1. Introduction to the OpenCV library

1.1. Introduction

The purpose of this laboratory is to acquaint the students with the framework application which will be used in the practical works related to the Image Processing course.

The background knowledge necessary to successfully complete the image processing laboratory are:

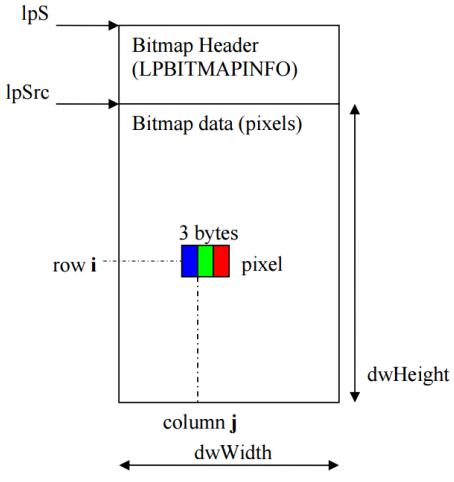
- **Compulsory**: C, Computer Programming, Data Structures and Algorithms.
- **Optional (recommended)**: *C++, Visual C++ 12.0 (Visual Studio 2013), Object Oriented Methods, Fundamental Algorithms, Programming Techniques, Linear Algebra and Geometry, Discrete Mathematics, Numerical Calculus, Special Mathematics*

1.2. The bitmap image format

The bmp format is used to store images in uncompressed form. It uses raster graphics to store digital images independently of the display device. It is capable of storing monochrome and color images with different encoding depth. The depth determines the number of possible colors and determines the image size. The file itself has the following structure:

- a bitmap file header - which contains a signature field, the file size and the offset to the pixel array;
- DIB header - which stores various information such as image dimensions, bits per pixel;
- color table (or look-up table) - for images with a color palette;
- the pixel array - contains the actual image information stored in a linearized manner and padded.

The following image illustrates the bitmap format for a 24bit color image. The image height and width are denoted dwHeight and dwWidth respectively.



Fig, 1.1. Bitmap image format

1.3. Overview of the OpenCV framework

The framework on which you will be working on contains the OpenCV library 2.4.13 bundled together with a Visual Studio 2013 solution. Include settings have been preconfigured, all static and dynamic libraries are included with the solution.

Your task is to create new functions and call them from the main function. You should group your work according to laboratory sessions and give suggestive names to functions. All code examples assume that you have included the cv namespace (**`using namespace cv`**), otherwise prepend **`cv::`** to all OpenCV classes and methods. A guideline for introducing new functions is given in the following code snippet (gray text indicates what you need to introduce):

```
void negative_image(){
   //implement function
}
int main(){
   int op;
   do{
        printf("Menu:\n");
        //...
        printf(" 7 - L1 Negative Image \n");
        //...
        printf(" 0 - Exit\n\n");
        printf("Option: ");
        scanf("%d",&op);
        switch (op)
        {
            //...
            case 7:
                negative_image();
                break;
        }
   }
   while (op!=0);
   return 0;
}
```

You should save your work from each session. The project can be cleaned with the clean.bat executable which deletes all build outputs and reduces the project size considerably. *Alternatively, to save space, just backup the main cpp file since the project solutions should not change.*

1.4. The Mat class

Images are stored as *Mat* objects in OpenCV. It is class for a generic matrix that can be used to hold other data as well, such as a normal 2x2 matrix or higher dimensional matrices.

Important fields of the *Mat* class are:

- rows - the number of rows of the matrix = the height of the image;
- cols - the number of columns of the matrix = the width of the image;
- data - pointer to the memory location of the actual image; it is of type **`unsigned char *`**, so it must be cast to the correct type for accessing operations

The simplest and cleanest way to create a *Mat* object called *img* is to use the 3 parameter constructor:

```
Mat img(rows, cols, type);
```

The last parameter encodes the type of data that is stored in the matrix. An example type would be CV_8UC1, which it represents: 8 bit, unsigned char, single channel. In general the first number after CV_ represents the number of bits required; the letter indicates the data type; and Cx shows the number of channels.

Type code	Data type	Used for
CV_8UC1	unsigned char	grayscale image (8bits/pixel)
CV_8UC3	Vec3b	color image (3x8bits/pixel)
CV_16SC1	short	data storage
CV_32FC1	float	data storage
CV_64FC1	double	data storage

Table 1. Common OpenCV data type codes

Example 1 - create a grayscale matrix of size 256x256:

```
Mat img(256,256,CV_8UC1);
```

Example 2 - create a color image of dimension with 720 rows and 1280 columns:

```
Mat img(720,1280,CV_8UC3);
```

Example 3 - create a 2x2 real matrix with values [1 2; 3 4], and print it:

```
float vals[4] = {1, 2, 3, 4};
Mat M(2,2,CV_32FC1,vals); //4 parameter constructor
std::cout << M << std::endl;
```

Notice, you can use the standard output stream with a *Mat* object.

For a detailed description of the *Mat* class, see the official documentation at: http://docs.opencv.org/2.4.13/modules/core/doc/basic_structures.html#mat

1.5. Opening/reading an image

To open an image and to store it as a Mat object use the **imread** function:

```
Mat img = imread("path_to_image", flag);
```

The first parameter contains the relative or absolute path to the image file; the second flag parameter can be:

- CV_LOAD_IMAGE_UNCHANGED (-1) - load the image in the same format as it was saved;
- CV_LOAD_IMAGE_GRAYSCALE (0) - load the image as a grayscale image; loading converts it to 8UC1 (1 channel unsigned char) image and performs grayscale conversion if required;
- CV_LOAD_IMAGE_COLOR (1) - load the image and convert it to a 8UC3 (3 channel unsigned char) image; it copies the grayscale channel to all color channels if required.

Example 1 - open an image in the current folder in the format it was saved:

```
Mat img = imread("cameraman.bmp", -1);
```

1.6. Accessing the data from an image

Matrix elements are indexed according to standard mathematical matrix notation. This means that the origin will be positioned at the top left corner of the image. The first index will indicate the row (increasing downwards) and the second index will indicate the column (increasing to the right). The following figure illustrates the indexing scheme:

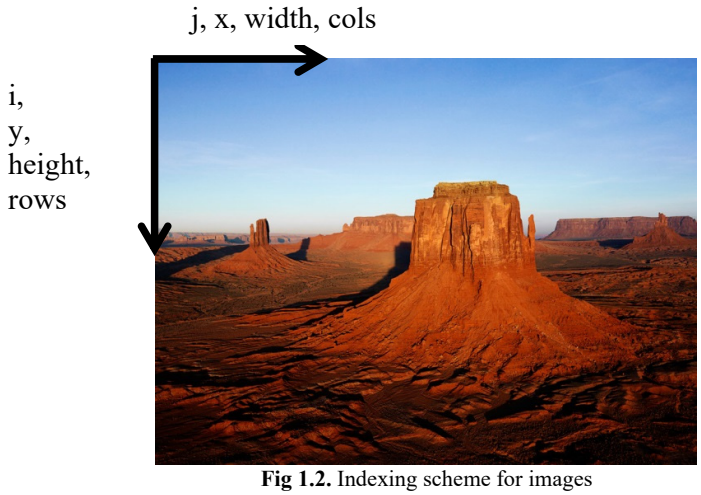


Fig 1.2. Indexing scheme for images

Always follow this convention to avoid indexing mistakes. When processing an image, first loop over the rows then over the columns.

To access the data from a grayscale image at row *i* and column *j* use the **at** method:

```
unsigned char pixel = img.at<unsigned char>(i,j);
```

Notice that you need to provide the data type which is stored in the matrix (**unsigned char**).

For faster access, we can use the **data** pointer and the **step** field directly:

```
unsigned char pixel = img.data[i*img.step[0] + j];
```

All data is stored in a linearized manner, row after row and from left to right, starting from the **data** pointer. Padding may be introduced so ***avoid*** accessing via **i*img.cols+j** because it might give wrong results for padded images.

You can also use a pointer to the data from the *i*-th row:

```
unsigned char pixel = img.ptr(i)[j];
```

To access the 3 component color at row *i* and column *j* from a color image, use the proper type:

```
Vec3b pixel = img.at< Vec3b>(i,j);
unsigned char B = pixel[0];
unsigned char G = pixel[1];
unsigned char R = pixel[2];
```

Vec3b is a vector with 3 byte (**unsigned char**) components. |It is recommended for manipulating color images.

The code can be simplified by using the **Mat_<T>** templated subclass of the *Mat* class, which enables omitting the type for access operations. At the creation of a **Mat_<T>** object you must provide the underlying type that is stored in the matrix.

```
Mat_<uchar> img = imread("fname",CV_LOAD_IMAGE_GRAYSCALE);
uchar pixel = img(i,j);
```

Here we have also used the type definition **uchar** which stands for **unsigned char**. Accessing a value from a certain position permits both reading and writing operations.

1.7. Viewing an image

To view a loaded image use the **imshow** function followed by a **waitKey** call:

```
imshow("image", img);
waitKey(0);
```

This shows the image in a new window titled *image* and waits for the user to input a key indefinitely. The **waitKey** function has only one parameter: how long it waits for a user input (measured in milliseconds). Zero means to wait forever.

Always follow each **imshow** operation with a **waitKey** command. Image windows can be moved and resized, which is desirable if you want to illustrate input and output side by side in the same configuration many times.

1.8. Saving/writing an image

To save an image to the disk use the *imwrite* function:

```
imwrite("fname", img);
```

The file name contains the path, the name and the extension, which determines the format of the image. You can save in multiple formats such as: *bmp, jpg, png*.

1.9. Sample function

The following sample code loads a grayscale image and transforms it into its negative image:

```
void negative_image(){
    Mat img = imread("Images/cameraman.bmp",
                     CV_LOAD_IMAGE_GRAYSCALE);
    for(int i=0; i<img.rows; i++){
        for(int j=0; j<img.cols; j++){
            img.at<uchar>(i,j) = 255 - img.at<uchar>(i,j);
        }
    }
    imshow("negative image",img);
    waitKey(0);
}
```

The image file must reside in the Images folder next to the project solution file.

1.10. Practical work

1. Download and build the *OpenCVApplication*.
2. Test the *negative_image* function.
3. Implement a function which changes the gray levels of an image by an additive factor.
4. Implement a function which changes the gray levels of an image by a multiplicative factor. Save the resulting image.
5. Create a color image of dimension 256 x 256. Divide it into 4 squares and color the squares from top to bottom, left to right as: white, red, green, yellow.
6. Create a 3x3 float matrix, determine its inverse and print it.
7. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms.**

1.11. References

[1] http://docs.opencv.org/2.4.13/modules/core/doc/basic_structures.html

2. Color spaces

2.1 Introduction

The purpose of the second laboratory work is to teach the basic color manipulation techniques, applied to the bitmap digital images.

2.2 The RGB color space

The color of each pixel, either in image acquisition devices such as cameras, and in image displaying devices such as the computer monitor and the TV screen, is obtained by combining three primary colors: Red, Green and Blue (additive color space – Fig. 2.1 and 2.2).

