CSP: Computer Service Labs
Image Analysis
AP3471

Cris L. Luengo Hendriks
Lucas J. van Vliet
Marjolein van der Glas

September 3, 2003

Contents

1 Introduction **4**

1.1 MATLAB 4

1.2 DIP*image* 5

1.3 Editing a MATLAB Command File 6

1.4 On Sampling and Images 6

2 Getting Started **8**

2.1 The Graphical User Interface 8

2.2 The Display Window 9

3 Basic Filtering **11**

3.1 Smoothing (Blurring) Filters 11

3.2 Derivative Filters: Gradient and Laplace 12

3.3 Rank-Value Filters 13

3.4 Sharpening 13

3.5 Filtering Applications 14

4 Point Operations **16**

4.1 Histogram-Based Operations 16

4.2 Thresholding 17

4.3 Other Point Operations 18

5 The Fourier Domain **19**
5.1 The Fourier Transform 19
5.2 Filtering in the Fourier Domain 21
5.3 Shift Estimation *(advanced)* 23

6 Binary Image Processing **24**
6.1 Neighborhood Relations 24
6.2 Binary Morphology 24
6.3 Selecting Objects 25

7 Morphology **28**
7.1 Morphological Filtering 28
7.2 Other Morphological Tools 30

8 Image Manipulation **31**
8.1 Coordinate System 31
8.2 Region Extraction 32
8.3 Transformations 33
8.4 Test Image Generation *(advanced)* 33

9 Measurements **35**
9.1 Extracting Single Objects 35
9.2 Measuring in Binary Images 35
9.3 Errors Introduced by Binarization 37
9.4 Measuring in Grey-Value Images *(advanced)* 38

10 Vector Images *(advanced)* **41**
10.1 Vector Image Operations 41
10.2 Color Spaces 42
10.3 Filtering Color Images 45

11 Adaptive Filtering *(advanced)* **48**
11.1 Kuwahara 48
11.2 Other Adaptive Filters 49

12 Other Advanced Topics *(advanced)* 50

12.1 Scale-Spaces 50
12.2 Hough Transform 51
12.3 Watershed Transform 54

A List of functions and operators 56

A.1 Functions 56
A.2 Mathematical Operators (grey in, grey out) 59
A.3 Statistical Operators (grey in, single value out) 60
A.4 Logical Operators (binary in, binary out) 60
A.5 Comparison Operators (grey in, binary out) 60
A.6 Tricks 61

1 Introduction

The goal of this laboratory course is to get hands-on experience with image processing. To do so, you will have to learn the image-processing environment: MATLAB and the DIP*image* toolbox for scientific 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.

First we want to stress one point: Try to understand what is happening, and do not be satisfied with just any answer you obtain! It is by far better that you seriously wor on half of the exercises and learn some image processing, than that you go through all of them, but understand nothing. This is a waste of your time.

book: FIP

In this laboratory manual, a lot of details about the theory have been left out. For these we refer to :

- Young, I.T., Gerbrands, J.J. and van Vliet, L.J., “The Fundamentals of Image Processing”, Department of Applied Physics, TUD.

It is available online (see `http://csp.tn.tudelft.nl/tn3531.html`) and at the *Dictatenverkoop*. On the margin of some paragraphs we specify a section of FIP you need to read before proceeding with that paragraph. Additionally, you can check-out from the library (or better: buy!) a book on image processing for a more in-depth discussion. We recommend:

- Jähne, B., “Digital Image Processing: Concepts, Algorithms, and Scientific Applications”, Springer, 1997.

Some sections, subsections and exercises are marked ‘advanced’. You are entitled to skip these if you are not going to do a Master’s Thesis on Image Processing at the Pattern Recognition Group. Even if you are not, you can study these advanced topics if you like. We recommend, however, that you do so only if you have finished all the compulsory exercises first.

1.1 MATLAB

This subsection is to re-acquaint you with MATLAB; if you use MATLAB regularly, skip this section. If you have never worked with MATLAB before, there is a small introduction you should ask for, which will take about one afternoon.

A MATLAB variable can contain anything from a single value (scalar) to a very complex data structure. However, the most common structure is an array. An array is typically two-dimensional (but higher-dimensionalities are also supported), with each element being a double-precision floating-point number. They serve as parameters to function calls, and unary and binary (== dyadic) operations can be applied to them.

```
>> a = b;
```

will cause whatever is in variable `b` 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). Similarly,

```
>> a = [100,200;50,0];
```

will assign a matrix into `a`. The square brackets catenate elements into an array; the comma (or space) separates elements horizontally, and the semi-colon vertically.

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));
```

This allows for complex operations involving any kind of functions and operators:

```
>> c = max([a/3+b/4,c-min(a,b)*4],0);
```

1.2 DIP*image*

DIP*image* is the toolbox we will be using under MATLAB to do image processing. At this time, we will review only the most relevant features. You will get to know this environment better throughout the course.

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 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 (Figure 1). Most of these functions correspond to image processing tasks, as we will see later.

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. The other method is through the command line.

When solving the problems in this laboratory course, we will be making text files (called scripts) that contain all commands we used to get to the result. 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

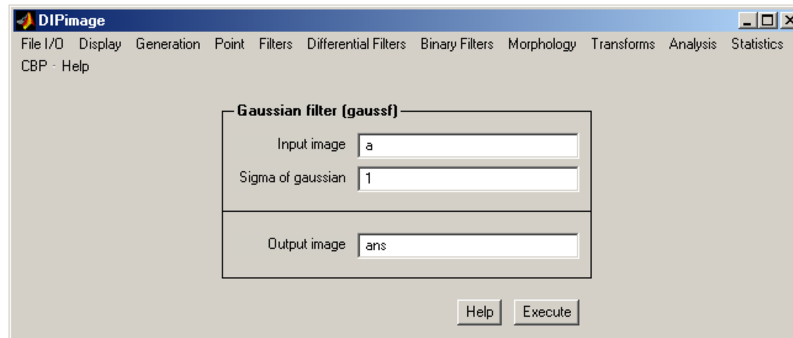


Figure 1: The DIPimage GUI.

to recognize. The file names should end in '.m', and should not conflict with existing function or variable names. 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.

If an element of the toolbox is not explained clearly enough in this manual, refer to the DIP*image* user guide:

- Luengo Hendriks, C.L., et. al., "DIP*image* User Guide", Delft, 2001.
  `http://www.ph.tn.tudelft.nl/DIPlib/docs/dipimage.pdf`

1.3 Editing a MATLAB Command File

To open the editor, type

```
>> edit
```

The MATLAB editor will be started (see Figure 2). We will type (or copy/paste) the commands we want to execute to the editor, and run the whole thing as a script. To do this, first save the file and then type its name on the MATLAB command prompt. Make sure the file name ends in '.m', but do not type this extension on the command line.

There is a "Run" menu item under the "Tools" menu. It can be used to let MATLAB run the file currently being edited.

1.4 On Sampling and Images

book: 5

During this course we will assume that images are correctly sampled, and the samples thus represent the underlying continuous, band-limited image completely. Note that some image processing operations break this assumption, which we need to take into account.

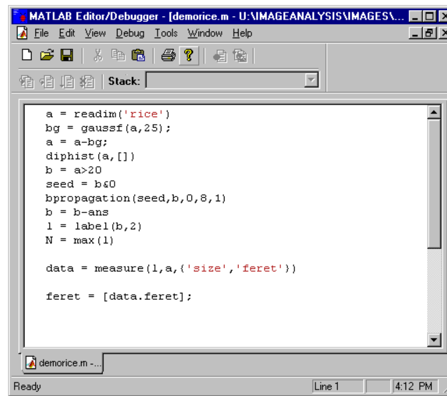


Figure 2: The MATLAB M-file editor

Each sample in a two-dimensional image is called a *pixel* (for PICTure ELement). For three-dimensional images the term *voxel* is often used (for VOlume ELement, with an extra X thrown in for good measure).

Besides spatial sampling, the sample values are also quantized. Very often pixel values are stored in an 8-bit integer, meaning that there are 256 different possible values for a pixel. This is enough for display purposes, since we cannot distinguish more than 64 or 128 different grey-values. However, for computation purposes, this is often not enough. Most operations and filters we will be using during this course produce images with floating-point pixel values. This is completely transparent to the user, and conversions from one data type to another do not need to be done explicitly.

2 Getting Started

This section will introduce the main elements of the user interface, which is composed of a GUI (with a menu system that contains image processing and analysis functions, and a body that changes to enable you to enter the parameters for these functions) and image display windows. Not all features will be shown to you at once, many elements will be introduced, as you need them, during the course. This is to avoid tedious enumerations that you won't be able to remember anyway. To get a complete description of the user interface we refer to the DIP*image* manual.

2.1 The Graphical User Interface

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

1. The image 'trui' is loaded into the variable `a`, and displayed to a figure window (see Figure 3).
2. The following lines (or something similar) appear in the command window:

```
>> a = readim('x:\c_ip\images\trui.ics','')
Displayed in figure 10
```

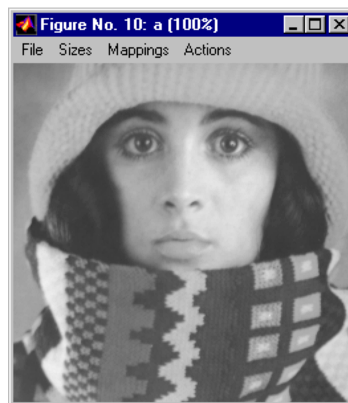


Figure 3: Display window with image 'trui' loaded.

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('trui')
```

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 '' (empty string)) 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('cermet');
```

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

```
>> a
```

2.2 The Display Window

You will have noticed that the image in variable `a` is always displayed in the top-left window. This window is "linked" to that variable. Likewise, variables `b` through `d` and `ans` are linked to a window. Images in all other variables are displayed to the sixth window. This can be changed, see the DIP*image* manual for instructions.

A grey-value image is displayed by mapping each pixel's value in some way to one of the 256 grey-levels supported by the display (ranging from black (0) to white (255)). By default, pixel values are rounded, negative values being mapped to 0, and values larger than 255 to 255. This behavior can be changed through the "Mappings" menu on each figure window. We will be using this menu very often, so try out the options now.

- Normal: the default mode, as explained above.
- Linear stretch: the lowest grey-value is mapped to 0, the highest to 255, and the other values are mapped linearly in between.
- Percentile stretch: 5% of the pixels with the lowest grey-values are all mapped to 0, and the highest 5% to 255; all other values are mapped linearly in between.
- Log stretch: The logarithm is applied to the values before linear stretching, thus improving discriminability of low grey-values in the presence of very high grey-values; this mode will be used when discussing the Fourier Transform.
- Based at 0: 0 is mapped to 128 (50% grey-value), and the rest is stretched linearly to fit.
- Angle: $-\pi$ is mapped to 0, and π to 255.
- Orientation: $-\frac{\pi}{2}$ is mapped to 0, and $\frac{\pi}{2}$ to 255.
- Labels: after rounding, each grey-value is assigned a color; we will use this mode later when discussing labelling.

Note that these mappings only change the way an image is displayed, not the image data itself.

Another menu on the figure window, “Actions”, contains some tools for interactive exploration of an image. The two tools we will be using are “Pixel testing” and “Zoom”. “Pixel testing” allows you to click on a pixel in the image (hold down the button) to examine the pixel value and coordinates. Note that you can drag the mouse over the image to examine other pixels as well. The right mouse button does the same thing as the left, but changes the origin of the coordinate system and also shows the distance to the selected origin (right-click and drag to see this).

The “Zoom” tool is to enlarge a portion of the image to see it more clearly. Click to double the pixel size, double-click to return to the original size. Click and drag to select the region to be enlarged.

3 Basic Filtering

book: 3.1 9.3

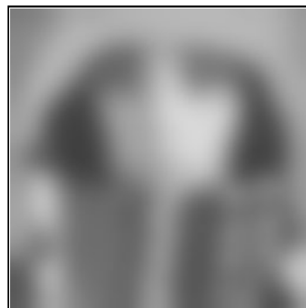
Filters are a set of tools available to process images. Basically, there are two types of filters: linear and non-linear filters. Linear filters can be implemented through a convolution, non-linear filters can not. Therefore, linear filters are easier to implement, and are important for Fourier Analysis. All filters discussed in this section (both linear and non-linear) can be implemented as a combination of the values in a neighborhood; that is, each pixel in the output image is computed out of a fixed set of pixels in the neighborhood of the corresponding pixel in the input image. Typical neighborhood shapes are round (`'elliptic'`) or square (`'rectangular'`). In the case of linear filters, the output value is a linear combination of the input values. We will also see some examples of non-linear filters.

Some linear filters, like the Gaussian filter, do not have a neighborhood shape as parameter. This is because their shape is fixed (i.e. a Gaussian kernel).

3.1 Smoothing (Blurring) Filters

book: 9.4

Find and select "Gaussian Filter (`gaussf`)" in the menu of the GUI. 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. The filter size (scale) is very important, as will be shown in Section 12.1.



Exercise 1: The uniform filter

Now try the uniform filter (`unif`). What are the similarities between the uniform filter and the Gaussian filter? In what do they differ?

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

It is possible to choose different horizontal and vertical sizes for the filters. This can be accomplished by separating the two values with a comma, and surrounding the whole thing with square brackets: `[4,10]` (a MATLAB array with two values). The first value will be used in the x-direction, and the second one in the y-direction.

3.2 Derivative Filters: Gradient and Laplace

book: 9.5

The menu “Differential Filters” contains a general derivative filter, a complete set of first and second order derivatives, and some combinations like gradient magnitude and Laplace.

Exercise 2: First order derivatives

Try out the first order derivatives (`dx` and `dy`). Why are these called Gaussian derivatives? What is the meaning of the parameter (“sigma of Gaussian”)?

Why is this derivative better than the discrete derivatives $[1\ -1]$ or $\frac{1}{2}[1\ 0\ -1]$ (as numerical approximations to the derivative)?

Look up the Sobel operator. How is it constructed, and how does it compare to the Gaussian derivative?

The Laplace operator is a second derivative. It is used to detect lines, like the gradient magnitude is used to detect edges, and is rotation invariant as well. 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?). The best mode to view these images is “Based at 0”, since this fixes the zero-level to a 50% grey-value, which makes it easy to compare the different derivatives.

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 (see Appendix A). 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`.

dxx

dyy

laplace

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. This can be accomplished in at least three ways:

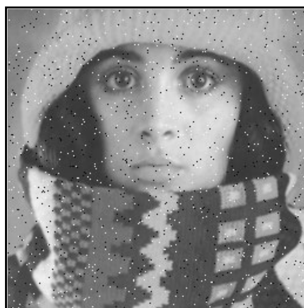
- Using the “Pixel testing” tool under the “Actions” menu in the display window. Click and hold down the left button, and move the mouse over the image, checking that the grey-value is exactly zero everywhere.

- Compare this image with zero (`d ~= 0`) or compare the two images directly (`b ~= c`). Either way a binary image is produced with true (displayed in red) or false (black) as pixel values. These actual pixel values are 1 and 0 respectively; the colors are used to show that it is a binary image, and not a grey-value image. Since this binary image is false everywhere (black), it is not a good example for a binary image. To obtain a binary image with both values, compare the result of the Laplace operator with zero: `c > 0`.
- Use the functions `max` and `min` to determine the maximum and minimum values. Both should be 0.

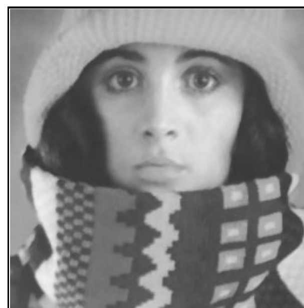
3.3 Rank-Value Filters

A rank-value filter orders all values in the selected neighborhood of a pixel, and takes one of these values as the value for the output pixel (for example the middle one, in case of a median filter). In DIP*image*, rank-value filters are implemented as percentile filters (`percf`). This way, specifying which value to take is done through a percentile value. The 50% value is the median (`medif`), and the 0% and 100% values are the minimum (`minf`) and maximum (`maxf`) respectively. Maximum and minimum filters are the basis of morphology and will be reviewed again in Section 7. The median filter is a non-linear smoothing filter. It is ideal for removing shot-noise (black and white pixels randomly scattered over the image). Compare to the result of the uniform filter, which blurs the image a lot to remove some of this noise.

book: 9.4.2



shot noise



median filtering

uniform filtering

3.4 Sharpening

book: 10.2.1

Now we will sharpen the image 'trui', which should still be in variable `a`; load it again if it got lost. Unsharp masking is often defined 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 the image 'trui'. 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.

Exercise 3: Unsharp masking

These two unsharp filters are not exactly the same. Draw their impulse response (point-spread functions, convolution kernels) to see how they differ. If you know about Fourier analysis, look at what these filters do in the Fourier domain.

Hint: to compute an impulse response, apply the filter to the discrete delta function (unit impulse). It can be generated with `deltaim` (see Section 8).

3.5 Filtering Applications

Exercise 4: Shading removal

book: 10.1

Load the image 'shading'. It contains some text on a shaded background. To remove this shading, we need to estimate the background. Once this is done, we can correct the original image. This is a common procedure, often required to correct for uneven illumination or dirt on a lens.

There are several background shading estimation methods:

- The most used one is the low-pass filter (`gaussf`). Try finding a relevant parameter for this filter to obtain an estimate of the background, then correct the original image.
- Another method uses maximum and minimum filtering. Since the text is black, a maximum filter with a suitable size will remove all text from the image, bringing those pixels to the level of the background. The problem is that each background pixel now also was assigned the value of the maximum in its neighborhood (see Figure 4). To correct this, we need to apply a minimum filter with the same size parameter. This will bring each background pixel to its former value, but the foreground pixels won't come back! This filter is called a morphological closing, and we will see more about it in Section 7. Use this estimate of the background to correct the original image.

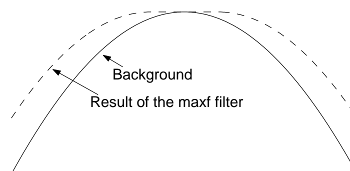


Figure 4: Background shading with result of maximum filter.

Exercise 5: Edge detection

Construct the gradient magnitude using the derivative filters from Subsection 3.2 (don't use the function `gradmag`!). The gradient magnitude is an edge detection filter. Use the image 'trui' to examine the result of your filter, and compare it to `gradmag`.

It is also possible to detect edges using the `maxf` and `minf` functions. For example, `a-minf(a,3)` gives an image similar to the previous one. There is one important difference, though. To see it, combine the two results in a color image:

```
>> joinchannels('RGB',stretch(b),stretch(c))
```

The result of this operation is a color image. Image `b` is the red component, and image `c` the green component. Assuming `b` and `c` contain the edges estimated with the two methods, red will show edges as estimated with the one method, and green the ones with the other one. Where they overlap, it will show in yellow.

Note how the edge estimates are not aligned. Why is this? Combine the results of the `maxf` and `minf` filters in such a way that the estimated edges are aligned with the ones produced by `gradmag`.

4 Point Operations

4.1 Histogram-Based Operations

book: 3.5.2

A histogram is a distribution of image grey-values. It is computed by counting the number of occurrences of each grey-value. The histogram gives global information on the image contents, and is used by certain algorithms, for example to determine a threshold that distinguishes objects from background (see Subsection 4.2).

book: 9.1

The function `diphist` (under the "Statistics" menu) plots the histogram of an image. Plot the histogram of the image 'trui'. You will notice that the lower 55 grey-values are not used, as are the upper 14. 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, which will be explained in more detail in Section 8. 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?

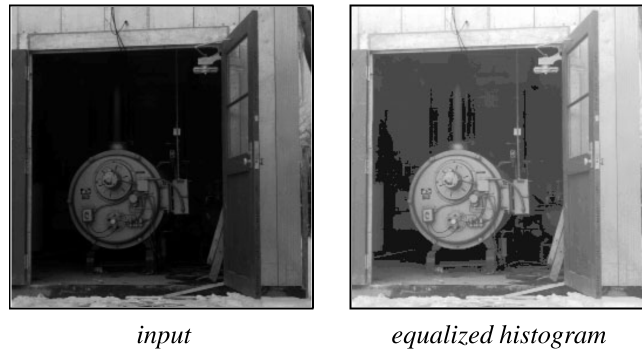
Exercise 6: Percentile stretch

Repeat the previous sequence of commands (you will need to read in 'trui' again) 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.

Most images have a very poorly distributed histogram. This simply means that some grey-values occur more often than others in the image. Sometimes this is not desirable, for example when comparing images acquired under different lighting circumstances. The `hist_equalize` function (also on the "Point" menu) works on the histogram to flatten it. For images with quantized pixel values, this is not possible, but the algorithm makes an approximation.

Exercise 7: Histogram equalization on quantized images

Apply `hist_equalize` to the image 'ketel'. Plot the histogram of this image before and after the histogram equalization. How does the algorithm solve the problem of the quantized grey-values?



input *equalized histogram*

4.2 Thresholding

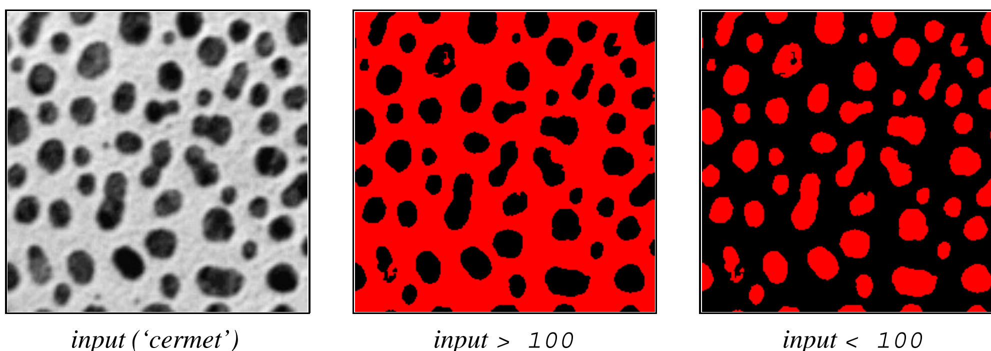
book: 10.3.1

Load the image 'cermet' again into variable `a`. The objects in it are clearly defined and are easy to segment. The "Point" menu contains the function `threshold`, which assigns 'object' or 'background' to each pixel depending on its grey-value. This is a very simple form of segmentation, but it is good enough for this image and many others. As the name indicates, a threshold is chosen. Pixels with a value above this threshold are considered part of the object (foreground). The output image is binary.

To select a threshold, there are several algorithms available (which use the histogram). We don't need them for this image. Select 'fixed' for the 'Type' parameter; this requires you to provide a threshold. Choose 100. The resulting image contains one object (red), with holes in it (black). That is because the objects we were looking for are dark instead of light. We can invert the image before thresholding to correct this, or invert it afterwards (look in Appendix A on how to do this). Another solution is to do the thresholding in an alternative way: using relational operators. Recall that thresholding returns an image with 'true' (1) where the image is larger than some threshold. This can be accomplished with the 'larger than' (>) operator:

```
>> b = a > 100
```

Since we want to find the dark objects, use the 'smaller than' (<) operator instead.



input ('cermet') *input* `> 100` *input* `< 100`

The thresholding algorithms available in the function `threshold` provide automatic ways of selecting a threshold. Each of them makes an assumption on the grey-value distribution of the image. See the book for more details.

We will be using thresholding in Sections 6 and 8.

4.3 Other Point Operations

book: 9.2

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`, etc.
- `abs`, `angle`, `real`, `imag`, `conj`, `complex`, etc.
- `log`, `log10`, `log2`, `exp`, `sqrt`, etc.
- `sign`, `round`, `floor`, `ceil`, etc.

There are also a large number of dyadic point operations, which act on two images pixel-by-pixel, or on one image and a constant. Examples:

- `min`, `max` etc.
- `atan2`, `mod`, `+`, `-`, `*`, `/`, etc.
- `==`, `~=` , `>`, `<`, etc.
- `&`, `|`, `xor`, etc.

5 The Fourier Domain

book: 3.3

Before you start this section, look up 'Fourier Transform' in your book. Note that the Discrete Fourier Transform (DFT) is not the same as the continuous version: since the image is sampled, the Fourier Transform is periodic, but the Fourier Transform itself must also be sampled (hence *Discrete*), thus the image must be assumed periodic too.

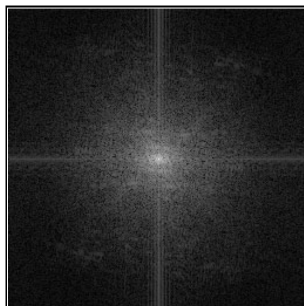
In DIP*image*, the FT is symmetric, in the sense that the normalization factor for the FT and the Inverse FT (IFT) are the same. In many text books this is not the case; they normalize the FT by dividing by N (the number of pixels), and don't normalize the IFT. In DIP*image* both are normalized by dividing by $\sqrt{N}$. This is important only when generating a filter in one domain, and using it in the other.

5.1 The Fourier Transform

Load the image 'trui' 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 stretch". Now all you have left is a single dot in the middle. The dynamic range is very large. "Log stretch" is usually employed to look at Fourier spectra.



The origin is at pixel location `floor(N/2)`, with `N` the number of pixels in one direction. This means it is just to the right of the center if N is even, and in the exact center if it is uneven. Coordinates are in the range $[-\frac{1}{2},\frac{1}{2})$, but $-\frac{1}{2}$ is not sampled if N is uneven (because $\frac{1}{2}$ would have to be sampled too). Don't confuse image coordinates (discrete pixel locations) with Fourier coordinates (frequencies). Note that DIP*image* uses the frequency f instead of radial frequency ω (as used by many books). These coordinate issues are important for example when using functions such as `xx` or `rr` with the parameter `'frequency'`.

Remember that the discreet Fourier spectrum is periodic (with period 1).

Now switch on the "Pixel testing" mode and look at the values in the spectrum (you might need to enlarge the window so that the values fit in the title bar). The values of

the Fourier image are complex. What you see as an image is just the amplitude of the spectrum. Under the “Mappings” menu you will now find a new section that controls how you look at the complex values. The default mode is “Magnitude”. The other modes are to look at the phase (angle), real and imaginary parts. Note that logarithmic stretching is not as useful for the real and imaginary parts, since these have both very large positive and negative values. This display mode is intended for use with the magnitude. The “Phase” mode is best used together with the “Angle” mode.

Exercise 8: Manipulating complex images (part I)

To decompose `b` into a real and an imaginary part, use the `real` and `imag` commands (these are not in the menus). Write these and the previous commands into a command file:

```
>> 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. Now write

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

book: 3.4

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 (look this up!). 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 negligibly small (they are due to round-off errors in the transforms). Remove them using the real function again.

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

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? Is it significant?

Exercise 9: Manipulating complex images (part II)

Another way of separating a complex image is in amplitude and phase. The amplitude is acquired using `abs`, the phase using `angle` or `phase`. However, the phase itself is not too interesting. Far more interesting is `exp(i*angle(b))` (let’s call it the ‘phase term’), because of its properties. Dividing the original spectrum by its amplitude also results in this phase term. Now compute the inverse transform of the amplitude and the phase term (make sure you look at the real part of the inverse transform, not the amplitude!). Which one contains more information?

To which filter is the inverse transform of the phase term similar, and how do these two differ?

5.2 Filtering in the Fourier Domain

A convolution between an image and a filter in the spatial domain corresponds to a multiplication of their Fourier transforms in the Fourier domain. This means that all linear filters (which are the ones that can be implemented by convolution) can be computed in the Fourier domain with a simple multiplication. Simple filters in the spatial domain (like the uniform filter) are very complex in the Fourier domain. To see this, make a 1D image with a single pixel set, and apply the uniform filter (use a large kernel); this results in the convolution kernel of the filter: the point-spread function (PSF). After applying the Fourier transform, we get an image with a sinc-like function (a large lobe in the middle, with ripples up to the edge of the image).

```
>> a = newim(200); % see the section on manipulation
>> a(100) = 255
>> a = unif(a,20)
>> b = ft(a)
```

Conversely, an ideal low-pass filter (a box in the Fourier domain) is a sinc-like function in the spatial domain. This ideal low-pass filter is used in the next exercise.

Exercise 10: Sub-sampling

Load the image 'trui' in `a`. We will down-sample it by multiplying it with a sampling signal (a pulse train):

```
>> s = newim(a);
>> s(0:8:end,0:8:end) = 1
>> b = a * s
```

In the Fourier domain, `b` contains many copies of the spectrum of `a`. The reason is that, by increasing the sample spacing from 1 to 8, we also decreased the period length of the Fourier spectrum (which goes from 1 to $\frac{1}{8}$: there are 8 copies of the spectrum side-to-side).

Using the "Log stretch" mode, you can see that all copies of the spectrum are identical, and that they also overlap. This is because the spectrum of 'trui' is too large to fit in the small window. How large is this window? Compare your calculation with the distance between two peaks in the image.

