Chapter 5

An Introduction to Image Processing

This chapter provides the reader with a few elements on image processing with MATLAB®, which comes equipped with 2D (two dimension) functions, necessary when working in this field, a field not too different from 1D signal processing.

This chapter is merely an introduction. The reader can benefit from reading [12], a rather extensive overview of what is done with images, both still and animated. Some important problems, related to sampling, rectangular, hexagonal or of another kind, to perception, to content aspects in terms of objects, etc., will not be discussed here. The only thing we will be dealing with is handling two dimension arrays. We will also explain how to program some of the functions contained in the "image" *toolbox*.

Examples in this chapter are illustrated by figures that cannot perfectly render the phenomena we are trying to underline. The printing process, whether monochrome or not, adds its own imperfections (quantization, weaving, number of colors, color transcription, etc.) when rendering images. In fact, every part of the digital processing chain, from the data recording device to the printer, has a role that will not be covered in this book.

5.1 Introduction

5.1.1 Image display, color palette

From now on, an image will be considered as a set of *pixels* (the contraction of *picture element*), associated with a rectangular grid of the original image (Figure 5.1).

In MATLAB®, there are several ways to display an image:

Digital Signal and Image Processing using MATLAB®: Volume 1 Fundamentals,
First Edition. Gérard Blanchet and Maurice Charbit.

Figure 5.1 – *Each point of the original image has an 8-bit coded "gray-level". Each pixel appears as a gray square*

– either directly with an $(N \times M \times 3)$ or $(N \times M \times 4)$ array depending on the color model: RGB (**R**ed, **G**reen and **B**lue), CMYK (**C**yan, **M**agenta, **Y**ellow and blac**K**), HSL (**H**ue, **S**aturation and **L**ightness), CIE Lab ("Commission Internationale de l'Eclairage": **L** is for luminance, and **a** and **b** are color component coordinates), etc.

  In the following example, an image in JPEG format is imported with the use of the `imread` function as a 3 dimension $800 \times 580 \times 3$ array, the 3 indicating that there are three RGB color planes. Notice that the data type used is the *8-bit unsigned integer*:

```
>> xx=imread('elido72.jpg','jpeg');
>> whos
  Name       Size                    Bytes  Class

  ans        1x94                      188  char array
  xx       800x580x3                1392000  uint8 array
```

– either by using a 2D (short for 2 dimension) array and a color palette. This is the display mode we will be using; it is called an *indexed representation.*

Let $\boldsymbol{A} = [a(i,j)]$, with $1 \leq i \leq N$ and $1 \leq j \leq M$, be an $N \times M$ array. The number $a(i,j)$, placed in line i and column j, indicates the color of the point with coordinates (i,j) in the image after it has been "sampled" and "quantified". The line index i represents the horizontal position, and the column index j represents the vertical position. The point with the coordinates $(1,1)$ is placed in the top-left corner (see Figure 5.2).

EXAMPLE **5.1 (Pixelizing an image)** Type:

```
>> image1=[32 0 48;0 16 0];
>> image(image1); colormap('gray')
```

The image displayed is comprised of 6 points, or *logical pixels*, and the one associated with `image1(1,1)` is the one in the top-left corner (Figure 5.2).

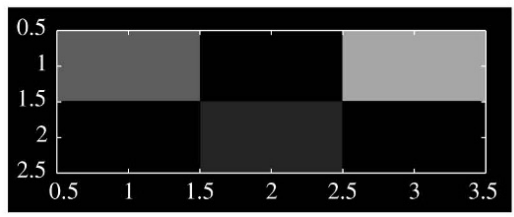

Figure 5.2 – *Six logical pixels: notice the integer x- and y-coordinate corresponding to the "center" of each pixel*

Notice that an element of the array with the index (i, j) can be associated with *several* physical pixels of the display window. In fact, there is no reason for the number of values of the matrix of elements $a(i, j)$ to be equal to the number of *physical pixels* of the display window. Hence, a point with the coordinates (i, j) can be represented by several *physical pixels*, just as a *physical pixel* can be used to represent several points with the coordinates (i, j). From now on, when we use the word *pixel*, we mean a *logical pixel*, that is to say elements identified by the pair (i, j).

If we want to display a figure and preserve its real size (one screen pixel corresponding to one image pixel), we will be using the properties `units`, `Position`, `AspectRatio`, etc. (these parameters can change from one MATLAB® version to the next). In example 5.1, a real-size display is achieved by typing:

```
>> set(gca,'units','pixels','Pos',[20 20 fliplr(size(image1))])
```

where the property `Pos`, or `Position`, is given in the form `[left bottom width height]`. The reading of relevant phase plot properties (called `CurrentAxes`) can be done using `get(gca)` or `et(gcf,'CurrentAxes')`.

In the indexed representation, $a(i, j)$ indexes a color array called the *palette* (Figure 5.3). The color palette is a $(P \times 3)$ array where each line is used to code a color according to its *Red*, *Green* and *Blue* components (RGB) using a real number between 0 and 1.

This representation is convenient since most bitmap editing programs can provide an image description in three planes, each one of them corresponding to a primary color R, G or B, encoded as an integer between 0 and $2^n - 1$ (n-bit encoding) (see section 5.2). The images we will be considering will be "in levels of gray". MATLAB® has a "gray" palette that can be activated using the `colormap('gray')` instruction.

Type `colormap('gray')` then `colormap`. You get a (64×3) array with three identical columns of values between 0 and 1:

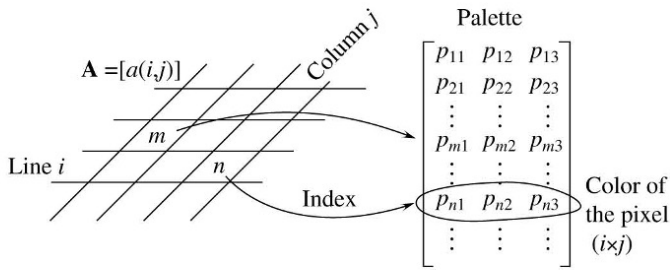

Figure 5.3 – *Connection between the image array and the palette*

```
ans =
         0         0         0
    0.0159    0.0159    0.0159
    0.0317    0.0317    0.0317
    0.0476    0.0476    0.0476
    ...
    ...
    ...
    0.9841    0.9841    0.9841
    1.0000    1.0000    1.0000
```

COMMENTS:

- in example 5.1 the zero values of the `image1` array are redefined as 1 and therefore index the color $(0, 0, 0)$, which is black (Figure 5.2);

- help for the commands `image`, `imagesc` and `colormap` should particularly be looked into;

- the "gray" palette is constructed linearly. Each column is of the type `[0:1/63:1]'` ($1/63 \simeq 0.0159$). This does not quite correspond to the perception we have of brightness. The visual response is roughly proportional to the logarithm of the intensity (*Fechner-Weber law*), hence the progression of the levels of gray should correspond to this law. In practice, the palette's linear conformation makes our work much easier since palette index lines and gray levels are related by an affine relation;

- other palettes come standard in the basic version of MATLAB® to make the user's work easier. Use the `help color` command to learn more about them. Also, nothing stops you from defining your own palettes. For example, to get a display with 256 levels of gray, all you need to do is create a `cmap` array as follows:

```
cmap=[0:255]'*ones(1,3)/255;
colormap(cmap);
```

5.1.2 Importing images

If you don't have an image you can perform tests on in MATLAB®, you can always create one based on raw format images (no header) using image processing software. At the same time, you can save the palette, if that is possible. The following function allows you to read and/or create a file that can be used directly by MATLAB®. The image that was chosen is an image universally used by "image processors" to compare results obtained for different implementations. It is referred to as *lena*. We assume that the data is stored as unsigned 8-bit coded integers.

Figure 5.4 – *Test image*

```
function pixc=raw2matf(fnameI,Nrow,Ncol,Tr,Fc,fname0)
%!=========================================================!
%! Reading a raw image file                                !
%! SYNOPSIS: pixc=RAW2MATF(fnameI,Nrow,Ncol,Tr,Fc,fname0)  !
%!   fnameI     = raw file ([.raw])                        !
%!   Nrow,Ncol  = image dimensions                         !
%!   Tr         = when 'T': transposing the image          !
%!   Fc         = when 'F': creating the file fname0 (.mat) !
%!   fname0     = resulting file ([.mat])                  !
%!=========================================================!
if nargin<6, fname0='fictrav.mat'; end
if nargin<5, Fc='N'; end
if nargin<4, Tr='N'; end
%===== raw image
nFS=findstr(fnameI,'.');
if isempty(nFS),
   NFE=[fnameI,'.raw'];
```

```
else
   NFE=fnameI; fnameI=fnameI(1:nFS-1);
end
fid=fopen(NFE,'r'); [pixc,Npix]=fread(fid,'uchar');
if (Npix ~= Nrow*Ncol)
sprintf('Dimension error: %d*%d ~= %d',Nrow,Ncol,Npix)
    return
end
pixc=reshape(pixc,Nrow,Ncol); if Tr=='T', pixc=pixc'; end
fclose(fid);
%===== creating the .MAT file
if Fc=='F',
   sprintf ('Creating the file %s',fname0)
   eval(['save ' fname0 ' pixc'])
end
return
```

The image can be loaded and displayed (Figure 5.4) by the following program:

```
%===== tstraw2mat.m
pixc=raw2matf('lena50',256,256,'T');
%===== palette construction
cmap=([255:-1:0]'/255)*[1 1 1];
%===== displaying with the new palette
imagesc(pixc); colormap(cmap); axis('image')
```

In this program, the palette is defined, but it can also be saved in the image processing application and stored in the `.mat` file.

COMMENTS:

- recent versions of MATLAB® allow you to directly load and save images in formats such as "bmp" (*bit map*), "tiff" (*Tagged Image File Format*), "jpeg" (*Joint Photographic Expert Group*), "pcx" (*Personal Computer Exchange*), etc. using the `imread` and `imwrite` functions;
- notice that when a palette is used, the `image` function works with an array of integer values (the non-integer values are rounded) between 1 and M. The values above M are constrained to M, and those below 1 are constrained to 1. Type at the end of the previous program:

  ```
  p256=pixc+256; subplot(121); image(p256); axis('image')
  p0=pixc-256; subplot(122); image(p0); axis('image')
  colormap(cmap)
  ```

  You should see a white square and a black square;
- it is usually preferable to use the `imagesc` function (suffix *sc* as in *scale*) which displays a version with the same scale as the original image: the values are changed to fit between 1 and `size(colormap,1)`;

– the image's color levels can have values such that it becomes difficult to display the image because of a few extreme values. The use of `image` or `imagesc` may not be satisfactory. The following function allows you to improve the display by modifying the color distribution:

```
function mydisp(pixr,cmap,stdpar,style)
%!=============================================!
%! Displaying with gray level control          !
%! SYNOPSIS: MYDISP(pixr, cmap, stdpar, style) !
%!    pixr   = image                           !
%!    cmap   = palette                         !
%!    stdpar = controls the min and max indices !
%!    style  = see AXIS function               !
%!=============================================!
if nargin<2,
    sprintf('Error on arguments');
    return
end
if nargin<4, style='image'; end
if nargin<3, stdpar=3; end
if (stdpar <= 0 | stdpar >10), stdpar=1; end
moy=mean(mean(pixr)); stdp=stdpar*std(std(pixr));
minp=moy-stdp; maxp=moy+stdp;
idx=1+(pixr-minp)*(size(cmap,1)-1)/(maxp-minp);
colormap(cmap); image(idx); axis(style)
return
```

– when using *scanners* or digital cameras, the standard sampling values, in "dots per inch" (*dpi*), are (300×600)[1], $(600 \times 1{,}200)$,$(1{,}600 \times 3{,}200)$, $(2{,}700 \times 2{,}700)$, etc., and for quantification, 8, 10, 12, etc. bits for each of the primary colors.

5.1.3 Arithmetical and logical operations

Because images in MATLAB® are matrices, the usual operations can be directly applied to them. In particular, arithmetic and logical operations between images, pixel by pixel, can be performed from the array values they are associated with.

Thus, the sum of two images `pix1` and `pix2` of the same size can be written `pix1 + pix2`, or just as the square root of `pix` can be written `sqrt(pix)`. You only have to make sure that the obtained values are consistent with the color palette, or you can use the functions `imagesc` or `mydisp`.

The logical operations have to be performed sur des "représentations entières". for "whole representations". The codes associated with pixels, indices or

[1] Meaning 300 dots per inch in one direction, and 600 dots per inch in the other.

values must be first converted to this format using functions `unint8`, `unint16`, etc. It must be ensured that the ranges are compatible with this representation. Here is an example: consider the two images in Figure 5.5 – we are going to perform the AND function between the figure on the left and the figure on the right.

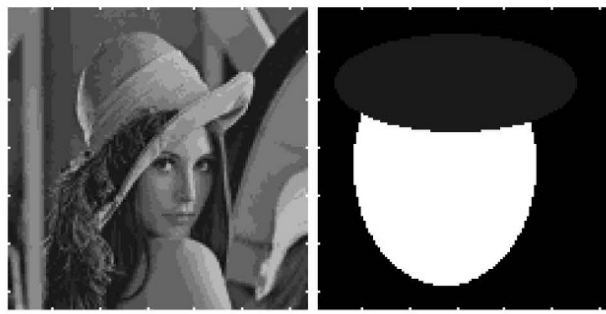

Figure 5.5 – *Logical operation AND*

The result is shown in Figure 5.6: the black areas of the image on the right in Figure 5.5, which are encoded as byte 0000 0000, force the corresponding areas of the resulting image to be black. This is because if `xxxx xxxx` is the value associated with a pixel from the first image, the logical AND of `xxxx xxxx` and `0000 0000` is `0000 0000`. The white areas of the image on the right are encoded as byte 11111111, leaving untouched the values of the corresponding pixels of *Lena*. This is because the logical AND of `xxxx xxxx` with `1111 1111` is `xxxx xxxx`. Finally, the areas of the image on the right which are encoded as `yyyy yyyy` lead to a pixel value with some bits unchanged, and others set to 0.

Figure 5.6 – *Result of the logical AND*

The `ANDlog` function performs the logical AND operation we have just described:

```
function pixr=ANDlog2(pix1,pix2)
%!=============================================!
%! Logical AND between two images              !
%! SYNOPSIS: pixr=ANDLOG(pix1,pix2) (UINT8)    !
%!    pix1 = first image (gray palette)        !
%!    pix2 = second image (gray palette) (UINT8) !
%!    pixr = image result (UINT8)              !
%!=============================================!
if (nargin<2), error('Parameters missing'); end
N1=size(pix1);
if (N1 ~= size(pix2)),
    error('Matrix dimensions are not appropriate')
end
pixr=bitand(pix1,pix2);
return
```

The program `testlogic.m` which uses the `ANDlog fonction`, leads to Figure 5.6:

```
%===== testlogic.m
clear all
load lena25;                  % loading and displaying
subplot(131); imagesc(pixc);  % the first image
colormap(cmap); axis('image');
load testlog1;                % loading and displaying
subplot(132); imagesc(pixtl); % the second image
axis('image')
%===== logical operation
pixr = ANDlog(uint8(pixc),uint8(pixtl));
subplot(133); imagesc(pixr); axis('image');
```

Exercise 5.1 (Logical functions) (see p. 437)

1. Write a function that uses the four basic logical operators AND, OR, EOR and NOT, as well as the comparison operators. Use the `bitand`, `bitor`, `bitxor`, `bitcmp` functions to implement it. It is assumed that the images in grayscale are indexed with an index between 0 and 255. The index is then converted to `UINT8` "8-bit unsigned integer" format before processing.

2. Write a test program for the logical operator NOT, as well as for the logical operator that is true when $a_m \leq b_m$, where a_m and b_m are the bits corresponding to two bytes we wish to compare.

5.2 Color spaces

There is considerable literature – [25], [24], [14], etc. – on aspects related to representations of color images. To simply specify a color space (indexed-color,

RGB, CMYK, Lab color, etc.) in image processing software it is necessary to have a few ideas of these spaces [28] [11].

The perception of color is, in all accounts, a complex process. When asked to describe the color of a light source, the answers involve three parameters: *hue*, for example "it pulls on the yellow with a hint of green", *brightness* of the source, which is relative to the ambient brightness and *saturation*, for example "this is a deep red".

The creation of a color representation system, in regards to *color space*, is built on these types of parameters. It is influenced by the following findings:

– The eye is made up of photoreceptors: the three types of *cones*, involved in the perception of color and *rods*, mainly sensitive to intensity. It is said that humans have a *tri-color* perception. Each cone is characterized by a sensitivity function called *spectral power distribution* (SPD) function of the wavelength λ. The three SPDs are denoted by $L(\lambda)$ (long), $M(\lambda)$ (medium) and $S(\lambda)$ (short) (Figure 5.7).

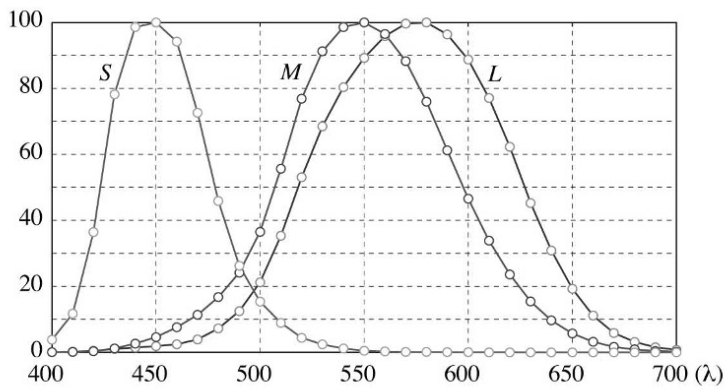

Figure 5.7 – *The normalized sensitivity functions (RLab-D65)*

In fact, these sensitivity functions are not the same from one individual to another and are only one element among many that are involved in vision. The perception of color depends indeed on many elements, ambient brightness, contrast, phenomena of vision adaptation, etc.