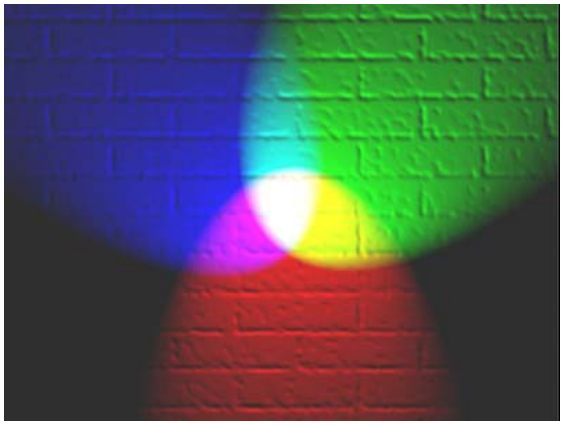


Fig. 2.1. Additive mixing of colors. When the primary colors are superposed, the secondary colors appear. When all three primary colors are superposed, the white color is obtained [1].

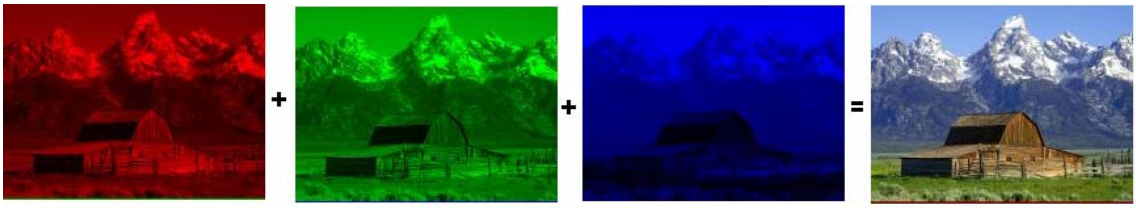


Fig. 2.2. The color image is obtained by pixel level combination of the primary colors. The three color channels are displayed.

Each image pixel will be defined by a triplet, containing a numerical value for each primary color. The color can be regarded as a point in a 3D RGB color space (Fig. 2.3). The origin of the coordinate axes corresponds to the color Black (0,0,0), and the opposite corner of the color space cube corresponds to the color White (255, 255, 255). The cube's diagonal, between black and white, corresponds to levels of gray (grayscale), defined by (R=G=B). Three of the corners correspond to the primary colors Red, Green and Blue. The other corners correspond to the complementary colors of Cyan, Magenta and Yellow. If the origin of the color space is moved to the White point, and the axes of the system are renamed as C, M and Y, one gets the complementary CMY color space, which is used in color printing devices.

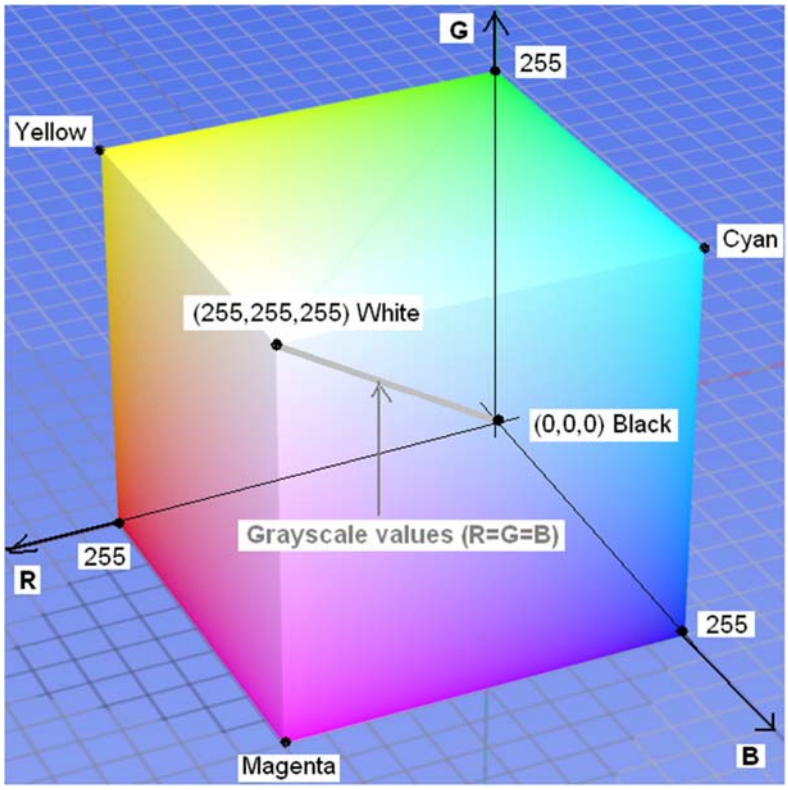


Fig. 2.3. The RGB color space mapped on a cube. Here, each color axis is represented on 8 bits (256 levels) (RGB24 bitmap images). The total number of colors is $2^8 x 2^8 x 2^8 = 2^{24} = 16.777.216$.

For RGB24 images, all possible color combinations can be displayed simultaneously. If the image contains a palette, and the color of a pixel is an index in the palette, only a subset of the colors can be displayed. In this context, the number of bits/pixel (the number of bits used to encode a color) is called "color depth" (Table 2.1):

Table 2.1. Color depth and image type

Color depth	Number of colors	Color mode	Palette (LUT)
1 bit	2	Indexed Color	Yes
4 bits	16	Indexed Color	Yes
8 bits	256	Indexed Color	Yes
16 bits	65536	True Color	No
24 bits	16.777.216	True Color	No
32 bits	16.777.216	True Color	No

There are other color models [2], which will not be discussed here.

2.3 Conversion of a color image to grayscale

In order to convert a color pixel to a grayscale pixel, its color components must be made equal. A widely used conversion method is to compute the intensity as the average of the three channels:

$$R_{Dst} = G_{Dst} = B_{Dst} = \frac{R_{Src} + G_{Src} + B_{Src}}{3} \quad (2.1)$$

2.4 Conversion of a grayscale image to binary (black and white)

A binary image, having only two pixel values (black and white) is obtained from a grayscale image through an operation called thresholding. This operation involves the comparison of the graylevel pixels with a value called "threshold". Thresholding is the simplest segmentation technique, which allows the separation of foreground objects from the background (Fig. 2.4).

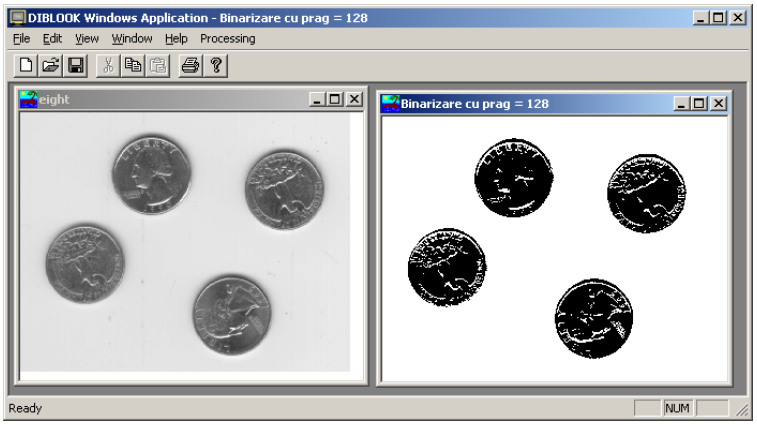


Fig. 2.4. Thresholding.

In this laboratory work you will implement the thresholding operation using a fixed, user defined threshold, for grayscale 8 bit images. The pixels from the source image will be compared to the threshold value, and the destination will be set to:

$$Dst(i,j) = \begin{cases} 0 \quad (black) \quad , \quad if \quad Src(i,j) < threshold \\ 255 \quad (white) \quad , \quad if \quad Src(i,j) \geq threshold \end{cases} \tag{2.2}$$

2.5 The HSV (Hue Saturation Value) color space

This color space tries to mimic the way the humans perceive color. The H component (hue) is the color itself, independent (invariant) of illumination, the S component (saturation) is the color's "purity" (how well defined the color is), and V (value, or intensity) is the brightness. This space is represented as a pyramid with a hexagonal base, or as a cone.

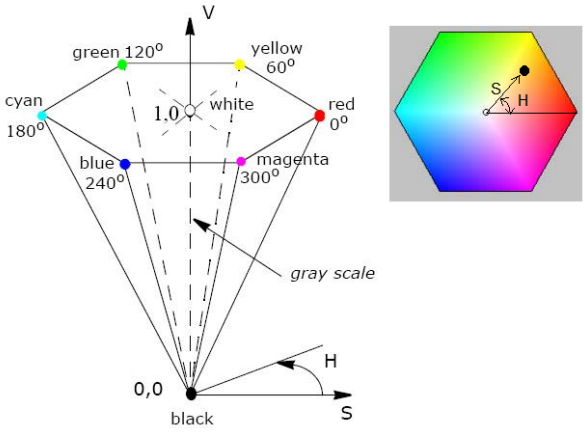


Fig. 2.5. The HSV color space.

Using the pyramid representation, the significance of the components is:

H – the angle between the current color and the ray corresponding to the color Red.
S – the distance from the current color to the central axis of the pyramid/code.
V – the height of the current color in the pyramid/cone.

2.6 The RGB → HSV transform

The equations for obtaining the HSV components from RGB are [3]:

```
r = R/255;  // r : the normalized R component
g = G/255; // g : the normalized G component
b = B/255; // b : the normalized B component
// Attention: please declare all variables as float
// If you have declared R as uchar, you have to use a cast: r = (float)R/255 !!!

M = max (r, g, b); //Attention: there is a default macro in Visual C for max and min, but
m = min (r, g, b);  //it only takes two parameters (no compiler error if you provide three)
C = M - m;
```

Value:

```
V = M;
```

Saturation:

```
If (V!=0)
    S = C / V;
Else // black
        S = 0;
```

Hue:

```
If (C!=0) {
    if (M == r) H = 60 * (g - b) / C;
    if (M == g) H = 120 + 60 * (b - r) / C;
    if (M == b) H = 240 + 60 * (r - g) / C;
    }
Else  // grayscale
    H = 0;
If (H < 0)
        H = H + 360;
```

The values for H, S and V computed with the previous equations will have the following range:

H = 0 .. 360
S = 0 .. 1
V = 0 .. 1

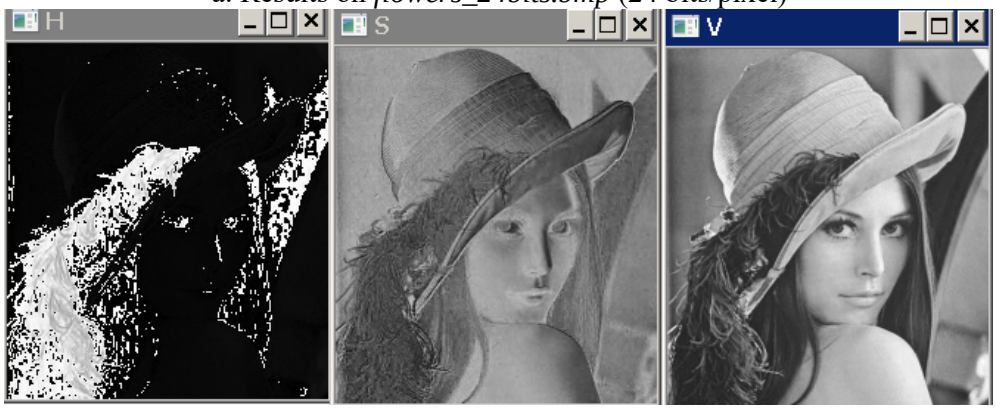
In order to display them as 8-bit grayscale images, you will need to scale them to the 0…255 interval:

H_norm = H*255/360
S_norm = S*255
V_norm=V*255

2.7 Practical work

1. Create a function that will copy the R, G and B channels of a color, RGB24 image (CV_8UC3 type) into three matrices of type CV_8UC1 (grayscale images). Display these matrices in three distinct windows.
2. Create a function that will convert a color RGB24 image (CV_8UC3 type) to a grayscale image (CV_8UC1), and display the result image in a destination window.
3. Create a function for converting from grayscale to black and white (binary), using (2.2). Read the threshold from the console. Test the operation on multiple images, and using multiple thresholds.
4. Create a function that will compute the H, S and V values from the R, G, B channels of an image, using the equations from 2.6. Store each value (H, S, V) in a CV_8UC1 matrix. Display these matrices in distinct windows. Check the correctness of your implementation using the example below.
5. Implement a function called *isInside(img, i, j)* which checks if the position indicated by the pair *(i,j)* (row, column) is inside the image *img*.
6. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms.**

a. Results on *flowers_24bits.bmp* (24 bits/pixel)



b. Results on *Lena_24bits.bmp* (24 bits/pixel)

Fig. 2.6. Examples of RGB to HSV conversion.