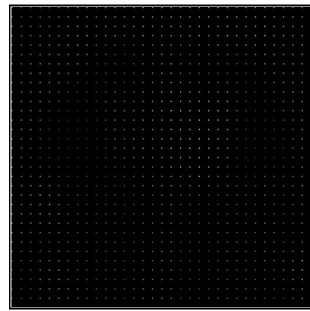
We will now construct a square low-pass filter:

```
>> d = 64 * ( max(abs(xx(c)),abs(yy(c))) < 60 )
```

`xx` and `yy` create images with the same size as `c`, filled with the x and y coordinates. Do the computation above step-by-step to see how this filter is constructed. We will do more of these in Section 8. Use the correct value in the threshold to obtain a filter of the desired size.

Multiply the Fourier spectrum of the sub-sampled 'trui' with this filter, and transform back (the filter was multiplied by 64 to compensate for the amount of intensity that was thrown away by the sampling). The resulting image shows the effects of aliasing. To avoid this, what do we have to do?

book: 5.1



sub-sampled 'trui'

low-pass filtered

Exercise 11: Wiener filtering *(advanced)*

book: 10.2.3

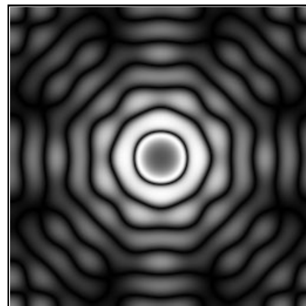
Read in the image 'blurr1'. We will apply some Wiener filtering to enhance this image and try to read the license plate. We know that the filter used to blur the image was a uniform filter with a circular support and a radius of 7 pixels. Its convolution kernel h_0 can be constructed like this (except for the normalization, which is not important here):

```
>> b = +(rr<=7)
```

You now have to implement the Wiener filter, and apply it in the Fourier Domain (to compute the complex conjugate, use the function `conj`). The ratio of noise to signal must still be estimated. If you choose a value that is too large, the effect of the Wiener filter will be too small, and no inverse filtering will be performed. Conversely, if you choose the value too small, you'll be sharpening the noise.

The ringing pattern around the edges of the image are caused by the filtering in the Fourier Domain. This is because of the periodic boundary conditions (i.e. the image is considered as one period of a periodic image, infinite in size).

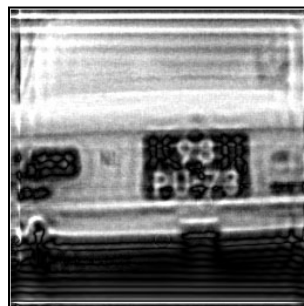
image 'blurr1'



wiener filter

restored license plate

too little



too much

5.3 Shift Estimation *(advanced)*

To find the shift between two images, one can compute the cross-correlation and find the maximum. A cross-correlation is very similar to a convolution,

$$\operatorname{cov}(f,g)(t) = \int f(\tau)g(t+\tau)d\tau = f(t) * g(-t) \qquad .$$

Exercise 12: Aligning images

Find the shift between the images 'imser1' through 'imser5', correct for this shift (use the function `shift`), and average the images. The resulting image should have less noise than the five input images. What is the variance of the noise in this image, assuming the input images all have white noise with a variance of σ^2?

Hint: instead of using `mirror` and `convolve`, you can compute the cross-correlation in the Fourier Domain by $F(\omega)G^*(\omega)$; the complex conjugate can be obtained with `conj`.

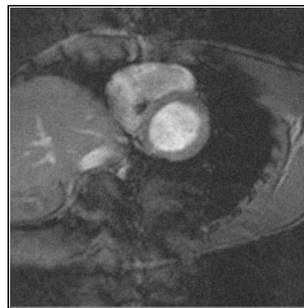
Hint: use the function `max` to find the location of the maximum. To know where the origin is, look for the maximum in the auto-correlation of one of the images.

A property of the Fourier Transform is that a shift in the spatial domain is equal to a phase-shift in the frequency domain. It is possible exploit this property to find a more accurate shift-estimator:

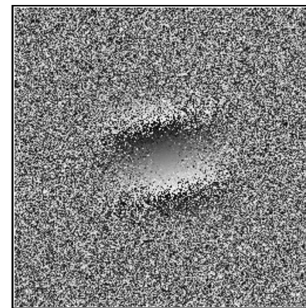
$$\begin{aligned} g(t) = f(t+a) \quad &\rightarrow \quad G(\omega) = e^{ia\omega}F(\omega) \\ g(t) * f(-t) \quad &\rightarrow \quad G(\omega)F^*(\omega) = e^{ia\omega}\left\|F(\omega)\right\|^2 \\ &\Rightarrow \quad \text{phase}\left\{G(\omega)F^*(\omega)\right\} = a\omega (\text{mod } 2\pi) \qquad , \end{aligned}$$

from which the shift a can be estimated using a least-squares fit (which we won't do here). Compute the phase of $G(\omega)F^*(\omega)$, using two of the images from the previous exercise. Also compute $F(\omega)F^*(\omega)$; it should have a phase of 0 everywhere, and an amplitude very similar to that of $G(\omega)F^*(\omega)$.

image 'imser1'



average of 5 images



phase of $G(\omega)F^*(\omega)$

6 Binary Image Processing

6.1 Neighborhood Relations

In binary images, an object is defined as *a connected set of pixels*. But which pixels are connected? In 2D images we can distinguish 4-connected and 8-connected objects. If an object is 4-connected, pixels touching each other diagonally are not considered to be connected; that is, each pixel has only four neighbors. In 8-connected objects, all 8 neighbors are considered connected.

This leads to the notion of distance. In a 4-connected world, a diagonal step has a distance of two (since we need to move horizontally first, and then vertically). This is called city-block-distance (imagine driving through New-York, where you can only drive in orthogonal directions). The 8-connected distance is called chessboard distance (compare to the steps the King can make in chess). A circle in these two metrics are a diamond and a square, respectively. These are bad approximations to the Euclidean distance. By alternating steps with these two metrics, a new metric (4-8 or 8-4 distance, depending on the first step taken) is obtained, in which a circle is octagonal. This is the best approximation possible if only nearest neighbors are to be taken into account.

In DIP*image*, these connectivities are specified as 1, 2 for the 4 and 8-connected steps (1 is only the direct neighbors, 2 are the next neighbors; this notion extends readily to 3D, where a connectivity of 3 can be added). -1 means alternating, starting with 1, and -2 means alternating, starting with 2.

In 3D, there are a 6-connected, a 18-connected and a 26-connected neighborhoods. These are represented in DIP*image* with connectivities of 1, 2 and 3 respectively.

6.2 Binary Morphology

Most binary image processing operations fall under the denominator morphology. In Section 7 we will extend these operations to grey-value images.

There are dedicated operations for binary images. The point operations 'not' (`~`), 'or' (`|`), 'and' (`&`), 'xor' (`xor`) can only be issued directly onto the command line (see A.4). The binary morphological filters can be found under the 'Binary Filters' menu.

book: 9.6.2

The dilation is an operation that 'grows' the binary objects. To see it in action, load the image 'cermet' and threshold it. Apply the function `bdilation` with different values for 'iterations' and 'connectivity'. Note how the connectivity affects the shape of the resulting objects. A connectivity of -1 or -2 produces the most circular borders.

Exercise 13: Neighborhood shapes

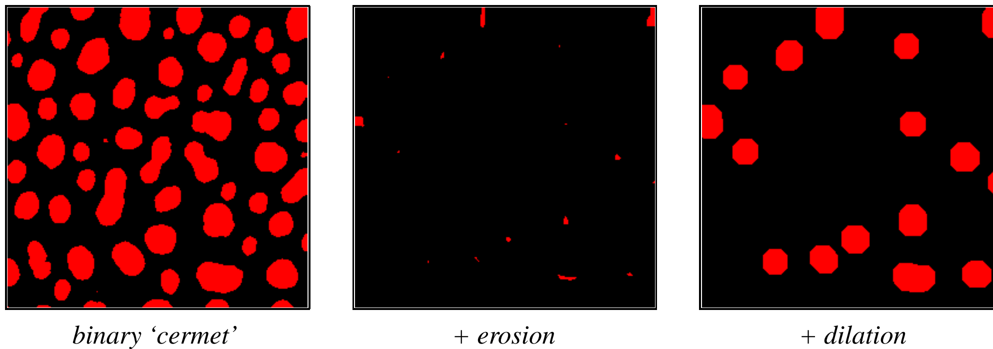
Compare the shapes imposed by the selection of a connectivity. To do so, make a binary image with one pixel set:

```
>> a = deltaim(256,256,'bin');
```

Apply several (64) steps of a dilation to it using the different connectivities. What is represented by these shapes?

Now try `berosion`. The erosion 'shrinks' objects, and produces the same result as applying a dilation to the background (`berosion(a) == ~bdilation(~a)`).

Note how an erosion completely removes the smaller objects, whereas the larger ones are reduced to small spots. If we were to dilate this image again, we somehow would reconstruct the original large objects, but the small ones, which had disappeared, cannot return.



binary 'cermet' *+ erosion* *+ dilation*

book: 9.6.4

This sequence of erosion and dilation is called an opening (`bopening`). The inverse sequence is a closing (`bclosing`). If the first one removes small objects, the second one will remove small holes in the objects.

6.3 Selecting Objects

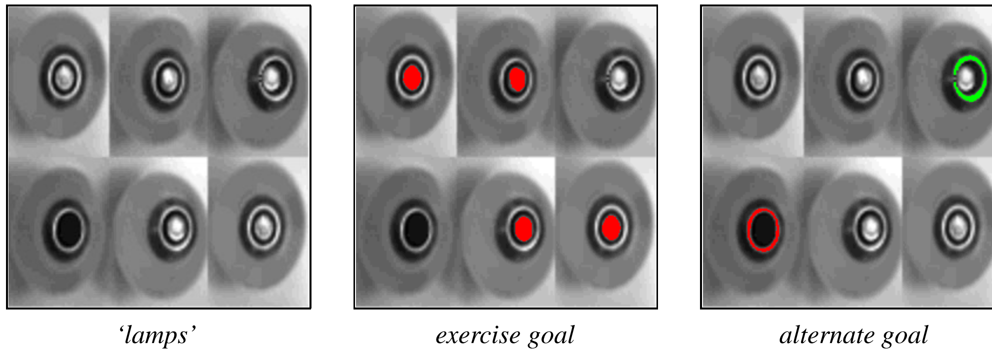
book: 9.6.8

If, instead of applying a dilation after the erosion, we apply a 'constrained' dilation, the opening is converted into an opening by reconstruction. There is no such function in DIP*image*, but the constrained dilation does exist. It is called binary propagation (`bpropagation`), and requires two input images: a seed image (the result of the erosion), and a mask image (the original binary image). What the function does is dilate the seed image, constraining it to the mask image. That is, the resulting objects will never be larger than the objects in the mask image. Try it out on the image we were working on. Make sure to set the edge condition to 0. This is the value of the pixels just outside the boundary of the image. If you set it to 1, all objects touching the border will also be reconstructed. This edge condition can be used to remove edge objects (as done in the function `brmedgeobjs`).

Exercise 14: Quality control of incandescent lamps

Load the image 'lamps'. It contains an image of six bulbs, two of which are to be discarded. The bulbs must have a contact at the bottom, and it must not touch the outer edge, which is the other contact.

Threshold at a low value, such that the bulb is merged with the background (we are only interested in the fitting, which is characterized by the two black rings). Now remove the background using `brmedgeobjs` (which is implemented using `bpropagation`). Now devise some operations using `not` (or `~`), `bdilation`, `berosion`, `bpropagation` and/or `brmedgeobjs` to detect either the good or bad bulbs (either make a program that rejects bad bulbs or accepts good bulbs).



'lamps' *exercise goal* *alternate goal*

The colored images were generated with the command `overlay`. It overlays a grey-value or color image with a binary image. The third (optional) parameter determines the color for the binary objects. It is possible to apply this function several times, each with a different binary image, which can thus be used to mark the image using several colors.

book: 9.6.7

The last operation we will discuss here is the binary skeleton (`bskeleton`). It is a conditional erosion: the objects are eroded until a single line remains. This line lies close to the geometrical center of the object, and has the same topological properties as the object (i.e. some shape characteristics are preserved). It can be used, as demonstrated in the next two exercises, to generate a seed image for the binary propagation, so as to select objects with specific shape properties (note the functions `getsinglepixel`, `getbranchpixel`, etc. in the "Binary Filters" menu).

Exercise 15: Distinguishing nuts from bolts

Now load the image 'nuts_bolts1'. Threshold it. Note that the threshold operation chooses the background as the object (because it is lighter). You will need to inverse the image before or after the thresolding.

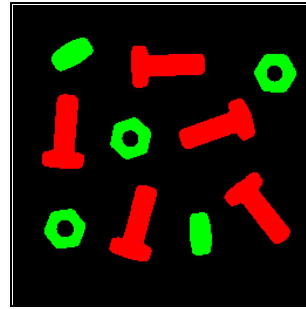
Use the `bskeleton` function (under the "Binary Filters" menu) to create a skeleton of the objects. What is the influence of the 'Edge Condition'? What does 'End-Pixel Condition' control?

With `'looseendsaway'` we can transform the nuts seen from the top (with the hole in them) into a form that is distinguishable from the other objects in the image. Now use the function `getsinglepixel` to extract the objects without a hole in them. This new image can be used as a seed image in `bpropagation`. The mask image is the original binary image. The objects with holes are retrieved with `b & ~c` (literally `b` and not `c`) if the output image for `bpropagation` was `c`.

Try extracting the last nut using the `bskeleton` and the `getbranchpixel` functions.

As a final test, load the image 'nuts_bolts2' and apply the same sequence of commands to it. You should be able to correctly identify the objects in this image.

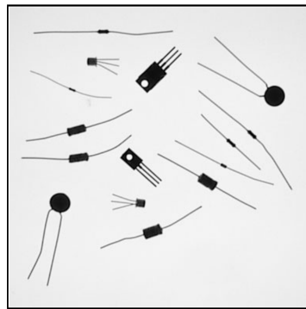
'nuts_bolts1'



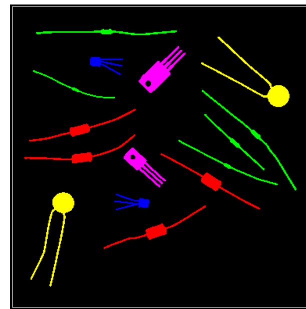
exercise goal

Exercise 16: Recognize components

Read in the image 'components' and threshold it (make sure that the objects are connected in the thresholded image). Now try to differentiate the transistors (three-legged), capacitors (big) and resistors (small) using the techniques learned in this chapter. Optionally, you can differentiate the current stabilizers from the transistors (they have a hole in them), and the ceramic capacitors from the electrolitic capacitors (the round ones are ceramic).



