Appendix B

Image Processing in Matlab

B.1 Exploring Matlab

Matlab is a very powerful program for technical and scientific numerical programming. It is so popular (and expensive) that several Matlab clones are available nowadays. Both commercially as well as freeware. Search on the Internet for 'Matlab' and'clone' and you will find 'SciLab', 'Octave', 'O-matrix' and several others.

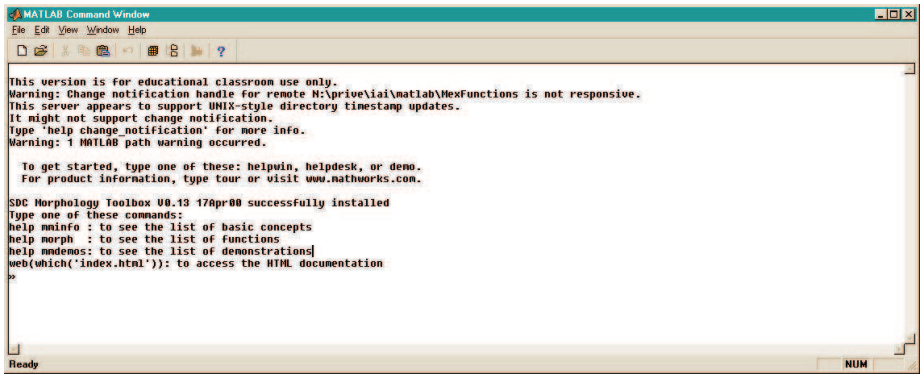
The best way to explore Matlab is by using it. The command `demo` brings up a GUI from which you can choose various basic tutorials and demonstration programs.

The help files are quite extensive. There is a plain ASCII text help system as well as a HTML based help system (the 'help desk'). You are advised to read the 'getting started' tutorial that is available on all Matlab installations.

B.2 Getting started

Image processing in Matlab is fun. It is simple, yet powerful for many of the tasks encountered in scientific image processing. This short introduction to Matlab and to image processing using Matlab is intended to help you get started enjoying the experience yourself.

We assume that you have Matlab 6.0 up and running on your PC. We also assume that the image processing toolbox is available. Your screen should look something like:



To get started right away, type in at the prompt (do not forget the semicolon!):

```
a = imread('kids.tif');
```

this will read the image in file `'kids.tif'` (an image that is part of the Matlab distribution) and store the pixel data in the Matlab variable `a`. Note that in Matlab there is no need to declare variables before using them. Assigning a value to a variable has the consequence that the variable becomes of the type of what was assigned to it. In this case the variable `a` is an array of bytes (one of the possibilities to represent images in Matlab).

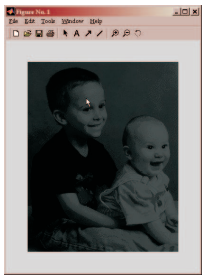
Within Matlab the command `whos` can be used to display the names of all variables and some associated information:

```
whos
     Name      Size          Bytes  Class
     a       400x318       127200  uint8 array
  Grand total is 127200 elements using 127200 bytes
```

Here we have used the typographic convention that all output from Matlab is preceded with a vertical line on the left.
Here we see that there is only one variable a and that it is a two dimensional 400×318 array of `uint8`'s (i.e. unsigned bytes). Matlab assumes that a 2D array of `uint8`'s is an image where the value corresponds with black and the value 255 corresponds with white. The image `a` can be shown on the screen with the command:

```
imshow(a);
```

A separate window should appear on your screen looking something like:



Indeed the image should look rather dark. This is easily understood when we look at the minimum and maximum grey value present in the image:

```
minval=min(min(a))
| minval = 0

maxval=max(max(a))
| maxval = 63
```

showing that this image only utilizes the very low (dark) grey values (remember that grey value 255 is white and 0 is black). Read the on-line Matlab documentation for the function `max` to understand why you have to nest the function call to `max` twice[1].

Matlab can display images represented as an `uint8` array but that is more or less all it can do with unsigned integers. To do some image arithmetic we first convert the image to a double (floating point) representation.

```
b = im2double(a);
```

This function automatically scales the image in such a way that now the range of grey values runs from 0 (black) to 1 (white).

Exercise. Now print the minimum and maximum grey value in the image `b`.

We can change the grey values in `b` in such a way that the entire dynamic range is utilized (from 0 to 1 in this case):

```
c = 1/maxval * b;
```

Display the image `c` to see what happened. You can also check by calculating the maximal value for this image.

Worth mentioning are the zoom capabilities. For the 5.3 Matlab version there are magnifier buttons visible in the figure windows. Pressing the '+' magnifier button will allow you to zoom in by clicking with the left button and zoom out with the right button. Matlab 5.2 users can achieve the same using the command `zoom`.