2.8 References

[1] http://en.wikipedia.org/wiki/RGB_color_model
[2] http://en.wikipedia.org/wiki/Color_models
[3] Open Computer vision Library, Reference guide, cvtColor() function, http://docs.opencv.org/2.4.13/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor

3. The histogram of image intensity levels

3.1. Introduction

This laboratory work presents the concept of image histogram together with an algorithm for dividing the image histogram into multiple bins and reducing the number of image gray levels (gray levels quantization).

3.2. The histogram of intensity levels

Given a grayscale image with the highest intensity value L (for an image with 8 bits/pixel $L=255$), the intensity (gray) level histogram is defined as a function $h(g)$ that is equal to as value the number of pixels in the image (or in the region of interest) that have intensity equal to g, for each intensity level $g \in [0 \dots L]$.

$$h(g) = N_g \tag{3.1}$$

N_g – the number of pixels in the image or in the region of interest that have the intensity equal to g.

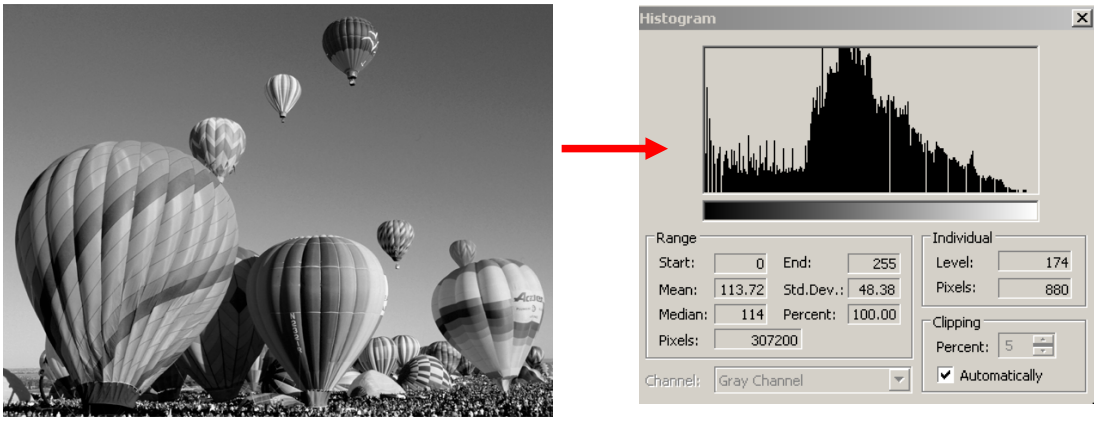


Fig. 3.1 Example: the histogram of a grayscale image

The function obtained by normalizing the histogram with the number of pixels in the image (in the ROI) is called the probability density function (PDF) of the intensity levels.

$$p(g) = \frac{h(g)}{M} \tag{3.2}$$

Where:

$M = image\_height \times image\_width$

PDF has the following properties:

$$\begin{cases} p(g) >= 0 \\ \int_{-\infty}^{\infty} p(g)dg = 1, \quad \sum_{g=0}^{L} \frac{h(g)}{M} = \frac{M}{M} = 1 \end{cases} \tag{3.3}$$

3.3. Application: Multilevel thresholding

In the following we describe an algorithm which determines multiple thresholds for reducing the number of image intensity (gray) levels.

Its first step is to determine the local maxima of the histogram. Then, each gray level is assigned to the closest maximum.

The following steps must be performed in order to determine the histogram maxima:

1. Normalize the histogram (transform it into a PDF)
2. Choose a window width *2*WH+1* (a good value for WH is 5)
3. Choose a threshold *TH* (a good value is 0.0003)
4. For each position (middle of the window) *k* from *0+WH* to *255-WH*
   - Compute the average *v* of normalized histogram values in the interval [*k-WH*, *k+WH*]. Remark: the value *v* is the average of *2*WH+1* values
   - If *PDF*[*k*]>*v+TH* and *PDF*[*k*] is greater or equal than all *PDF* values in the interval [*k-WH*, *k+WH*] then *k* corresponds to a histogram maximum. Store it and then continue from the next position.
5. Insert 0 at the beginning of the maxima position list and 255 at the end (this allows the colors black and white to be represented exactly).

The second step is thresholding. Thresholds are located at equal distances between the maxima. Therefore the algorithm for thresholding is simply to assign to each pixel the color value of the nearest histogram maximum.

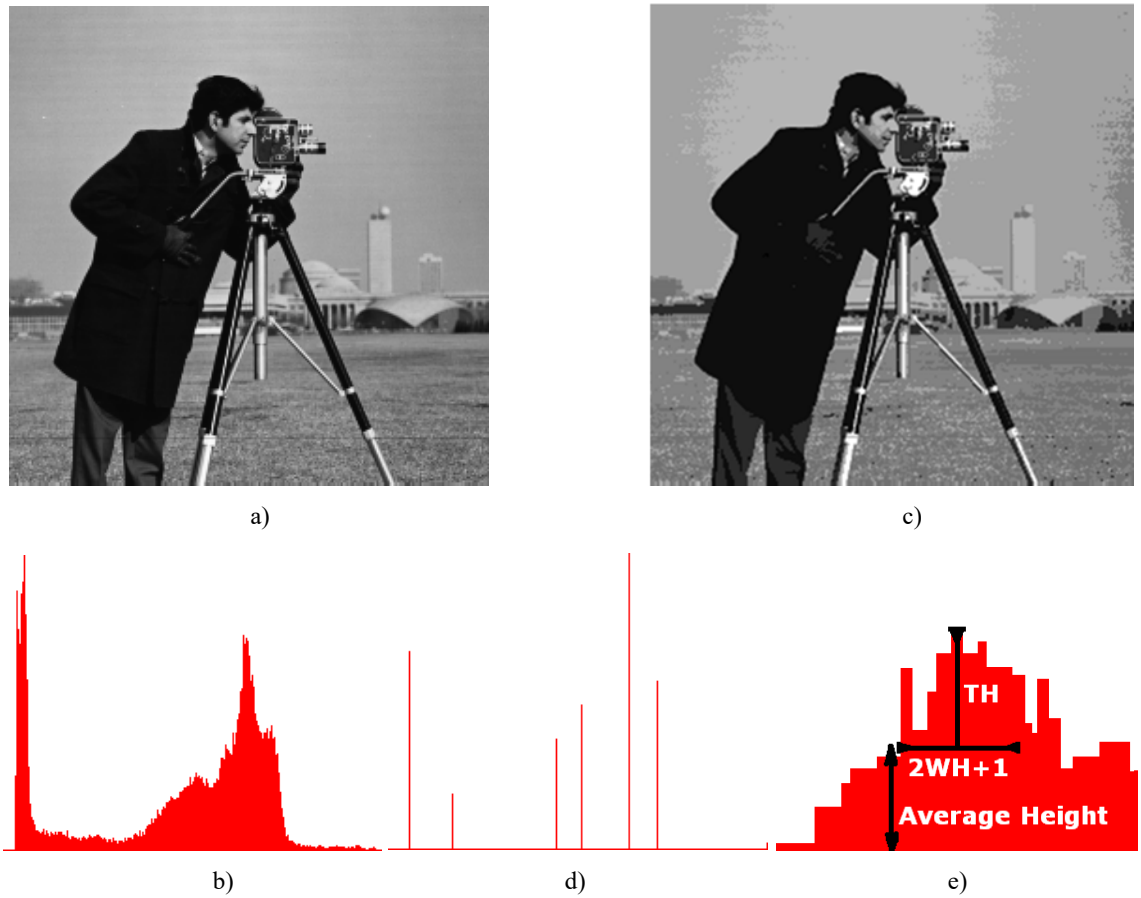


Fig. 3.2 a) The initial image; b) The histogram of the initial image; c) The obtained multilevel thresholded image; d) The histogram of the multilevel thresholded image; e) The histogram maxima computation algorithm

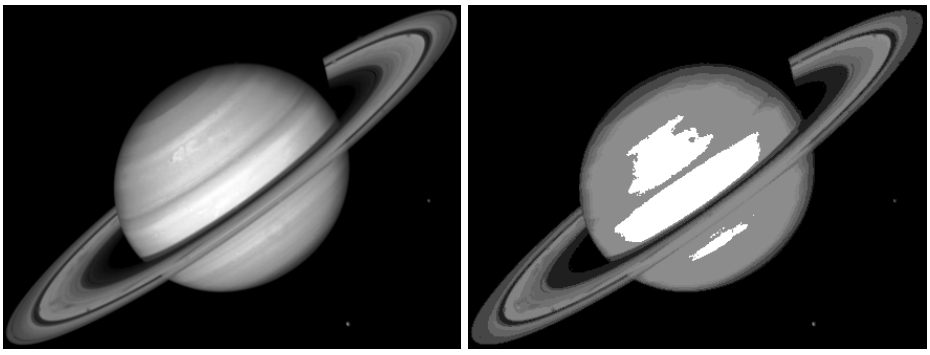
3.4. Floyd-Steinberg dithering

As seen in Fig. 3.3b, the results are visually unacceptable when the number of gray levels is small. To correct this, a dithering algorithm can be applied. Such an algorithm spreads the quantization error to multiple pixels. An example of a dithering algorithm is the Floyd-Steinberg algorithm:

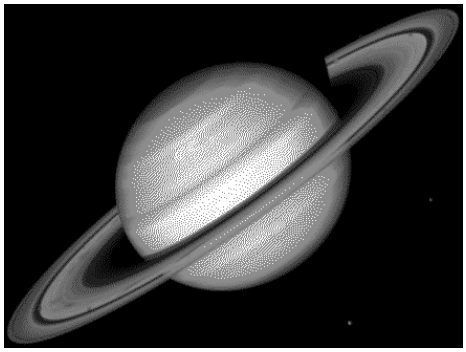
```
for i from [0, rows-1]
   for j from [0, cols-1]
      oldpixel := img(i,j)
      newpixel := find_closest_histogram_maximum(oldpixel)
      img(i,j) := newpixel
      error := oldpixel - newpixel
      if (i,j+1) inside img then
         img(i,j+1)   := img(i,j+1)   + 7*error/16
      if (i+1,j-1) inside img then
         img(i+1,j-1) := img(i+1,j-1) + 3*error/16
      if (i+1,j) inside img then
         img(i+1,j)   := img(i+1,j)   + 5*error/16
      if (i+1,j+1) inside img then
         img(i+1,j+1) := img(i+1,j+1) + error/16
```

This algorithm computes the quantization error and spreads it to the neighboring pixels according to the following fraction matrix (**X** = current pixel's location):

0	0	0
0	**X**	7/16
3/16	5/16	1/16



a) b)



c)

Fig. 3.3 a) The initial image; b) The obtained multilevel thresholded image; c) Dithering on the initial image using the Floyd-Steinberg algorithm

3.5. Implementation details

3.5.1. Displaying the histogram as an image

The histogram can be viewed as an image by making a bar plot. For each gray level draw a bar with height proportional to the number of appearances. The function below **`showHistogram`** plots a histogram (available in the **`OpenCVApplication`** framework). You need to provide the computed histogram, the number of bins, and the height of the desired output image. Bars/lines are automatically rescaled to fit the image but they remain proportional to the histogram values.

```
void showHistogram(const string& name, int* hist, const int  hist_cols,
                   const int hist_height) {
     Mat imgHist(hist_height, hist_cols, CV_8UC3, CV_RGB(255, 255, 255));
                                                 // constructs a white image
     //computes histogram maximum
     int max_hist = 0;
     for (int i = 0; i<hist_cols; i++)
     if (hist[i] > max_hist)
          max_hist = hist[i];
     double scale = 1.0;
     scale = (double)hist_height / max_hist;
     int baseline = hist_height - 1;
     for (int x = 0; x < hist_cols; x++) {
          Point p1 = Point(x, baseline);
          Point p2 = Point(x, baseline - cvRound(hist[x] * scale));
          line(imgHist, p1, p2, CV_RGB(255, 0, 255)); // histogram bins
                                                      // colored in magenta
     }
     imshow(name, imgHist);
}
```

3.5.2. Histogram with custom number of bins

The image histogram can be computed using a custom number of bins $m \leq 256$. This entails dividing the range 0-255 into m equal parts, then counting all the gray levels falling into each of the m bins or buckets. Such a representation is useful since it is lower dimensional.

