Introduction to DIPimage

Contents

Quantitative Imaging Group
Department of Imaging Science & Technology
Faculty of Applied Sciences
Delft University of Technology
Delft, The Netherlands

T: +31 15 278 1416
E: Secr-QI-TNW@tudelft.nl
I: http://www.ist.tudelft.nl/qi
I: http://www.diplib.org/

1 Introduction

The goal of this laboratory work is to get hands-on experience with image processing. To do so, you will have to learn the image-processing environment: MATLAB and the DIPimage toolbox for multi-dimensional image processing. To facilitate a quick start, there will not be an in-depth explanation of all the features in the software. We will briefly present the things you need at this moment. We have marked the sections that explain something about the environment with the ☝ symbol, so that they stand out. Let us start with a short introduction to the MATLAB environment.

2 MATLAB

This section is to make you familiar with MATLAB; if you already are, skip this section.

☝ MATLAB is a computing/programming environment especially designed to work with data sets as a whole such as vectors, matrices and images. These data sets can be treated the same as a mathematical expression of scalar variables. MATLAB provides only a command line and graphics capabilities. The idea is that you give it commands through the command line, which are executed immediately. The MATLAB syntax will become clear during this laboratory session. Here just a few short comments:

```
a = b;
```

will cause whatever is in variable **b** (a scalar or an image) to be copied into variable **a**. Whatever was in variable a gets lost. If you omit the semicolon at the end of the command, the new contents of a will be printed (scalars, vectors and matrices will printed in the command window, whereas images displayed in a separate window). If **max** is the name of a function, then

```
a = max(a,b);
```

will call that function, with the values **a** and **b** as its parameters. The result of the function (its return value) will be written into **a**, overwriting its previous contents. If no explicit assignment is done, the output of a function will be put into a variable called **ans**:

```
max(a,b);
```

is the same as

```
ans = max(a,b);
```

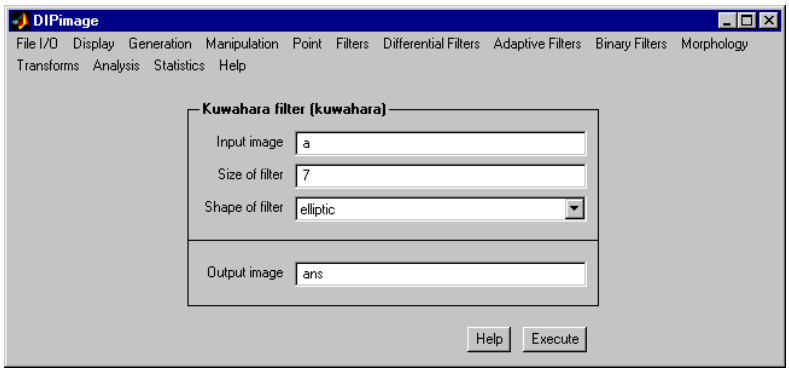
It is possible to use the result of a function call as a parameter in another function:

```
a = max(max(a,b), max(c,d));
```

3 DIPimage

DIPimage is the toolbox we will be using under MATLAB to do image processing. This section takes you through the most relevant features.

☝ You may have noticed the windows that appeared around the screen when you started MATLAB. The one on the top-left is the GUI (Graphical User Interface). The other windows are the ones used to display the images in. The GUI contains a menu bar. Spend some time exploring the menus. When you choose one of the options, the area beneath the menu bar changes into a dialog box that allows you to enter the parameters for the function you have chosen:



There are two ways of using the functions in this toolbox. The first one is through the GUI, which makes it easy to select filters and its parameters. We will be using it very often at first, but gradually less during the course of the afternoon. The other method is through the command line.

When solving the problems in this laboratory session, we will be making text files that contain all commands we used to get to the result (we will call them exercise command files). This makes the results reproducible. It will also avoid lots of repetitive and tedious work. We recommend you make a new file for each exercise, and give them names that are easy to recognize.