– In 1931, the CIE (*Compagnie Internationale de l'Eclairage*) standardized three weighting functions for modeling, by the application of the SPD of a light source, the process of color perception in a human being.

These three functions are designated by the *Color Matching Functions* (CMF) and denoted by $\overline{x}$, $\overline{y}$ and $\overline{z}$ (Figure 5.8 obtained using the `chromaticity.m) program`.

```
%===== chromaticity.m
% A Guided Tour of Color Space, C. Poynton
% (A Technical Introduction to Digital Video
% Wiley & Sons, New York, 1996)
vallambda=false;
[xb,yb,zb,lambda]=ciecmf();
N=length(xb);
%===== CIE color matching functions
figure(1); plot(lambda,[xb',yb',zb']), grid
%===== (x,y) chromaticity diagram
tsm=[xb;yb;zb]; stsm=sum(tsm);
x=xb ./ stsm; y=yb ./ stsm;
figure(2), plot([x x(1)],[y y(1)],'-r'), grid on
hold on, plot(x,y,'o')
%===== D65 CIE illuminant
xD65=.3128; yD65=.3290; plot(xD65,yD65,'x');
text(xD65+.01,yD65,'D65')
if vallambda
    for k=1:N, text(x(k)+.01,y(k),num2str(lambda(k))), end
end
hold off
```

```
function [xb,yb,zb,lambda]=ciecmf()
xb=[.0143 .0435 .1344 .2839 .3483 .3362 .2908 .1954 ...
    .0956 .0320 .0049 .0093 .0633 .1655 .2904 .4334 ...
    .5945 .7621 .9163 1.0263 1.0622 1.0026 .8544 .6424 ...
    .4479 .2835 .1649 .0874 .0468 .0227 .0114];
yb=[.0004 .0012 .0040 .0116 .0230 .0380 .0600 .0910 ...
    .1390 .2080 .3230 .5030 .7100 .8620 .9540 .9950 ...
    .9950 .9520 .8700 .7570 .6310 .5030 .3810 .2650 ...
    .1750 .1070 .0610 .0320 .0170 .0082 .0041];
zb=[.0679 .2074 .6456 1.3856 1.7471 1.7721 1.6692 ...
    1.2876 .8130 .4652 .2720 .1582 .0782 .0422 .0203 ...
    .0087 .0039 .0021 .0017 .0011 .0008 .0003 .0002 ...
    0 0 0 0 0 0 0 0];
lambda=[400:10:700];
return
```

Knowledge of the CMF allows a system of color representation to be built;

– the representation system defined by the CIE uses three components, X, Y and Z, where Y represents the sensitivity of humans to brightness. If $p(\lambda)$ represents the SPD of a light source based on the wavelength λ, its brightness is equal to $\int p(\lambda)\overline{y}(\lambda)d\lambda$. With $X = \int p(\lambda)\overline{x}(\lambda)d\lambda$ and $Z = \int p(\lambda)\overline{z}(\lambda)d\lambda$, triplet (X, Y, Z) is called "XYZ-tristimulus".

The luminosity is usually set relative to a "reference white" by a number between 1 and 100 instead of a candelas per square meter.

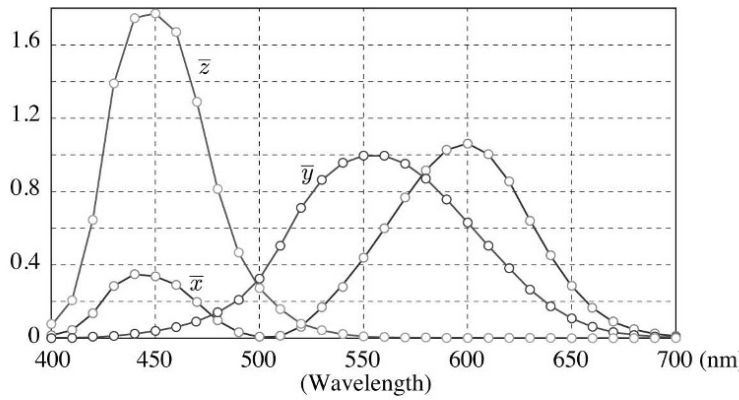

Figure 5.8 – *CIE color matching functions*

The function giving X, Y and Z from L, M and S is linear.

Many color systems are built from components X, Y and Z;

– in addition to brightness, the CIE defines the two other components of chromaticity by:

$$x = \frac{X}{X+Y+Z} \text{ and } y = \frac{Y}{X+Y+Z} \tag{5.1}$$

A color can therefore be described by the triplet (Y, x, y). It is denoted by CIE-Yxy model. There are several similar representations derived from the latter: CIE-Yuv, CIE-$Yu'v'$ (1960), CIE-$Y^*u^*v^*$ (1976), CIE-$Y^*a^*b^*$ (1931), all of which aim to standardize the perception when considering the differences between points located at equal distances in the diagram.

The chromaticity diagram (Figure 5.9) is obtained by the `chromaticity.m` program.

The linear combination $\alpha XYZ_1 + \beta XYZ_2$ of two XYZ-tristimuli gives a linear relationship between x and y (straight line $\Delta_{\alpha\beta}$ of Figure 5.9):

– a simple way to build a color is to leave the so-called three "primary" colors and to perform a linear combination of the SPD. Color additive synthesis, is often spoken about. The RGB (Red, Green, Blue) system is widely used in image reproduction systems, video projectors for example with three beams R, G and B, or to encode color images. The reproduction of an image on a screen or computer monitor requires a reference for the "white" to be defined. The CIE has done this digitally by defining "whites" according to the lighting conditions within their chromaticity (for example the D_{65} point in Figure 5.9). White can also be defined

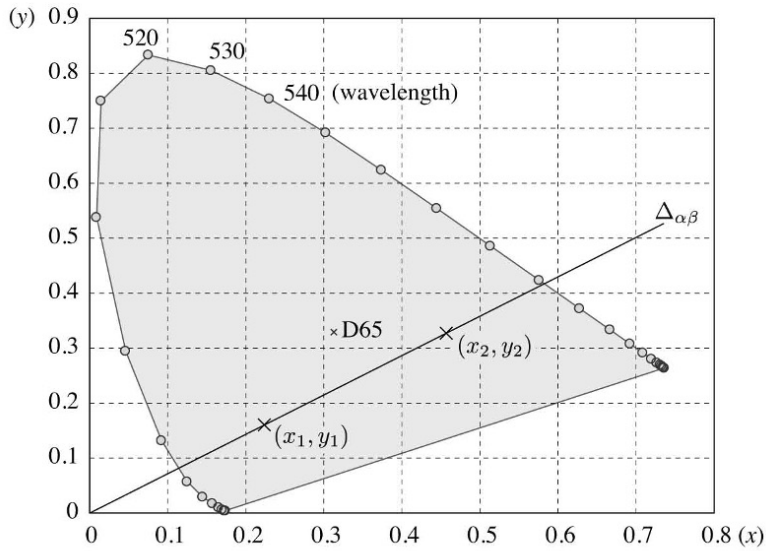

Figure 5.9 – *(x,y) chromaticity diagram. The "CIE D65 illuminant" corresponds to a dayligth reference for the western and northern europe.*

from the light emission of a black body worn at a given temperature (expressed in degrees Kelvin). A system of encoding RGB is therefore specified by the chromaticity of its three primary colors and the reference white.

What follows is three of the most commonly used systems: the RGB (Red, Green, Blue), system the HSV system (Hue, Saturation, Value), also known as HSB (Hue, Saturation, Brightness), and the CMYK (Cyan, Magenta, Yellow, and K for Black) system. MATLAB® uses RGB to specify the colors of the objects graphics or images, indexed or not.

5.2.1 RGB coding

The RGB system allows color to be defined by mixing the three primary colors, red, green and blue. The values of the three components are often coded on 8 bits between 0 and 255. So the triplet (255, 255, 255) codes white, (0, 0, 0) black, (255, 0, 0) a pure red, (100, 100, 100) a gray, etc.

In MATLAB®, each component can be coded with "floating-point" number between 0 and 1 or with a 8 or 16 bit integers.

EXAMPLE **5.2 (Coding a color with 8 bit numbers)** Taper:

```
>> myimage=zeros(1,8,3,'uint8');
>> myimage(:,:,1)=[255 255 255 255 0 0 0 0]; % red plane
>> myimage(:,:,2)=[0 0 255 255 255 255 0 0]; % green plane
>> myimage(:,:,3)=[0 255 0 255 0 255 0 255]; % blue plane
>> image(myimage), axis('image')
```

Obviously the results obtained depend on the characteristics of the visualization system.

RGB coding is implemented in many digital devices: input devices – color scanners, digital photo equipment, camcorders, etc. – or output devices – screens, printers, etc. However, for image processing, it is often more convenient to use a HSV representation.

5.2.2 HSV coding

The HSV model was introduced in 1978 by A. R. Smith [32]. It is a nonlinear transformation of the RGB color space. It is better than the RGB system to specify a tint. In fact, there are three representation systems best suited to the human perception of color: HSL (Hue, Saturation and Lightness), HSI (Hue, Saturation and Intensity) and HSV. Only the last representation will be addressed. HSV defines a color with three components:

1. the tint, or hue, H describes the perceived color, such as red, blue, or yellow, and uses values from 0 to 360. Thus, 0 is red, 45 is a shade of orange and 55 is a shade of yellow;

2. the *saturation* S, or the color intensity, ranges from 0 to 100%. 0 indicates "no color" and 100 indicates an intense color;

3. the *value* V, or color brightness, ranges from 0 to 100%. 0 is always black. According to saturation, 100 can be white or can mean more or less color saturation.

The MATLAB® functions `hsv2rgb` and rgb2hsv are used to perform conversions between color spaces. The `testHSV.m` program gives all the colors obtained with saturation and brightness equal to 1. Type:

```
%===== testHSV.m
h=(0:2:360)'; nbr=length(h); HSVcm=zeros(nbr,3);
HSVcm(:,1)=h/360;          % H
HSVcm(:,2)=ones(nbr,1)*1; % S
HSVcm(:,3)=ones(nbr,1)*1; % V
imr=20; myimage=zeros(imr,nbr,3);
RGBcm=hsv2rgb(HSVcm);
myimage(:,:,1)=ones(imr,1) * RGBcm(:,1)'; % red plane
```

```
myimage(:,:,2)=ones(imr,1) * RGBcm(:,2)'; % green plane
myimage(:,:,3)=ones(imr,1) * RGBcm(:,3)'; % blue plane
image(myimage), axis('image')
```

Other systems such as CIELAB [20, 19] (CIE L*a*b*, 1976 and CIE 15.2 publication, 1986) or CIECAM02 [13] (used in Microsoft Windows Vista) are better suited to the separation of the primary colors.

5.2.3 CMYK coding

The CMYK coding is widely used when printing images to reproduce a wide color spectrum from the three base colors (cyan, magenta and yellow) to which black is added. The CMYK model works by partially or entirely hiding certain colors on a typically white background. Cyan, magenta, and yellow are the three primary colors in Subtractive synthesis, contrary to the red, green and blue (RGB) which use additive synthesis.

Black is used to obtain grays, which would be more difficult to obtain by mixing the three primary colors. They may, however, be added as an extra color to black to accentuate the shades of gray (for example, a brown or orange ink). This method is used, in particular, with black and white photography. It is also possible to add a color (usually of cyan) in order to print a more intense black.

Conversion RGB to CMYK

Color values are meant to be real numbers between 0 and 1. The proposed conversion works from *a* RGB model to *a* CMYK model. In this case the CMYK color uses the most black (K) possible with the least amount of color (CMY) possible. Thus, the #808080 color (gray) will be converted to $(0, 0, 0, 0.5)$ and not $(0.5, 0.5, 0.5, 0)$.

The RGB to CMY conversion is described by:

$$\begin{bmatrix} C' \\ M' \\ Y' \end{bmatrix} = \begin{bmatrix} 1-R \\ 1-G \\ 1-B \end{bmatrix} \tag{5.2}$$

The CMY to CMYK conversion is described by:

$$K = \min\{C', M', Y'\} \tag{5.3}$$

$$\begin{cases} \text{if } K = 1 \text{ then } [C, M, Y, K] = [0, 0, 0, 1] \\ \text{else } \begin{bmatrix} C \\ M \\ Y \\ K \end{bmatrix} = \begin{bmatrix} \frac{C'-K}{1-K} \\ \frac{M'-K}{1-K} \\ \frac{Y'-K}{1-K} \\ K \end{bmatrix} \end{cases} \tag{5.4}$$

```
function CMYK = rgb2cmykm(rgb)
%!=======================================!
%! SYNOPSIS: CMYK = rgb2cmyk(rgb)        !
%! rgb     : (1,3)-matrix -> (R,G,B)     !
%! CMYK    : (1,4)-matrix -> (C,M,Y,K)   !
%!=======================================!
cmy=[1-rgb(1),1-rgb(2),1-rgb(3)];
k=min(cmy);
if(k==1), CMYK=[0,0,0,1]; return; end
CMYK=[(cmy(1)-k)/(1-k),(cmy(2)-k)/(1-k),(cmy(3)-k)/(1-k),k];
return
```

Conversion of CMYK to RGB

The CMY is used as an intermediate coding to perform the conversion between RGB and CMKY. It then converts the CMY value to RGB.

$$\begin{bmatrix} C' \\ M' \\ Y' \end{bmatrix} = \begin{bmatrix} C(1-K)+K \\ M(1-K)+K \\ Y(1-K)+K \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1-C' \\ 1-M' \\ 1-Y' \end{bmatrix} = \begin{bmatrix} (1-C)(1-K) \\ (1-M)(1-K) \\ (1-Y)(1-K) \end{bmatrix}$$

```
function RGB = cmyk2rgbm(cmyk)
%!=======================================!
%! SYNOPSIS: RGB = cmyk2rvb(CMYK)        !
%! CMYK    : (1,4)-matrix -> (C,M,Y,K)   !
%! RGB     : (1,3)-matrix -> (R,V,B)     !
%!=======================================!
cmy=[cmyk(1)*(1-cmyk(4))+cmyk(4),...
    cmyk(2)*(1-cmyk(4))+cmyk(4),...
    cmyk(3)*(1-cmyk(4))+cmyk(4)];
RGB=[1-cmy(1),1-cmy(2),1-cmy(3)];
return
```

NOTE: the two `rgb2cmykm` and `cmyk2rgbm` conversion functions are an unsatisfactory practice. The basic relationship "1−RGB" is indeed only approximative. It originates from the fact that, for example, blue is achieved by simply removing red, green, then yellow. If so, all the yellow from the white source should be filtered, then blue should be obtained. In reality, the three filters, magenta, cyan and yellow absorb light across the spectrum. There are interactions between the three when filtering.

The otained color space (*gamut*) remains less "extended" than that given by the RGB model, the transformation is not linear, and the manipulation of colors is even less intuitive than the RGB model. In practice, these conversions are based on tables or polynomial transformations.

5.2.4 How to extract the RGB information from an image

The RGB information of a digital image can be extracted in two ways, depending on whether the image is indexed or not:

1. an *indexed image* ($n \times m$), as previously shown, is an array of indexes "pointing-out" to a color map (`colormap` function). The latter is, in MATLAB®, a (64×3) matrix (default size). This allows each of the $n \times m$ triplets to be revealed in the table to therefore retrieve the RGB components;

2. non-indexed images ($n \times m$) ("structured images") are composed of three ($n \times m$) arrays giving the values of the red, green and blue components. The following program retrieves the three R, G and B arrays:

```
%===== drawmo.m
myimg=imread('morisse.jpg','JPEG');
size(myimg)
nbcl=64; mcol=[0:nbcl-1]'/(nbcl-1);
mypal=zeros(nbcl,3,3); mypal(:,1,1)=mcol;
mypal(:,2,2)=mcol; mypal(:,3,3)=mcol;
for k=1:3
    figure(k), imagesc(myimg(:,:,k))
    colormap(mypal(:,:,k)); axis('image')
end
```

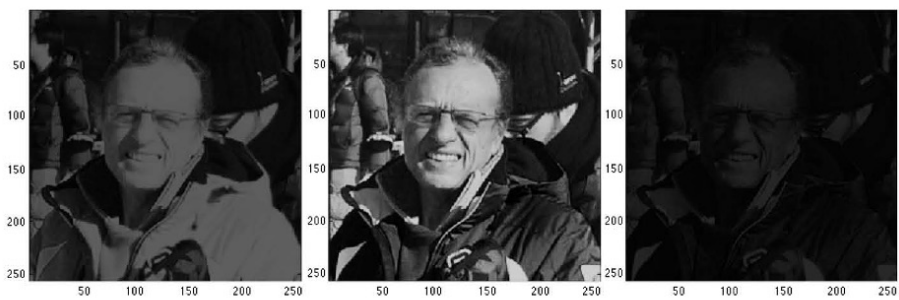

Figure 5.10 – *The R, G and B components*

5.2.5 Converting from color to grayscale

Color to grayscale conversion is based on the perception of luminance. The C.I.E proposes two formulas to characterize this information:

1. in its 709 recommendation, concerning "real" or natural colors:

$$Y = \text{gray} = 0.2125 \times \text{red} + 0.7154 \times \text{green} + 0.0721 \times \text{blue}$$

2. in its 601 recommendation for colors with gamma correction (for example, images seen on a video screen):

$$Y = \text{gray} = 0.299 \times \text{red} + 0.587 \times \text{green} + 0.114 \times \text{blue}$$

The weights used to calculate luminosity are related to the monitor's phosphorus. These weights are explainable by the fact that for equal amounts of color, the eye is most sensitive to green, then red, then blue. This means that for equal "amounts" of green and blue light, the green will look much brighter than the blue.

5.3 Geometric transformations of an image

5.3.1 The typical transformations

The simple geometric transformations, such as translations, rotations and torsions are problematic because of the "integer" nature of the pixels' position in an image (Figure 5.11).