3.6. Practical work

1. Compute the histogram for a given grayscale image (in an array of integers having dimension 256).
2. Compute the PDF (in an array of floats of dimension 256).
3. Display the computed histogram using the provided function.
4. Compute the histogram for a given number of bins $m \leq 256$.
5. Implement the multilevel thresholding algorithm from section 3.3.
6. Enhance the multilevel thresholding algorithm using the Floyd-Steinberg dithering from section 3.4.
7. Perform multilevel thresholding on a color image by applying the procedure from 3.3 on the Hue channel from the HSV color-space representation of the image. Modify only the Hue values, keeping the S and V channels unchanged or setting them to their maximum possible value. Transform the result back to RGB color-space for viewing.
8. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms.**

3.7. Bibliography

[1] R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.
[2] Floyd-Steinberg algorithm, http://en.wikipedia.org/wiki/Floyd-Steinberg_dithering

4. Geometrical features of binary objects

4.1. Introduction

This lab work presents some important geometric properties of binary images and the algorithms used for computing them. The properties described are the area, the center of mass, the elongation axis, the perimeter, the thinness ratio, the aspect ratio and the projections of the binary image.

4.2. Theoretical considerations

After applying segmentation and labeling algorithms, we obtain a new image where each object can be referred separately.

An object 'i' in the image is described by the function:

$$I_i(r,c) = \begin{cases} 1, & \text{if } I(r,c) \in \text{object labeled 'i'} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{where } r \in [0...Height-1] \text{ and } c \in [0...Width-1]$$

The geometric properties of the objects can be classified into two categories:

- position and orientation properties: the center of mass, the area, the perimeter, the elongation axis
- shape properties: aspect ratio, thinness ratio, Euler's number, the projections, the Feret diameters of the objects

4.2.1. Area

$$A_i = \sum_{r=0}^{H-1}\sum_{c=0}^{W-1} I_i(r,c)$$

The area A_i is measured in pixels and it indicates the relative size of the object.

4.2.2. The center of mass

$$\overline{r_i} = \frac{1}{A_i}\sum_{r=0}^{H-1}\sum_{c=0}^{W-1} rI_i(r,c)$$

$$\overline{c_i} = \frac{1}{A_i}\sum_{r=0}^{H-1}\sum_{c=0}^{W-1} cI_i(r,c)$$

The equations above correspond to the row and column where the center of mass is located. This attribute helps us locate the object in a bi-dimensional image.

4.2.3. The axis of elongation (the axis of least second order moment)

$$\tan(2\varphi_i) = \frac{2\sum_{r=0}^{H-1}\sum_{c=0}^{W-1}(r-\overline{r_i})(c-\overline{c_i})I_i(r,c)}{\sum_{r=0}^{H-1}\sum_{c=0}^{W-1}(c-\overline{c_i})^2 I_i(r,c) - \sum_{r=0}^{H-1}\sum_{c=0}^{W-1}(r-\overline{r_i})^2 I_i(r,c)}$$

If both the nominator and the denominator of the above equation are equal to zero, then the object has a circular symmetry, and any line that passes through the center of mass is a symmetry axis.

For finding the direction of the line (the angle) one must apply the arctangent function. The arctangent is defined on the interval (-∞, +∞) and it takes values in the interval (-π/2, π/2). The evaluation of the arctangent becomes unstable when the denominator of the fraction tends to zero.

The signs of the numerator and of the denominator are important for determining the right quadrant in which the result lays. The arctangent function does not make the difference between directions that are opposed. For this reason, the usage of the function "atan2" is suggested. The "atan2" function has as arguments the numerator and the denominator of such fraction, and it returns a result in the interval (-π, π).

The axis of elongation gives information about how the object is positioned in the field of view, more exactly, its orientation. The axis corresponds to the direction in which the object (seen as a plane surface of constant width) can rotate most easily (has a minimum kinetic moment).

After the φ_i angle is found, the correctness of the resulted value can be validated by drawing the axis of elongation. The axis of elongation will correspond to the line that pass through the center of mass and determines the φ_i angle with Ox axis.

4.2.4. The perimeter

The perimeter of the object helps us determine the position of the object in space and it also gives information about the shape of the object. The perimeter can be computed by counting the number of pixels on the contour (pixels of value 1 and having at least one neighbor pixel of value 0).

A first approach to contour detection is the scanning of the image, line by line and counting the number of pixels in the object that satisfy the condition mentioned above. A main disadvantage of this method is that we cannot distinguish the exterior contour from the interior contours (if they exist they are generated by the holes in the object). As the pixels of digital images represent distributions on a rectangular raster, the length of curves and oblique lines in the image cannot be correctly estimated by counting the pixels. A first correction is given by the multiplication by π/4 of the perimeter that resulted in the previous algorithm. There are other methods for length correction. These methods take into account the type of neighborhood used (4 neighbors, 8 neighbors etc.).

Another method for detecting the contour of an object involves the usage of an existing algorithm for edge detection, the thinning of the edges until they become 1 pixel thick and in the end the counting of the resulted edge pixels.

Methods of type "chain-codes" represent complex methods for contour detection and offer a high accuracy.

4.2.5. The thinness ratio (circularity)

$$T = 4\pi\left(\frac{A}{P^2}\right)$$

The function above has the maximum value equal to 1, and for this value we obtain a circle. The thinness ratio is used for determining how "round" an object is. If the value of T is close to 1, the object tends to be round.

The value of the thinness ratio also offers information on how regular an object is. The objects that have a regular contour have a greater value of T than the objects of irregular contours. The value 1/T is called irregularity factor of the object (or compactness factor).

4.2.6. The aspect ratio

This property is found by scanning the image and keeping the minimum and maximum values of the lines and columns that form the rectangle circumscribed to the object.

$$R = \frac{c_{\max} - c_{\min} + 1}{r_{\max} - r_{\min} + 1}$$

4.2.7. The projections of the binary object

The projections give information about the shape of the object. The horizontal projection equals the sum of pixels computed on each line of the image, and the vertical projection is given by the sum of the pixels on the columns.

$$h_i(r) = \sum_{c=0}^{W-1} I_i(r,c) \qquad v_i(c) = \sum_{r=0}^{H-1} I_i(r,c)$$

The projections are used in applications of text recognition in which the interest object can be normalized.

4.3. Implementation details

In order to distinguish between the various objects present in an image, we will suppose that each one of them is painted using a different color. These colors may be the result of a previous labeling step, or may be generated manually (see Fig. 4.1).

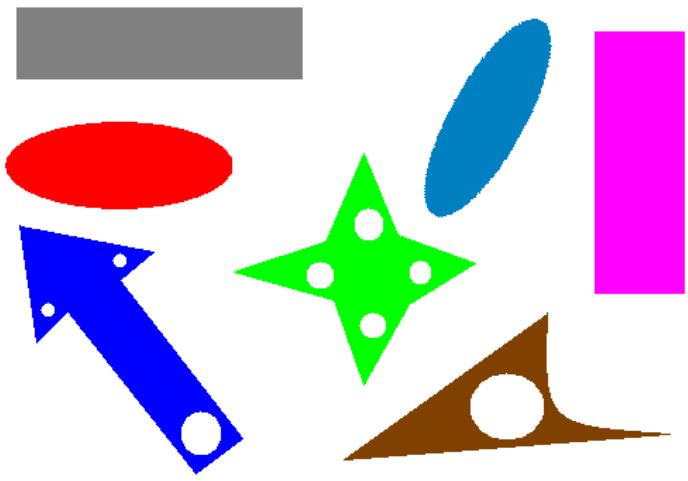


Fig. 4.1 Example of a labeled image on which the described algorithms could be tested

There are various approaches for implementing the geometrical properties extractors:

4.3.1. Compute the geometrical features for all objects in an image at once

For each object, the compound pixels are selected based on the object unique label (color) and the corresponding geometrical features are computed. This procedure is applied to each object from the input labeled image.

4.3.2. Compute the geometrical features for a specific object selected with the mouse

The user should position the mouse pointer over a pixel belonging to the desired object and *click* on it. In response to this action, the geometrical features of the desired object should be computed and displayed in the standard output.

In order to add an event *handler*, we will use the `setMouseCallback` function from OpenCV which has the role to set a *handler* for the mouse in a specific window.

```
void setMouseCallback(const string& winname, MouseCallback onMouse, void* userdata=0)
```

winname – window title,
onMouse – *callback* function name that is called when a mouse event occurs on the **winname** window
userdata – optional parameter that may be passed to the *callback* function.

The computation of the desired features will be implemented in **onMouse** function.

```
void onMouse (int event, int x, int y, int flags, void* param)
```

event – is the mouse event and can take the following values:
- EVENT_MOUSEMOVE
- EVENT_LBUTTONDOWN
- EVENT_RBUTTONDOWN
- EVENT_MBUTTONDOWN
- EVENT_LBUTTONUP
- EVENT_RBUTTONUP
- EVENT_MBUTTONUP
- EVENT_LBUTTONDBLCLK
- EVENT_RBUTTONDBLCLK
- EVENT_MBUTTONDBLCLK

x, y – are the x and y coordinates where the event occurred,
flags – specific condition whenever a mouse event occurs,
param – corresponds to the userdata pointer passed through **setMouseCallback** function.

In *OpenCVApplication* framework, an example of event handler is presented in the **testMouseClick()** function.

In order to draw the elongation axis, use the **line** function from OpenCV to draw the line:

```
void line( Mat img,  Point pStart, Point pEnd, Scalar color, int thickness )
```

img – image where the line segment is drawn
pStart, **pEnd** – the two points that define the line segment
color – line color
thickness – line thickness

4.4. **Practical work**

1. For a specific object in a labeled image selected by a mouse *click*, compute the object's area, center of mass, axis of elongation, perimeter, thinness ratio and aspect ratio.
    a. Display the results in the standard output
    b. In a separate image (source image clone):
        - Draw the contour points of the selected object
        - Display the center of mass of the selected object
        - Display the axis of elongation of the selected object by using the *line* function from OpenCV.
    c. Compute and display the projections of the selected object in a separate image (source image clone).
2. Create a new processing function which takes as input a labeled image and keeps in the output image only the objects that:
    a. have their *area* < *TH_area*
    b. have a specific orientation *phi*, where *phi_LOW* < *phi* < *phi_HIGH*

    where *TH_area*, *phi_LOW*, *phi_HIGH* are given by the user.
3. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

4.5. Bibliography

[1] Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8.