If you start each file with the commands

```
clear
dipclf
```

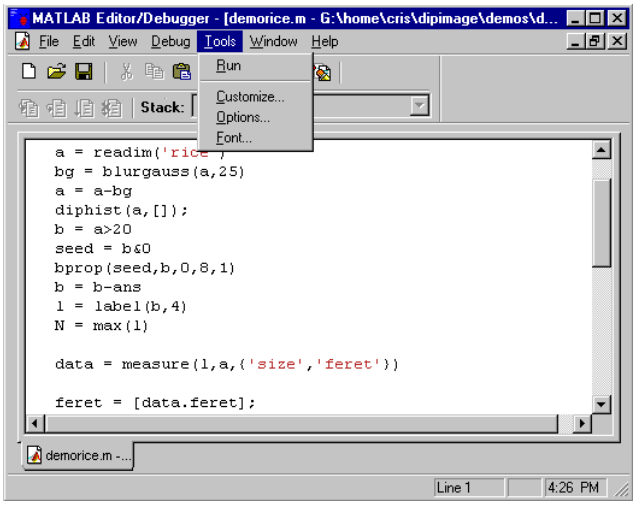
then the variables and the figure windows will be cleared before your commands are executed. This avoids undetected errors.

Edit a MATLAB command file

To open the editor, type

```
edit
```

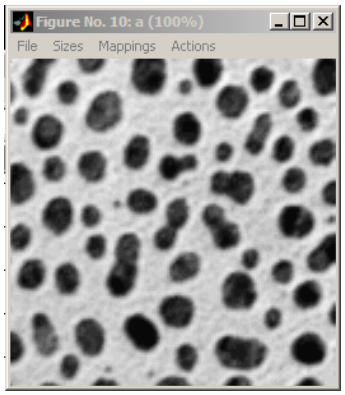
The MATLAB editor will be started. We will type (or copy/paste) the commands we want to execute in it. There is a "Run" menu item under the "Tools" menu. It can be used to let MATLAB run the file currently being edited. You can also execute the script by typing its name in the MATLAB prompt.



4 Loading and displaying an image

We need to load an image (from file) into a variable before we can do any image processing. The left-most menu is called "File I/O", and its first item "Read image (readim)". Select it. Press the "Browse" button, and choose the file **`cermet.ics`**. Change the name of the output variable from **`ans`** to **`a`**. Now press the "Execute" button. Two things should happen:

1) The image 'cermet' is loaded into the variable **`a`**, and displayed to some figure window:



2) The following lines (or something similar) appear in the command window:

```
» a = readim('c:\matlab\toolbox\dipimage\images\cermet.ics','')
Displayed in figure 10
```

This is to show you that exactly the same would have happened if you had typed that command directly in the command window. Try typing this command:

```
b = readim('cermet')
```

The same image will be loaded into the variable **`b`**, and again displayed in a window. Note that we omitted the **`'.ics'`** extension to the filename. **`readim`** can find the file without you having to specify the file type. We also didn't specify the second argument to the **`readim`** function, since **`''`** denotes the default value. Finally, by not specifying a full path to the file, we asked the function to look for it either in the current directory or in the default image directory.

Copy the command as printed by the GUI into the editor (Windows: select with the mouse, Ctrl+C to copy the text; go to the editor, Ctrl+V to paste. Unix: select with the mouse, go to the editor, and click with the middle mouse button to paste.)

To suppress automatic display of the image in a window, add a semicolon to the end of the command:

```
a = readim('qdna1');
```

Note that the contents of **`a`** changed, but the display is not updated. To update the display, simply type:

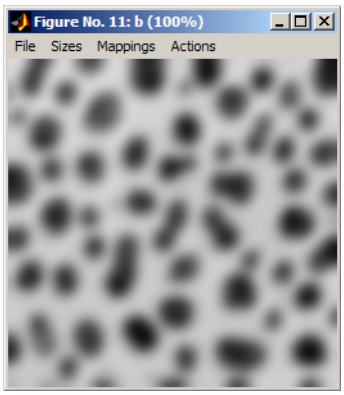
```
a
```

5 Image filtering

The "Filters" menu contains a series of image filters. A filter is a neighborhood operations in which the output value is a function of the values in a local window around the current pixel.

Blurring filters

Choose "Gaussian filter". The name between parentheses on the menu indicates the name of the function that implements this filter. The required input image should already reside in one of the variables, e.g. **`a`**. Type any name for the output image, for example **`b`**. Now we need to choose the size of the Gaussian filter: the standard deviation in pixels. Try out different values for it, and see what happens.