'components'



exercise goal

This colored image was generated with the command

```
>> joinchannels('RGB',(cerco+elco+stab)*255,...
                  (cerco+res)*255,(trans+stab)*255)
```

7 Morphology

In Section 6 we introduced binary morphology. In this section we will extend morphology to grey-value images. Morphological operations are non-linear, and have a wide range of applications.

book: 9.6.10

We already mentioned the maximum and minimum filters in Subsection 3.3. In morphology, these are called dilation and erosion (`dilation`, `erosion`), and constitute the basic morphological operations. By putting them in sequence we obtain the closing and the opening (`closing`, `opening`):

```
>> b = erosion(dilation(a))
```

(since the structuring element is symmetric, we don't need to mirror it). Note that the filter window is called 'structuring element' in morphology. These terms are interchangeable.

The closing is an extensive filter: the output is always greater or equal to the input. This is one of the important properties of the closing. The opening is anti-extensive. Furthermore, both are idempotent: applying the operation a second time does not further modify the image.

7.1 Morphological Filtering

Linear filters are best suited to solve problems due to linear phenomena (motion, blurring, etc.). Other tasks should be tackled with non-linear filters. As such, morphological filters provide solutions to a wide variety of problems. Noise is one such problem. Linear filters are often used to reduce noise, but they do not preserve edges, as non-linear filters can.

Morphological filtering is often used because of their ability to distinguish structures based on size, shape or contrast (whether the object is lighter or darker than the background). They can be employed to remove some structures, leaving the rest of the image unchanged. In this sense, morphology is one step ahead of other image processing tools towards image interpretation.

book: 9.6.11

The closing and the opening are smoothing filters. They remove small local minima or maxima without affecting the grey-values on larger structures. A sequential combination of these two filters is a morphological smoothing, and known under the names open-close and close-open. Note that it matters which one is applied first.

Exercise 17: Morphological smoothing

Apply a closing and an opening to the image 'erika' in both orders. What is the difference between these two smoothing filters? If you take the difference between the two results, you will notice that one is mostly greater or equal to the other: one is biased towards dark objects and one towards light ones. However, there is no ordering relation between the original image and these two results: they are not extensive nor anti-extensive filters.

Note that the size of the structuring element is an important parameter. Construct a smoothing filter that removes most of the hair, but leaves the face recognizably human.

A morphological smoothing with a small structuring element is an ideal tool to reduce noise in an image.

'erica'

open-close

close-open

Exercise 18: Morphological sharpening

In Subsection 3.5 we saw some edge detectors constructed with `maxf` and `minf` (which are the same as `dilation` and `erosion`, respectively). They are morphological gradient magnitudes:

$$\begin{aligned} Edge_1 &= dilation(A) - A \\ Edge_2 &= A - erosion(A) \end{aligned} \quad .$$

In a similar way, we can construct a morphological second derivative:

$$MSD = \frac{dilation(A) - 2A + erosion(A)}{2} = \frac{Edge_1 - Edge_2}{2} \quad .$$

Note the (1,-2,1) across the edge, like in the Laplace operator. Apply it to the image 'trui' (use a small size, for example 3) and compare it to the linear Laplace. Use it to sharpen the input image.

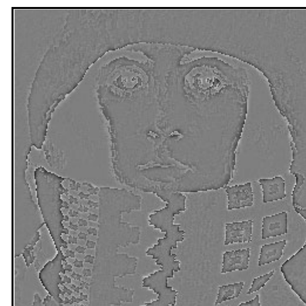
A sharper version of the morphological Laplacian can be computed by taking the minimum value of the two edge detectors. Note that the sign of the morphological Laplacian is used for this purpose (use the function `sign`).

$$MSD_{sharp} = sign(Edge_1 - Edge_2) \cdot min(Edge_1, Edge_2) \quad .$$

Apply it to the image 'trui' and compare it to the other two results. Use it to sharpen the input image.

linear Laplace

morphological Laplace


sharp morphological Laplace

Exercise 19: Edge and/or texture sensitivity

The morphological gradient magnitudes defined above are sensitive to both edges and noise (or small detail). We can modify them such that they are sensitive to edges only:

$$\begin{aligned} Edge_3 &= dilation(A) - closing(A) \\ Edge_4 &= opening(A) - erosion(A) \quad . \end{aligned}$$

Use these edge detectors to compute the morphological Laplacian and the sharp morphological Laplacian. Can you explain why these filters are insensitive to the small detail in the image (like the nose and the checkered pattern to the left)?

Repeat the previous exercise again, this time use

$$\begin{aligned} Edge_5 &= closing(A) - A \\ Edge_6 &= A - opening(A) \quad . \end{aligned}$$

What do you expect to see this time, and why?

If the differences are not clear enough, generate this test image:

```
>> a = gaussf((rr<64)*128+64,3);
>> a = noise(a,'uniform',0,64);
```

and apply the three MSD filters you constructed in this exercise and the previous one to it.

7.2 Other Morphological Tools

As in binary morphology, there is a grey-value equivalent to the skeleton and the propagation. Additionally, there are algorithms used for segmentation (watershed transform, see Subsection 12.3), recognition and measurement (granulometries, see Subsection 12.1). There is not enough time in this course to study all of these, but the interested student is referred to

- Soille, P., “Morphological Image Analysis, Principles and Applications”, Springer, 1999.

8 Image Manipulation

This section will explain a bit further how to manipulate images in DIP*image*. This is a non-trivial topic, because there are many functions that you should be aware of. Knowing these functions will avoid a lot of work during your research.

If you are skipping the advanced sections, it will suffice if you skim over the next three sub-sections, and use them as reference later on.

8.1 Coordinate System

Unlike MATLAB arrays, images in DIP*image* are indexed starting at 0. Also, the first index indicates the x-coordinate (horizontal) instead of the row number. This is an important difference that might be a bit confusing at the beginning. Know the type of the object you are indexing. To extract the value of a pixel, use the syntax `b=a(30,10)`. This syntax can also be used to assign a value to a pixel: `a(30,10)=255`. The y-axis is inverted in the display of images. This is very common in image processing. However, this is only in the display, the coordinate system used is Cartesian.

Many computations require the coordinates of each of the pixels being addressed. This is easy if you write loops over the pixel values:

```
>> for x=0:255
>>   for y=0:255
>>     b(x,y) = function(a(x,y),x,y);
>>   end
>> end
```

However, this type of code is very slow, and often unnecessary. Using the functions `xx`, `yy` and `zz` you can create images containing the coordinates of all pixels (additionally, the functions `rr` and `phiphi` provide polar coordinates). These can be used in *vectorized* code, code that applies the same operation on all pixels at once:

```
>> b = function(a,xx(a),yy(a));
```

For example, to compute the center of mass of an image:

```
>> x = sum( a * xx(a,'corner') ) / sum(a)
>> y = sum( a * yy(a,'corner') ) / sum(a)
```

Note the option `'corner'` in the call to `xx` and `yy`. It causes the origin to be in the upper left corner (the same as in indexing). If this option is omitted, the origin is in the center of the image (the same 'center' used by the Fourier Transform `ft`, see Subsection 5.1).

Finally, to retrieve a list with coordinates of non-zero pixels (especially useful on binary images) use `findcoord`. The standard MATLAB function `find` returns indices into the array, which can also be used to index. Indices also start at 0 for images, and run (in de MATLAB way) first down and then across:

```
[ 0 3 6
  1 4 7
  2 5 8 ]
```

To convert from coordinates into indices:

```
>> I = y + x*size(a,2) + z*size(a,2)*size(a,1)
```

Note that:

```
>> x = xx(a,'corner');      % image with x-coordinates
>> crd = double(x(m));      % select elements in mask
```

is the same as

```
>> crd = findcoord(m);      % get coordinates of mask pixels
>> crd = crd(:,1);          % keep only x-coordinates
```

(Indexing with a mask image (`m`) is explained below, read on.)

8.2 Region Extraction

To access more than one pixel from an image there are several possibilities.

- The easiest is indexing a rectangular patch:

  ```
  >> b = a(64:127,0:63)
  >> a(64:127,0:63) = b*2
  ```

  Note that `64:127` is an array with 64 elements. This can be exploited to do things like:

  ```
  >> a([0,end],:) = 0
  >> a(:,[0,end]) = 0
  ```

  In indices, the colon (`:`) indicates all elements; to indicate a range, as above, the colon is used between two elements. If you need a regular sub-set of pixels, use the notation `64:3:127`, which takes one pixel and skips two (64, 67, 70, etc.). `end` means the last pixel in that dimension.

  Using this syntax it is not possible to access a set of isolated pixels:

  ```
  >> a([x1,x2,x3],[y1,y2,y3])
  ```

  retrieves not only the values `[x1,y1]`, `[x2,y2]` and `[x3,y3]`, but also all values `[x1,y2]`, `[x1,y3]`, `[x2,y1]`, etc.

- To index a set of isolated pixels, you will need to use indices into the image, as shown earlier (it is possible to use an array of indices). The values returned are kept in the same order as the indices were given (that is, `a([3,1])` returns the values for pixels number 3 and 1).

- The third indexing method is using a mask image. A mask image is a binary image with 1 at the locations of the pixels being indexed. It must, of course, have the same size as the image being indexed into. The next piece of code shows indexing using a mask image and indices.

  ```
  >> m = a>100
  >> a(m) = 100
  >> I = find(m)
  >> a(I) = 0
  ```

8.3 Transformations

Another important image manipulation class are the functions that perform rotations, mirroring, shifting, inversions, resampling, etc. Most of these are located under the "Transforms" menu. We will not go too deeply into them here, since their use is quite obvious. You already might have used some of these, and you will certainly need them in the future. Be aware of their existence.

8.4 Test Image Generation *(advanced)*

Using the functions `xx`, `yy`, etc. introduced above, we can construct test objects. Test objects are very often necessary to test an algorithm, and compare its results to what we know it should produce (which we do not on natural images).

This example produces a Gaussian kernel:

```
>> sigmax = 20 ; sigmay = 10;
>> exp(-0.5*((xx/sigmax)^2+(yy/sigmay)^2))/(2*pi*sigmax*sigmay)
```

Examine the code carefully, and execute it in portions to see what each one does. Compare the code to the mathematical formula of the Gaussian kernel,

$$\frac{\exp\left(-\frac{1}{2}\left[\left(\frac{x}{\sigma_x}\right)^2+\left(\frac{y}{\sigma_y}\right)^2\right]\right)}{2\pi\sigma_x\sigma_y} \quad ,$$

and note how we don't need to apply it for each pixel separately, but can compute the whole image at once.

Exercise 20: Generation of a rotated Gaussian kernel

Using the code given above and Figure 5 as a guide, generate a rotated Gaussian kernel (do not rotate an image of a Gaussian kernel, use a rotated coordinate system).

Exercise 21: Generation of a rotated binary rectangle

Change the code you wrote for the previous exercise to generate a rectangle of certain size and orientation.

Hint: you can construct a rectangle by comparing both coordinates to the required sizes, i.e. `abs(x)<30 & abs(y)<50`.

Note: Keep this function, you will use it in Section 9.

To convert your script into a function, add the following line at the top of the file:

```
function out = rect(sz,phi)
```

and save it as `rect.m`. This will also be the name of the function. You will be able to call it like this:

```
>> a = rect([10,20],pi/6)
```

Within your function, `sz` will have the value `[10,20]`, and `phi` the value $\frac{\pi}{6}$.

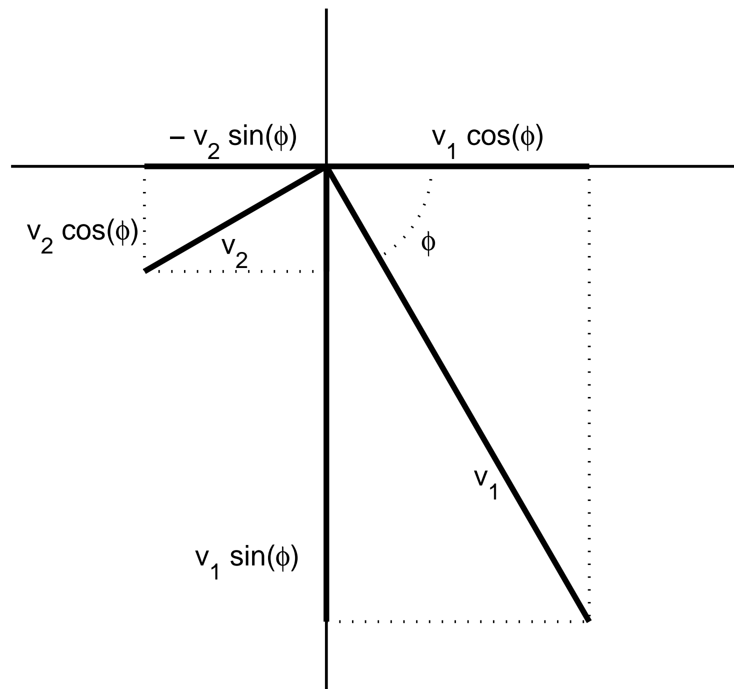
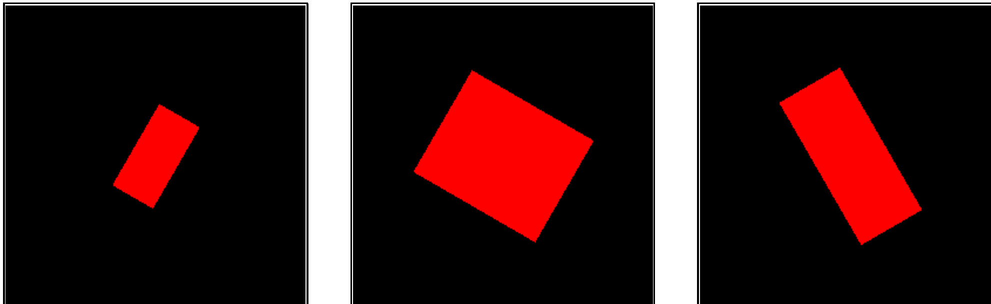


Figure 5: Coordinate system for the rotated Gaussian



The function `noise` adds noise to an image. The default noise type is `'gaussian'` (additive, Gaussian-distributed noise), but it can also produce `'uniform'` and `'poisson'` noise (both are also additive, but with different distribution). This is important to be able to test your algorithms.