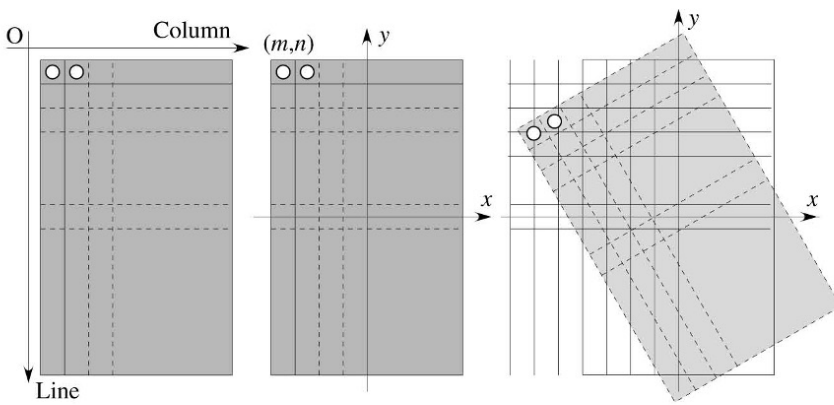

Figure 5.11 – *Rotations of an image*

EXAMPLE **5.3 (Rotation of an image)**
We wish to rotate an image. To make things simpler, we will be using an indexed image in levels of gray:

1. the rotation matrix has the expression:

$$\boldsymbol{M} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

2. we create an array for the pixel coordinates (change from the line and column numbers over to the x, y coordinates). Applying the rotation to every point provides, after rounding the resulting value, and changing back to the line, column representation, leads us to the final image:

```
%===== geomtransf.m
% Geometric transformations / rotation
fmt='jpeg'; fn='imageGG.jpg'; pixc=imread(fn,fmt);
%=====
subplot(121), imagesc(pixc); Spix=size(pixc);
Nr=Spix(1); Nc=Spix(2);
mycmap=[0:1/255:1]'*[1 1 1];       %=== gray colormap
colormap(mycmap); set(gca,'DataAspectRatio',[1 1 1])
set(gca,'units','pixels','pos',[40 40 fliplr(Spix)])
axis off, % turns off labels, marks...
%===== rotation center
xor=(1+Nc)/2; yor=-(1+Nr)/2;
%===== tbidx=indices (columnwise)
tbx=ones(Nr,1)*[1:Nc]; tby=[1:Nr]'*ones(1,Nc);
tbidx=[reshape(tby,1,Nr*Nc);reshape(tbx,1,Nr*Nc)];
idtb=tbidx(1,:)+(tbidx(2,:)-1)*Nr; %=== linear indices
%===== tbcoord=coordinates pixels/rotation center
tbcoord=[tbidx(2,:)-xor;-tbidx(1,:)-yor];
%===== rotation
theta=25; thet=theta*pi/180;
MRot=[cos(thet) -sin(thet);sin(thet) cos(thet)];
tbv=round(MRot*tbcoord);
xmin=min(tbv(1,:)); xmax=max(tbv(1,:)); ncol=xmax-xmin+1;
ymin=min(tbv(2,:)); ymax=max(tbv(2,:)); nlig=ymax-ymin+1;
%===== index reconstruction
tbidxR=[-tbv(2,:)-ymin+1;tbv(1,:)-xmin+1];
pixcR=zeros(nlig,ncol); pixcR2=pixcR-1;
idtbR=tbidxR(1,:)+(tbidxR(2,:)-1)*nlig;
pixcR(idtbR)=pixc(idtb); pixcR2(idtbR)=pixc(idtb);
save pixcR2 pixcR2 thet mycmap Nr Nc; %===
%===== displaying the result
subplot(122), imagesc(pixcR); colormap(mycmap)
set(gca,'DataAspectRatio',[1 1 1]);
set(gca,'units','pixels',...
    'pos',[70+Spix(2) 40 fliplr(size(pixcR))])
axis off
%===== saving the image for median filtering
pxRmn=min(min(pixcR)); pxRmx=max(max(pixcR));
pixcRn=255*(pixcR-pxRmn)/(pxRmx-pxRmn)+1;
imwrite(pixcRn,mycmap,'imageGGR.bmp','bmp')
```

Notice the use of the `imread` and `imwrite` functions, making it possible to read and save images in a given format, "jpeg" in this example.

Rotating two neighboring pixels can result, after rounding, in identical coordinates. This leads us to the conclusion that there are "holes" in the target image. These are clearly visible in the image resulting from the rotation (Figure 5.12). We will see in exercise 5.11 how to deal with these isolated points. It is also possible to process the pixels with identical coordinates using a weighted mean of the source pixels.

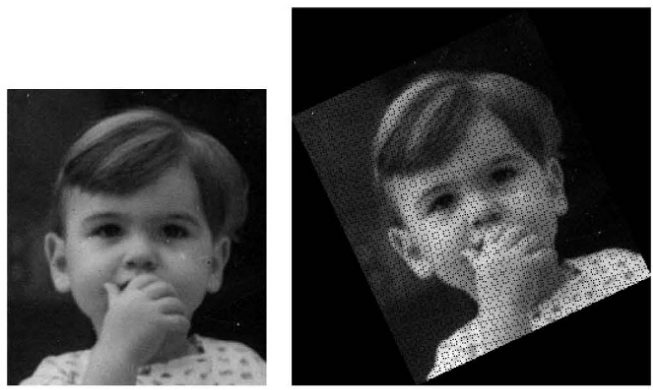

Figure 5.12 – *Flaws due to the rotation*

Generally speaking, *affine* transformations are represented with expression 5.5:

$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.5}$$

x and y are the coordinates of the source $\mathcal{S}$, X and Y those of the target image $\mathcal{T}$. t_x and t_y define the translation applied to the image.

Likewise, the word *torsion* (see exercise 5.3) is used when the relation between $\mathcal{S}$ and $\mathcal{T}$ is of the type 5.6:

$$\begin{bmatrix} U \\ V \\ T \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ e & f & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ and } X = \frac{U}{T}, Y = \frac{V}{T} \tag{5.6}$$

These two types of transformations pose a problem for interpolation(paragraph 5.6.2) and/or undersampling (paragraph 5.6.1) which we will discuss later.

There is no rule that says you have to use an xOy axis system instead of a "line, column" coordinate system (LC coordinates). In the case of a rotation, it allows the transformation matrix to preserve its usual form. In general, the transformation matrix can be identified immediately in LC coordinates.

Exercise 5.2 (Plane transformation) (see p. 439)
Describing a transformation can be done in an interactive way using a simple shape. Here we are going to use the triangle to define the affine transformation we will apply to the image:

1. write a linear transformation function of an $n \times m$ pixel image, knowing that the (2×2) transformation matrix is described in an xOy system;
2. write a program asking the user to define two triangles interactively, which then calculates the (3×3) affine transformation matrix used to go from one triangle to the other;
3. apply the transformation to an image.

Exercise 5.3 (Transformation of a rectangular selection) (see p. 441)
Many image processing applications allow you to deform a rectangular-shaped selection by having an effect on each corner of the selection. Consider expression 5.6 of the "torsion". For a corner with the coordinates x_k, y_k, the coordinates X_k and Y_k after modifications can be expressed:

$$X_k = \frac{U_k}{T_k} = \frac{ax_k + by_k + t_x}{ex_k + fy_k + 1}$$
$$Y_k = \frac{V_k}{T_k} = \frac{cx_k + dy_k + t_y}{ex_k + fy_k + 1}$$

$$\begin{cases} X_k(ex_k + fy_k + 1) = ax_k + by_k + t_x \\ Y_k(ex_k + fy_k + 1) = cx_k + dy_k + t_y \end{cases} \tag{5.7}$$

If applied to all four corners, these expressions make it possible to determine the eight coefficients of the transformation matrix:

1. using 5.7, determine the linear system needed to find the transformation matrix;
2. apply this transformation to an image by assuming that the rectangular selection is applied to the whole image.

5.3.2 Image registration

Many applications – biometrics, identification number recognition, handwriting recognition, etc. – require that an image be forced to fit a certain size before undergoing whatever processing is needed. A method called the *Procrustes method* is often used to perform this operation.

The idea is to start with a simplifed model based on characteristics points. Thus, for a face, we can choose a model such as the one illustrated in Figure

5.13. In the case of a hand, you can either choose points on the outline of the hand, or points on the outline of each finger. For a license plate, the natural choice would be the four corners, etc.

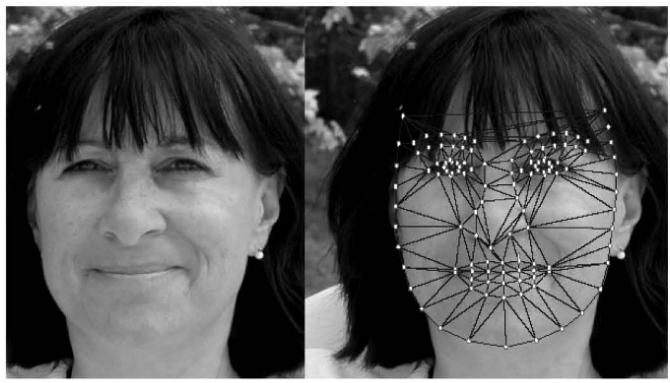

Figure 5.13 – *Triangle-based model*

Once we have a reference model $\mathcal{A}$ (the pattern on the left in Figure 5.14), we can start searching for a transformation that drags the characteristic points of the figure $\mathcal{B}$ to be analyzed (the pattern on the right in Figure 5.14) over onto the points of the reference model, according to a criterion used to evaluate the distance between two sets of points.

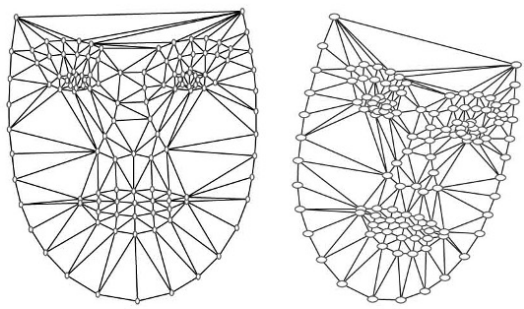

Figure 5.14 – *Characteristic points and Delaunay triangulation: the pattern on the left serves as a reference, the set of points on the right corresponds to the figure we wish to align*

The fact that the two sets of points $\mathcal{A}$ and $\mathcal{B}$ must correspond exactly adds a difficulty. When the points are provided by the automatic image analysis, $\mathcal{A}$ and $\mathcal{B}$ do not necessarily have the same number of points, meaning that some manual corrections may turn out to be unavoidable.

Let $\boldsymbol{A} \in \mathbb{R}^{r \times s}$ and $\boldsymbol{B} \in \mathbb{R}^{r \times s}$ be two $r \times s$ matrices. In our case, the matrix

size is $(N, 2)$ or $(2, N)$, where N is the number of charateristic points. We are trying to determine the $(r \times r)$ matrix $\boldsymbol{Q}$, solution to the problem, for which we define a constraint:

$$\min_{\boldsymbol{Q}} \|\boldsymbol{A} - \boldsymbol{QB}\|_F \quad \text{with} \quad \boldsymbol{Q}^T\boldsymbol{Q} = \sigma^2\boldsymbol{I}_r \tag{5.8}$$

The matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are centered. If their size is $(N, 2)$:

$$\boldsymbol{M} = \begin{bmatrix} X_1 & Y_1 \\ X_2 & Y_2 \\ \vdots & \vdots \\ X_N & Y_N \end{bmatrix}$$

they are centered by typing `M - ones(N,2)*mean(M)`.

For a matrix $\boldsymbol{M}$, the Frobenius norm is defined by $\|\mathbf{M}\|_F^2 = \mathrm{Tr}\{\mathbf{M}\mathbf{M}^T\}$. We have:

$$\begin{aligned} \|\boldsymbol{A} - \boldsymbol{QB}\|_F^2 &= \mathrm{Tr}\{\boldsymbol{AA}^T\} - 2\mathrm{Tr}\{\boldsymbol{QBA}^T\} + \sigma^2\mathrm{Tr}\{\boldsymbol{BB}^T\} \\ &= \mathrm{Tr}\{\boldsymbol{AA}^T\} - 2\sigma\mathrm{Tr}\{\boldsymbol{PBA}^T\} + \sigma^2\mathrm{Tr}\{\boldsymbol{BB}^T\} \end{aligned} \tag{5.9}$$

where $\boldsymbol{Q} = \sigma\boldsymbol{P}$ where $\boldsymbol{P}$ is a unitary matrix. Hence, for a given σ, the minimization problem amounts to the maximization problem of $\mathrm{Tr}\{\boldsymbol{PBA}^T\}$ under the constraint $\mathbf{P}^T\mathbf{P} = \boldsymbol{I}_s$.

The matrix $\boldsymbol{BA}^T$ is an $r \times r$ square matrix. Its singular value decomposition can be written as follows:

$$\boldsymbol{BA}^T = \mathbf{UDV}^T$$

where $\mathbf{U}$ and $\mathbf{V}$ are unitary. This leads us to:

$$\mathrm{Tr}\{\boldsymbol{PBA}^T\} = \mathrm{Tr}\{\boldsymbol{P}\mathbf{UDV}^T\} = \mathrm{Tr}\{\mathbf{V}^T\boldsymbol{P}\mathbf{UD}\}$$

If we assume $\mathbf{Z} = \mathbf{V}^T\boldsymbol{P}\mathbf{U}$, and because $\mathbf{D}$ is diagonal, we have:

$$\mathrm{Tr}\{\boldsymbol{PBA}^T\} = \mathrm{Tr}\{\boldsymbol{ZD}\} = \textstyle\sum_i z_{ii}d_{ii}$$

where the z_{ii} and d_{ii} are the diagonal terms of $\mathbf{Z}$ and $\boldsymbol{D}$ respectively. But, because $\mathbf{Z}$ is a unitary matrix, $|z_{ij}| < 1$ for i, j any pair. Indeed using $\boldsymbol{Z}^T\boldsymbol{Z} = \boldsymbol{I}$, for all j we have $\sum_i |z_{ij}|^2 = 1$. This means that, for any matrix $\boldsymbol{P}$:

$$\mathrm{Tr}\{\boldsymbol{PBA}^T\} \leq \textstyle\sum_i d_{ii}$$

The upper bound $\sum_i d_{ii}$, which is independent of $\boldsymbol{P}$, can be reached if we let $\mathbf{Z} = \mathbf{V}^T\boldsymbol{P}\mathbf{U} = \mathbf{I}$, that is to say:

$$\boldsymbol{P} = \boldsymbol{VU}^T \tag{5.10}$$

which is unitary. Hence (5.10) is the solution we were looking for. To sum up, after starting with $\mathbf{A}$ and $\mathbf{B} \in \mathbb{R}^{r \times s}$, we calculate, one after the other:

1. $\mathbf{C} = \boldsymbol{B}\boldsymbol{A}^T$;
2. the singular value decomposition: $\mathbf{C} = \mathbf{U}\mathbf{D}\mathbf{V}^T$;
3. $\boldsymbol{P} = \boldsymbol{V}\boldsymbol{U}^T$;
4. notice that minimizing 5.9 in regard to σ leads to:

$$\sigma = \frac{\mathrm{Tr}\{\boldsymbol{P}\boldsymbol{B}\boldsymbol{A}^T\}}{\mathrm{Tr}\{\boldsymbol{B}\boldsymbol{B}^T\}} \Rightarrow \boldsymbol{Q} = \frac{\mathrm{Tr}\{\boldsymbol{P}\boldsymbol{B}\boldsymbol{A}^T\}}{\mathrm{Tr}\{\boldsymbol{B}\boldsymbol{B}^T\}}\boldsymbol{P}$$

Notice that if $\boldsymbol{A} = \boldsymbol{B}$, $\boldsymbol{Q} = \boldsymbol{I}$.

The counterpart to the problem posed by expression (5.8) is determining the $r \times r$ unitary matrix $\mathbf{R}$, solution to the problem:

$$\begin{cases} \min_{\mathbf{R}} \|\boldsymbol{A} - \boldsymbol{B}\boldsymbol{R}\|_F \\ \mathbf{R}^T\mathbf{R} = \beta^2\boldsymbol{I}_s \end{cases} \tag{5.11}$$

Of course, its solution can be inferred from the previous one if you notice that 5.11 is equivalent to:

$$\begin{cases} \min_{\mathbf{R}} \|\boldsymbol{A}^T - \boldsymbol{R}^T\mathbf{B}^\mathbf{T}\|_\mathbf{F} \\ \mathbf{R}^T\mathbf{R} = \beta^2\boldsymbol{I}_s \end{cases} \tag{5.12}$$

the solution of which is $\mathbf{R} = \beta\mathbf{Y}\mathbf{W}^T$ where $\boldsymbol{B}^T\mathbf{A} = \mathbf{Y}\mathbf{D}'\mathbf{W}^T$.

Notice in the example above that if one of the dimensions is always equal to 2, for example:

$$\boldsymbol{A} = \begin{bmatrix} X_1 & X_2 & \dots & X_s \\ Y_1 & Y_2 & \dots & Y_s \end{bmatrix} \text{ and } \boldsymbol{B} = \begin{bmatrix} x_1 & x_2 & \dots & x_s \\ y_1 & y_2 & \dots & y_s \end{bmatrix}$$

the algorithm's development is true for any $r \times s$ pair. In particular, the pixels of two images we wish compare can be directly used.

5.4 Frequential content of an image

Just as it was done for 1D discrete-time signals, we are going to define the Fourier transform $X(\nu, \mu)$ of an image, referred to as the 2D-DTFT.

Definition 5.1 (2D-DTFT) *Let $x(k, \ell)$ be a two index sequence. The 2D-DTFT is the function of ν and of μ defined by:*

$$X(\mu, \nu) = \sum_{k=-\infty}^{+\infty} \sum_{\ell=-\infty}^{+\infty} x(k, \ell) e^{-2\pi j(k\mu + \ell\nu)} \tag{5.13}$$

Because of its definition, $X(\mu, \nu)$is periodic with period 1 for the two variables μ and ν.