Also explore the "Uniform filter". What is the difference with the "Gaussian filter"? What parameters would you choose to make the result of the uniform filter similar to the result from the Gaussian filter? Why can't you make the results exactly the same?

Make sure you copy some of the function calls you make to your exercise command file.

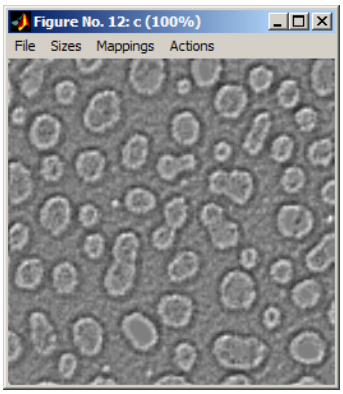
It is possible to choose a different filter size parameter for the horizontal and vertical blur. This can be accomplished by separating the two values with a comma, and surrounding the whole thing with square brackets: **`[4,10]`**. This is the way that arrays are constructed in MATLAB. The first value will be used in the *x*-direction, and the second one in the *y*-direction.

Derivative filters: gradient and Laplace

The menu "Differential Filters" contains a general derivative filter (**`gauss_derivative`**), and a complete set of first and second order derivatives. There are also some more filters based on combinations of derivatives, such as **`laplace`** and **`gradmag`**.

As you have learned today, the Laplace operator is a general second derivative. Let's make one based on the basic derivatives only. This will allow us to show you how to compute with images. First we need the second derivatives in the *x* and *y*-directions. Put them in variables named **`a_xx`** and **`a_yy`** (any name is as good as the other, isn't it?).

You will notice that the second derivative does not yield very high grey-values. The display looks black. By default, images are displayed by mapping the value 0 to black, and the value 255 to white. Since all grey-values in this particular image are (approximately) between -30 and 30, this mapping is not adequate. We can change this mapping through the menus of the figure windows. Open the "Mappings" menu. It contains a couple of options that allows you to change the display mapping. Choose "Based at 0". This mode will map the image values linearly between black and white, forcing the value 0 to 50% grey. Try out the other modes too.



Now we need to add these two derivatives together. This is accomplished with the command

```
b = a_xx + a_yy
```

(Note that there is no menu item for adding two images). Images can be subtracted, multiplied and divided in a similar way. It is also possible to use a constant value instead of either image.

Let's compare the result with the Laplace operator (**laplace**). Put its result in **c**. Now compare **b** and **c** by subtracting the two images. Since both are equal, the result is completely black. But we want to make sure that the difference is zero everywhere, and not just very small.

The "Actions" menu allows you to specify the mouse action on the figure window. Select "Pixel testing", and press the mouse button while pointing somewhere in the image (keep the button down). The figure caption changes to show the coordinates of the mouse in the image and the value of the pixel at those coordinates. Try moving the mouse while holding the button down, and check that the values are all indeed zero. Try this out on another image too.

Another option on the "Actions" menu is used to zoom in on an image. Try it out too.

Make sure you have copied every command you executed to the exercise command file.

(Un)sharpening

Now we will sharpen the image 'qdna1', load it in variable **a** by the command:

```
a = readim('qdna1')
```

Unsharp masking has been defined today as the original image minus the Laplace of the image. We can write this very easily using only the command line:

```
a - laplace(a)
```

The answer is put into the variable **ans**.

Note that unsharp masking gets its name from a procedure employed by photographers long before the days of computers or image processing. What they used to do was print an unsharp version of the image on film, and use that to mask the negative. The two combined produced a sharper version of the photograph. The trick is that the unsharp print masks the low-frequency components, but not the high frequencies; the procedure thus implements a high-pass filter. Let's reproduce that trick with our 'cermet'. Type this:

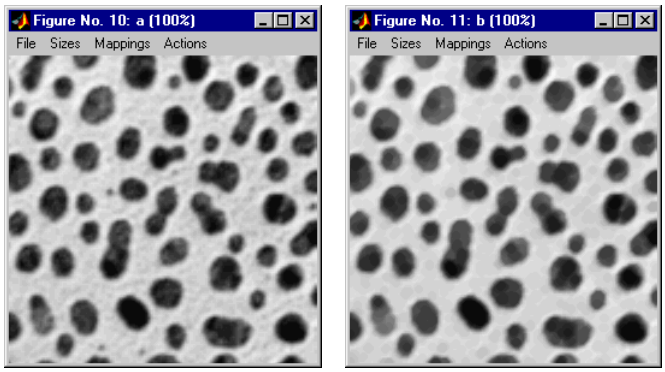
```
2*a - gaussf(a)
```

By multiplying the image by two, we multiply both the low and high frequencies. The low frequency components are then subtracted again, thus remaining in their original intensity. Only the high-frequency components are effectively multiplied by two.

The two unsharp filters ("**a-laplace(a)**" and "**2*a - gaussf(a)**") are not the same. In what do they differ? How are they alike?

Local maximum and minimum filters

The local "minimum and maximum filters" (**minf** and **maxf**) assign respectively the minimum or the maximum value of a local window to the current pixel in the output image. The general names for these operations are erosion and dilation, and can be found as such under the "Morphology" menu. Dilation is a local maximum filter, erosion is a local minimum filter, opening is the sequence erosion and dilation, and closing is the inverse sequence. For example, try a closing with different size parameter on the image 'cermet' (which is a microscopic image with some dark objects in it). Note how the image is made smoother, without making the object edges less sharp. Also, note how the smaller objects are removed from the image.



Try the other filters on the same image and watch the size of objects. Try different filter sizes as well.

Using max/min filters (equivalent to **dilation**/**erosion**) we will construct a morphological gradient that will be modified into a morphological sharpening operation. Load the image 'cermet' into variable **a**. Subtract the original from the dilated image and put the result in **b**. Subtract the eroded image from the original and put the result in **c**.

```
a = readim('cermet')
b = dilation(a) - a
c = a - erosion(a)
```

Both results detect abrupt changes in grey-level. Notice the subtle difference in the eyes. (Remember to write all commands in an exercise command file.)

6 Point operations

There exist a large number of monadic point operations that are only accessible through the command line. They act on the image pixel-by-pixel. Examples are the mathematical functions:

```
sin, cos, tan, atan2, etc.
abs, angle, real, imag, conj, complex, etc.
log, log10, log2, exp, sqrt, etc.
```

Histogram-based operations

The function **diphist** (under the "Statistics" menu) plots the histogram of an image. Open a new image: 'sca'; in this image are bloodcells of a sickle cell anaemia patient shown. Plot the histogram of the image 'sca'. You will notice that the lower 98 grey-values are not used, as are the upper 4. Correct this using the function **stretch** (under the "Point" menu). To see the difference with the original image, make sure that the display mode is set to "Normal". Plot the histogram of the new image.

This stretching method is very sensitive to noise. For example, set a single pixel in the original image **a** to 10. This can be accomplished by indexing. It is not very important how this works exactly, since manipulating an image in this way is not part of the course. Type:

```
a(0,0) = 10
```

Now plot the histogram again. You will *not* notice the difference. However, the stretching algorithm will. Plot the histogram of the stretched image to see this. Why is the lower part of the histogram flat?

Repeat the previous sequence of commands with the lower and upper percentiles in the stretch function 1 and 99 respectively. This causes the lower and upper 1% of the grey-values to be clipped before stretching.

The histogram of 'sca' is also very poorly distributed. This is a feature of most images. It simply means that some grey-values occur more often in the image than others. Sometimes this is not desirable, for example when comparing images acquired under different lighting circumstances. Apply the **hist_equalize** function (also on the "Point" menu) to the image '**sca**', and then plot the histogram again. Is in this image a stretch or an equalization of the histogram preferable?

```
a = readim('sca')
b = hist_equalize(a)
```

Repeat the previous sequence of commands with the image 'maan', one of the early images of our moon.

```
a = readim('maan')
b = hist_equalize(a)
```

Plot the histogram before and after histogram equalization. Is in this image a stretch or an equalization of the histogram preferable?