9 Measurements

This section describes the basic Image Analysis procedures: measuring properties of objects in images. There are other tools that fall under the denomination of *Image Analysis*, such as morphology (selecting objects based on their shape, see Section 6) and scale-spaces (discussed in Subsection 12.1).

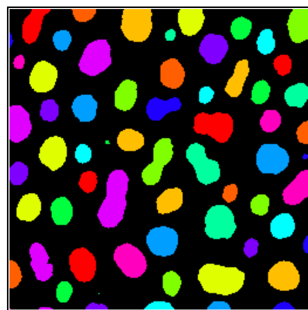
9.1 Extracting Single Objects

In a binary image, an object is considered a connected set of pixels. As discussed in Subsection 6.1, there are different 'connectivity modes' that define which pixels are considered connected (and thus belonging to the same objects). Before we can measure a property of such an object (say, the number of pixels that define it), we need to extract the object from the image. The common way of doing this is to label all objects. Labelling involves finding any foreground pixel in the image, give it a value (the label ID), and recursively give the same value to all pixels that are connected to it. This process is repeated until all foreground pixels have been assigned to an object. To extract one object, all we now need to do is get all pixels with its ID.

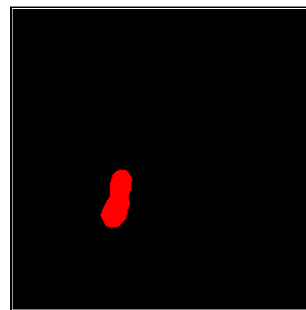
Load the 'cermet' image and threshold it. Now find the function `label` in the "Transforms" menu. Look at the label result using the "labels" mapping mode, in which each object is displayed in a different color. This makes it easy to see if objects have been correctly separated or not. Note that there are only a small number of different colors. If there are more objects, some will share a color. Use the "Pixel testing" mode on the result to check what values each object has. To extract object number 38 from the image, we can now do (assuming `la` is the label image):

```
>> la == 38
```

Note the double equal sign, it is the equality operator (as opposed to the assignment operator). The area of this object is now easily obtained with `sum(ans)`.



labeled objects



object number 38

9.2 Measuring in Binary Images

In Exercise 15 (Subsection 6.3) we tried to separate nuts from bolts using binary morphology. Here we will do the same exercise by measuring different object properties, such as the area, perimeter and lengths.

Load the ‘nuts_bolts1’ image again. Threshold and label it, making sure to keep the original (grey-value) image. Now we are ready to do some measuring. Select the `measure` function in the “Analysis” menu. The object image is the labeled image, and the grey-value image the original image before segmentation. It won’t be used by the measurements we will do, but it has to be provided. Select `'size'` as the measurement (it computes the area by counting the number of pixels). If you leave ‘Object IDs’ empty, all objects will be measured. Put the output in a variable called `data`. Now

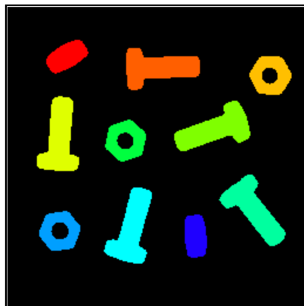
```
>> sz = data.size
```

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

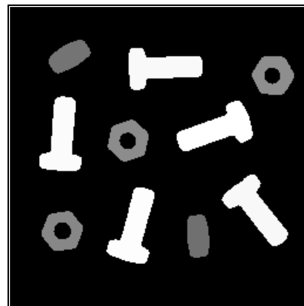
```
>> diphist(sz,[1,1500],500)
```

This will create a histogram for the areas. There are obviously two area categories. Let’s say that areas up to 1000 pixels are for the nuts, and larger areas for the bolts. There exist ways of doing this automatically (similar to the automatic thresholding techniques), but we won’t go into them now. We will use the function `msr2obj` 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:

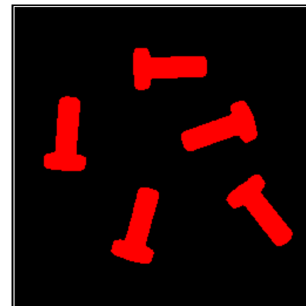
```
>> nuts = xor(b,bolts)
```



labeled ‘nuts_bolts1’



‘painted’ objects



threshold at 1000

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.

Note that it is possible to obtain a whole series of measurements at once, by specifying more than one measurement name. To extract all measurements for one object, index the returned measurement object using the label ID of the object you are interested in. The next example illustrates the four types of indexing:

```
>> data(4)          % properties of object with ID 4
>> data.size        % size of all objects
>> data(4).size     % size of object with ID 4
>> data.size(4)     % size of 4th. element
```

9.3 Errors Introduced by Binarization

Note that the area, perimeter, etc. you measured earlier are not the exact measurements you could have done on the objects in the real world. Because of the binarization, the object boundary was discretized, introducing an uncertainty in its location. The true boundary is somewhere between the last on-pixel and the first off-pixel. The pixel pitch (distance between pixels) determines the accuracy of any measurement.

Exercise 22: Thought experiment

Imagine you drive a car with an odometer that indicates the distance travelled in 100 meter units. You plan to use this car to measure the length of a bridge. When you cross the bridge, sometimes the odometer advances one unit, sometimes two. Can you use this set-up to measure the length of the bridge accurately? How can you determine the accuracy? What special measures do you need to take to make sure your measurement is not biased?

Exercise 23: Errors in area measurement

The object area (`'size'`) is computed counting the number of pixels that comprise the object. The error made depends on the length of the contour. Quantify this error for round objects of various sizes. What happens with the accuracy as a function of the radius? Why?

Hint: make sure you generate the objects with a random offset to avoid a biased result. To do so, use the function `rand`:

```
>> a = ((xx+rand)^2+(yy+rand)^2) <= 64^2
```

Hint: on these images, you can use the function `sum` instead of `measure`, since you only have one object in each image.

book: 3.6.1 5.2.2

The perimeter is measured using Freeman chain codes. The weight assigned to each of the steps and the corner count is chosen such that the mean square error is minimized for lines under an arbiraty orientation. This means that some error is made under all orientations. The original Freeman method counts horizontal and vertical steps with a weight of 1, and diagonal steps with a weight of $\sqrt{2}$. This causes horizontal and vertical lines, as well as lines under an angle of 45°, to be measured accurately, but all other lines to be measured with some error. The largest error would be at 22.5°. To decrease this maximum error, we need to introduce a small error in the measurements of lines under 0° and 45°. By also counting the number of corners - that is, the number of points at which the chain code changes - the errors can be further reduced.

Exercise 24: Errors in perimeter measurement *(advanced)*

To see the errors made by this algorithm, we will measure the perimeter of rectangles. All sides of a rectangle are under the same orientation (modulo $\frac{\pi}{2}$), and thus will be measured with the same error. Using the function you made in Exercise 21 (Subsection 8.4), generate a series of rectangles with random offset, under different orientations, and with a fixed size. For each of these rectangles, measure its perimeter and `scatter` it versus its orientation. What is the influence of the orientation on the accuracy and precision of the measurement?

What is the influence of the object size on the accuracy of the measurement?

9.4 Measuring in Grey-Value Images *(advanced)*

In the previous sections we threw out a lot of information by binarizing the images before measuring the objects. The original grey-value images contain a lot more information (assuming correct sampling) than the binary images derived from them (which are not correctly sampled). As long as we only apply operations that are sampling-error free, we can perform measurements on grey-value images as if we were applying them directly to analog images. In this case, measurement error is only limited by the resolution of the image, noise, and imperfections caused by the acquisition system.

The following code produces a band-limited disk (with height or intensity 255 and radius 64):

```
>> a = testobject(a,'ellipsoid',255,64)
```

To measure its area, sum all the pixel values. The result should be very close to the true area of the disk times its height (to see really how close they are, subtract the two results from each other). Compare to the result obtained on the binarized version of this image.

The perimeter can be obtained by integrating the gradient magnitude:

```
>> sum(gradmag(a,2))/255
```

Note that the scale of the gradient (the sigma of the Gaussian derivative) has an influence on the result of this measurement. If it is taken too large, some of the grey-values will disappear over the edge of the image; if it is taken too small, discretization errors of the filter will have the upper-hand.

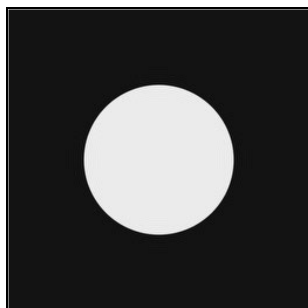
The curvature is defined as the rate of change of orientation along the contour, and can be computed using the second derivative along the contour (`dcc` in 2D):

```
>> b = gradmag(a,2)
>> c = max(b,max(b)/5) % Avoid division by 0
>> c = -dcc(a,2)/c
```

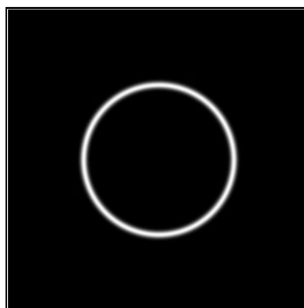
Make sure you only look at the results near the boundary; elsewhere the curvature is ill-defined. Mean curvature is given by

```
>> mean(c(threshold(b)))
```

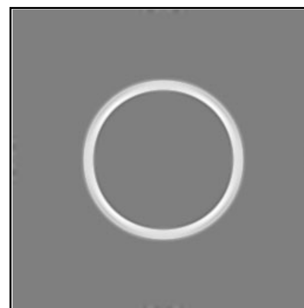
and should equal one over the radius ($\frac{1}{64} = 0.0156$).



disk image



perimeter



curvature

Exercise 25: Bending energy

Bending energy is defined as the square of the curvature, integrated over the perimeter. Make a script that computes the bending energy of the disk in `a`. The result should be $\frac{2\pi}{64}$.

Hint: to integrate over the perimeter, you need a mask image `m` that indicates the pixels that belong to the perimeter (derived from the gradient magnitude, for example), and sum the pixel values that fall within this mask (`sum(a(m))`).

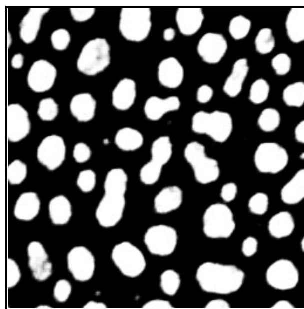
Exercise 26: Area, perimeter and bending energy of 'cermet'

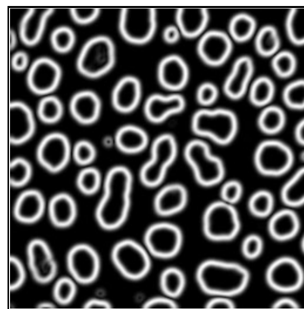
Read in the image 'cermet', and apply an `erfclip` around 128, with a range of 64. This should make the edges more pronounced and suppress intensity fluctuations in the foreground and the background. 'cermet' now looks more like the ideal image of the disk we used earlier. Note that `erfclip` (error-function or soft clipping) does introduce some aliasing, but not nearly as much as regular (hard) clipping or thresholding. As a last step, invert and stretch the image. Now the height of the objects is 255, like in our disk image.

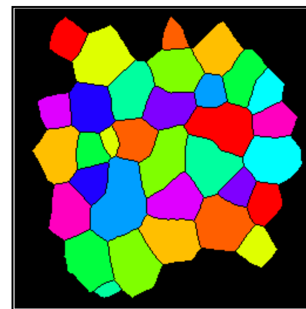
The measurements we did earlier (`sum`) now need to be done separately for each of the objects. For this purpose we will use the `measure` function. But we will require a label image in which each label covers one object and the area around it. Try to make such a label image. (*Hint:* use a skeleton of the background.) See if you are able to remove the labels for the objects that touch or are close to the edge of the image, without extending the labels of the other objects.

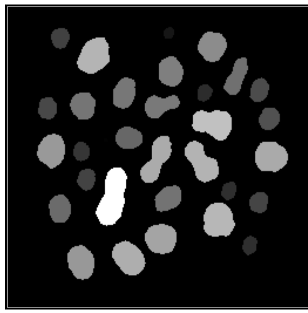
Compute the sum over each of the regions in your label image of the soft-clipped (`erfclip`) 'cermet' (using `measure`). This is the area for each object. Do the same thing using the gradient magnitude of the image. This is the perimeter for each object. Compare these results to measurements on the binarized image (`'size'` and `'perimeter'`). (We don't know the true values, so this comparison is a bit useless.)

Now compute the bending energy of each object. Use the function `msr2obj` to paint each label (in the label image) with the bending energy. Now multiply this image with the binarized 'cermet' image. This allows you to examine the result more closely. Make sure that smaller objects have a higher bending energy, as do objects with sharp bends in their contour.

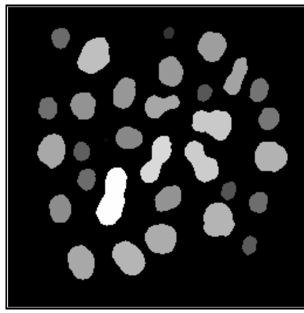

soft-clipped 'cermet'


perimeter

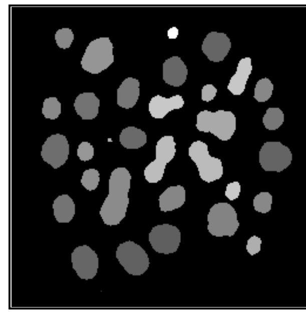

labeled regions



area



perimeter



bending energy

10 Vector Images *(advanced)*

Until now all images we have seen were represented by a single value for each pixel (i.e. they were functions $f : \mathbb{R}^n \rightarrow \mathbb{R}$, where n is the dimensionality of the image. A more general representation of an image is a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where each pixel is represented by m values. Depending on the meaning of these values, this can be:

- a color image: each value is recorded through a different color filter,
- a multi-spectral image: each value is the intensity in a very arrow band of wavelengths,
- an image of a sample dyed with different fluorescent dyes, each value being the response to a different excitation laser,
- {any other way of combining different information from the same location}, or
- a mathematical construct: the values are computed from the original image, for example the derivative in orthogonal directions.