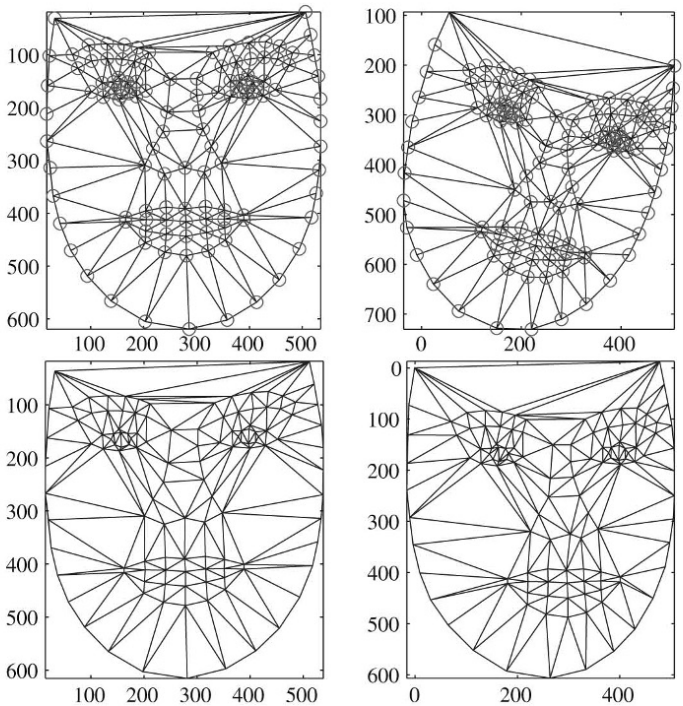

Figure 5.15 – *Applying the multiplications on the right and on the left in the example of Figure 5.14. In the bottom-right, the size of the transformation matrix is* (2×2). *In the bottom-left, the size of the matrix is* $(N \times N)$ *where* N *is the number of points in the mesh*

In practice, the images processed have a finite size $K \times L$ and we have:

$$X(\mu, \nu) = \sum_{k=0}^{K-1} \sum_{\ell=0}^{L-1} x(k, \ell) e^{-2\pi j(k\mu + \ell\nu)} \tag{5.14}$$

In this case, $X(\mu, \nu)$ poses no existence problems, since the values of $x(k, \ell)$ are bounded and the sequence is finite.

The inverse formula leading to $x(k, \ell)$ from $X(\mu, \nu)$ is:

$$x(k, \ell) = \int_{-1/2}^{1/2} \int_{-1/2}^{1/2} X(\mu, \nu) e^{2\pi j(k\mu + \ell\nu)} d\mu d\nu \tag{5.15}$$

Property 5.1 (2D convolution) *2D convolution is the name of the operation that associates the two sequences* $x(k, \ell)$ *and* $y(k, \ell)$ *with the sequence*

$z(k,\ell)$:

$$z(k,\ell) = (x \star y)(k,\ell) = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{\infty} x(i,j)y(k-i,\ell-j) \tag{5.16}$$

The 2D-DTFT of the 2D convolution of x with y is the product of the respective 2-DTFTs. In other words:

$$(x \star y) \leftrightarrow X(\mu,\nu) \times Y(\mu,\nu) \tag{5.17}$$

Just as for 1D, the problem of the numerical calculation of the 2D-DTFT leads to the introduction of the 2D-DFT, which corresponds to the 2D-DTFT's expression calculated in points *regularly spread-out over the* $(0,1)\times(0,1)$ *block.* Without being at all specific, and as for 1D, the number of points before and after the transformation can be considered the same, by completing with zeros if necessary. This leads to the following definition.

Definition 5.2 (2D Discrete Fourier Transform (2D-DFT))
The 2D discrete Fourier transform, or 2D-DFT, of the finite sequence $\{x(k,\ell)\}$, *with* $k \in \{0, \ldots, M-1\}$ *and* $l \in \{0, \ldots, N-1\}$, *is the sequence defined, for* $m \in \{0, \ldots, M-1\}$ *and* $n \in \{0, \ldots, N-1\}$, *by:*

$$X(m,n) \quad = \quad \sum_{k=0}^{M-1} \sum_{\ell=0}^{N-1} x(k,\ell) \exp\left\{-2\pi j \left(\frac{km}{M} + \frac{\ell n}{N}\right)\right\} \tag{5.18}$$

If we change the expression of $X(m,n)$ to:

$$X(m,n) = \sum_{\ell=0}^{N-1} \left(\exp\left\{-2\pi j \frac{\ell n}{N}\right\} \sum_{k=0}^{M-1} x(k,\ell) \exp\left\{-2\pi j \frac{km}{M}\right\} \right) \tag{5.19}$$

for each value of ℓ, the 1D-DFT of the sequence $x(k,\ell)$ for the variable k appears in the parenthesis. With MATLAB®, the N FFTs corresponding to expression 5.19 are calculated by applying the `fft` function to the $(M \times N)$ array `x`. The 2D-DFT is then achieved simply by performing another FFT on the resulting *transpose* array.

To sum up, the 2D-DFT is obtained by doing:

```
fft(fft(x).').'
```

The `fft2` function, available in the basic version of MATLAB®, performs the same operation. As was the case with 1D signals, typing `fft2(x,M,N)` completes, if necessary, the array `x` with zeros so as to have an $M \times N$ array. Again, as it was the case for 1D signals, it is often preferable to display the spatial frequencies with values between 0 and 1, or between 0 and 1/2. This is what we did in example 5.4.

EXAMPLE **5.4 (2D-DTFT of a square block)**
The following program calculates the 2D-DTFT of a square block and displays its modulus. The result is shown in Figures 5.16 and 5.17:

```
%===== tstfftblock.m
block=zeros(8,8); delta=4;
block(1:delta,1:delta)=ones(delta,delta);
set(gcf,'color',[1 1 1])
subplot(131); imagesc(block); colormap('gray');
axis('image'); set(gca,'xcolor',[0 0 0],'ycolor',[0 0 0])
%===== spectral content
M=32; N=32; blockFqs=fft2(block,M,N);
%===== normalized spatial frequencies
mu=(0:M-1)/M;nu=(0:N-1)/N;
subplot(132); contour(nu,mu,abs(blockFqs),20);
axis('square'); set(gca,'xlim',[0 .5],'ylim',[0 .5])
set(gca,'xcolor',[0 0 0],'ycolor',[0 0 0])
subplot(133); imagesc(nu,mu,abs(blockFqs))
axis('square'); set(gca,'xlim',[0 .5],'ylim',[0 .5])
set(gca,'xcolor',[0 0 0],'ycolor',[0 0 0])
```

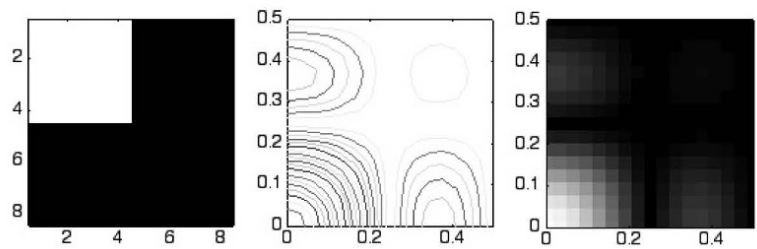

Figure 5.16 – *2D-FFT applied to the rectangular block by restricting the frequencies to* $([0, 1/2] \times [0, 1/2])$

The lobes are similar to the ones obtained for the discrete-time sine cardinal (Figure 5.17).

The properties of the 2D-DFT are similar to those of the 1D-DFT:

Property 5.2 (Inverse 2D-DFT) *The 2D-inverse-DFT of* $X(m, n)$ *has the expression:*

$$x(k, \ell) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m, n) \exp\left\{2\pi j \left(\frac{km}{M} + \frac{\ell n}{N}\right)\right\}$$

where $k \in \{0, \ldots, M-1\}$ *and* $\ell \in \{0, \ldots, N-1\}$,

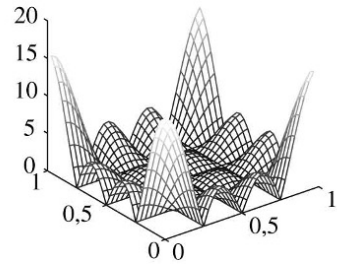

Figure 5.17 – *2D-FFT applied to the rectangular block with the frequencies belonging to* $([0,1] \times [0,1])$

This result is obtained by using the relation:

$$\begin{aligned} g(k,\ell) &= \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \exp\left\{2\pi j \left(\frac{km}{M} + \frac{\ell n}{N}\right)\right\} \\ &= \begin{cases} 1 & \text{if } k = 0 \bmod M \text{ and } \ell = 0 \bmod N \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Property 5.3 (Circular convolution (2D-DFT))
Let $x(k,\ell)$ *and* $y(k,\ell)$ *be two images with the same finite size* $M \times N$*. Let* $X(m,n)$ *and* $Y(m,n)$ *be their respective 2D-DFTs calculated over* $M \times N$ *points. Then the inverse 2D-DFT of* $Z(m,n) = X(m,n)Y(m,n)$ *has the following expression, for* $k \in \{0,\ldots,M-1\}$ *and* $\ell \in \{0,\ldots,N-1\}$*:*

$$z(k,\ell) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} x(u,v)y((k-u) \bmod M, (\ell - v) \bmod N)$$

where the indices of y *are calculated modulo* M *and modulo* N *respectively.*

Property 5.4 (Real image and hermitian symmetry (2D-DFT))
If the image $x(k,\ell)$ *is real then its 2D-DFT is such that:*

$$X(m,n) = X^*(-m \bmod M, -n \bmod N)$$

where the first indices[2] *are calculated modulo* M *and the second indices modulo* N*.*

Thus, $X(0,0) = X^*(0,0)$ which is therefore real. If $M = 8$ and $N = 16$, $X(4,3) = X^*(8-4, 16-3) = X^*(4,13)$.

[2]Bear in mind that the array indices start at 1 and not 0.

EXAMPLE **5.5 (2D-DFT of a checkerboard)**
The following program calculates the 2D-DFT of a checkerboard the horizontal frequency of which is `f0x=0.2` and the vertical frequency `f0y=0.3`, and displays its modulus (Figure 5.18). The resulting graph shows lobes at the spatial frequencies (0.2, 0.3) and $(1-0.2,\ 1-0.3)$, since the image is real. You can try other values of `f0x` and `f0y`.

```
%===== tstfftmo.m
%===== checkerboard
clear; mside=8; bloc=zeros(mside,mside);
f0x=0.2; f0y=0.3;
dom=f0x*(0:mside-1)'*ones(1,mside)+f0y*ones(mside,1)*(0:mside-1);
chkbd=cos(2*pi*dom)+1;
set(gcf,'color',[1 1 1])
subplot(121); imagesc(chkbd);
colormap('gray'); axis('image')
set(gca,'xcolor',[0 0 0],'ycolor',[0 0 0])
%===== spectral content
M=128; N=128; chkbdFqs=fft2(chkbd,M,N);
mu=(0:M-1)/M;nu=(0:N-1)/N;
subplot(122);
contour(nu,mu,abs(chkbdFqs),20); %imagesc(nu,mu,abs(chkbdFqs));
set(gca,'xcolor',[0 0 0],'ycolor',[0 0 0])
axis('square'); grid
```

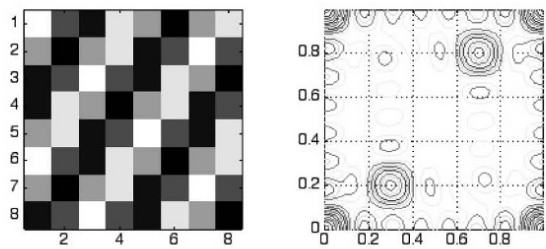

Figure 5.18 – *2D-FFT applied to a checkerboard*

If $x(k,\ell)$ is separable, that is if $x(k,\ell) = x_1(k)x_2(\ell)$, the 2D-DFT can be expressed as the product of two 1D-DFTs (meaning that $X(m,n)$ is separable). Thus, we can write:

$$\begin{aligned} X(m,n) &= \sum_{k=0}^{M-1} x_1(k) \exp\left\{-2\pi j \frac{km}{M}\right\} \times \sum_{\ell=0}^{N-1} x_2(\ell) \exp\left\{-2\pi j \frac{ln}{N}\right\} \\ &= X_1(m) \times X_2(n) \end{aligned}$$

The calculation then becomes quite simpler.

5.5 Linear filtering

The `filter2` function, used for 2D filtering, is available in the basic version of MATLAB®. This function uses the 2D-convolution function, the command line of which is, in MATLAB®, `c=conv2(a,b)`. The 2D-convolution is a *built-in function.*

Definition 5.3 *2D-linear filtering is the operation that associates the image $x(k,\ell)$ with the image $y(k,\ell)$ defined by:*

$$y(k,\ell) = (x \star h)(k,\ell) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} x(k-m,\ell-n)h(m,n) \tag{5.20}$$

The two index sequence $h(k,\ell)$, characteristic of the filter, is called the Point Spread Function, *or* PSF.

As was the case for 1D-filtering, the operation denoted by "$\star$" in expression 5.20 is linear and *space-invariant*, and the sequence $h(k,\ell)$ is the equivalent of the impulse response for the one dimension case. Once again, property 5.1 leads to a simple expression of the filtering operation in the frequential range. This gives us the following property.

Property 5.5 *Consider a 2D-linear filter with the PSF $h(k,\ell)$. $H(\mu,\nu)$ denotes the 2D-DTFT of its PSF. It is called the* optical transfer function, *or* OTF. *Because of property 5.1 we have:*

$$Y(\mu,\nu) = H(\mu,\nu)X(\mu,\nu)$$

where $X(\mu,\nu)$ and $Y(\mu,\nu)$ refer to the 2-DTFTs of $x(k,\ell)$ and $y(k,\ell)$ respectively.

Thus, the *identity filter* has the PSF $h(k,\ell) = \delta(k)\delta(\ell)$, where $\delta(k)$ is equal to 1 if $k = 0$ and 0 otherwise. Its OTF is equal to 1 for any frequency pair (μ,ν). This filter leaves the input image untouched. Because of property 5.5, we can also say that the identity filter passes all frequencies.

In MATLAB®, unlike the `filter(b,a,x)` function for 1D use, which allows the user to design an infinite impulse response filter using the input coefficients `a`, the `filter2(B,x)` function performs only the 2D equivalent of a finite impulse response filtering, the expression of which is:

$$y(k,\ell) = (x \star h)(k,\ell) = \sum_{m=M_1}^{M_2} \sum_{n=N_1}^{N_2} x(k-m,\ell-n)h(m,n) \tag{5.21}$$

The `filter2` function has an additional parameter that allows the user to set how the side effects should be taken into account: `'same'` to have an

output image with the same size as the input image (this is the default option), `'valid'` to keep only the part of the image unaffected by the side effect (the resulting image is smaller than the original), and `'full'` to keep all of the points, including the ones resulting from the filter's impulse response (this leads to an image larger than the original).

The concept of *stability* is essential, as it was with 1D signals. It states that to any bounded input corresponds a bounded output. Because we will only be considering filters characterized by expression 5.21 and similar to the 1D FIR filters, the stability condition will always be met from now on.

On the other hand, the concept of *causality*, although fundamental when it comes to signals, has very little significance in the case of images. This is because there is no reason for the quantity calculated for the coordinates (k, ℓ) to be dependent only on the points placed "before" (k, ℓ), that is to say $(k - m, \ell - n)$, where m and n are positive. In 2D processing, all of the points around (k, ℓ) can contribute to the calculated value.

EXAMPLE **5.6 (Circular filter)**
Consider what is called the *circular filter*, $h(k, l)$, defined in the program:

```
%===== smooth1.m
h =[0 0 1 1 1 0 0;
    0 1 1 1 1 1 0;
    1 1 1 1 1 1 1;
    1 1 1 1 1 1 1;
    1 1 1 1 1 1 1;
    0 1 1 1 1 1 0;
    0 0 1 1 1 0 0];
h = h / sum(sum(h));
load wenmanu; subplot(121); imagesc(pixc);
colormap(cmap); axis('image');
set(gca,'units','pixels','DataAspectRatio',[1 1 1])
pixr=filter2(h,pixc); subplot(122); imagesc(pixr);
set(gca,'units','pixels','DataAspectRatio',[1 1 1])
axis('image')
```

This program smooths the image 5.19.

As was the case with 1D smoothing filters, this filter tends to "erase" high frequencies, particularly the ones contained in the contours, and therefore produces a blurred image (Figure 5.19) compared to the original image.

Definition 5.4 *A filter is said to be separable when its PSF is such that:*

$$h(k, \ell) = h_x(k) h_y(\ell) \tag{5.22}$$

In the case of a finite PSF, if $\boldsymbol{h}$ *is the matrix with* $h(k, \ell)$ *as its elements, and if* $\boldsymbol{h}_x$ *and* $\boldsymbol{h}_y$ *are the vectors with the respective components* $h_x(k)$ *and*

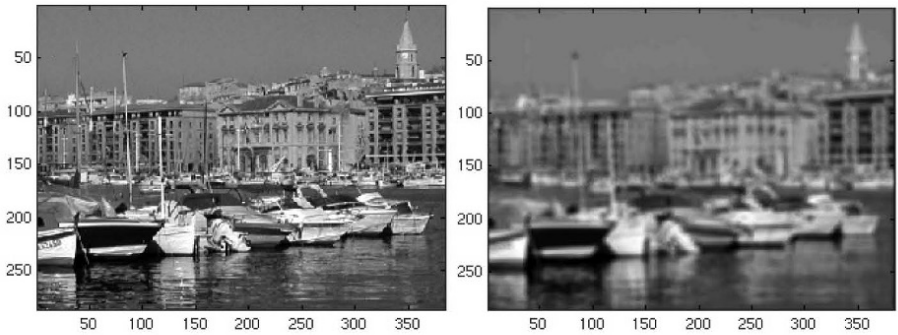

Figure 5.19 – *Smoothing of an image test using the circular filter*

$h_y(\ell)$, relation 5.22 is equivalent to:

$$\boldsymbol{h} = \boldsymbol{h}_x \boldsymbol{h}_y^T \tag{5.23}$$

We are going to show that a separable 2D filtering can be performed by combining two consecutive 1D filters. This is how it works:

$$\begin{aligned}(x \star h)(m,n) &= \sum_{k=K_1}^{K_2} \sum_{\ell=L_1}^{L_2} x(m-k, n-\ell) h(k,\ell) \\ &= \sum_{k=K_1}^{K_2} h_x(k) \left(\sum_{\ell=L_1}^{L_2} x(m-k, n-\ell) h_y(\ell) \right) \\ &= \sum_{\ell=L_1}^{L_2} h_y(\ell) \left(\sum_{k=K_1}^{K_2} x(m-k, n-\ell) h_x(k) \right)\end{aligned}$$

Bear in mind that if the `filter` function is used for a separable 2D filtering, you must take into account the fact that `filter` implements a causal design (exercise 5.4).