Thresholding

Load and stretch the image 'sca' again into variable **a**. The objects in it are clearly defined and are easy to segment. The **threshold** function contains a couple of different algorithms to do this. The simplest one, 'fixed', requires you to provide a parameter: the threshold level. The other three estimate this parameter based on the histogram of the image in different ways. For this image, the parameter is not very critical; let's use 100. The red portion of the segmented image is the object and the black is the background. On this image it goes all wrong: the background has become the object, and the particles have become 'holes' in the object. That is because grey-values larger than the threshold are considered as part of the object. One solution is to invert the image before or after thresholding. This can be done with the negation operator for the binary image (**~threshold(a,'fixed',100)**), or the minus

operator for the grey-value image (**`threshold(-a,'fixed',-100)`**, note how the threshold value should also be changed). Another way is to change the thresholding operation. We can use the relational operators to do thresholding (**`<`**, **`>`**, **`==`**, **`~=`**, etc). In our case:

```
b = a < 100
```

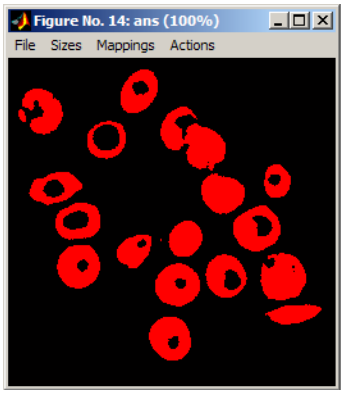
Now we have a binary image, recognizable by the red coloring of the display. This leads us to binary morphology.

7 Binary morphology

There are dedicated operations for binary images. The point operations 'not' (**`~`**), 'or' (**`|`**), 'and' (**`&`**), 'xor' (**`xor`**) can only be issued directly onto the command line. The binary morphological filters can be found under the "Binary Filters" menu.

For example, we can use the "Binary Opening" (**`bopening`**) to remove small objects. In the same way, we can use **`bclosing`** to remove small holes in the objects. What is the effect of the "Edge condition"? What connectivity gives the best results for this image? What is the effect of the specified number of iterations?

Now load the image 'sca', stretch and threshold it (<175). Make sure that the thresholding operation retains the whole objects. The cells, which are not fully inside the picture should be removed. This can be done by the function remove edge objects (under "Binary filters").



We will use the skeleton operations to separate the healthy from the sickle cells.

Use the **`bskeleton`** function (under the "Binary Filters" menu) to create a skeleton of the objects. What is the influence of the 'Edge Condition'? What does 'End-Pixel Condition' control?

Since most of the normal cells have a reflection, which is also visible in the thresholded image, the skeleton image shows a circle for a normal cell. A sickle cell will show a straight line.

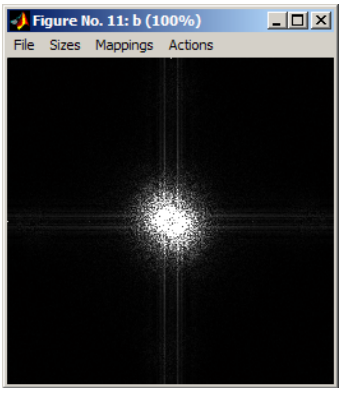
The picture reveals one straight line, several ellipses and some curved lines (originating from two cells overlapping).

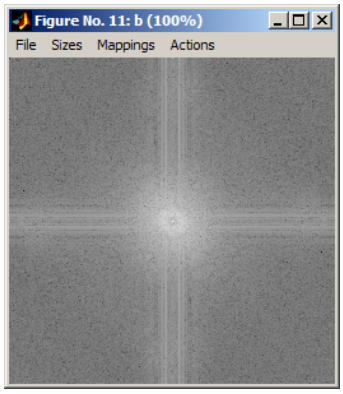
8 The Fourier transform

Load the image 'qdna1' into variable **a**. Under the "Transforms" menu, you will find the forward and inverse Fourier transforms. Apply the forward transform to the image in variable **a**, and store the Fourier spectrum in variable **b**.

```
b = ft(a)
```

The result looks like a white cloud in a black background; this is because the default display mapping is not the most adequate. Try linear stretching. Now all you have left is a single dot in the middle. The dynamic range is very large. Logarithmic stretching is usually employed to look at Fourier spectra.