5 Connected-component labeling

5.1 Introduction

This laboratory work presents algorithms for labeling distinct objects from a black and white image. As a result, every object will be assigned a unique number. This number, or label, can be used to process the objects separately.

5.2 Theoretical foundations

We will present several algorithms for labeling. The input for the algorithms is a binary image. The output is a label matrix which has the same dimensions as the input image. It should be capable of storing sufficiently large label values.

In the input binary image the objects are represented as connected components of color black (0), the background is assigned the color white (255). To define what a connected component is, we need to introduce different neighborhood types.

The 4-neighborhood of a position (i,j) is defined to be the set of positions:

$$N_4(i,j)=\{(i-1,j),\ (i,j-1),\ (i+1,j),\ (i,j+1)\},$$

i.e. the upper, left, lower and right neighbors.

The 8-neighborhood consists of all neighboring positions differing by at most 1:

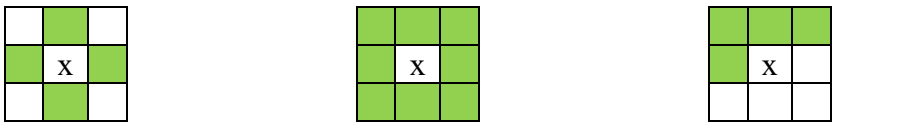
$$N_8(i,j) = \{(k,l) \mid |k-i|\leq 1,\ |l-j|\leq 1,\ (k,l)\neq(i,j)\},$$

so it includes the 4-neighborhood and the neighbors situated diagonally.

When traversing the image in a particular direction we can define the previous neighbors with regard to this traversal. The previous neighbors for normal top-down, left-right traversal for a position (i,j) is:

$$Np(i,j)=\{(i,j-1),\ (i-1,j-1),\ (i-1,j),\ (i-1,j+1)\}.$$

The presented definitions are illustrated below.



a) 4-neighborhood b) 8-neighborhood c) previous neighbors

We will define a graph generated by a binary image. The set of vertices is formed by all object pixel positions. The neighboring object pixels determine the edges of the graph. Two positions are neighboring if one is part of the other's neighborhood. We will use N_4 and N_8 so the generated graph is undirected. In this setting a connected component is a set of vertices in which for each pair there is path from vertex 1 to vertex 2.

5.2.1 Algorithm 1 - Breadth first traversal

We start the description with a straightforward method for labeling which relies on breadth first traversal of the graph defined on the image. The first step is to initialize the label matrix to zeroes which indicates that everything is unlabeled. Then algorithm searches for an unlabeled object pixel. If it finds one, it gives it a new label and propagates the label to its neighbors. We repeat this until all object pixels are given a label. In the following we present the steps of the algorithm:

```
label = 0
labels = zeros(height, width)   //height x width matrix with 0
for i = 0:height-1
    for j = 0:width-1
        if img(i,j)==0 and labels(i,j)==0
            label++
            Q = queue()
            labels(i,j) = label
            Q.push( (i,j) )
            while Q not empty
                q = Q.pop()
                for each neighbor in N8(q)
                    if img(neighbor)==0 and labels(neighbor)==0
                        labels(neighbor) = label
                        Q.push( neighbor )
```

Algorithm 1 - Breadth first traversal for connected-component labeling

The queue data structure maintains the list of points that need to be labeled. Since the queue uses a FIFO policy we obtain a breadth first traversal. We mark visited nodes by setting the label for their position. Changing the data structure to a stack would result in a depth first traversal of the image graph.

5.2.2 Algorithm 2 - Two-pass with equivalence classes

Labeling can be achieved by performing two linear passes over the image and some additional processing on a smaller graph. This approach uses less memory. In the previous algorithm we needed the store a list of points. If there is a large connected component, the size of the list is roughly the same as the size of the image.

The current algorithm performs the first pass and labels all object pixels with initial labels. For each pixel we need to consider the previously visited and labeled pixels, so we use the N_p neighborhood defined above. After inspecting the labels of the previous positions we can have the following cases:

- If no previous neighbor was labeled, we create a new label.
- Otherwise, we take the smallest label, called *x*, from the neighbors. Afterwards, we mark each neighboring label *y* as equivalent to *x*.

We assign the label found in the previous step to the current position and continue. After the first pas we have assigned initial labels to each position. However, several labels are equivalent so we need to assign new ones to each equivalence class.

The equivalence relations define an undirected graph on the labels. This graph is usually much smaller than the original graph defined on the whole image. It consists of nodes labeled from 1 to the maximum label value. The edges of the graph indicate the equivalence relations. We can apply Algorithm 1 on this smaller graph to obtain a new list of labels. All labels equivalent to label 1 get relabeled to 1. The next connected component not equivalent to 1 gets relabeled to 2, and so on. A new pass over the labels matrix is necessary to update the labels.

```
label = 0
labels = zeros(height, width)
vector<vector<int>> edges
for i = 0:height-1
    for j = 0:width-1
         if img(i,j)==0 and labels(i,j)==0
              L = vector()
              for each neighbor in Np(i,j)
                   if labels(neighbor)>0
                        L.push_back(labels(neighbor))
              if L.size() == 0        //assign new label
                   label++
                   labels(i,j) = label
              else                    //assign smallest neighbor
                   x = min(L)
                   labels(i,j) = x
                   for each y from L
                     if (y <> x)
                       edges[x].push_back(y)
                       edges[y].push_back(x)

newlabel = 0
newlabels = zeros(label+1)   //an array of zeroes of length label+1
for i = 1:label
    if newlabels[i]==0
         newlabel++
         Q = queue()
         newlabels[i] = newlabel
         Q.push( i )
         while Q not empty
              x = Q.pop()
              for each y in edges[x]
                   if newlabels[y] == 0
                        newlabels[y] = newlabel
                        Q.push( y )

for i = 0:height-1
    for j = 0:width-1
         labels(i,j) = newlabels[labels(i,j)]
```

Algorithm 2 - Two-pass connected-component labeling

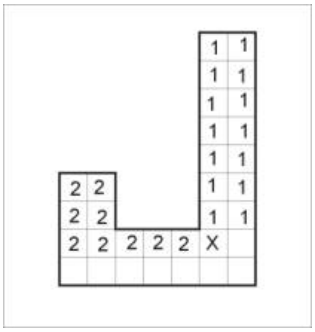


Fig. 5.1 Example of a case when the previous neighbors have different labels. Labels 1 and 2 are marked as equivalent at this step.

5.3 Implementation details

The following code illustrates how to visit the 4-neighborhood of a pixel. It can be easily modified to 8-neighborhood, or to only consider the upper and left neighbors of the pixel.

```
int di[4] = {-1,0,1,0};
int dj[4] = {0,-1,0,1};
uchar neighbors[4];
for(int k=0; k<4; k++)
    neighbors[k] = img.at<uchar>(i+di[k], j+dj[k]);
```

Pay attention to stay within the bounds of the image.

Store the labels in a matrix capable of holding the maximum number of labels:

$2^8 = 256$ - **`uchar (CV_8UC1)`**
$2^{16} = 65536$ - **`short (CV_16SC1)`**
$2^{32} \sim 2.1e9$ - **`int (CV_32SC1)`**

You can use the **`std::stack`** and **`std::queue`** container for storing points for Algorithm 1 to obtain DFS and BFS traversal, respectively. Sample code for initializing and performing operations on a queue:

```
#include <queue>
queue<Point> Q;
Q.push ({j,i}); // add element onto the top of the queue (newest)
Point p = Q.front(); // access the element from the bottom
                     // of the queue (oldest element)
Q.pop(); // removes the element from the top of the queue (newest)
```

The equivalence relations that define the edges of the smaller graph can be stored using adjacency lists in a **`vector<vector<int>>`**. Sample code to initialize and insert edges:

```
vector<vector<int>> edges;
//ensure that edges has the proper size
edges.resize(label+1);
//if u is equivalent to v
edges[u].push_back(v);
edges[v].push_back(u);
```

To display the label matrix as a color image you need to generate a random color for each label. You should use the default random generator from the standard library. It is better than a call to **`rand()%256`**.

```
#include <random>
default_random_engine gen;
uniform_int_distribution<int> d(0,255);
uchar x = d(gen);
```

5.4 Labeling examples

Fig. 5.2 Labeling examples

5.5 Practical Work

1. Implement the breadth first traversal component labeling algorithm (Algorithm 1). You should be able to easily switch between the neighborhood types of 4 and 8.
2. Implement a function which generates a color image from a label matrix by assigning a random color to each label. Display the results.
3. Implement the two-pass component labeling algorithm. Display the intermediate results you get after the first pass over the image. Compare this to the final results and to the previous algorithm.
4. Optionally, visualize the process of labeling by showing intermediate results and pausing after each step to illustrate the order of traversal a selected algorithm.
5. Optionally, change the queue to a stack to perform DFS traversal.
6. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

5.6 Bibliography

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8
[2]. Robert M. Haralick, Linda G. Shapiro, *Computer and Robot Vision*, Addison-Wesley Publishing Company, 1993.

6. Border Tracing Algorithm

6.1. Objectives

The purposes of this laboratory session are:

- to extract the objects' contours using a border tracing algorithm;
- to represent efficiently each extracted contour using chain codes;
- to take advantage of using chain codes in representing the objects' contours (border reconstruction, matching, merging etc.).

6.2. Theoretical Background

6.2.1. Border Tracing Algorithm

The border tracing algorithm is used to extract the contours of the objects (regions) from an image. When applying this algorithm it is assumed that the image with regions is either binary or those regions have been previously labeled.

Algorithm's steps:

1. Search the image from top left until a pixel of a new region is found; this pixel P_0 is the starting pixel of the region border. Define a variable *dir* which stores the direction of the previous move along the border from the previous border element to the current border element. Assign
   (a) *dir* = 0 if the border is detected in 4-connectivity (Fig. 6.1a)
   (b) *dir* = 7 if the border is detected in 8-connectivity (Fig. 6.1b)
2. Search the 3x3 neighborhood of the current pixel in an anti-clockwise direction, beginning the neighborhood search at the pixel positioned in the direction
   (a) (*dir* + 3) *mod* 4 (Fig. 6.1c)
   (b) (*dir* + 7) *mod* 8 if *dir* is even (Fig. 6.1d)
   (*dir* + 6) *mod* 8 if *dir* is odd (Fig. 6.1e)
   The first pixel found with the same value as the current pixel is a new boundary element P_n. Update the *dir* value.
3. If the current boundary element P_n is equal to the second border element P_1 and if the previous border element P_{n-1} is equal to P_0, stop. Otherwise repeat step (2).
4. The detected border is represented by pixels $P_0 \ldots P_{n-2}$.