Exercise 5.4 (The rectangular filter) (see p. 442)

Consider the rectangular filter defined by:

$$\mathbf{h} = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Write a program that:

1. performs the filtering of the test image;
2. performs the same filtering using two separate 1D filterings.

Exercise 5.5 (The conical filter) (see p. 443)

The conical filter is defined by:

$$\mathbf{h} = \frac{1}{25}\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 2 & 2 & 0 \\ 1 & 2 & 5 & 2 & 1 \\ 0 & 2 & 2 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Apply the conical filter to the test image.

Definition 5.5 (The Gaussian smoothing filter) *The generating element of the Gaussian smoothing filter's PSF is:*

$$h(k,\ell) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{k^2+\ell^2}{2\sigma^2}\right)$$

This filter is separable, since we can write $h(k,\ell) = h_x(k)h_y(\ell)$.

The smaller the σ parameter is, the more the filter behaves like an identity filter, that is to say that it passes all the frequencies of the plane. The gain filter can of course be modified by multiplying it by a constant.

Exercise 5.6 (The Gaussian smoothing filter) (see p. 443)

1. Write a MATLAB® function that calculates the PSF of Gaussian smoothing filter using σ. Consider using the `meshgrid` function;
2. apply the Gaussian filter to the test image.

2D-DFT frequency filtering

Starting with the circular convolution property 5.3, it is possible to consider performing a filtering by simply multiplying the 2D-DFT of an image by the 2D-DFT of the filter's PSF. Of course, just like in 1D, this process must take into account the circular convolution property.

Consider the PSF $h(k,\ell)$ of a $K \times L$ filter (number of non-zero coefficients), and an $M \times N$ image $x(k,\ell)$. We will assume $M > K$ and $N > L$. $H(m,n)$ and $X(m,n)$ refer to the 2D-DFTs of the PSF and of the image respectively. Both are calculated for $M \times N$ points. According to property 5.3, the inverse

2D-DFT of the product $H(m,n)X(m,n)$ can be written, for $k \in \{0,\ldots,M-1\}$ and $\ell \in \{0,\ldots,N-1\}$:

$$y(k,\ell) = \sum_{u=0}^{K-1}\sum_{v=0}^{L-1} h(u,v)x(k-u \bmod M, \ell - v \bmod N)$$

For $k \geq K-1$, $(k-u \bmod M) = k-u$: there is no index "aliasing" when we sum u from 0 to $(K-1)$. This also true for $\ell \geq L-1$, $(\ell - v \bmod N) = \ell - v$. In this case, the calculated points do correspond to those of the convolution associated with the filtering. However, for $k < K-1$ and/or $\ell < N-1$, there is an index "aliasing" which leads to an incorrect result. One way of avoiding this phenomenon is by completing the image with K zeros along the horizontal axis, and L zeros along the vertical axis.

Derivative operations

To display the variations of a 2D function, the concept of derivative can be used, as it was in 1D. The difference is that in 2D, the derivative comprises two components corresponding to the two directions of the plane. Thus, to perform a 2D-derivative, you can use a first derivative filter along the horizontal direction, and a second one along the vertical direction. Base derivative filters are used for PSF matrices (5.24):

$$\boldsymbol{D}_x = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } \boldsymbol{D}_y = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \tag{5.24}$$

The filters that derive in one direction and smooth in the other direction are generally preferred. Prewitt and Sobel filters are good examples of this.

Definition 5.6 (Prewitt derivative filter)
The PSF of a Prewitt derivative filter along the vertical direction is given by:

$$\mathbf{h}_v = \frac{1}{3}\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

and the PSF of a Prewitt derivative filter along the vertical direction by:

$$\mathbf{h}_h = \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

These two filters are therefore separable. Decomposing $\boldsymbol{h}_v$ clearly shows:

- a smoothing function along one direction, smoothing corresponding to the vector $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}/\sqrt{3}$;

– a derivative function in the other direction, corresponding to the vector $\begin{bmatrix}1 & 0 & -1\end{bmatrix}/\sqrt{3}$. Remember that the 1D causal filter defined by this vector associates the input $u(n)$ with the output $v(n) = (u(n) - u(n-2))/\sqrt{3}$, which can be seen as the derivative.

Definition 5.7 (Sobel derivative filter)
The PSF of a Sobel fitler along the vertical axis is given by:

$$\mathbf{h}_v = \frac{1}{4}\begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1\\ 2\\ 1\end{bmatrix}\begin{bmatrix}1 & 0 & -1\end{bmatrix} \tag{5.25}$$

The PSF of a Sobel fitler along the vertical axis is given by:

$$\mathbf{h}_h = \frac{1}{4}\begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1\\ 0\\ -1\end{bmatrix}\begin{bmatrix}1 & 2 & 1\end{bmatrix} \tag{5.26}$$

Sobel filters are separable. Notice that they perform a derivative along one axis, and a smoothing operation along the perpendicular axis.

Applying formula 5.23 leads to a 2D filter that derives in both directions without any smoothing:

$$\boldsymbol{h} = \alpha\begin{bmatrix}1\\ 0\\ -1\end{bmatrix}\begin{bmatrix}1 & 0 & -1\end{bmatrix} = \alpha\begin{bmatrix}1 & 0 & -1\\ 0 & 0 & 0\\ -1 & 0 & 1\end{bmatrix}$$

where α is a normalization coefficient.

Starting off in 1D, we can also design a 2D filter that approximates the second derivative, by convoluting the impulse response filter $\begin{bmatrix}-1 & 1\end{bmatrix}$, which is an approximation of the first derivative, with itself. If you type `conv([-1 1],[-1 1])` in MATLAB®, the result is `[1 -2 1]`. By combining the two directions, we get a 2D filter that performs a second derivative in both directions, defined by:

$$\mathbf{h} = \alpha\begin{bmatrix}1\\ -2\\ 1\end{bmatrix}\begin{bmatrix}1 & -2 & 1\end{bmatrix} = \alpha\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix} \tag{5.27}$$

where α is a normalization coefficient.

Generally speaking, it is of course possible to design other filters using one dimension design methods, and then inferring a 2D separable filter with formula 5.23.

Exercise 5.7 (The Sobel derivative filter) (see p. 444)

1. Apply the Sobel filters 5.25 and 5.26 to the test image;
2. apply the filter 5.27 to the same image.
3. by using a similar method to the window method, calculate a derivative filter;
4. same question for a second derivative filter.

Definition 5.8 (Gaussian derivative filter)
Consider the function defined as the difference between two Gaussians, also called a Difference of gaussians mask, or DoG mask:

$$g(k,\ell) = \frac{1}{2\pi\sigma_1^2}\exp\left(-\frac{k^2+\ell^2}{2\sigma_1^2}\right) - \frac{1}{2\pi\sigma_2^2}\exp\left(-\frac{k^2+\ell^2}{2\sigma_2^2}\right)$$

with $\sigma_2 = r\sigma_1$ *and* r *between* 1.4 *et* 1.8. *The Gaussian derivative filter is the filter with the following PSF:*

$$h(k,\ell) = g(k,\ell) - \sum_k \sum_\ell g(k,\ell)$$

implying that $\sum_k \sum_\ell h(k,\ell) = 0$.

The imposed condition, $\sum_k \sum_\ell h(k,\ell) = 0$, is related to the fact that a derivative filter has a gain equal to 0 at the frequency 0. This result is similar to the one obtained in 1D in exercise 4.10.

The graph of a Gaussian derivative filter PSF is shaped like the one in Figure 5.20.

EXAMPLE **5.7 (Gaussian derivative filter)**

1. Write a MATLAB® function that calculates the PSF of a Gaussian derivative filter. Write it so that $\sum_k \sum_l h(k,l) = 0$;
2. apply this filter to the test image for three values of $\sigma_1 = \{1,2,3\}$ and for $r = 1.4$. Save the results in three different files (use the functions `sprintf` and `eval` to change in a program the name of the saved file).

HINT:

1. Type:

```
function hd=dergauss(sigma)
%!================================!
%! Gaussian derivative filter     !
%! SYNOPSIS: hd = DERGAUSS(sigma) !
```

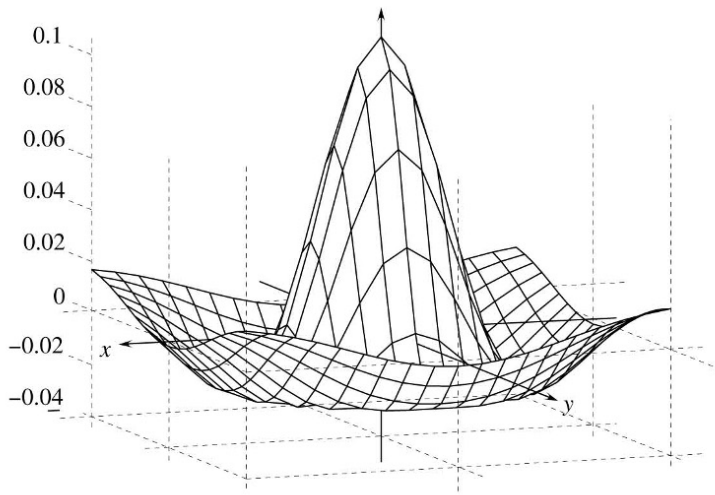

Figure 5.20 – *Graph shape of a Gaussian derivative filter's PSF*

```
%!   sigma = Standard deviation    !
%!   hd    = filter with (N*N)-PSF !
%!================================!
rho=[-sigma*3:sigma*3]; N=length(rho);
rp=1.4; s2=2*sigma^2; s22=s2*rp*rp;
idx= ([1:N]-(N+1)/2)' * ones(1,N); idy=idx';
idxa=[1:N]' * ones(1,N); idya=idxa';
%=====
inds(1,:)=reshape(idx,1,N*N); inda(1,:)=reshape(idxa,1,N*N);
inds(2,:)=reshape(idy,1,N*N); inda(2,:)=reshape(idya,1,N*N);
rho2=sum(inds .* inds); rho=sqrt(rho2);
for k=1:N*N
    g1=(1/sigma)*exp(-rho2(k) / s2);
    g2=(1/sigma/rp)*exp(-rho2(k) / s22);
    hd(inda(2,k),inda(1,k))=g1-g2;
end
hd=hd-sum(sum(hd))/N/N;
return
```

2. Applying the filter to the test image (Figure 5.21):

```
%===== tstdergauss.m
%===== loading the image
clear; load lena; subplot(221); imagesc(pixc+1);
colormap(cmap); axis('image')
set(gca,'Xcolor',[0 0 0],'Ycolor',[0 0 0])
%===== gaussian derivative filter
```

```
for k=1:3
    hd=dergauss(k);
    pixr=round(filter2(hd,pixc));
    subplot(2,2,k+1); imagesc(pixr); axis('image')
    set(gca,'Xcolor',[0 0 0],'Ycolor',[0 0 0])
end
set(gcf,'Color',[1 1 1])
```

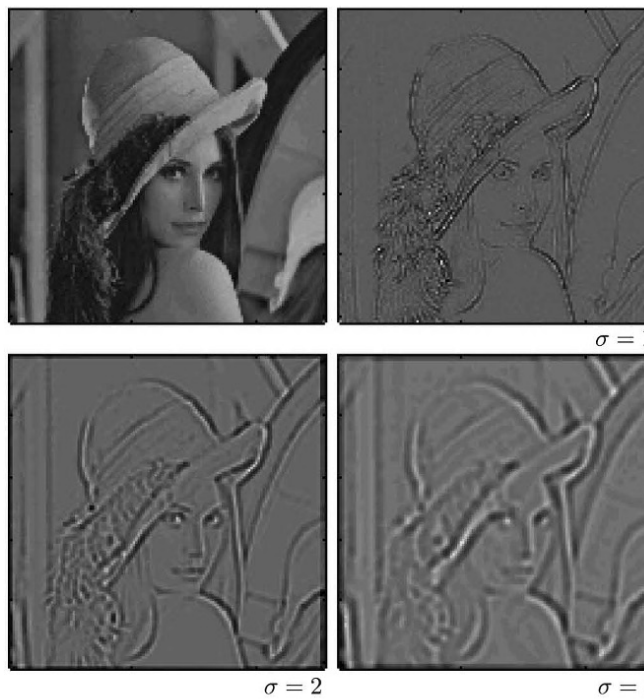

Figure 5.21 – *Gaussian derivative for* $\sigma_1 = \{1, 2, 3\}$ *and* $\sigma_2 = 1.4\sigma_1$

The programs we have just described are used in the contour detection program.

Definition 5.9 (Gaussian derivative-smoothing filter)
Consider the rotation of angle θ *that changes the point with coordinates* (u, v) *according to the expression:*

$$\begin{bmatrix} u(k,\ell) \\ v(k,\ell) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} k \\ \ell \end{bmatrix} \tag{5.28}$$

Consider the Gauss function:

$$h_1(k,\ell) = \frac{1}{\sigma_1\sqrt{2\pi}} \exp\left(-\frac{u(k,\ell)^2}{2\sigma_1^2}\right) \tag{5.29}$$

and the derivative function:

$$h_2(k,\ell) = -\frac{v(k,\ell)}{\sigma_2^3\sqrt{2\pi}} \exp\left(-\frac{v(k,\ell)^2}{2\sigma_2^2}\right) \tag{5.30}$$

The Gaussian derivative-smoothing filter is the filter that performs a Gaussian smoothing filtering (function h_1) along the direction $\theta \in (0, 2\pi)$ and a Gaussian derivative filtering (function h_2) along the perpendicular direction.

The expression of its PSF's generating element is:

$$h(k,\ell) = h_1(k,\ell)h_2(k,\ell) - \sum_k \sum_\ell h_1(k,\ell)h_2(k,\ell)$$

which verifies $\sum_k \sum_\ell h(k,\ell) = 0$.

Exercise 5.8 (Gaussian derivative-smoothing filter) (see p. 447)

1. Write a MATLAB® function that calculates the PSF of a Gaussian derivative-smoothing filter;
2. apply this filter to the test image.

5.6 Other operations on images

5.6.1 Undersampling

As it was the case with one dimension signals, the undersampling has to meet some conditions to avoid the aliasing phenomenon. Remember that aliasing occurs when the sampling rate is too slow compared to the frequencies found in the image, and causes frequential artifacts to appear. In an image, high frequencies correspond to important variations in color and/or brightness concentrated on small surfaces. Take the example of the images represented in Figure 5.22. They were obtained with the following program:

```
%===== aliasingtrains.m
close all; clear all
load trainsV4, sxx1=size(xx1);
xx1se=xx1(1:5:sxx1(1),1:5:sxx1(2)); % undersampling
lwpass=ones(5,5)/25;
yy1=filter2(lwpass,xx1);            % filtering before
yy1se=yy1(1:5:sxx1(1),1:5:sxx1(2)); % under-sampling
subplot(221); imagesc(xx1); colormap('gray'), axis('image')
subplot(222); imagesc(xx1se); colormap('gray'), axis('image')
subplot(223); imagesc(yy1); colormap('gray'), axis('image')
subplot(224); imagesc(yy1se); colormap('gray'), axis('image')
set(gcf,'color','w')
```

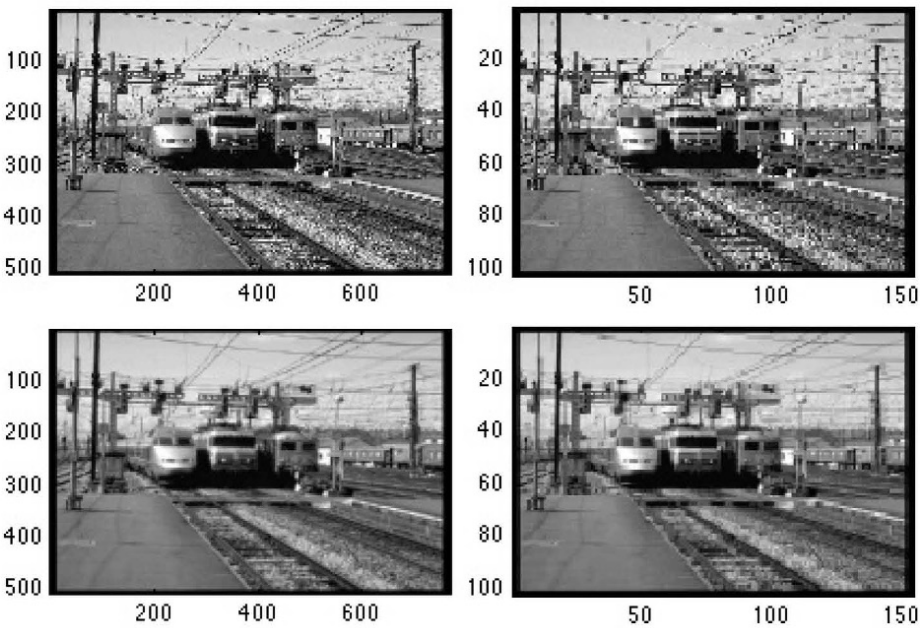

Figure 5.22 – *Effects of spectrum aliasing. In the top-left corner, the original image* (512×768). *In the top-right corner, the same image undersampled by a factor of* 5. *In the bottom-left corner, the image filtered by a smoothing filter over a* 5×5 *square. In the bottom-right corner, the image filtered and undersampled by a factor of* 5

In the top-left corner, you can see the original image, a 512×768 array. This image contains "high frequencies", particularly around the electric cables and the tracks, where the shapes are in some places less than a few pixels wide. In the image represented in the top-right corner, obtained by taking 1 out of 5 pixels horizontally and vertically, you can clearly see major and erratic variations in some areas of the image, due to aliasing.