10.1 Vector Image Operations

The vector image as a mathematical construct is really only useful in the mathematical context itself. We use it in some advanced applications to ease the implementation of operations. By using these vector images, we avoid the hassle of defining an image for each of the components. Take for example the vector image returned by `gradientvector`. The first component is the gradient in the x-direction, and the second one that in the y-direction. Apply the function `gradientvector` (or just `gradient`) on the image 'erika'. The result is a "2x1 tensor image" (a tensor is just the more general form of a vector: a tensor with all values along a row or a column is a vector). There are three techniques to see the information in this image:

- extract an image with only one of the vector components: `b{1}`,
- add colorspace information, converting it into a color image; this allows to display three components at once in the RGB colorspace: `colorspace(b,'RGB')`, or
- if it is a one-dimensional or two-dimensional image, catenate the individual components into a new dimension: `cat(3,b)` (not useful for higher-dimensional images, because of the display limitations in the environment).

The next few commands show how to compute the gradient in an arbitrary direction (the angle $-\frac{\pi}{3}$ is used; this is the direction from Erika's chin to her forehead):

```
>> b = gradientvector(a)
>> alpha = -pi/3;
>> v = [cos(alpha);sin(alpha)]
>> c = b'*v
```

By multiplying (`*` is the vector product) the gradient vector image `b` with a vector `v`, we project the gradient upon this vector (note we need to transpose the image `b` to

align the components correctly for this operation). Thus, what we are doing here is the same as:

```
>> c = dx(a)*cos(alpha) + dy(a)*sin(alpha)
```

In this example, using a vector image might seem more complicated that necessary, but the code that uses the tensor image works for any size of vector, whereas the explicit form is for two-element vectors only. Also, it is easier to see what the code with the vector image is doing, since it better resembles mathematical notation.

10.2 Color Spaces

Images as stored in computer memory or files usually specify color in either RGB (red, green, blue) or CMYK (cyan, magenta, yellow, black). The first directly maps to the computer monitor, which uses red, green and blue phosphors. The second form directly maps to printers, which use those four colors of ink. However, there are many more representations for color.

RGB is a linear representation, since it directly maps to light intensities of the various frequencies. However, human vision is *logarithmic*, in the sense that the perceived contrast is based on the ratio of two intensities, not the difference (the smallest contrast that we can perceive is one intensity being 1.01 times the other). Thus, RGB is a (perceptually) non-uniform color space.

CIE XYZ is also a linear representation, but differs from RGB in that it is a standard. RGB values can be interpreted in many ways, and are usually tuned to a specific set of monitor phosphors. The XYZ color space does not have this dependency, and its weighting curves are tuned to human vision. Like RGB, it is not perceptually uniform. The Y value is the luminance component. Luminance is the intensity per unit area weighted by the spectral sensitivity of the human eye (units: $\mathrm{cd} \cdot \mathrm{m}^{-2}$), in contrast to radiance, which is the total intensity of radiated energy (units: $\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}$).

A perceptually uniform color space is very difficult to define. After more than a decade of research, the CIE decided on standardizing two systems, since neither was ideal. These are $L^*u^*v^*$ and $L^*a^*b^*$ (also written as CIELUV and CIELAB). Both have one lightness channel and two chroma channels. Their drawback is that it takes quite a while to compute the transformation to RGB space for display. The lightness L^* is defined as the cube of the luminance Y, except for very low values, where the relation is linear:

$$L^* = \begin{cases} 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16 & , \quad 0.008856 < \frac{Y}{Y_n} \\ 903.3\frac{Y}{Y_n} & , \quad \frac{Y}{Y_n} < 0.008856 \end{cases} ,$$

where $\frac{Y}{Y_n}$ goes from 0 to 1 (Y_n being the luminance of the reference white). Because of the offset, this curve can be approximated with a 0.4-power function. This leads to the (non-linear) $R'G'B'$ space, which is quite close to perceptual uniformity:

$$\begin{aligned} R' &= R^{0.4} \\ G' &= G^{0.4} \\ B' &= B^{0.4} \end{aligned}$$

Finally, there exist some other color spaces like $Y'C_BC_R$ (*luma* plus two *chroma* channels, non-linear), HSB (Hue, Saturation and Brightness) and HLS (Hue, Lightness, Saturation). These last two are neither linear nor perceptually uniform, and should no longer be used. HSB and HLS were developed in an age when users had to specify colors numerically, but are not useful anymore. Instead, you should use either a linear color space or a perceptually uniform color space. The mayor drawbacks are that the "lightness" or "brightness" are not proportional to Y nor L^*; and that the hue (an angle) has a discontinuity at 360° (it is not possible to perform arithmetic mixtures of colors expressed in polar coordinates).

In DIP*image* there are several of these color spaces implemented. The function `colorspace` converts color images from one representation to another. Note that images in any color space are converted to RGB for display.

```
>> a = readcolorim('gogh')
>> b = colorspace(a,'Lab')
```

Exercise 27: Segmentation on color (part I)

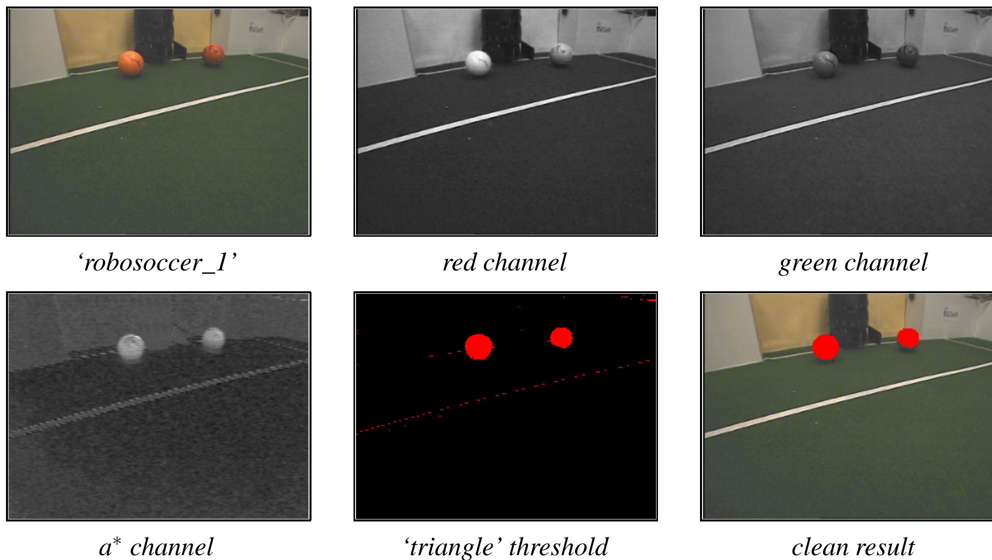
Some images are easily segmented when the correct color space has been chosen. This is very often $L^*a^*b^*$, and this exercise and the next will show why.

Read in the image 'robosoccer_1'. This is an image recorded by a soccer-playing robot. You'll see the dark green floor, greyish walls, a yellow goal, a black robot (the goal keeper), and two orange balls (of different shades of orange). We will write an algorithm to find these balls.

Look at the R, G and B components (extracting them with `a{1}`, `a{2}`, etc.). You'll notice that it is not easy to segment the balls using any one of these three images. One problem is that the bottom side of the balls is darker than the top part. We need to separate color from luminance, as does the $L^*a^*b^*$ color space.

Convert the image into $L^*a^*b^*$. The a^* channel (red-green) makes the segmentation very easy (by chance: we are looking for objects with lots of red, and the balls are the only such objects in the image). A `'triangle'` threshold will extract the balls. Note that the thin lines along strong edges are caused by incorrect sampling in the camera. This is a common problem with single-chip CCD cameras, where the three colors of a single pixel are actually recorded at different spatial locations. If you zoom in on such a strong edge in the input image, you'll notice the color changes. These thin lines in our thresholded image are easy to filter out using some simple binary filtering.

The images 'robosoccer_2' through 'robosoccer_5' contain the same scene recorded with smaller diaphragms (less light reaches the detector). Test that this algorithm still works for these worse lighting conditions.



'robosoccer_1' *red channel* *green channel*

a^* channel *'triangle' threshold* *clean result*

Based on $L^*a^*b^*$, we can define

$$\begin{aligned} C^*_{ab} &= \sqrt{a^{*2} + b^{*2}} \qquad \textit{and} \\ h_{ab} &= \arctan\left(\frac{b^*}{a^*}\right) \qquad , \end{aligned}$$

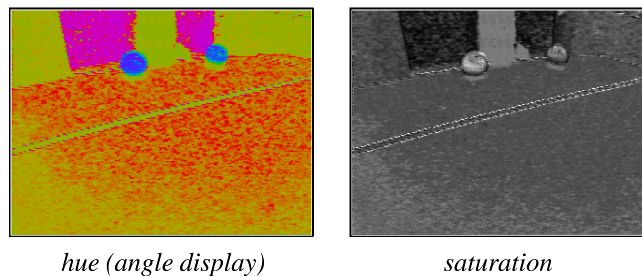
respectively chroma and hue (we can define similar quantities based on $L^*u^*v^*$, for example). Hue is an angle in 4 quadrants, and can be computed using the function `atan2`. In $L^*a^*b^*$ space there is no definition for saturation, but in e.g. $L^*u^*v^*$ there is:

$$s_{uv} = \frac{C^*_{uv}}{L^*} \qquad .$$

Exercise 28: Segmentation on color (part II)

The a^* channel provides a good solution to our problem. However, if there were a red or purple object in the scene (like a robot adversary), this technique wouldn't work. We want to be able to differentiate orange not only from yellow, but also from red and purple. The hue should provide us with a nice tool for this purpose.

Compute the hue (h_{ab}) image from 'robosoccer_1' and use it to segment the balls. Try your program on the other images in the series.



hue (angle display) *saturation*

10.3 Filtering Color Images

Processing color images is a very difficult topic, and many filters have never been applied satisfactorily to color images yet.

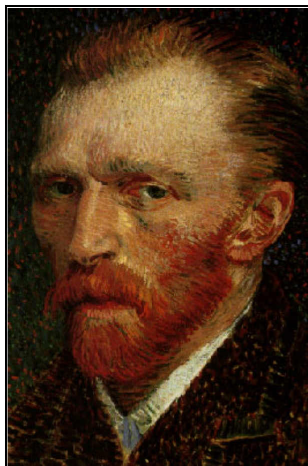
Let's start with linear filters. Since a color is represented as a vector, addition, subtraction and scalar multiplication can be performed on pixels. Furthermore, these operations are all performed on an element-per-element basis, which means that these operations can be performed on the different color channels separately. Thus, all linear filters (which only require these three operations) can be applied on each channel separately. This converts the complexity of filtering a 3-component vector image to filtering three grey-value images. *However*, the fact that these vectors are linear, does not imply that they are visually linear. For example, the average of 'pure' green and 'pure' red might be visually closer to green than to red, or might even be some shade of green and not look like yellow at all. Being half-way between two colors in some color space does not imply being perceptually half-way. That is where perceptually uniform color spaces come in.

As a second note of warning, color spaces are usually a sub-set of a three-dimensional space. The possible values for a color is called the gamut, and depends on the reproduction capabilities of a device. For example, the three RGB values are confined to the range [0,255]. Being outside the gamut means that clipping will occur somewhere along the way between the computer memory and your eyes. Since this clipping might occur in a different color space than the one you were computing in, it is possible that the color is changed. We will see an example of this later.

You'll notice that color images are not supported directly by the filters in DIP*image*. To apply the same filter to each of the color channels, use the function `iterate` (which actually works for any type of image array, it doesn't need to be a tensor image).

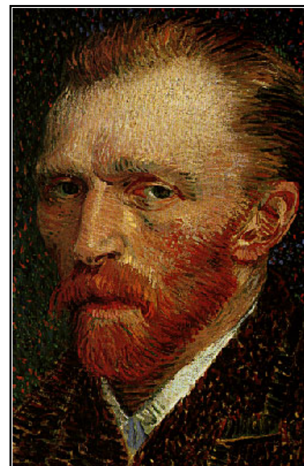
```
>> a = readcolorim('gogh')
>> b = iterate('gaussf',a,10)
```

If you like, try unsharp masking.



'gogh'

Gaussian filter



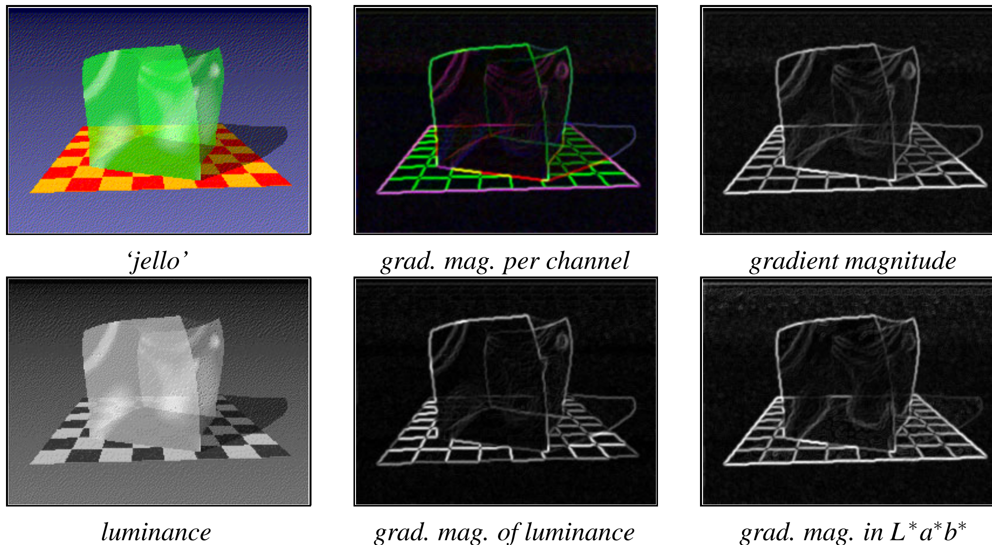
unsharp mask

Exercise 29: Color edge detection

Load the color image 'jello' (remember to use `readcolorim`). Compute the

magnitude (`norm`) of the `gradmag`, and compare to the `gradmag` resulting from the luminance image (the luminance can be obtained by converting to the 'grey' color space; the result is a grey-value image). Which one is preferable? Why?

Convert the image to L*a*b* color space and compute the gradient magnitude there. Notice how features are weighted differently now (for example, the shadow's edge does not produce a strong response).