Remember:

- using double images in Matlab is much easier.
- convert images to doubles using `im2double` (or do the conversion to the $[0, 1]$ interval yourself).
- when you change the pixel values of an image you need to display it again. Images on display are just snapshots of the values at the time the image was displayed
- the 'help' button on the Matlab window command window brings up the help browser. All Matlab functions are documented.

[1] A dimension independent way of calculating the maximum value in a d-dimensional array is `(max(a(:))`. Here we use the Matlab notation `a(:)` to denote the 1D vector of all elements in `a`.

B.3 Array Indexing

Let us read another image from file:

```
a = imread('flowers.tif');
```

this is a color image which is obvious when you display it (using the `imshow` command). For now we only process gray value images, so:

```
a = rgb2gray(a);
```

will result in a grey value image. Changing the resulting `uint8` image to the double format:

```
a = im2double(a);
```

Display this image and show the coordinate axes:

```
imshow(a); axis on;
```

(note that we can put two statements on one line if they are separated by a semicolons.) This will change the display to:

A surprise in using Matlab for image processing is the choice for the coordinate axes. In Matlab images are represented using array data structures. The standard array indexing technique is used to access the individual pixels in the image. The value of a pixel at index (i,j) is obtained in Matlab as:

```
a(50,300)
  ans =
      0.3608
```

The answer from Matlab is appropriately assigned to the variable `ans` and displayed on the screen. Note that in the above command no semicolon was typed. Without ending a command with a semicolon, Matlab assumes that you want to display the result from a command (besides the 'side effects' a command might have, like rendering an image in a separate window). Ending the line with a semicolon prevents Matlab from displaying the result. Thus please end your image processing commands with a semicolon, else a very large number of pixel values will be displayed in the command window (pressing control-c will end this).

The value `a(50,300)` cannot only be obtained, but it can also be changed:

```
a(50,300) = 1;
```

Displaying the image again:

```
imshow(a);
```

shows that indeed something has changed: there is a small white spot visible somewhere. Convince yourself that it is at the location that you expected. Experiment with changing some pixel values to understand the coordinate system that Matlab is using. Is `a(0,0)` a valid indexing in the array?

In Matlab everything is an array. And because arrays are central to the Matlab universe it is easy to work with them. A vector of N numbers is represented in Matlab as a $1 \times N$ array and can be easily constructed from the command line (and in programs):

```
v=[ 8 5 1 9 ]
  v =
      8     5     1     9
```

The individual elements in the vector can be obtained and assigned to as `v(1)`, `v(2)`, `v(3)` and `v(4)`. Besides indexing in an array with just numbers, also indexing with vectors is possible. As an example, define the vector:

```
x = [ 1 3 ];
```

and use this to index into the vector `v`:

```
v(x)
  ans =
      8     1
```

You can also use this indexing construction to assign several elements from the array in one command:

```
v(x) = [ 11 34 ]
  v =
     11     5    34     9
```

These indexing constructions are so often used in Matlab that there is a special constructor to make arrays with a series of scalars: the colon construction:

```
1:10
  ans =
      1  2  3  4  5  6  7  8  9  10
```

Of course the start and end values can be chosen arbitrarily. A step value can also be specified:

```
100:-15:10
  ans =
     100    85    70    55    40    25    10
```

For two dimensional arrays the indexing constructions are very handy indeed. For example to find the 3x3 matrix of values around a pixel with indices (45, 100) we can simply write:

```
a(49:51, 299:301)
  ans =
      0.3451    0.3569    0.3922
      0.3529    1.0000    0.3686
      0.3373    0.3412    0.3569
```

Remember:

- ending a command with a semi-colon prevents the answer to be printed on the screen.
- typing CONTROL-C stops the printing of all image values (or other large data arrays) when you have omitted the semi-colon.
- indexing in Matlab starts at 1
- indexing in Matlab uses the matrix indexing scheme (row,column)
- a sequence (vector) of numbers can be easily generated with `start:step:end`
- index vectors (matrices) can be used to index a matrix

B.4 Displaying images and other plots

We assume that in `a` is the image from the last section. In case we take a scalar for one of the indices and let the other index run from 1 to the maximum we get a vector of values being either the pixel values in a row or column:

```
line = a( 128, :);
```

Please note that in an indexing construction the colon without any numbers stands for `1:end` where `end` is the maximum allowed for the array involved. In case you didn't type the semicolon at the end of the previous command you will have seen a lot of numbers printed on your screen. There is however a better way to look at one dimensional data: the plot command.

```
plot(line)
```

In the graphical window where `a` was displayed you should now be able to see a grey value profile, i.e. the grey values in the 128th row plotted as a function of the column index. Note that in Matlab there is no need to have just one command per line. The previous two commands can be combined:

```
plot( a( 128, :));
```

Most often it is convenient to show both the image and the image profile. You can do that in Matlab in several ways. The 'difficult' way is to display both the image and the profile in one window (look at the `subplot` command documentation). It is easier to create two windows. Every `figure` command creates a new graphical (figure) window. All image rendering and plot commands will use the 'actual' window (the last one created or the last one used).

```
imshow(a); figure; plot( a(128,:) );
```

B.5 Programming in Matlab

Often you will find yourself typing the same sequences of commands over and over again. Then it's time to collect these commands in a function.