Just like in 1D, a low-pass filtering must be performed before the undersampling. This can be done with a simple filter, calculating the mean over 5×5 cells. This operation is performed by `filter2(lwpass,xx1)`, which uses the `filter2` function. The resulting image is shown in the bottom-left corner. The filter causes a slight "blur". The image in the bottom-right corner shows the previous image after the undersampling operation. Most of the artifacts are gone. The tracks in particular show less unwanted fluctuations.

5.6.2 Oversampling

As for the oversampling of 1D signals, the interpolation operation can be performed by the insertion of "zeros" (the zero's significance is not the same here) followed by a low-pass filter. In the following example, we isolated the part of the original image containing the clock, on the platform to the left. This portion of the image is shown on the left-hand side of Figure 5.23. In order to improve the image rendering, we oversampled by a factor of 4, horizontally and vertically. The low-pass filter is a separable filter with a PSF of the type $\sin\alpha/\alpha$, to which a Hamming window is applied in order to reduce the ripples in the resulting image. The following program was used to obtain the image on the right of Figure 5.23:

```
%===== oversamp2ds.m
%===== over-sampling ratio
clear; Mx=4; My=4; cmap='gray';
load trainsV4; ima=xx1;       % the file "trains" --> xx1
%===== zooming in on the clock
pixc=ima(180:210,80:120);
[Lig,Col]=size(pixc);
%===== low-pass filter PSF (Lfft>N)
N=30; [X,Y]=meshgrid(-N:1:N, -N:1:N); X=X+eps; Y=Y+eps;
FEP=Mx*My*(sin(pi * X/Mx) ./ X) .* (sin(pi * Y/My)./ Y);
%===== Hamming window
W = (0.54 - 0.46*cos(2*pi*(X+N)/(2*N))) ...
    .* (0.54 - 0.46*cos(2*pi*(Y+N)/(2*N)));
FEP=FEP .* W;
%===== expansion and filtering
pixcz=zeros(Mx*Lig,My*Col);
pixcz(1:Mx:Mx*Lig,1:My:My*Col)=pixc;
pixcSE=filter2(FEP,pixcz);
%===== displaying the result
subplot(121); imagesc(pixc); axis('image'); colormap(cmap);
subplot(122); imagesc(pixcSE); axis('image'); colormap(cmap);
set(gcf,'Color',[1 1 1])
```

These techniques, taken directly from signal processing, are not the only ones used. The bibliography shows some sources of information for bilinear interpolations, cubic interpolations, etc.

For example, the bilinear interpolation consists of constructing "intermediate" pixels P from four pixels P_{00}, P_{01}, P_{10} and P_{11}:

$$P = P_{00}(1-t_x)(1-t_y) + P_{01}(1-t_x)t_y + P_{10}t_x(1-t_y) + P_{11}t_xt_y$$

by making the values of the parameters t_x and t_y vary from 0 to 1.

EXAMPLE **5.8 (Bilinear interpolation)** The `bilintrimg` function performs the bilinear interpolation of an image with $(n \times m)$ pixels:

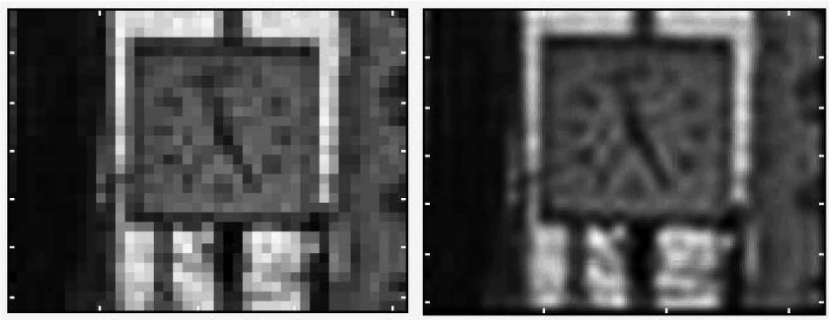

Figure 5.23 – *Image on the left: zoom-in on the clock in the original image from Figure 5.22. Image on the right: oversampling by a factor of* 4 *in both directions*

```
function pixcR=bilintrimg(pixc,Rintx,Rinty)
%!=============================================!
%! Bilinear interpolation of an image          !
%! SYNOPSIS: pixcR=BILINTRIMG(pixc,Rintx,Rinty) !
%!    pixc = image (nl*nc) pixels              !
%!   Rintx = interpolation rate (x)            !
%!   Rinty = interpolation rate (y)            !
%!=============================================!
Spix=size(pixc); Nr=Spix(1); Nc=Spix(2);
txt=[0:Rintx-1]/Rintx; ty=[0:Rinty-1]'/Rinty;
nrow=(Nr-1)*Rinty+1; ncol=(Nc-1)*Rintx+1;
pixcR=zeros(nrow+Rinty,ncol+Rintx);
M00=(1-ty)*(1-txt); M01=(1-ty)*txt;
M10=ty*(1-txt); M11=ty*txt;
pixc=[pixc zeros(Nr,1);zeros(1,Nc+1)];
for kl=1:Nr
    for kc=1:Nc
        tl=(kl-1)*Rinty+[1:Rinty]; tc=(kc-1)*Rintx+[1:Rintx];
        PC=pixc(kl,kc)*M00+pixc(kl,kc+1)*M01+...
           pixc(kl+1,kc)*M10+pixc(kl+1,kc+1)*M11;
       pixcR(tl,tc)=PC;
   end
end
pixcR=pixcR(1:nrow,1:ncol);
return
```

The program creates interpolated points based on the four points $A(0,0,1)$, $B(1,0,0)$, $C(0,1,0.5)$ and $D(1,1,0.5)$ with interpolation ratios equal to 4 (Figure 5.24).

```
%===== tstbilinr.m
pixc=[1 0;.5 .5]; % A, B, C, D
pixcR=bilintrimg(pixc,4,4);
```

```
[X,Y]=meshgrid((0:4)/4,(0:4)/4);
Xr=reshape(X,25,1); Yr=reshape(Y,25,1);
pixcRr=reshape(pixcR,25,1);
mesh(X,Y,pixcR), view(20,48)
```

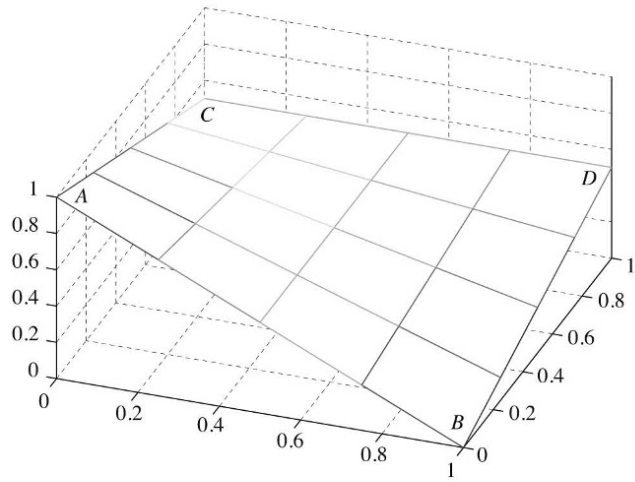

Figure 5.24 – *Applying the bilinear interpolation*

5.6.3 Contour detection

Contour detection is a common application of image processing. A simple method is to start by extracting the portions of the image with a significant gradient. This can be done with a derivative filter. We then need to define a boolean information, for each pixel, stating whether or not the pixel belongs to a contour. This can be done simply by comparing the obtained results to a threshold. The following program uses an image obtained by differentiation in example 5.7 (gaussian derivative filter):

```
%===== thresholdg.m
%===== file loading
clear; load lenabool2; % after differentiation
subplot(121); mydisp(pixr,cmap);
%===== threshold with manual choice for alpha
alpha=0;
decal=round((max(max(pixr))+min(min(pixr)))/2);
subplot(122); imagesc(-sign(pixr+decal+alpha));
axis('image')
```

The result of the thresholding is represented in Figure 5.25.

Figure 5.25 – *Results of the thresholding after derivative filtering*

Combining a Gaussian low-pass filter with first order horizontal and vertical derivatives, such as in the previous example, is a very common method for contour detection. J. Canny [5] showed that this method is very similar to applying a filter that optimizes a criterion related to precision and stability.

It can be wiser to search for local maxima – peaks – in the result of the derivative.

Exercise 5.9 (Contours using Sobel filtering) (see p. 448)

1. Apply the Sobel filters 5.25 and 5.26 to the test image;
2. using the resulting pixels $p_v(k,l)$ and $p_h(k,l)$, construct the image of the pixels $\sqrt{p_v^2(k,l) + p_h^2(k,l)}$. By defining an appropriate threshold value, extract the contours of the image.

The function `fminsearch` implementing the Nelder-Mead algorithm can be used for the seaching of extrema.

Hough method

For shape recognition, it may be useful to detect the presence of basic shapes, such as circles, ellipses, straight lines, etc. The Hough method is one of the most common.

Consider for example the case of line detection in an image previously processed so as to outline the contours. For straight contour detection, we then use sets of lines where each straight line is defined by the pair of paramaters (ρ, θ) (Figure 5.26). ρ refers to the distance to the origin and θ the angle to the direction perpendicular to the line.

In each point of the contour, a set of concurrent lines is built, with the parameters ρ and θ (see Figure 5.26). Figure 5.27 shows that two sets of lines

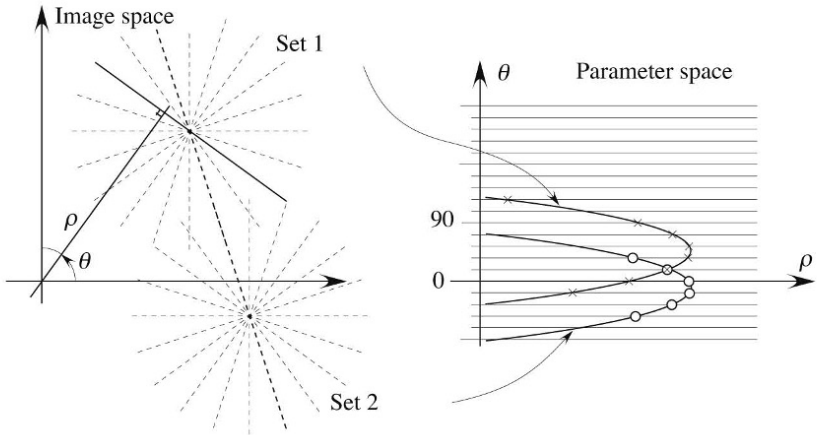

Figure 5.26 – *Setting the parameters for the set of lines*

on a portion of a line share a common point, or to be less specific, in the same neighborhood, on the parameters.

The accumulation, resulting from all the filters associated with this portion of a line, leads to a maximum in the neighborhood of this point. We then proceed to partitioning the parameter space, so as to obtain a quantization grid, then we count the points inside each box of the grid. The resulting values are then used for different kinds of processing.

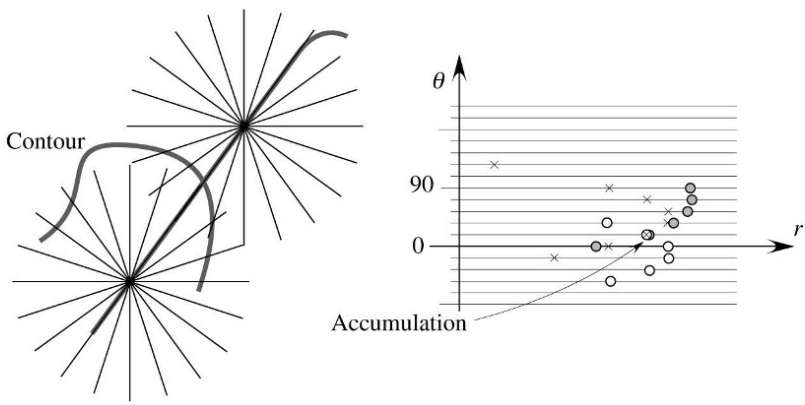

Figure 5.27 – *Using sets of lines*

EXAMPLE **5.9 (Implementing the Hough method)**
Consider an image (Figure 5.29, image on the left), assumed to have been

obtained by contour extraction. The following program performs a search for straight lines

```
%===== hough.m
load hough; [nlig,ncol]=size(pixc);
figure(1); subplot(121); colormap(cmap);
imagesc(pixc); axis('image')
%===== contour extraction
indx=find(pixc==0); Nidx=length(indx);
%=====
Nt=60; thetad=[0:180/Nt:180]; theta=thetad*pi/180;
thet=thetad(1:Nt); tbl=zeros(Nt,Nt);
figure(2)
%===== for each point and each value
%      of theta, rho is computed
for k=1:Nidx
    nc=floor((indx(k)-1)/nlig)+1; nl=indx(k)-(nc-1)*nlig;
    for m=1:Nt, rho(m)=nc*cos(theta(m))+nl*sin(theta(m)); end
    tbl(:,k)=rho';
    plot(rho,thet); hold on
end
rhomax=sqrt(nlig*nlig+ncol*ncol);
set(gca,'Xlim',[0 rhomax]); grid; hold off
%===== result (visual examination)
figure(1); subplot(122); imagesc(pixc); colormap(cmap);
axis('image'); hold on
plot([0 54.5*cos(70*pi/180)], [0 54.5*sin(70*pi/180)])
hold off
```

The observation of the resulting set (Figure 5.28) provides us with a direction.

The perpendicular direction is indicated in Figure 5.29 (the image on the right). An automatic search requires searching for zones with a high point density, hence the idea to use a 2D histogram to extract the (ρ, θ) positions of the maxima.

One example of an application for this type of processing is the search for the writing line directions in a handwritten text. The method described here is known as the *Hough method* [18], or *Hough transform method*, and allows the extraction of directions, one of which still has to be chosen. The presence of a high point density along a line can help find them.

The Hough method can also be used to search for other shapes. The idea is the same. For example, for circular shapes, the parameter $\{r, \theta, R\}$ can be used, where the pair (r, θ) refers to the polar coordinates of the circle's center and R is its radius.

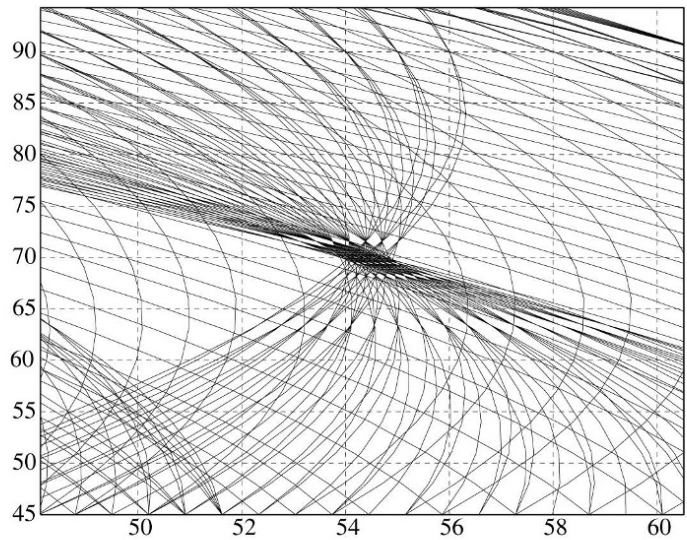

Figure 5.28 – *Zoom-in on the set of lines: we find the values* $\rho = 54.5$ *and* $\theta = 70^\circ$

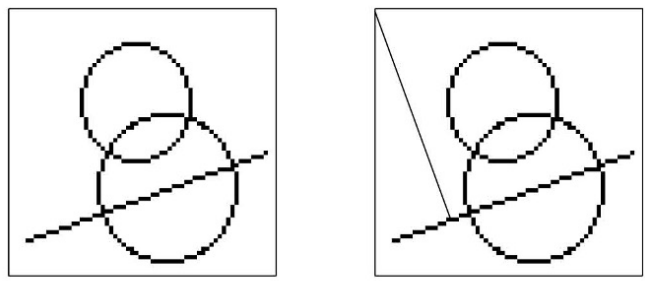

Figure 5.29 – *Search for a straight line*

5.6.4 Median filtering

Compared to other non-linear filters, median filtering is both simple and efficient. Just like a linear low-pass filtering, it smooths the image and can therefore eliminate certain of the image's imperfections. However, unlike a linear low-pass filter, which inevitably adds a blur around the contours, it better preserves the sharp variations of the image.

Definition 5.10 (Median filter)
Let $\{a(k,\ell)\}$ *be an image. The median filter associates the mean value* $m(k,\ell)$ *with the point with coordinates* (k,ℓ)*, in the* $(M \times N)$ *rectangular window,*

centered on (k, ℓ). *If we assume* N *and* M *to be odd, and if* $u(n)$ *denotes the sorted sequence* $(u(n) \geq u(n-1))$ *obtained from the array* $[a(i,j)]$ $(M \times N)$ *where* $i \in \{(k-(M-1)/2, \ldots, k+(M-1)/2\}$ *and* $j \in \{(\ell-(N-1)/2, \ldots, \ell-(N+1)/2)\}$, *we have:*

$$m(k,\ell) = u((MN+1)/2)$$

Exercise 5.10 (Median filtering) (see p. 448)
Apply this program to the test image:

```
%===== snowing.m
load lena
dims = size(pixc);
msnow = (randn(dims)>-2);
pixcmsnow = pixc .* msnow;
imagesc(pixcmsnow); axis('image')
```

This causes white points to randomly riddle the image, a bit like snow. Compare the effect of a Gaussian smoothing filter with the effect of a median filtering on the "snowy" image.