'jello' — *grad. mag. per channel* — *gradient magnitude*

luminance — *grad. mag. of luminance* — *grad. mag. in $L^*a^*b^*$*

Exercise 30: Filtering in another color space

It is often better to filter images in a perceptually uniform color space, such as L*a*b* (which is supposed to be Euclidean). However, these color spaces have a strange boundary: for different values of L*, a* and b* have different ranges. While processing in this space, it is necessary to keep this in mind. It can be seen on a simple test image what happens if we don't:

```
>> a = newimar(3,1);
>> a{1:3} = newim(256,256);
>> a(128:255,128:255) = 255;
>> a{1}(128:255,0:127) = 255;
>> a{2}(0:127,128:255) = 255;
>> a = colorspace(a,'RGB')
```

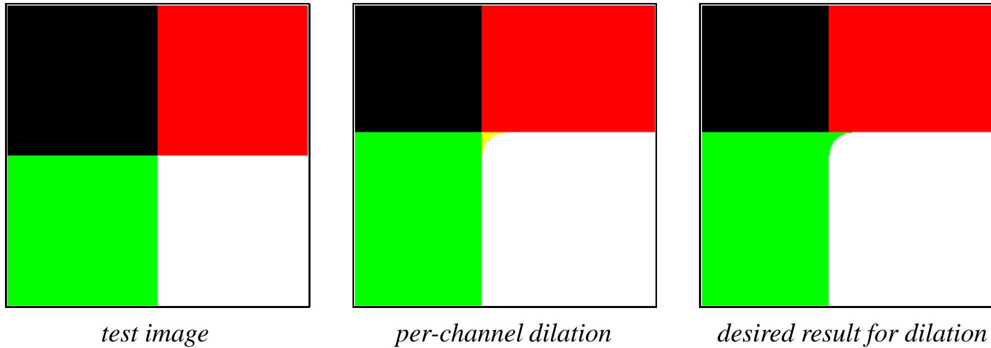
Apply the Gaussian filter as above to this image, then convert it to the L*a*b* color space and apply it again. Compare the results. Why is this different? What strange effects occur to the image in L*a*b* space?

For non-linear filters, it very often is not this clear how they should be applied to color images. For example the morphological operations, that never should introduce new colors (the values of the output are selected from the input by maximum or minimum operations), are particularly difficult to implement.

Exercise 31: Color morphology

Apply the dilation to each of the components of the test image of the previous

exercise (use an elliptic structuring element with a large size). Where does the little yellow corner come from?



test image *per-channel dilation* *desired result for dilation*

It has been proposed to 'sort' the RGB values in some way so that the maximum (or minimum or median) value can be selected from a set. The most logical way of doing this is to treat the RGB value with the highest intensity as the largest value. To resolve ties, one of the colors must be given 'priority' (for example, the one with the largest green value). This indeed solves the problem of introducing new colors in morphological operations, but it also produces a biased result (since green is favored). We won't implement this operation in this course.

Further reading about color spaces and color (or vector) filtering:

- Poynton, C., "Frequently Asked Questions about Color", `http://Home.InfoRamp.Net/~poynton/ColorFAQ.html`.
- Poynton, C., "Frequently Asked Questions about Gamma", `http://Home.InfoRamp.Net/~poynton/GammaFAQ.html`.
- Sangwine, S.J. and Horne, R.E.N., "The Colour Image Processing Handbook", Chapman & Hall, London, 1998.

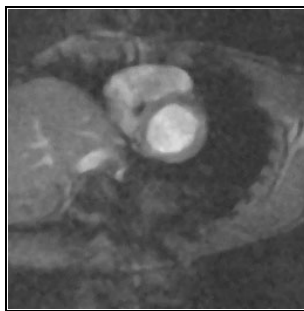
11 Adaptive Filtering *(advanced)*

All filters we had seen up to now had a fixed filter window. By changing this window for each location in the image (based on local properties), it is possible to construct more complex non-linear filters capable of, for example, smoothing an image and enhancing its edges at the same time.

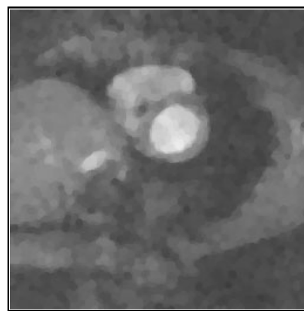
11.1 Kuwahara

book: 9.4.2

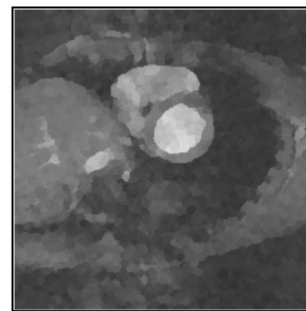
The simplest form of adaptive filtering is the one where the neighborhood is shifted to minimize some criterion. The Kuwahara filter does this to smooth an image while enhancing the edges. Try it on the image 'imser1', and compare your result with other non-linear filters such as median filtering and close-open filtering.



median filter



close-open filter



Kuwahara filter

Exercise 32: Constucting the Kuwahara filter

What the Kuwahara filter does is take the average (uniform filter) over a neighborhood, shifted so that the variance over that neighborhood is minimized (the pixel for which this computation is being done should always be included in the neighborhood). We implement this through a filtered image (`unif`) and a selection image (`varif`), in which the minimum in a neighborhood is found. The value of the filtered image at this point is used as the result of this filter.

This is done by the selection filter (`selectionf`), which is much like the minimum filter, but doesn't return the value of the minimum in the input image. Instead, it returns the value of another input image at that same position. Thus, Kuwahara can be written as:

```
>> b = selectionf(unif(a,5),varif(a,5),5)
```

Try variants of this filter, like substituting `varif(a)` for `varif(unif(a))`, and using different smoothing filters. Note that the filter size is used three times: for both of the input images of the selection filter, as well as for the selection filter itself. Is it important that both input images are computed with the same filter size? Does the selection filter require the same filter size as its input images?

Two morphological filters we have seen (the opening and the closing) are actually also some sort of adaptive filtering (although they are implemented as the sequence of two fixed-filter operations): the location of the filter over which the maximum is taken (in

case of the closing) is chosen such that this maximum has the lowest possible value. We can see that by comparing the result of the closing with that of the selection filter applied to the result of a maximum-filtered image:

```
>> a = readim
>> c = closing(a,5)
>> b = maxf(a,5)
>> d = selectionf(b,b,5)
```

Note that `selectionf(b,b)` is the same as `minf(b)`.

11.2 Other Adaptive Filters

It is possible to change the shape of a filter as well as its location. For example, some filters will use a larger or smaller neighborhood depending on the local variance. Other filters will align themselves to the local structure, using, for example, an elliptic neighborhood whose eccentricity is related to the anisotropy, and is oriented to match possible lines. These filters are not very general, and therefore not directly available under DIP*image*. These are the kind of filters you have to implement yourself if the need arises.

Filters that are easy to turn are the derivatives, since the derivative in any direction can be computed using derivatives in orthogonal directions. As we saw in Subsection 10.1, we can create a derivative in the direction $-\frac{\pi}{3}$ by a linear combination of the derivatives along the x and y axes:

```
>> b = gradient(a)
>> v = [cos(-pi/3);sin(-pi/3)];
>> b = b'*v
```

Exercise 33: Second derivative along the contour

In this exercise you will construct a second derivative that aligns itself to the contour in each image point. For this you will need the direction of the gradient θ (computed using `atan2` and `gradient`), the Hessian matrix H (`hessian`),

$$H = \begin{pmatrix} \partial_{xx} & \partial_{xy} \\ \partial_{yx} & \partial_{yy} \end{pmatrix} \qquad ,$$

and a way of linearly combining the elements of the Hessian according to the gradient direction:

$$\partial_{\phi\phi} = v^t H v \qquad ,$$

with

$$v = \begin{pmatrix} \cos(\phi) \\ \sin(\phi) \end{pmatrix} \qquad .$$

Compare your result to that of the function `dcc`, which does the same thing.

12 Other Advanced Topics *(advanced)*

This section highlights a few advanced topics often used within the Pattern Recognition Group. If you are planning on graduating there, you need to study this chapter.

12.1 Scale-Spaces

Scale-spaces are an image analysis tool which is important because often we do not know at which scale to filter an image to obtain the desired result. In a scale-space, we filter the image at all scales (or, in practice, a selection of scales), and examine the resulting image. A scale-space augments the image with a new dimension for the scale,

$$f(\mathrm{x},\sigma) = \Phi_\sigma f(\mathrm{x}) \qquad ,$$

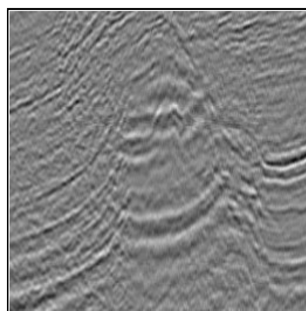
where Φ_σ is a filter at scale σ. The most common filter here is the Gaussian filter, but any one can be used, as long as some properties are satisfied (such as causality). Scale-space theory often involves partial differential equations (PDEs), where going up in scale is equivalent to increasing time in a diffusion process. A Gaussian scale-space can also be written as a PDE and is equivalent to isotropic diffusion.

The difference between two scales of a Gaussian scale-space (DoG, which is similar to a Laplace filter) is a band-pass filter, which can be used to obtain the energy of a frequency band, as is demonstrated in the next exercise.

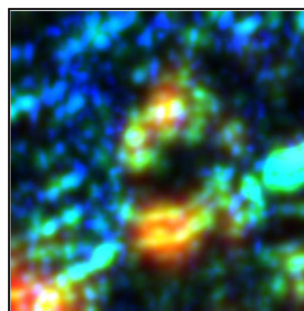
Exercise 34: Power scale-space

Read in the image 'seismic1'. Apply a Gaussian filter at the scales 1, 2, 4 and 8, and take the difference of subsequent scales (for example, `gaussf(a,8)-gaussf(a,4)`). Square these images and blur them with another Gaussian filter with the same scale as the larger of the two images. This is the power of the chosen frequency band. Now place the three images in different planes of a color image (use `colorspace` as shown in Section 10). Make sure that the low-frequency power is represented by red, and the high-frequency power by blue.

Play around with the chosen scales. The functions `scalespace` and `scale2rgb` should make this easier.



'seismic1'



power scale-space in color

A non-linear scale-space can be built (for example) using the closing or opening operations. Such a scale-space is also known as a sieve, since at each scale all image features smaller than σ have been removed. Integration over the images results in a granulometry, which can be normalized to obtain a cumulative size distribution.

Exercise 35: Granulometry

Read in the image 'cermet', and apply `erfclip` around grey-value 128, with a range of 64. This removes the intensity fluctuations in the light part. We will measure the size distribution of this image in two ways: with a granulometry, and by measuring the binarized objects with `measure`. The following commands perform the latter:

```
>> data = measure(a<128,a,{'size','feret'});
>> [binx,I] = sort(data.feret(2,:));
>> biny = cumsum(data.size(I));
>> biny = biny/biny(end);
>> figure; plot(binx,biny,'b.-')
```

Study this code carefully, and understand what it does: `data.feret(2,:)` is the second Feret diameter for all objects (the smallest diameter). The x-axis is formed by these values. We plot the weight of the objects against this diameter, in a cumulative distribution. The variable `I` contains the order of the objects after being `sorted`, and is used to put the `size` array in the same order. Finally, the cumulative weight is normalized to 1.

Now apply a closing to the image `a` at the scales `greyx = sqrt(2).^[1:12]`, and compute the mean grey-value for each of the results (do this using a `for` loop). These values should be in an array `greyy`. Now normalize this array using `mean(a)` and `max(a)` as the lower and upper bounds for the cumulative distribution (the first one is the result at scale 0, the other at scale inf. Now plot this distribution to the same figure window by bringing the previous plot to the foreground, and executing the following commands:

```
>> plot(greyx,greyy,'ro-')
>> legend({'binary','granulometry'},2)
>> set(gca,'xscale','log')
```

(The last command sets the x-scaling to logarithmic, which is a good idea since we also measured our distribution logarithmically). Note that the points at which we measured the distribution with the granulometry are in good agreement with the binary version. However, we don't know what happens in between those points. We can compute any point in this distribution independently from the other points. Figure 6 shows what you should see as a result.

12.2 Hough Transform

The Hough Transform is a technique to detect pre-defined shapes. The original Hough Transform is used to detect straight lines; detection of other shapes can be done in a similar way. If you want to know more about the Hough Transform see:

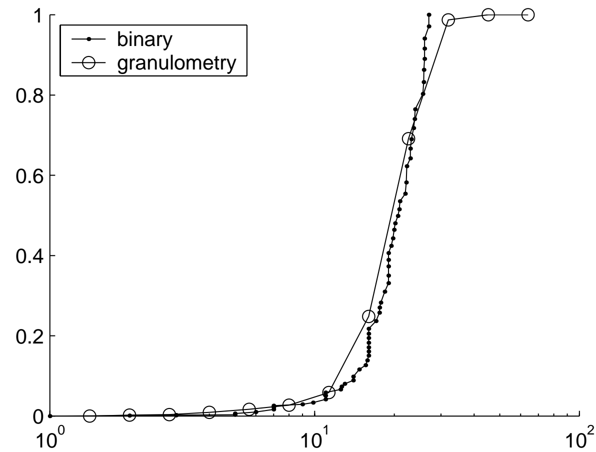


Figure 6: Result for Exercise 35.

- Leavers, V.F., "Shape detection in computer vision using the Hough transform", Springer-Verlag, 1992.

A line can be parameterized by (see Figure 7)

$$p_0 = x\cos(\theta_0) + y\sin(\theta_0) \qquad ,$$

where p is the algebraic length of the normal of the line that passes through the origin, and θ is the angle that this normal makes with the x-axis.

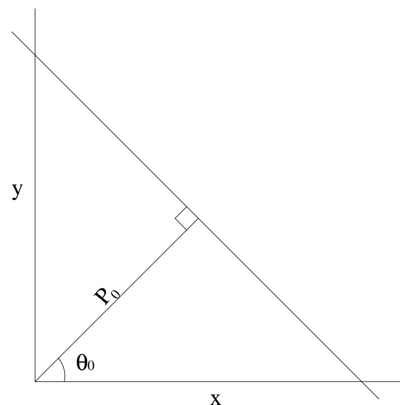


Figure 7: Line parameterization.