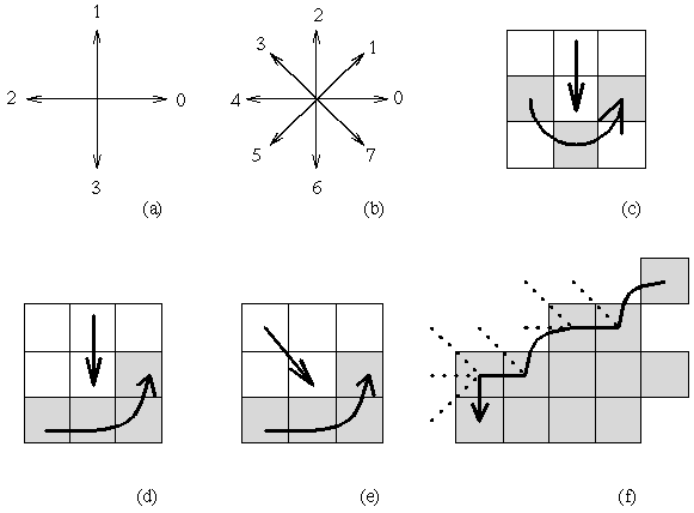


Fig. 6.1 (a) Direction notation, 4-connectivity, (b) 8-connectivity, (c) pixel neighborhood search sequence is 4-connectivity, (d),(e) search sequence in 8-connectivity, (f) boundary tracing in 8-connectivity (dashed lines show pixels tested during the border tracing).

Remarks:

- The above algorithm works for all regions larger than one pixel.
- Looking for the border of a single-pixel region is a trivial problem.
- This algorithm is able to find region borders but does not find borders of region holes.
- To search for the object's holes' borders as well, the border must be traced starting in each region or hole border element if this element has never been a member of any border previously traced.
- Note that if objects are of unit width, more conditions must be added.

6.2.2. Chain Codes Extraction

The *chain code* provides a storage-efficient representation for the boundary of an object in a binary image. The chain code representation incorporates such pertinent information as the length of the boundary of the encoded object, its area, and moments. Chain codes lend to efficient calculation of certain curve parameters. Additionally, chain codes are invertible in that an object can be reconstructed from its chain code representation.

The basic idea behind the chain code is that each boundary pixel of an object has an adjacent boundary pixel neighbor whose direction from the given boundary pixel can be specified by a unique number between 0 and 7 (8-connectivity neighborhood). Chain codes could also be defined using a 4-connectivity neighborhood. A 4-connecivity neighborhood chain codes example it is presented in Fig. 6.4.

In the following we discussion we use the 8-connectivity neighborhood. Given a pixel, consider its eight neighboring pixels. Each 8-neighbor can be assigned a number from 0 to 7 representing one of eight possible directions from the given pixel (see **Fig. 6.2**). This is done with the same orientation throughout the entire image.

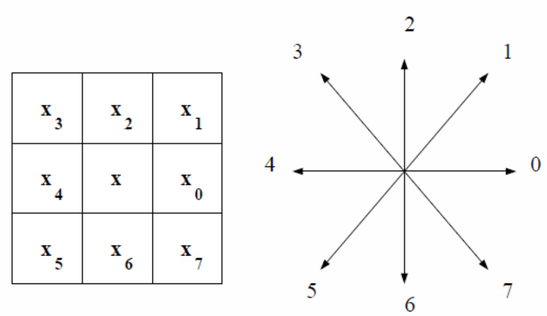


Fig. 6.2 The 8-neighborhood and the associated eight directions

The chain code for the boundary of a binary image is a sequence of integers $c=\{c_0, c_1, \dots, c_{n-1}\}$, having each c_i from the set $\{0,1, \dots ,7\}$ for i=0, 1, … , n-1. The number of elements in the sequence c is called the length of the chain code. The elements c_0 and c_{n-1} are called the *initial* and *terminal point* of the code, respectively. Starting at a given base point, the boundary of an object in a binary image can be traced out using the head-to-tail directions that the chain code provides.

Fig. 6.3 illustrates the process of tracing out the boundary of a triangle by following direction vectors. Suppose we choose the topmost left feature pixel of Fig. 6.3 as the base point (x=109, y=61) for the boundary encoding. The chain code for the boundary of the triangle is the following sequence of 245 codes: 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 5 6 7 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 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 0 0 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2.

Given the base point and the chain code, the boundary of the triangle can be completely reconstructed. The chain code is an efficient way of storing boundary information because it requires only three bits ($2^3 = 8$) to determine any one of the eight directions.

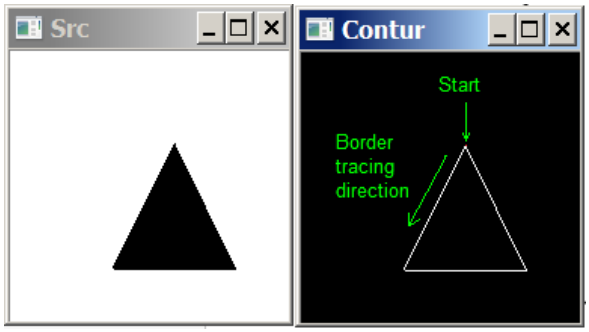


Fig. 6.3 Chain code directions with associated direction numbers

Chain codes may be made position-independent by ignoring the "start point". If they represent closed boundaries they may be "start point normalized" by choosing the start point so that the resulting sequence of direction codes forms an integer of minimum magnitude.

The "derivative" of the chain code is useful because it is invariant under boundary rotation. The derivative (really a first difference mod 4 or 8) is simply another sequence of numbers indicating the relative direction of chain code segments; the number of left hand turns of $\pi/2$ or $\pi/4$ needed to achieve the direction of the next chain segment. A *mod* 4 or *mod* 8 difference is called a chain code *derivative* (see Fig. 6.4!).

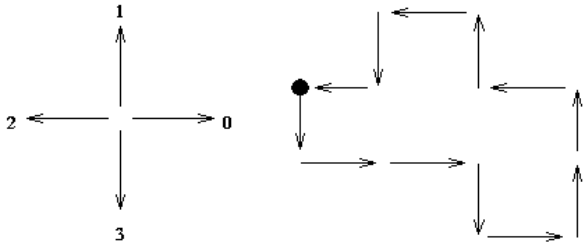


Fig. 6.4 Chain code in 4-connectivity and its derivative.

Code: 3, 0, 0, 3, 0, 1, 1, 2, 1, 2, 3, 2
Derivative: 1, 0, 3, 1, 1, 0, 1, 3, 1, 1, 3, 1

Chain codes properties:

- Chain codes describe an object by a sequence of unit-size (4-connectivity) line segments with a given orientation.
- The first element of such a sequence must bear information about its position to allow reconstruction of the region.
- Even codes {0, 2, 4, 6} correspond to horizontal and vertical directions; odd codes {1, 3, 5, 7} correspond to the diagonal directions.
- Each code can be considered as the angular direction, in multiples of 45 degrees that we must move to go from one contour pixel to the next.
- The absolute coordinates of the first contour pixel (e.g. top, leftmost) together with the chain code of the contour represent a complete description of the discrete region contour.
- When there is a change between two consecutive chain codes, then the contour has changed direction. This point is defined as a *corner*.

6.3. Practical Work

Using the *OpenCVApplication* framework and the laboratory's additional images and files:

1. Implement the border tracing algorithm and draw the object contour on an image having a single object.
2. Starting from the border tracing algorithm write the algorithm that builds the chain code and derivative chain code for an object. Compute and display (command line or output text file) both codes (chain code and derivative chain code) for an image with a single object.
3. Implement a function that reconstructs (draws) the border of an object over an image having as inputs the start point coordinates and the chain code in 8-neighborhood *(reconstruct.txt*). Load the image *gray_background.bmp* and apply the function that reconstructs the border. You should obtain the contour of the word "EXCELLENT" (having all the letters connected).
4. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

Additional info:

The test images with a single object have:

- 8 bits/pixel
- index 0 for object's pixels (black pixels)
- other index value for background pixels (white pixels)

The file *reconstruct.txt* is a text file having:

- on the first line the start point coordinates (row column) separated with a space;
- on the second line the number of chain codes;
- on the third line the chain codes (sequence of directions in 8-connectivity) separated with a space.

6.4. Bibliography

[1] Border Tracing – Digital Image Processing lectures, The University of Iowa, http://www.icaen.uiowa.edu/~dip/LECTURE/Segmentation2.html#tracing
[2] Contour Representations – Quantitative Imaging Group, Delft University http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Contour.html#Heading27
[3] G.X. Ritter, J.N. Wilson – Handbook of Computer Vision Algorithms in Image Algebra Second Edition – Chapter 10.4 Chain Code Extraction and Correlation, CRC Press, New York 2001.
[4] Chain Codes – Digital Image Processing lectures, The University of Iowa http://www.icaen.uiowa.edu/~dip/LECTURE/Shape2.html#chaincodes
[5] Representation of Two-Dimensional Geometric Structures, http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/LIB/bandb8_12.pdf

7. Morphological operations on binary images

7.1. Introduction

Morphological operations are affecting the form, structure or shape of an object. Usually, they are applied on binary images (black & white images – images with only 2 colors: *black* and *white*). They are used in pre- or post- processing (filtering, thinning, and pruning) or for getting a representation or description of the shape of objects/regions (boundaries, skeletons convex hulls).

7.2. Theoretical considerations

The two principal morphological operations are *dilation* and *erosion* [1]. Dilation allows objects to expand, thus potentially filling in small holes and connecting disjoint objects. Erosion shrinks objects by etching away (eroding) their boundaries. These operations can be customized for an application by the proper selection of the structuring element, which determines exactly how the objects will be dilated or eroded.

Notations:

Object / foreground pixels: pixels of interest (on which the morphological operations are applied)

Background pixels: the complementary set of the object / foreground pixels

7.2.1. The dilation

The *dilation* process is performed by laying the structuring element **B** on the image **A** and sliding it across the image from left to right, top to bottom. The result image has the same size as image **A**. Its pixels are initialized to 'background'. The operation is non-linear and can be described as follows:

1. If the origin of the structuring element coincides with a 'background' pixel in the image **A**, there is no change; move to the next pixel.
2. If the origin of the structuring element coincides with an 'object' pixel in the image, label all pixels covered by the structuring element as 'object' pixels in the result image.

Notation:

$A \oplus B$

The structuring is a compound of 'object' pixels organized in any shape. Typical shapes are presented below:

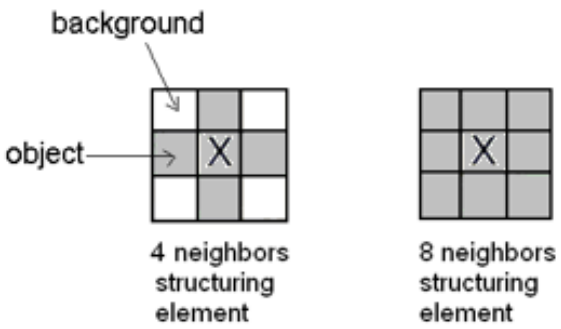


Fig. 7.1 Typical shapes of the structuring elements **B**.

An example is shown in Fig. 7.2. Note that with a dilation operation, all the 'object' pixels in the original image will be retained, any boundaries will be expanded, and small holes will be filled.