Exercise 5.11 (Processing the result of a rotation) (see p. 449)
Use the saved image in example 5.3:

1. perform a 3×3 median filtering on the resulting image. Try several rotation angles;

2. perform a processing of the "missing" points by calculating a mean on the surrounding pixels.

There are many possible methods for processing an image after it has undergone geometric transformations: interpolations, morphological filtering (paragraph 5.6.7), median filterings, etc. or other methods adapted to the case in question. There are no absolute rules in the field.

5.6.5 Image binarization

Image binarization consists of intensifying the contrast until complete saturation is reached. Black and white are the only two levels kept after this operation. It is used in particular for Optical Character Recognition, or OCR. The technique described below is based on a method suggested by N. Otsu in 1979 [26]. It requires the calculation of an histogram first, followed by a separation in two categories, $\underline{\mathcal{C}}$ and $\overline{\mathcal{C}}$, associated with the two colors. It is quite simple to adapt this method to a greater number of categories:

1. The histogram calculation consists of initializing with zeros an array $\boldsymbol{H} = [h(k)]$ with $P = 256$ entries. These entries correspond to P levels of gray. The entire image is covered, and for each pixel (n, m) with a level of gray k, the entry for $h(k)$ in the array $\boldsymbol{H}$ is incremented. The histogram is normalized by dividing $\boldsymbol{H}$ by the number N of pixels in the image. The $h(k)$ can then be interpreted as estimated values for the probabilities of finding the 256 levels of gray in the image.

2. Separating the image pixels in two categories can be done by directly comparing levels of gray with a threshold value defined by observing the previous histogram.

This very simple method can give disappointing results. There are two main drawbacks. First, isolated pixels can belong to an area and not be part of that area's category. In particular, this can lead to highly contrasted textures. The second drawback concerns images showing the shadow of certain objects. It is not always a good thing to have them belong to the same category as the object they came from, whatever the lighting may be.

Consider for example the original image in Figure 5.31. The following program first draws the histogram for the 256 levels (Figure 5.30), then uses it to calculate threshold values. Based on these values, the program displays two binarization examples. The results, for two threshold values, are shown in Figure 5.31:

```
%===== binar1.m
load elido72
[nlig ncol]=size(pixc); nbpix=prod(size(pixc));
%===== global histogram
histog=zeros(1,256); pixc3=zeros(nlig*ncol,1); pixc3(:)=pixc;
histog=hist(pixc3,256)/nlig/ncol;
figure(1); plot([0:255],histog); grid
%===== thresholds based on a visual examination
%      of the histogram
figure(2); subplot(131);
imagesc(pixc); axis('image'); colormap(cmap);
%===== threshold 1
pixc2=zeros(nlig,ncol);
seuil=152; idxy=find(pixc>seuil);
pixc2(idxy)=255*ones(size(idxy)); subplot(132); imagesc(pixc2);
axis('image'); colormap(cmap)
%===== threshold 2
pixc2=zeros(nlig,ncol);
seuil=90; idxy=find(pixc>seuil);
pixc2(idxy)=255*ones(size(idxy)); subplot(133); imagesc(pixc2);
axis('image'); colormap(cmap)
save histog histog
```

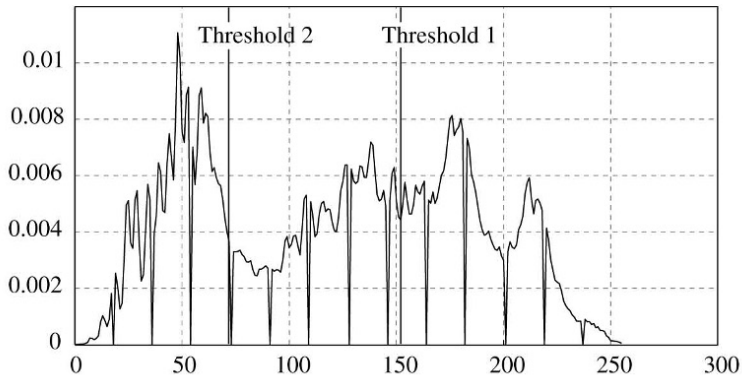

Figure 5.30 – *Histogram and global thresholds*

Figure 5.31 – *Binarization of the image above for two threshold values*

Automatic threshold calulation: the Otsu method

We will now see how to make the choice of the threshold automatic. In order to do this, we will write $h(k)$ to refer to the percentage of values from the image that are equal to k, where $k \in \{0, \ldots, 255\}$, as it was calculated in the previous histogram. $h(k)$ provides an *estimation for the probability* of level k.

Let s be the threshold. s defines two categories of values: category $\mathcal{C}_I$ for values below s, and category $\mathcal{C}_S$ for values above s. The method suggested by Otsu [26] simply consists of choosing, as the threshold value s, the integer for which the quadratic error is minimal between the observed random value k and its corresponding value $\mu(K)$ in one of the two categories.

This method is merely the particular case for 1 bit of the N scalar quantification problem, the solution of which is known as the Lloyd-Max solution.

Let μ_I and μ_S be the number of pixels in the categories $\mathcal{C}_I$ and $\mathcal{C}_S$ respec-

tively. The expression of the criterion we wish to minimize, with respect to s, μ_I and μ_S is:

$$L(s, \mu_I, \mu_S) = \sum_{k=0}^{s-1} (k - \mu_I)^2 h(k) + \sum_{k=s}^{P-1} (k - \mu_S)^2 h(k) \tag{5.31}$$

Minimizing this ratio as a function of μ_I and μ_S can be achieved by zeroing the partial derivatives of $L(s, \mu_I, \mu_S)$ with respect to μ_I and μ_S.

In the case of μ_I for example, this leads to:

$$\frac{\partial L(s, \mu_I, \mu_S)}{\partial \mu_I} = -2 \sum_{k=0}^{s-1} (k - \mu_I) h(k) = 0$$

the solution of which is:

$$\mu_I(s) = \frac{\sum_{k=0}^{s-1} k h(k)}{\sum_{k=0}^{s-1} h(k)} \tag{5.32}$$

Likewise, we have:

$$\mu_S(s) = \frac{\sum_{k=s}^{P-1} k h(k)}{\sum_{k=s}^{P-1} h(k)} \tag{5.33}$$

Notice that there is an obvious interpretation for μ_I and μ_S: they are the respective means of each category. By replacing these two expressions of $L(s, \mu_I, \mu_S)$ in 5.31, we get an expression $J(s)$, dependent only on s, which needs to be minimal. The solution cannot be obtained analytically, but the numerical solution can be found by calculating $J(s)$ for the 256 possible values of s.

There are two equivalent expressions of $J(s)$ that are best adapted for the numerical calculation. Let:

$$P_I(s) = \sum_{k=0}^{s-1} h(k) \text{ and } P_S(s) = \sum_{k=s}^{P-1} h(k) \tag{5.34}$$

such that $P_I(s) + P_S(s) = 1$. The minimizing of $J(s)$ with respect to s is equivalent to *maximizing*:

$$G(s) = P_I(s)\mu_I^2(s) + P_S(s)\mu_S^2(s) \tag{5.35}$$

with respect to s. This is because:

$$\begin{aligned} J(s) &= \textstyle\sum_{k=0}^{s-1} k^2 h(k) - 2 \sum_{k=0}^{s-1} k \mu_I(s) h(k) + \sum_{k=0}^{s-1} \mu_I^2(s) h(k) + \\ &\quad \textstyle\sum_{k=s}^{P-1} k^2 h(k) - 2 \sum_{k=s}^{P-1} k \mu_S(s) h(k) + \sum_{k=s}^{P-1} \mu_S^2(s) h(k) \\ &= \textstyle\sum_{k=0}^{P-1} k^2 h(k) - (\mu_I^2(s) P_I(s) + \mu_S^2(s) P_S(s)) \end{aligned}$$

Hence, because the first term is independent of s, minimizing $J(s)$ is equivalent to maximizing $G(s) = \mu_I^2(s)P_I(s) + \mu_S^2(s)P_S(s)$.

The maximizing of $G(s)$ found in 5.35 is equivalent to the maximizing of:

$$H(s) = P_I(s)P_S(s)(\mu_I(s) - \mu_S(s))^2 \tag{5.36}$$

Notice that the quantity:

$$P_I(s)\mu_I(s) + P_S(s)\mu_S(s) = \sum_{k=0}^{P-1} k^2 h(k)$$

(see expressions 5.32, 5.33 and 5.34) is independent of s. We can calculate its square and substract this square value from $G(s)$ without changing the maximizing with respect to s. We get:

$$\begin{aligned} H(s) &= G(s) - (P_I(s)\mu_I(s) + P_S(s)\mu_S(s))^2 \\ &= \mu_I^2(s)P_I(s) + \mu_S^2(s)P_S(s) - \mu_I^2(s)P_I^2(s) - \mu_S^2(s)P_S^2(s) \\ &\quad -2P_I(s)P_S(s)\mu_I(s)\mu_S(s) \\ &= \mu_I^2(s)P_I(s)(1 - P_I(s)) + \mu_S^2(s)P_S(s)(1 - P_S(s)) \\ &\quad -2P_I(s)P_S(s)\mu_I(s)\mu_S(s) \end{aligned}$$

If we use $P_I(s) + P_S(s) = 1$, we get the expected result 5.36.

Exercise 5.12 (Application of the Otsu method) (see p. 452)

1. Using MATLAB®, write the function that calculates the threshold obtained by maximizing expression 5.36;
2. apply this function to a test image.

Figure 5.32 gives the result obtained by calculating the threshold with the Otsu method.

Figure 5.32 – *Binarization for the threshold calculated with the Otsu method*

5.6.6 Modifying the contrast of an image

Differences in *contrast* were classified by J. Itten [21] into seven different types: "saturation", "light and dark", "extension", "complements", "hue", "warm and cool", "hue-primary" and "simultaneous". These refer to differences perceived between objects of color or brightness. Images in "grayscale" will now be examined and also the methods used to improve transitions between different areas of grayscale.

Global changes

The first and simplest modification consists of reallocating the different colors that appear in the brightness histogram. Thus, if the lighter or darker areas are not represented, it is easy to redistribute the other brightness settings on all other available codes.

Consider the image in Figure 5.33. This image, in 8-bit grayscale, is a little too contrasted. This means that as a whole the luminosity of the histogram is focusing on a relatively narrow brightness spectrum (Figure 5.34).

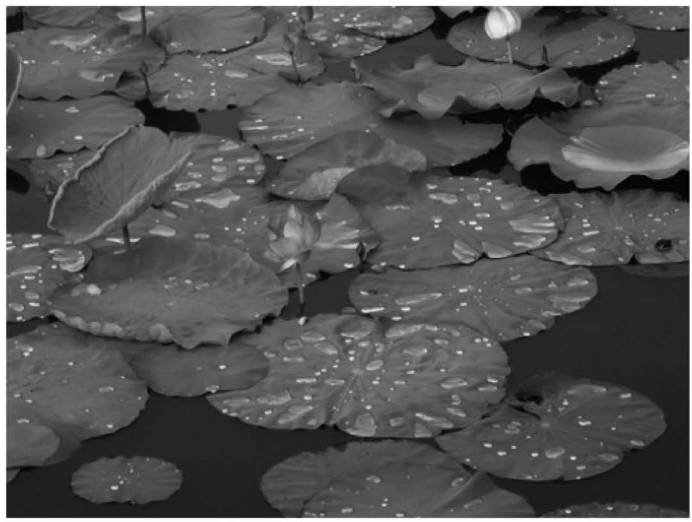

Figure 5.33 – *Details of the image to be modified*

The histogram reveals that the extreme gray values, lower than $h_{\min}$ and higher han $h_{\max}$, are not used. Other gray values ($h_{\min} \leq p(x,y) \leq h_{\max}$) are then distributed between 1 and 256 (fonction `hlinmod`).

```
function pixm=hlinmod(pixc,hmn,hmx)
%!=======================================!
%! SYNOPSIS: pixm=HLINMOD(pixc,hmn,hmx)     !
```

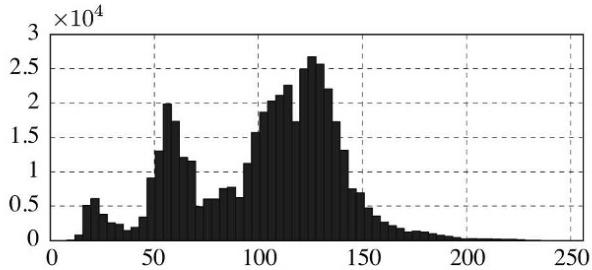

Figure 5.34 – *Histogram of the image to be modified*

```
%! pixc = image array (8-bit, gray)          !
%! hmn, hmx = limits for modifying contrast !
%!=========================================!
[nr,nc]=size(pixc); pixm=ones(nr,nc);
idxm=find(pixc<hmn);idxM=find(pixc>hmx);
pixm(idxm)=hmn; pixm(idxM)=hmx;
pixm=ones(nr,nc)+(pixc-hmn)*(256-1)/(hmx-hmn);
return
```

The `hmn` ou `hmx` parameters can be chosen manually, from the histogram, or automatically by choosing a minimum threshold for the number of representatives in the extreme classes.

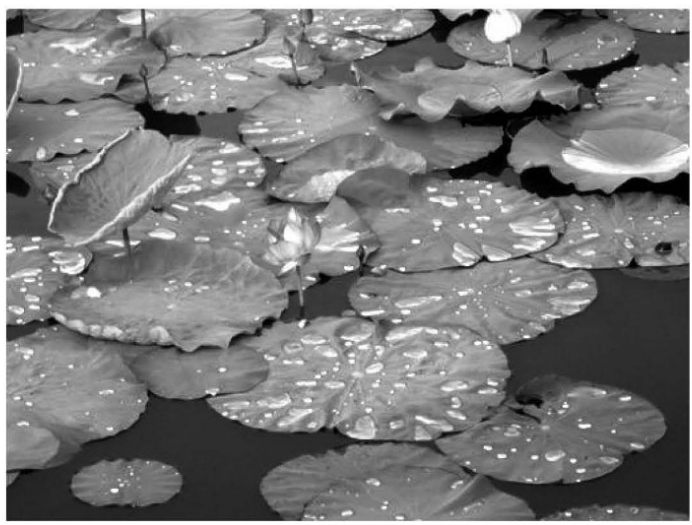

Figure 5.35 – *Details of the image*

The lighter or darker areas appear symmetrically in the histogram, and

can be a distribution different at both ends of the code (fonction `hadaptmod` function for an adaptive modification).

```
function pixm=hadaptmod(pixc,medpx,hmn,hmx)
%!================================================!
%! SYNOPSIS: pixm=HADAPTMOD(pixc,medpx,hmn,hmx) !
%! Adapted enhancement of contrast               !
%! pixc  = image array (8-bit, gray)             !
%! medpx = median(pixc)                          !
%! hmin, hmax = limits for modifying contrast    !
%!================================================!
[nr,nc]=size(pixc); pixm=ones(nr,nc);
theta1=(1-medpx)/(hmn-medpx);
theta2=(256-medpx)/(hmx-medpx);
Ct=ones(nr,nc)*medpx;
idx1=find(pixc<=medpx); idx2=find(pixc>medpx);
pixm(idx1)=Ct(idx1)+theta1*(pixc(idx1)-Ct(idx1));
idx=find(pixm<1); pixm(idx)=1;
pixm(idx2)=Ct(idx2)+theta2*(pixc(idx2)-Ct(idx2));
idx=find(pixm>256); pixm(idx)=256;
return
```

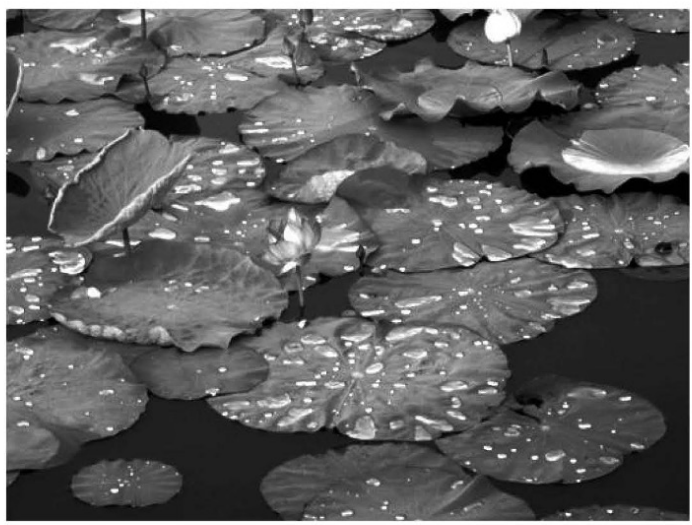

Figure 5.36 – *Modified image*

Local modifications

The presence of very light or very dark areas in an image restricts other areas making them set at lower code levels. Image processing software offers the

possibility of globally changing this distribution interactively (applying distribution curves to the histogram). Nevertheless, this remains a global solution like the previous ones.

In practice, avoid modifying areas too much at the expense of others. Try to make local changes of the contrast. The principle of this is explained in Figure 5.37.

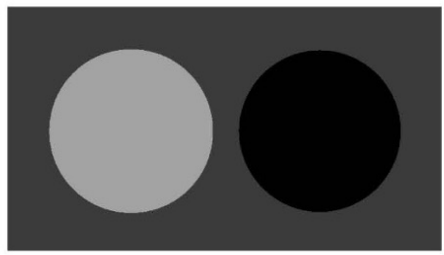

Figure 5.37 – *An image to be modified*

A low-pass filter (Gaussian low-pass filter) was applied to the image in Figure 5.37. The transitions are diminished. Then the difference between the original and the filtered image is used to produce the final image (Figure 5.38).