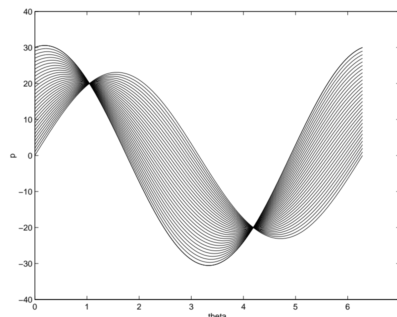
To demonstrate the Hough Transform, we first have to make the vectors x and y, that together compose a line.

```
>> x = 0:30;
>> p0 = 20; theta0 = pi/3;
>> y = (p0-x*cos(theta0))/sin(theta0);
>> plot(x,y)
```

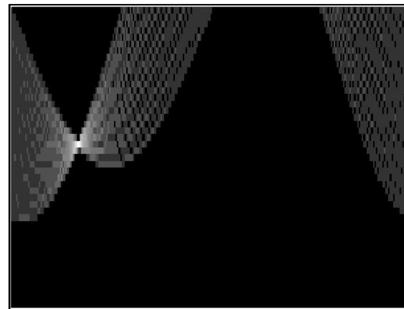
There are a lot of lines that go through a point (x_1, y_1). However, there is only one line that goes through all points (x_i, y_i). At each point we determine all lines (combinations of p and θ) that go through that point:

```
>> theta = 0:pi/256:2*pi;
>> p = x(1) * cos(theta) + y(1) * sin(theta);
>> plot(theta,p)
```

This results in a parameter space as shown. The axes of this space are the parameters you are looking for (in this case p and θ).



Parameter space



Sampled parameter space

Exercise 36: Understanding the parameter space

In the figure, you can see two points where all lines get together.

- What do these points represent?
- Why are there two points?
- Compare this result to a parameter space of another line.
- How could you reduce the size of the parameter space?
- Are there any advantages or disadvantages of reducing the size?

Exercise 37: Implementing the Hough Transform

Now that the basic idea of the Hough Transform has been explained, we have to implement the Hough transform so that you can apply it on binary images and do measurements in the parameter space.

1. Make an binary input image of size 32x32 containing one or more lines.
2. Determine the necessary size of the parameter space if you want to measure θ from 0 to 2π with an accuracy of $\pi/128$, and p from 0 to $32\sqrt{2}$ with an accuracy of 1.
3. Make an empty parameter space image of the determined size.
4. Fill the parameter space:
   - For each object point in the image determine all possible combinations of p and θ.
   - For each combination of p and θ determine the corresponding pixel in the parameter space image and increment the value of that pixel by one.
5. Find the maximum in the parameter space.
6. Determine the corresponding values of p and θ.

Apply your Hough Transform to a binarized version of the image 'schema'.

Even for small images, the Hough transform is a time consuming process. Smart programming will decrease the execution time dramatically. For shorter execution times, the number of `for`-loops has to be reduced.

Exercise 38: Reducing execution time

Compare the calculation times `t1` and `t2` of

```
nx = 100;  ny = 100;  x = 0:99;
a = newim(nx,ny);  b = newim(nx,ny);
tic
for q = 5:5:30;
   y = round((1+cos((x+q)/25))*40+10);
   for ii=1:length(x)
       a(x,y) = a(x,y)+1;
   end
end
t1 = toc
tic
for q = 5:5:30;
   y = round((1+cos((x+q)/25))*40+10);
   I = y + x*ny;
   b(I) = b(I)+1;
end
t2 = toc
```

Use this to speed up your Hough Transform. The variable `I` in the second part is an array containing linear indices into the image `b`. Note how it is computed: the column number multiplied by the height of each column, plus the row number. MATLAB arrays (and thus also images) are stored column-wise.

12.3 Watershed Transform

The watershed transform is a morphological segmentation tool. Imagine a 2D image as a 3D landscape, and imagine that landscape being flooded. Each of the local minima is an independent source, and as the water level raises, we want to keep the different pools separated. To do so, we raise watersheds (dykes) in between them to avoid one's water to mix with the other's (see Figure 8). The pools form a tessellation (segmentation) of the image, and the watersheds are the boundaries between them. To reduce the number of regions found, it is common to apply some smoothing operation to the input image (one that reduces the number of local minima). Even so, objects are often segmented into many pieces, which must be joined in a post-processing step, based on similarity (e.g. variance of the pixels of both segments together). The watershed transform (`watershed`) as it is implemented in DIP*image* can merge regions while they are being grown, based on their size and 'depth' (grey-level difference between the lowest and the highest point in a region, at the moment the merging takes place). This produces acceptable results, but is not as flexible as a post-processing step.

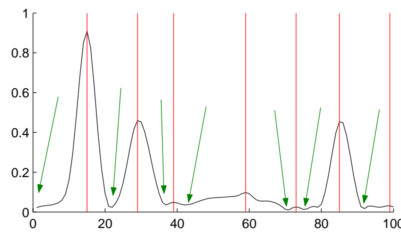
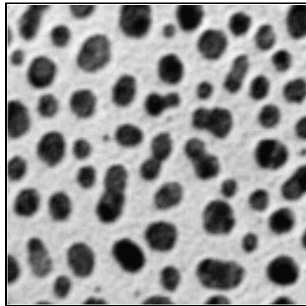
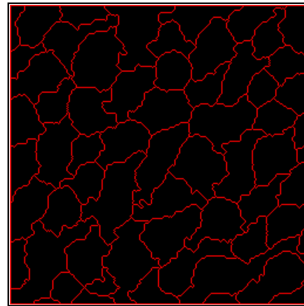


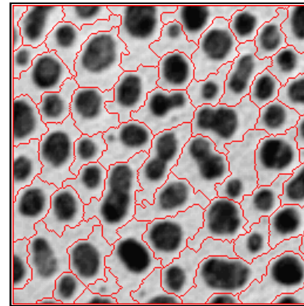
Figure 8: Watershed on 1D image. The arrows indicate the local minima that act as sources. The red lines are the watersheds.



'cermet'

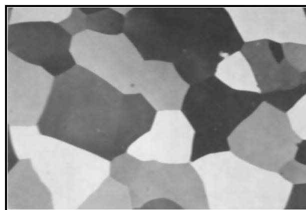


watershed

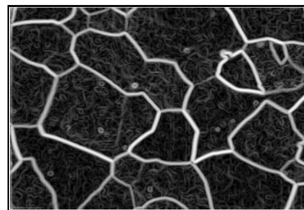


overlay

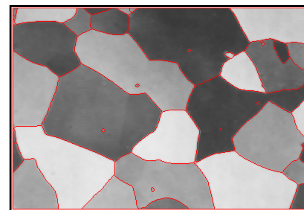
A watershed can be applied directly to an image where the objects are dark, and separated by light borders, like 'cermet', but not to an image where we want to separate objects with different grey-values, like 'alumgrns'. In such a case, you can apply the algorithm to the result of an edge-detection filter (like `gradmag`).



'alumgrns'



gradient magnitude



watershed overlay

A List of functions and operators

A.1 Functions

File I/O

`readim`	Read image from file
`writeim`	Write image to file
`readcolorim`	Read color image from TIFF file
`writecolorim`	Write color image to TIFF file
`readavi`	Read AVI
`writeavi`	Write AVI

Display

`overlay`	Overlay image with mask
`orientationplot`	Orientation plot
`dipgetimage`	Retrieves an image from a display
`dipcrop`	Crop image from display
`dipgetcoords`	Get coordinates of clicks
`diproi`	Interactive rigion of interest selection
`diptruesize`	Set figure size
`dipclf`	Clear all image windows

Generation

`newim`	New image
`ramp`	Ramp
`xx`	Creates an image with x coordinates
`yy`	Creates an image with y coordinates
`zz`	Creates an image with z coordinates
`rr`	Creates an image with r coordinates
`phiphi`	Creates an image with phi coordinates
`testobject`	Creates bandlimited test objects
`noise`	Add noise to an image
`drawline`	Creates a line in an image
`drawpolygon`	Creates a polygon in an image
`gaussianblob`	Sets a Gauss shaped spot into an image

Manipulation

`shift`	Shift an image
`rotation`	Rotate an image around an axis
`rotation3d`	Rotate a 3D image freely
`mirror`	Mirror an image

`resample`	Resample an image
`subsample`	Subsample an image

Point

`clip`	Grey-value clipping
`erfclip`	Grey-value error function clipping
`stretch`	Grey-value stretching
`hist_equalize`	Histogram equalization
`threshold`	Thresholding
`lut`	Look-up table (with interpolation)
`get_subpixel`	Retrieves subpixel values in an image

Filters

`convolve`	General convolution filter
`gaussf`	Gaussian blurring filter
`unif`	Uniform blurring filter
`maxf`	Maximum filter
`minf`	Minimum filter
`medif`	Median filter
`percf`	Percentile filter
`varif`	Variance filter
`gabor`	Gabor filter
`gabor_click`	Interative Gabor filter

Differential Filters

`gauss_derivative`	Gaussian derivatives
`dx`	First Gaussian derivative in the X-direction
`dy`	First Gaussian derivative in the Y-direction
`dz`	First Gaussian derivative in the Z-direction
`gradmag`	Gradient magnitude
`gradientvector`	Gradient vector
`dxx`	Second Gaussian derivative in the X-direction
`dyy`	Second Gaussian derivative in the Y-direction
`dzz`	Second Gaussian derivative in the Z-direction
`dxy`	Second Gaussian derivative in the XY-direction
`dxz`	Second Gaussian derivative in the XZ-direction
`dyz`	Second Gaussian derivative in the YZ-direction
`dgg`	Second Gaussian derivative in the gradient-direction
`dcc`	Second Gaussian derivative in the contour-direction
`laplace`	Laplace operator
`laplace_plus_dgg`	Laplace + Dgg
`laplace_min_dgg`	Laplace - Dgg
`hessian`	Hessian matrix of an image

Adaptive Filters

`kuwahara`	Kuwahara filter for edge-preserving smoothing
`selectionf`	Selection filter
`tframehessian`	Second derivatives driven by structure tensor
`gsdif`	Geometry steerd diffusion
`gaussf_adap`	Adaptive Gaussian Filtering
`percf_adap`	Adaptive Percentile Filtering

Binary Filters

`bdilation`	Binary dilation
`berosion`	Binary erosion
`bopening`	Binary opening
`bclosing`	Binary closing
`hitmiss`	Hit-Miss operator
`bskeleton`	Binary skeleton
`bpropagation`	Binary propagation
`brmedgeobjs`	Remove edge objects
`countneighbours`	Count neighbours
`bmajority`	Binary majority voting
`getsinglepixel`	Get single-pixels from skeleton
`getendpixel`	Get end-pixels from skeleton
`getlinkpixel`	Get link-pixels from skeleton
`getbranchpixel`	Get branch-pixels from skeleton

Morphology

`dilation`	Grey-level dilation
`erosion`	Grey-level erosion
`opening`	Grey-level opening
`closing`	Grey-level closing
`dilation_se`	Dilation with a user-defined structuring element
`erosion_se`	Erosion with a user-defined structuring element
`closing_se`	Opening with a user-defined structuring element
`opening_se`	Closing with a user-defined structuring element
`rankmax_opening`	Rank-max opening
`rankmin_closing`	Rank-min closing
`rankmax_opening_se`	Rank-max opening with a user-defined structuring element
`rankmin_closing_se`	Rank-min closing with a user-defined structuring element
`reconstruction`	Reconstruction by dilation
`watershed`	Watershed

Transforms

`ft`	Fourier Transform (forward)

`ift`	Fourier Transform (inverse)
`dt`	Euclidean Distance Transform
`vdt`	Vector Distance Transform
`gdt`	Grey-value Weighted Distance Transform
`label`	Label objects in a binary image
`hull`	Creates the convex hull of binary image

Analysis

`measure`	Measures objects in an image
`msr2obj`	Label each object in the image with its measurement
`msr2ds`	Convert a measurement structure to a PRTOOLS dataset
`measurehelp`	Provides help on the measurement features
`scalespace`	Gaussian scale-space
`morphscales`	Morphological scale-space
`scale2rgb`	Convert scale-space to RGB image
`structuretensor`	Computes Structure Tensor for 2D images
`structuretensor3d`	Computes Structure Tensor for 3D images
`curvature`	Curvature calculation
`opticflow`	Optic flow
`findshift`	Finds shift of two images

Statistics

`diphist`	Displays a histogram
`chordlength`	Chord lengths of the phases in a labeled image
`paircorrelation`	Pair correlation of the phases in a labeled image
`radialdistribution`	Chord Length
`radialmax`	Radial maximum
`radialmin`	Radial minimum
`radialmean`	Radial mean
`radialsum`	Radial sum
`mse`	Mean square error
`mre`	Mean relative error

A.2 Mathematical Operators (grey in, grey out)

Unary

`-`	negate image	`-a`
`round, floor, etc.`	image with only integer pixel values	`round(a)`
`abs`	absolute	`abs(a)`
`sin, log, sqrt, etc.`	math operations on pixel values	`sin(a)`

Binary

Operator	Description	Example
`+`	sum of two images	`a+b`
`-`	minus	`a-b`
`*`	times	`a*b`
`/`	division	`a/b`
`^`	power	`a^2`
`mod`	modulus (signed remainder of a/b)	`mod(a,b)`
`max, min`	selecting pixel values	`max(a,b)`

A.3 Statistical Operators (grey in, single value out)

Unary

Operator	Description	Example
`sum`	sum of pixel values	`sum(a)`
`mean`	mean pixel value	`mean(a)`
`std`	standard deviation of pixel values	`std(a)`
`max`	maximum pixel value	`max(a)`
`min`	minimum pixel value	`min(a)`
`median`	median pixel value (50 percentile)	`median(a)`
`percentile`	p percentile of the pixel values	`percentile(a,p)`

A.4 Logical Operators (binary in, binary out)

Unary

Operator	Description	Example
`~`	negate image	`~ b`

Binary

<table>
<tr><td>&</td><td>and</td><td>b & c</td></tr>
<tr><td>|</td><td>or</td><td>b | c</td></tr>
<tr><td>xor</td><td>xor</td><td>xor(b,c)</td></tr>
</table>

A.5 Comparison Operators (grey in, binary out)

Binary

Operator	Description	Example
`==`	equality	`a == b`
`~=`	inequality	`a ~= b`
`<`	smaller than	`a < b`
`<=`	smaller or equal than	`a <= b`
`>`	greater than	`a > b`
`>=`	greater or equal than	`a >= b`

A.6 Tricks

`a(:)  = 0`	put all pixel values to zero
`c = newim(a)`	create an empty image with size of `a`
`c = newim(a,'bin')`	create an empty binary image with size of `a`
`c = a(left:right,top:bottom)`	extract a square portion of an image
`a = +b`	convert binary image into greyvalue image
`b = rr(a) <= r`	create binary disk with radius r
`joinchannels('RGB',r,g,b)`	create an RGB image with `r`, `g` and `b` components
`colorspace(a,'Lab')`	convert a color image to L*a*b* color space
`[v,p] = max(a)`	value and location of the global maximum of `a`