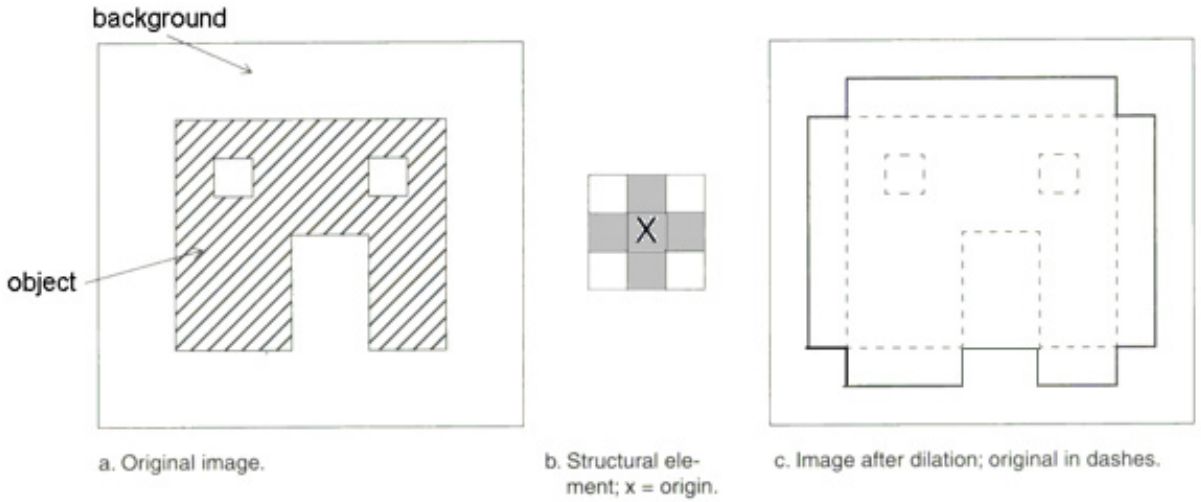


Fig. 7.2 Illustration of the dilatation process.

a. b.

Fig. 7.3 Example of the dilation (object = black / background = white): a. Original image **A**; b. The result image: $\mathbf{A} \oplus \mathbf{B}$.

7.2.2. The erosion

The *erosion* process is similar to dilation, but the effect is somehow opposite. The result image, of same size as image A, is initialized to 'background'. The same image scanning techniques is adopted as for dilation. The structuring element slides its position over pixels from image **A**. Each new position applies the following steps:

1. If the structuring element covers only 'object' points, its corresponding pixel in the result image is set as 'object' pixel.
2. If the structuring element covers any 'background' point, the pixel in the result image keeps its 'background' label.

Notation:

$A \ominus B$

In Fig. 7.4 the only remaining pixels coincide to the origin of the structuring element

for the case when it was completely contained by any existing object. Because the structuring element is 3 pixels wide, the 2-pixel-wide right leg of the image object was eroded away, but the 3-pixel-wide left leg retained some of its center pixels.

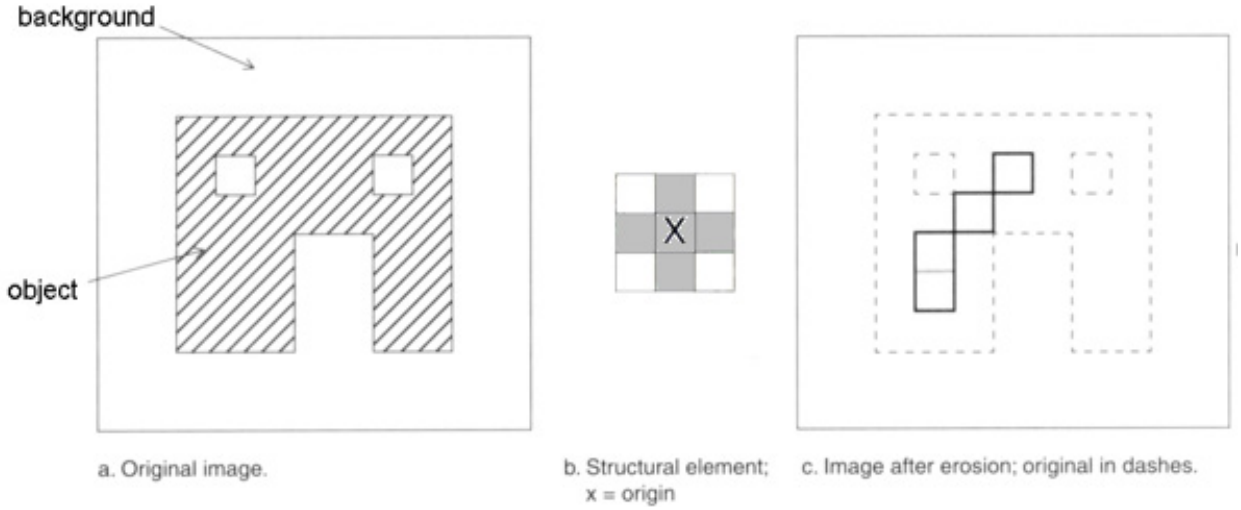
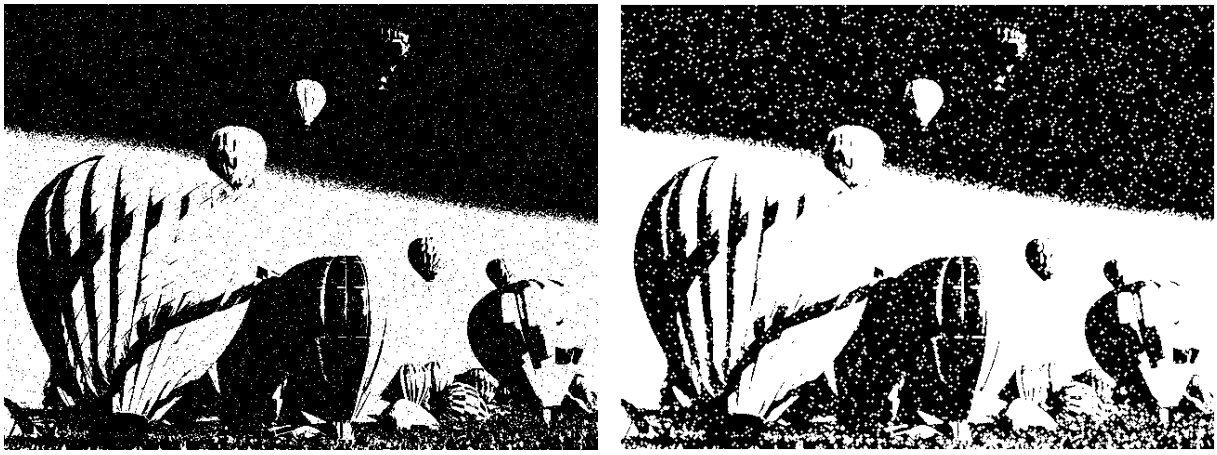


Fig. 7.4 Illustration of the erosion process.



a. b.

Fig. 7.5 Example of the erosion (object = black / background = white): a. Original image **A**; b. The result image: **A** Θ **B**.

7.2.3. Opening and closing

These two basic operations, dilation and erosion, can be combined into more complex sequences. The most useful of these for morphological filtering are called opening and closing [1]. *Opening* consists of an erosion followed by a dilation and can be used to eliminate all pixels in regions that are too small to contain the structuring element. In this case the structuring element is often called a probe, because it is probing the image looking for small objects to filter out of the image. See Fig. 7.6 for the illustration of the opening process.

Notation:

$$A \circ B = (A \ominus B) \oplus B$$

Closing consists of a dilation followed by erosion and can be used to fill in holes and small gaps. In Fig. 7.7 we see that the closing operation has the effect of filling in holes and closing gaps. Comparing the left and right images from Fig. 7.8, we see that the order of

operation is important. Closing and opening will generate different results even though both consist of erosion and dilation.

Notation:

$A \bullet B = (A \oplus B) \Theta B$

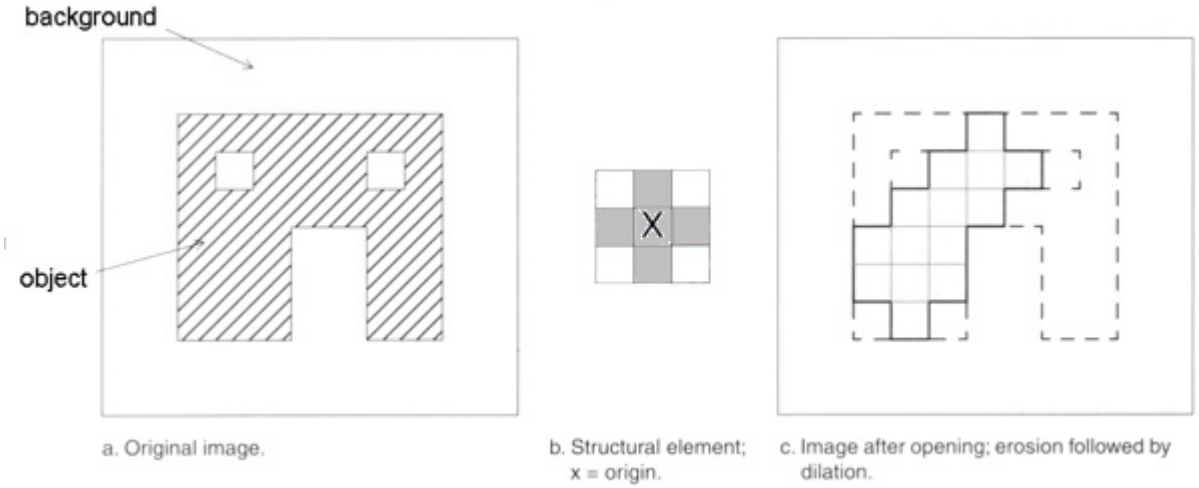


Fig. 7.6 Illustration of the opening process

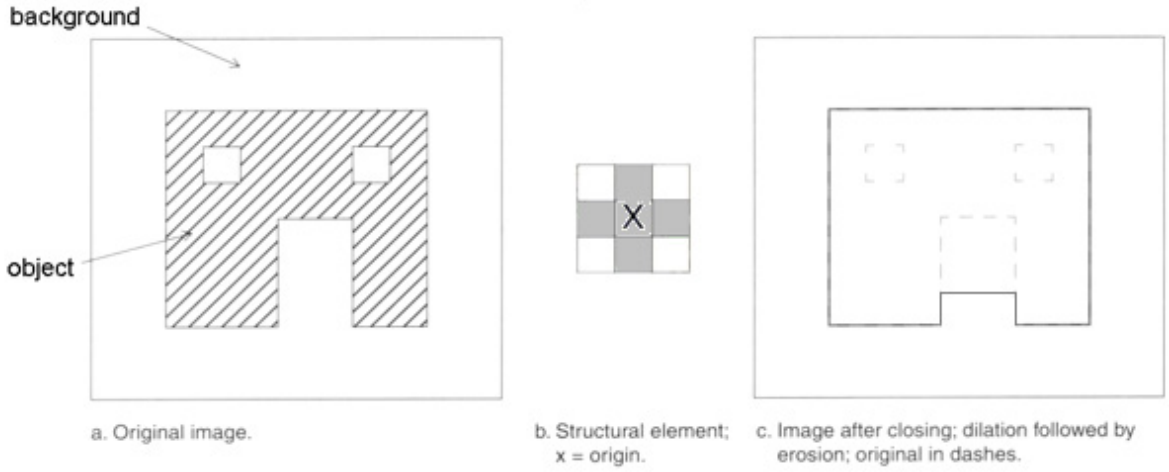


Fig. 7.7 Illustration of the closing process.

a. b.

Fig. 7.8 Results of the opening (a) and closing (b) operations applied on the original image from Fig. 7.5a (object = black / background = white).

7.2.4. Some basic morphological algorithms [2]

7.2.4.1. Boundary extraction

The boundary of a set A, denoted by $\beta(A)$, can be obtained by first eroding A by B and then performing the set differences between A and its erosion. That is,

$$\beta(A)=A-(A\Theta B)$$

where

B is a suitable structuring element.

