_75 in chapter V., we could have obtained the same result by using a relational operator:

myimage_bw_75_bis = myimage_gray > 0.75 whos myimage_bw_75_bis

Every pixel of value larger than 0.75 will be a logical 1 (or true) otherwise it will be a logical 0 (or false).

c/ Logical operators

& : elementwise AND
| : elementwise OR
~ : elementwise and scalar NOT && : scalar AND
|| : scalar OR

Logical 1s and non-zero numbers are considered as true, logical and numerical 0s are considered as false.
See what these do, and try to understand:

A = [1 2 0; 0 4 5] B = [1 -2 3; 0 1 1] a =3
b = logical(1)

A& B A |B ~A

~A & B a && b a && ~b a || ~B

There exist in Matlab a few logical functions that may be useful: to get the list of those functions in Matlab documentation, type:

doc is

10. Linspace, Meshgrid, Find

These commands together are extrenely useful for creating image/filters in MATLAB. As an example first create two vectors of 128 numbers starting at -63 with each element increasing by 1 till we get to 64.

xvec= linspace(-63,64,128);

yvec = linspace(-63,64,128);

Then use

[XX,YY] = meshgrid(xvec,yvec);

This creates matrices XX, YY which are 128x128 matrices which contain respectively the x and y values of every pixel.

The next command creates a matrix/image which stores for every pixel the number of pixels to the central point of the matrix/image (with the centre defined in the case as pixel 64, 64)

RR= sqrt(XX.^2 + YY.^2);

Next we can use the *find* command and the principles of indexing in MATLAB to create an image of a circle of radius of say 10 pixels. First in variable *index* store the location of all pixels closer than 10 pixels from the centre.

rad=10;

index= find(RR<rad);

Then initialize the *circ* image as zero then and after that set all the pixels within the defined radius to be one i.e.

circ = zeros(128,128):

circ(index) = 1;

Then use *imshow(circ,[])* to display an image of a circle.

11. Exercise

Let's use all this to process some images! Do this exercise in a new m-file.

1. Load the cameraman.tif image into the variable: cam Display it using imshow.

2. Convert it to class double using the full dynamic range [0,1] with function mat2gray and call it cam_gray

3. Display the negative of this image (call it cam_neg)

4. Convert the negative image into black and white using the function im2bw with a threshold of 0.75, and call it cam_neg_bw. Display it.
It should look like Image 1 (see below).

5. Using the help command, try to understand what this does:

x = linspace(-8*pi, 8*pi, size(cam,2)); y = linspace(-8*pi, 8*pi, size(cam,1)); [X,Y] = meshgrid(x, y);
myimage = cos(sqrt(X.^2 + Y.^2)); figure; imshow(myimage, []);

6. Rescale myimage between 0.2 and 0.8
To do that, first check the min and max values in the array. Then rescale it to have a dynamic range of 0.6 (= 0.8-0.2). Finally shift the intensity to be centered around 0.5 (the central value

between 0.2 and 0.8).
Call it myimage_rescaled and display it.

7. Use cam_neg_bw as a filter on myimage_rescaled by simply multiplying the two arrays (using the array multiplication symbol .*).
You should get Image 2 (see below). Call it myimage_filtered1.

8. Try to produce Image 3 (see below) using logical indexing (chapter VII.c/) on myimage_rescaled. Call it myimage_filtered2.

9. Create a new digital filter that will cover all the remaining buildings on the horizon of cam_neg_bw. For that, display cam_neg_bw with imtool and look for the lower and upper y-coordinates of the objects on the horizon.
Then create an all-zero matrix of the size of cam_neg_bw using the Matlab function zeros(). Call it filter2.

Add a horizontal band of ones between those two coordinates in filter2.
10. Now merge the 2 filters: cam_neg_bw and filter2 using a single logical operator.

Call it filter_final. Display it.
11. Produce Image 4 (see below) with your favorite method.

Call it myimage_filtered3 and display it.
12. Save it under the name: cameraman_filtered.tif

Image 1

Image 2

Image 3

Image 4