Consider the function `nbh` that returns a (2N+1) square neighborhood centered at the pixel at coordinates `(i,j)` in a given image `im`. The coordinates are easily generated:

```
N = 3; i = 128; j = 164; % these will be the parameters
                         % to our function
iind = i + (-N:+N);
jind = j + (-N:+N);
```

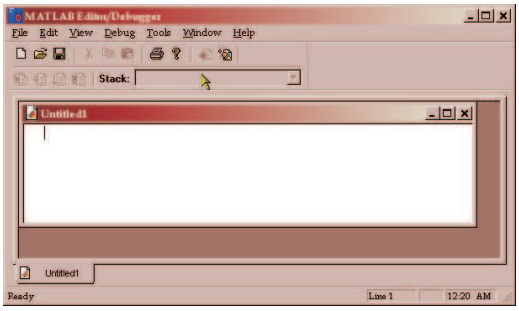
Note that we have used a new language construction here. We have added a constant to a vector (array). The constant then is added to all elements from the vector. We cannot use `im(iind, jind)` to get the pixel values for all values of `i` and `j` . In case we are close to the border the neighborhood would not be (entirely) within the image and an error (index out of range) will be issued by the Matlab interpreter. Let `imax` and `jmax` be the maximum index in both axes:

```
im = imread('bonemarr.tif'); % bone marrow image
im = im2double(im);
[imax, jmax] = size(im);
iind = max( min(iind, imax ), 1 );
jind = max( min( jind, jmax ), 1 );
```

Carefully note that a Matlab function can return multiple values as the `size` function does . Also note that again a scalar is 'distributed' over an array. The `imax` parameter in the `min` function is compared with each of the elements in the `iind` vector. Convince yourself that the above code fragment takes care of the 'border problem'. Let us cast this little piece of code in a small function.

In the command window press the button to start editing a new function:

This will open a text editor window:



Type in the following function:

```
function nbharray = nbh( image, i, j, N )
% nbh: get the 2N+1 squared neighborhood
% of the pixel (i,j) in 'image'
[imax, imin] = size(image);
iind = i + (-N:+N);
jind = j + (-N:+N);
iind = max( min( iind, imax ), 1 );
jind = max( min( jind, jmax ), 1 );
nbharray = image( iind, jind );
```

Notice that in Matlab you don't have to explicitly return a value, just assign it to the 'return variable' (in the above function `nbharray`). Save this function in a directory that you have selected for this purpose. Give the file the same name as the function with extension `.m`, in this case `nbh.m`. Make sure that you add this directory to the path along which Matlab searches for files. You can do this by clicking the 'pathbrowser button'

on the Matlab tool bar. Now you are ready to use the function:

```
nbh(im,1,1,3)

  ans =
     0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
     0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
     0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
     0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
     0.8549  0.8549  0.8549  0.8549  0.8588  0.9059  0.8902
     0.8588  0.8588  0.8588  0.8588  0.8784  0.9137  0.9059
     0.8706  0.8706  0.8706  0.8706  0.8784  0.9294  0.9137
```

You may not have noticed, but by now we are capable of writing our own convolution operation using a uniform kernel: enumerate all points `i,j` in the image and at each point calculate the average of the neighborhood pixel values.

```
function r = slowuniform( image, N )
[imax, jmax] = size(image);
r = image; % not strictly needed
           % but only here for 'efficiency'
np = (2*N+1)^2;
for i=1:imax
    for j = 1:jmax
        r(i,j) = sum(sum( nbh( image, i, j, N )))/np;
    end
end
```

Make sure you try the function (just to see that it is worthy of its name).

It should be noted that the syntax of a for-loop in Matlab is `for <varname> = <array>`. So instead of writing `for i=1:imax` we could have written `range = 1:imax; for i=range ....`

Remember:

- one function per file works OK.
- a function name and its file name must be the same
- store your own functions in your own directory and set the Matlab path to it (use the menu item 'set path' in the file menu or click the 'path browser button').

B.6 Image Processing in Matlab

By now you certainly found out that writing your own image processing function in *interpreted* Matlab code is very inefficient. Rule 1 in Matlab is that `for` or any other loop over the individual pixels is to be avoided. Either use a function available in the image processing toolbox or try to use generic array processing functions. In case everything else fails you could also write your own functions in C and use them from within Matlab as if it were a standard function. You need a C-compiler to do so.

The Matlab version used in the practical has the 'image processing toolbox' available. Browse the help files to see the functions in this toolbox. During the practical course you will be pointed to several of the functions from the toolbox to solve image processing problems efficiently.