Now switch on the "Pixel testing" mode and look at the values in the spectrum. The values are complex. What you see as an image is just the amplitude of the spectrum. Now we will decompose it into a real and an imaginary part. We will do this through the **real** and **imag** commands (these are not in the menus). Write these and the previous commands into a command file (we assume that b is the Fourier spectrum):

```
re = real(b)
```

```
im = imag(b)
```

You will notice that the image **im** contains real values. We need to multiply it by **i**. If you overwrote variable **i** with an image, you can use **j**. If you also overwrote it, clear them with

```
clear i j
```

This will return them to their original use, the imaginary number $\sqrt{-1}$. Now write

```
im = i*imag(b)
```

According to the theory, the inverse transform of **re** should be the even component of the original image, and the inverse transform of **im** should be the odd component. Furthermore, both should be real. However, the inverse transforms are not real, but complex. By examining the images, you can see that the imaginary parts are negligible small. Remove them using the **real** function again.

```
real(ift(real(b)))
```

```
real(ift(i*imag(b)))
```

Now the displayed images are not the amplitude of the complex images, so negative values are visible. Make sure the images are truly even and odd, then add them up and compare with the original image. Where does the difference come from?

Another way of separating a complex image is in amplitude and phase. The amplitude is acquired using **abs**, the phase using **angle**. However, the phase itself is not too interesting. Far more interesting is **exp(i*angle(b))** (we will call this the phase term). We can also calculate this by dividing the original spectrum by its amplitude. Now compute the inverse transform of the amplitude and the phase term (make sure you take the real part of the inverse transform, not the magnitude!).

```
real(ift(abs(b)))
```

```
real(ift(exp(i*angle(b))))
```

Which one contains more information? What does the transform of the phase term look like? How do we get the original image back? Try

```
real(ift(abs(b)*exp(i*angle(b))))
```

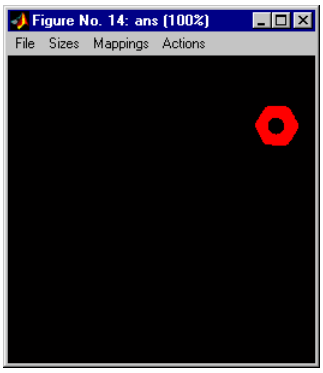
9 Measurements in images

In section 7 we tried to distinguish nuts from bolts using binary morphology. Here we will do the same exercise by measuring different object sizes, like the area, perimeter and lengths. Load the 'nuts_bolts1' image again. Threshold it. Now find the function **label** in the "Transforms" menu. What this function does is give each separate object in the image a different grey-value. What objects are considered separate can be controlled by the 'Connectivity' parameter. Use the "Pixel testing" mode on the result to check what values each object has. To extract object number 3 from the image, we can now do (assuming **la** is the label image):

```
la == 3
```

Note the double equal sign. This is the equality operator.

There is a special display mode to look at labeled images. Turn it on. Now each object is displayed in a different color. This makes it easy to see if objects have been correctly separated or not. Note that there are only a small number of different colors. If there are more objects, some will share a color.



Now we are ready to do some measuring. Select the **`measure`** function in the "Analysis" menu. The 'Object image' is the labeled image **`la`**. The 'Grey-value image' is the original image before segmentation; it won't be used by the measurements we will make, so it can be left out. Select **`'size'`** as the measurement. If you leave 'Object IDs' empty, all objects will be measured. Put the output in a variable called **`data`**. Now

```
sz = data.size
```

is a MATLAB array with the sizes of the objects. Now type

```
diphist(sz,[1,2000],100)
```

This will create a histogram for the sizes. There are obviously two size categories. Let's say that sizes up to 1000 are for the nuts, and larger sizes for the bolts. There exist ways of doing this automatically, but we won't go into them now. We will use the function **`mrs2obj`** to 'paint' each object with their measured size. Choose the label image as the input, and **`data`** as the measurement data. We can now threshold this image at the chosen value of 1000 to retrieve the bolts. The nuts can be obtained by **`xor`**-ing the original binary image and the bolts image:

```
nuts = xor(b,bolts)
```

We could do the same thing with other measurements of the objects, like the length (**`'feret'`**), the size of bounding box (**`'dimension'`**), or the perimeter (**`'perimeter'`**). Try them out.