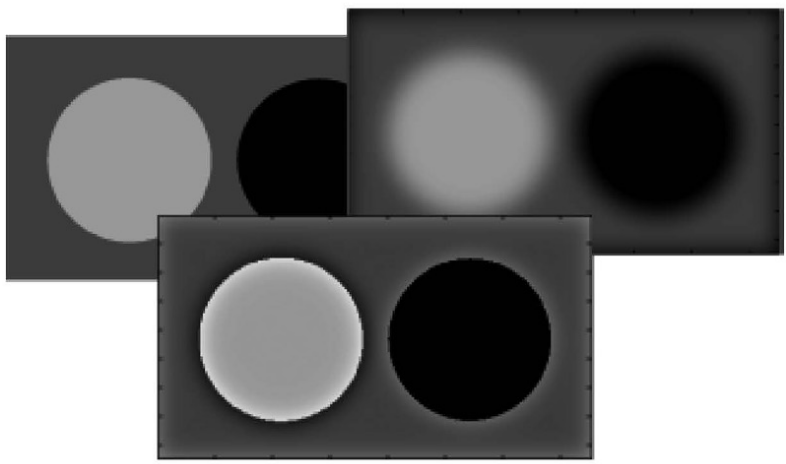

Figure 5.38 – *Results with enhancement on the transitions*

The effect of these operations can be viewed by looking at the gray levels on one of the manipulated image lines. Each transition is best underlined: the lighter side is darkened while the darker side is lightened (Figure 5.39).

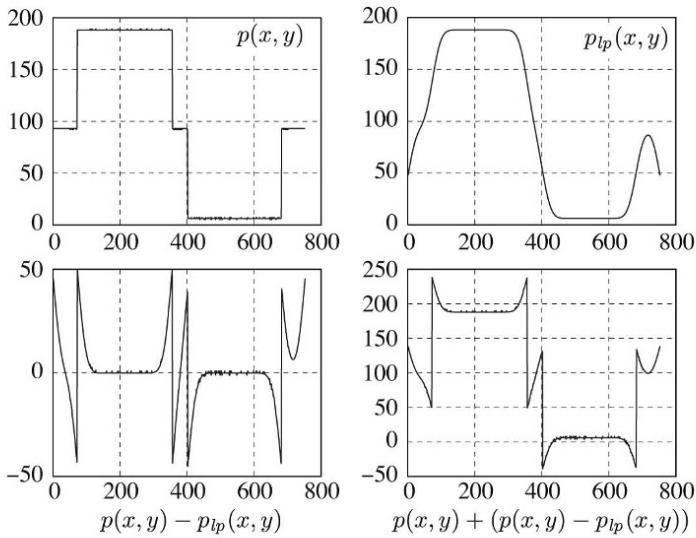

Figure 5.39 – *Modification effect*

Exercise 5.13 (Local contrast changes) (see p. 454)
Applying the method of local modification to the image in Figure 5.33, write a program:

1. which displays the original image,
2. which displays the image that has had the local changes applied,
3. and finally, the end result of the adaptive modification.

In the case of color images, direct application of this principle to each map may cause the appearance of undesired colors. This is the reason why it is advisable to work with codings that show a component of brightness. This is the map that was then applied to the method.

5.6.7 Morphological filtering of binary images

A morphological filtering is a filtering that uses `min` and `max` operations. This can be symbolized as follows:

$$p_{k,l} = \mathcal{F}\left(\mathcal{B}(P)\right) \tag{5.37}$$

where P is an image, $\mathcal{B}(P)$ a portion of the image extracted using a window $\mathcal{B}$, and $\mathcal{F}$ is a logical operation applied to pixels isolated by the window $\mathcal{B}$. The following function, called *erosion*, illustrates the process 5.37 applied to a binary image when the `min` operation amounts to a logical AND:

```
function ppx=erosion(block,mtool)
%!=================================================!
%! SYNOPSIS: ppx=EROSION(block,mtool)              !
%!   block = data block of the same size as the tool !
%!   mtool = matrix of the tool shape (boolean)    !
%!   Example: [0 1 0;1 1 1;0 1 0] defines a cross. !
%!   ppx   = resulting pixel value                 !
%!=================================================!
[nr,nc]=size(block); L=nr*nc;
bb8=uint8(block); mm8=uint8(mtool*255);
bm=bitand(bb8,mm8);
ppx=bm(1);
for k=2:L, ppx=bitand(ppx,bm(k)); end
ppx=double(ppx);
end
```

In this function, the windowing matrix associated with the operator $\mathcal{B}$ is referred to as the *structuring element*. It consists of a boolean matrix. A "1" indicates a pixel that needs to be taken into account by the logical function processing. Hence the processing can be symbolized by:

$$p_{k,l} = \bigcap_{\{n,m:b_{n,m}=1\}} p_{n,m}$$

The program `exerosion.m` illustrates the `erosion` function call:

```
%===== exerosion.m
% processing using uint8 data
clear
mdisp=true; NbLevel=256;
cmap=flipud([0:NbLevel-1]'/NbLevel)*[1 1 1];
load exerosion, [nl0,nc0]=size(pixc);
if mdisp
    subplot(131); imagesc(pixc); axis('image')
    colormap(cmap);
end
%===== defining the tool
mtool=ones(3,3);
[nr,nc]=size(mtool);
Nrowu=fix(nr/2); Nrowb=Nrowu;
Ncoll=fix(nc/2); Ncolr=Ncoll;
if rem(nr,2)==0, Nrowu=Nrowu-1; end
if rem(nc,2)==0, Ncoll=Ncoll-1; end
%===== the image must be coded between 0 and NbLevel-1
pixc=[ones(nl0,Ncoll) pixc ones(nl0,Ncolr)];
pixc=[ones(Nrowu,nc0+nc-1);pixc;ones(Nrowb,nc0+nc-1)];
ppxe=zeros(nl0+nr-1,nc0+nc-1); ppxd=ppxe;
%=====
for nl=Nrowu+1:nl0+Nrowu
```

```
    for nc=Ncoll+1:nc0+Ncolr
        blk=pixc(nl-Nrowu:nl+Nrowb,nc-Ncoll:nc+Ncolr);
        ppxe(nl,nc)=erosion(blk,mtool);   %=====
        ppxd(nl,nc)=dilation(blk,mtool);  %=====
    end
end
if mdisp
    subplot(132);
    imagesc(ppxe(Nrowu+1:nl0+Nrowu,Ncoll+1:nc0+Ncoll));
    axis('image')
    subplot(133);
    imagesc(ppxd(Nrowu+1:nl0+Nrowu,Ncoll+1:nc0+Ncoll));
    axis('image')
end
```

In this program, the structuring element is a (3×3) square. Its execution is particularly slow. MATLAB® is not well suited for this type of processing comprising many loops. The best method would once again be to write a dedicated "`.mex`" function. The image *toolbox*, of course, provides such functions.

If, in the `erosion.m` function, the AND ($\cap$) function is replaced with an OR ($\cup$) function, the result is a *dilation* function. This means we are dealing with the implementation of the `max` function for binary images.

```
function ppx=dilation(block,mtool)
%!=================================================!
%! SYNOPSIS: ppx=DILATION(block,mtool)             !
%!   block = data block of the same size as the tool !
%!   mtool = matrix of the tool shape (boolean)    !
%!   Example: [0 1 0;1 1 1;0 1 0] defines a cross. !
%!   ppx   = resulting pixel value (double)        !
%!=================================================!
[nr,nc]=size(block); L=nr*nc;
bb8=uint8(block); mm8=uint8(mtool*255);
bm=bitand(bb8,mm8);
ppx=bm(1);
for k=2:L, ppx=bitor(ppx,bm(k)); end
ppx=double(ppx);
end
```

Figure 5.40 illustrates the respective effects of erosion and dilation. In the case of erosion, any pattern not covered by the window disappears. The contours of the objects in the foreground are "eroded". Dilation, on the contrary, emphasizes the image's details by "increasing" their size.

5.7 JPEG lossy compression

The JPEG format (*Joint Photographic Experts Group*) for coding image files is widely used because of the compression rates it can achieve without significant

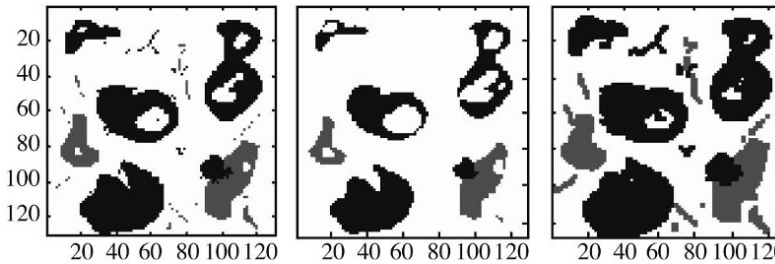

Figure 5.40 – *Effects of erosion and dilation: original, eroded and dilated images from left to right respectively*

quality loss. We are going to construct the functions of this coding, without trying, however, to construct the final binary flux.

The idea behind this coding has to do with the use of the *discrete cosine transform*, or DCT.

5.7.1 Basic algorithm

The JPEG compression (lossy compression) algorithm can be very briefly summed up as follows [15]:

- the image is divided in blocks of 8 by 8 pixels, to which a DCT is applied (the blocks are read line by line, from top to bottom and from left to right). The basic process implies that the levels associated with each pixel are 8 bit coded. To make things simpler, we will assume that the images we are going to process are given in "levels of gray";
- the 64 coefficients of the DCT are quantified (rounded);
- the "mean value" (DCT value at the frequency 0) is subtracted from the same term of the next block;
- the 63 other terms are read in "zigzags" (Figure 5.41);
- the sequence of the obtained values is coded (Huffman entropic coding);
- each non-zero coefficient is coded by the number of zeros preceding it, the number of bits needed for its coding, and its value. The coding rules are imposed by the [33] standard.

We assume that we will keep the floating-point representation coding. We will not try to optimize the size of the coded DCTs.

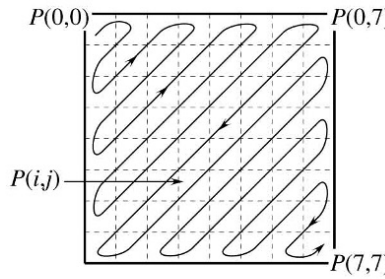

Figure 5.41 – *Reading of the DCT coefficients*

5.7.2 Writing the compression function

Writing the DCT calculation and quantification functions

Consider an (8×8) array of pixels $p(x, y)$ $(x, y \in \{0, \ldots, 7\})$, the DCT's expression for $u, v \in \{0, \ldots, 7\}$:

$$P(u,v) = \frac{1}{4} C(u)C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} p(x,y) \cos \frac{(2x+1)\pi u}{16} \cos \frac{(2y+1)\pi v}{16} \quad (5.38)$$

with $C(0) = \frac{1}{\sqrt{2}}$ and $C(k) = 1$ for $k = 1 \ldots 7$. Once the coefficients are obtained, the array is weighted and quantified:

$$P_q(u,v) = \texttt{round} \frac{P(u,v)}{Qtab(u,v)}$$

where `Qtab` is a quantification table for chrominance included with the standard as an example. It is supposed [35] to provide good results for the type of coding performed here and for most images commonly dealt with.

The following initialization function returns the `Qtab` table as well as the indices used for the "zigzag" reading of the DCT array:

```
function [Qtab,zig,zag]=initctes
%!================================================!
%! Init. of the constants for the JPEG algorithm !
%!================================================!
global mNORM mYV mUX
mUX = cos([0:7]'*(2*[0:7]+1)*pi/16);
mYV = cos((2*[0:7]'+1)*[0:7]*pi/16);
mNORM = [1/2 ones(1,7)/sqrt(2); ones(7,1)/sqrt(2) ones(7,7)]/4;
Qtab=[16 11 10 16 24 40 51 61;
      12 12 14 19 26 58 60 55;
      14 13 16 24 40 57 69 56;
      14 17 22 29 51 87 80 62;
```

```
      18 22 37 56 68 109 103 77;
      24 35 55 64 81 104 113 92;
      49 64 78 87 103 121 120 101;
      72 92 95 98 112 100 103 99];
zig=[1 9 2 3 10 17 25 18 ...
     11 4 5 12 19 26 33 41 ...
     34 27 20 13 6 7 14 21 ...
     28 35 42 49 57 50 43 36 ...
     29 22 15 8 16 23 30 37 ...
     44 51 58 59 52 45 38 31 ...
     24 32 39 46 53 60 61 54 ...
     47 40 48 55 62 63 56 64];
zag=zig(64:-1:1);
return
```

Exercise 5.14 (Writing basic functions) (see p. 454)

1. Write the calculation function of the DCT using the vectors `mNORM`, `mYV` and `mUX`, which will be declared `global mNORM mYV mUX` and initialized with the `initctes.m` function;
2. write the quantification function;
3. check the processing using the following data [1].

The data we are working with are coded on one byte, a value between 0 and 255, that you need to bring back between −128 and 127:

```
%===== dataex.m
pix=[139 144 149 153 155 155 155 155;
     144 151 153 156 159 156 156 156;
     150 155 160 163 158 156 156 156;
     159 161 162 160 160 159 159 159;
     159 160 161 162 162 155 155 155;
     161 161 161 161 160 157 157 157;
     162 162 161 163 162 157 157 157;
     162 162 161 161 163 158 158 158];
```

The result has to be:

```
    15     0    -1     0     0     0     0     0
    -2    -1     0     0     0     0     0     0
    -1    -1     0     0     0     0     0     0
    -1     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

COMMENT: the quantification table is usually associated with a *quality factor* F_q. The previous table corresponds to $F_q = 50\,\%$. The following function can be used to generate tables for other values of F_q:

```
function [Qtab]=TabQuantif(Fq)
%!=====================================!
%! SYNOPSIS: [Qtab]=TABQUANTIF(Fq)     !
%!     Fq   = quality factor (0 to 100) !
%!     Qtab = weighting table           !
%!=====================================!
if nargin<1, Fq=50; end
%===== table for a quality factor = 50
Qtab=[16 11 10 16 24 40 51 61 ;
      12 12 14 19 26 58 60 55 ;
      14 13 16 24 40 57 69 56 ;
      14 17 22 29 51 87 80 62 ;
      18 22 37 56 68 109 103 77 ;
      24 35 55 64 81 104 113 92 ;
      49 64 78 87 103 121 120 101 ;
      72 92 95 98 112 100 103 99];
%=====
if (Fq<50)
   scal = 5000/Fq; else scal = 200 - Fq*2;
end
Qtabnew=floor(((Qtab.*scal)+50)./100);
idz=find(Qtabnew<=0); Qtabnew(idz)=ones(size(idz));
idz=find(Qtabnew>255); Qtabnew(idz)=255*ones(size(idz));
Qtab=Qtabnew;
return
```

Exercise 5.15 (Writing the compressed frame) (see p. 457)

1. Using the functions written in exercise 5.14, write the function that creates the compressed frame for *one block*. The starting mean value is assumed to be zero. For the previous exercise, the result, with some comments, should be:

```
%==== Test block
5        number of bits used to code the difference
15       diff. with the previous block's mean
         (0 in this case)
1,2,-2   1 zero before the -2, which is 2 bit coded
0,1,-1   no zero before the -1 which is 1 bit coded
0,1,-1   idem
0,1,-1   idem
2,1,-1   2 zeros before the -1 which is 1 bit coded
0,1,-1   no zero before the -1 which is 1 bit coded
0,0      there is nothing left but zeros
```

Do not calculate, for now, the number of bits needed for coding each of the DCT's coefficients. We will assume it is the same for all of them, and that its value is 17.

Save the compressed data to the file `unbloccode.dat`, but after having added at the beginning of the file the number of line blocks and of column blocks as follows:

```
fid=fopen('unbloccode.dat','w');
fwrite(fid,nby,'integer*1');
fwrite(fid,nbx,'integer*1');
% Writing the compressed data
for k=1:...
    fwrite(fid,...,'integer*1');
end
fclose(fid);
```

2. apply the obtained program to the test image. Save the compressed data to the file `imgtstcode.dat`.

5.7.3 Writing the decompression function

Inverse DCT

The inverse DCT, referred to as the ICDT, is given by 5.39:

$$p(x,y) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} P(u,v)C(u)C(v)\cos\frac{(2x+1)\pi u}{16}\cos\frac{(2y+1)\pi v}{16} \quad (5.39)$$

Exercise 5.16 (Decompression) (see p. 458)

1. Using the `Qtab` table given on page 235, write the "dequantization function" of the DCT coefficients;

2. write the inverse DCT function;

3. test the "decompression" operation by applying it to the previously used test block which is coded as follows:

```
A=[1,1,17,15,1,17,-2,0,17,-1,0,17,-1,0, ...
   17,-1,2,17,-1,0,17,-1,0,0]
```

The first two terms indicate that there is only one block;

4. apply the program to the file `imgtstcode.dat` obtained in exercise 5.15 (the result obtained with the test image that was chosen is shown in Figure 5.42).

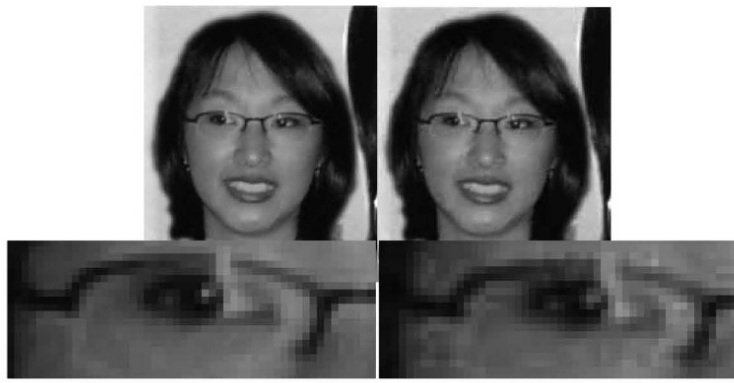

Figure 5.42 – *Comparing original images with images obtained by coding and decoding for a quality factor of* $\approx 30\%$