'–' is the difference operation on sets (illustrated in Fig. 7.10)

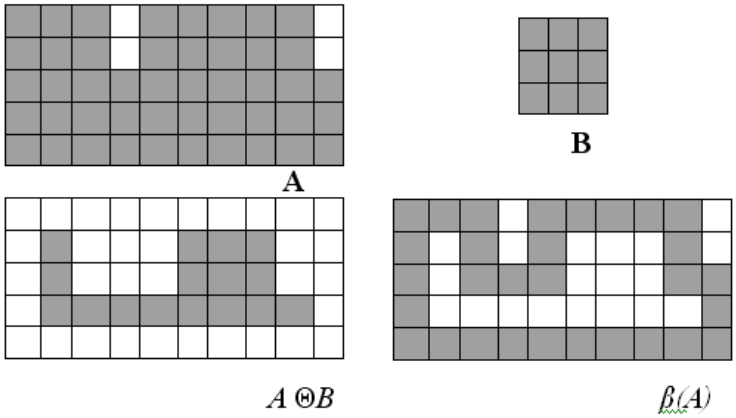


Fig. 7.9 Illustration of the boundary extraction algorithm.

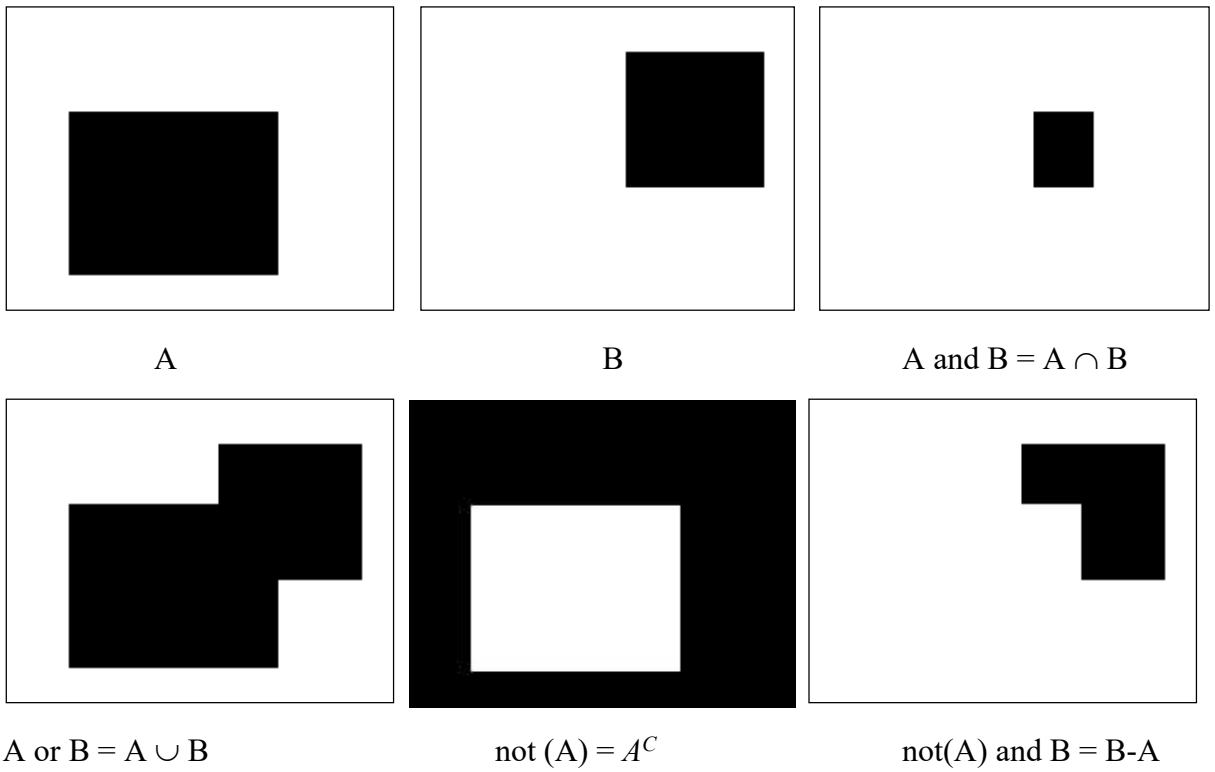


Fig. 7.10 Illustration of the main operations on sets.

7.2.4.2. Region filling

Next, we develop a simple algorithm for region filling based on set dilations, complementation, and intersections.

Beginning with a point p inside the boundary, the objective is to fill the entire region with 'object' pixels. If we adopt the convention that all non-boundary points are labeled 'background', then we assign the value/label 'object' to p to begin. The following procedure then fills the region with 'object' pixels:

$$X_k = (X_{k-1} \oplus B) \cap A^C \qquad k=1,2,3,\ldots$$

where

$X_0=p$,
B is the symmetric structuring element
$\cap$ - is the intersection operator (see Fig. 7.10)
A^C – is the complement of set A (see Fig. 7.10)

The algorithm terminates at iteration step k if $X_k=X_{k-1}$. The set union of X_k and A contains the filled set and its boundary.

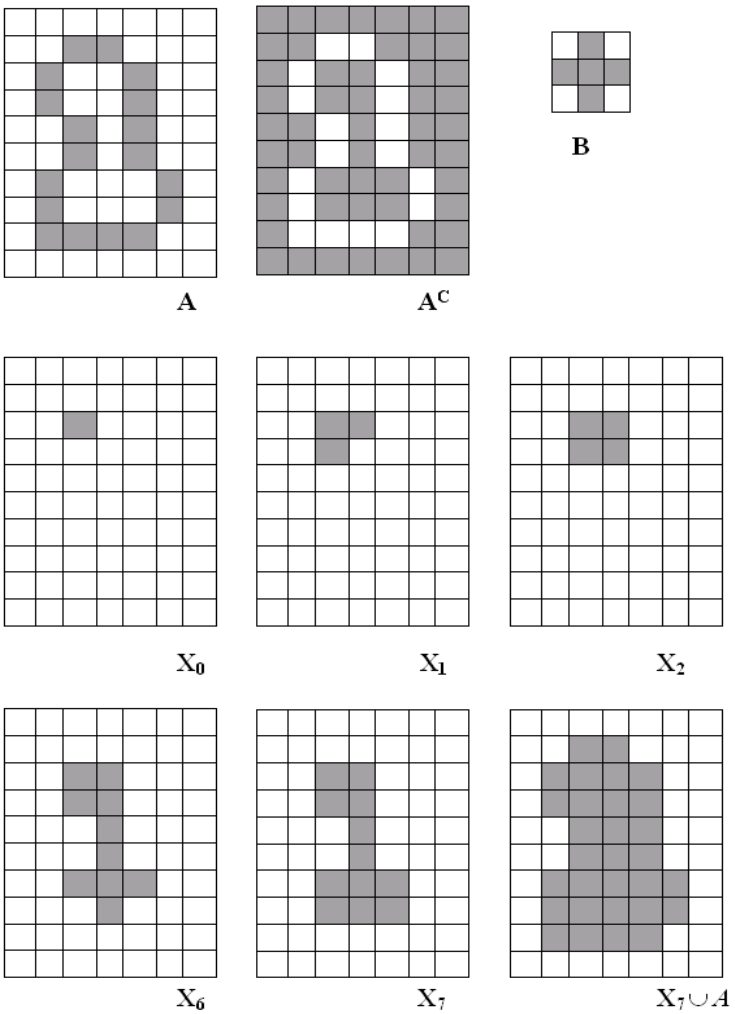


Fig. 7.11 Illustration of the region filling algorithm.

7.3. Implementation hints

7.3.1. Using a supplementary image buffer for chain processing

The results of the basic morphological operations (dilation and erosion) should be applied in the following manner:

Destination image = *Source image* (operator) *Structuring element*

The source image shouldn't be affected in any way!

For the implementation of the combined morphological operations (opening and closing) or of the repeated operations (for example: *n* consecutive erosions) in a single processing function a supplementary image buffer should be created and used.

7.4. Practical work

1. Add to the *OpenCVApplication* framework processing functions, which implement the basic morphological operations.
2. Add the facility to apply the morphological operations repeatedly (*n* times). Input the value of *n* from the command line. Remark the 'idempotency' property of the opening/closing operations (therefore there is no use to apply them repeatedly).
3. Implement the boundary extraction algorithm.
4. Implement the region filling algorithm.
5. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

7.5. References

[1] Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8.
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.

8. Statistical properties of grayscale images

8.1. Introduction

This laboratory work presents the main statistic features that characterize the distribution of intensity levels in a grayscale image or in an area / region of interest (ROI) of the image. These statistic features can be applied similarly to color images, on each color component.

The following notation will be used throughout this lab:

- L=255 highest intensity level
- $h(g)$ histogram function, counts the number of pixels with gray level g
- $M = H$ x W, number of pixels in the image
- $p(g) = h(g)/M$ gray level probability distribution function (PDF).

8.2. The mean value of intensity levels

The mean value of intensity levels is a measure of the mean intensity of the given image or of the region of interest. A dark image has a low mean value (**Fig. 8.1**a), and a bright image has a high mean value (**Fig. 8.1**b).

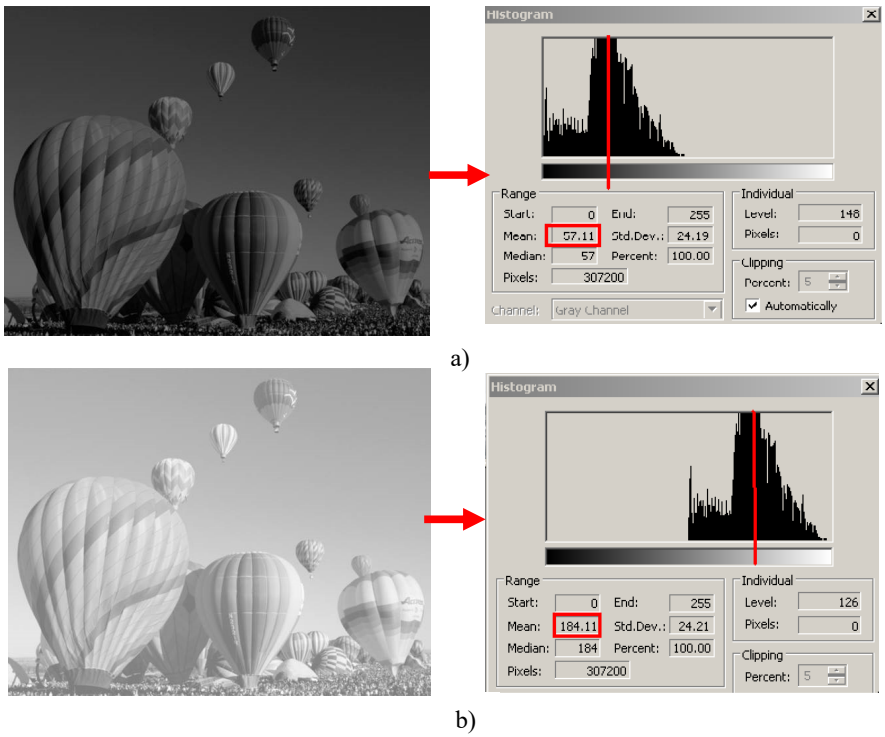


Fig. 8.1 The position of the histogram and the mean value of the intensity levels for a dark image (a) and a bright image (b).

The mean intensity value is computed as follows:

$$\bar{g} = \mu = \int_{-\infty}^{+\infty} g \cdot p(g) dg = \sum_{g=0}^{L} g \cdot p(g) = \frac{1}{M} \sum_{g=0}^{L} g \cdot h(g) \tag{8.1}$$

$$\bar{g} = \mu = \frac{1}{M}\sum_{i=0}^{H-1}\sum_{j=0}^{W-1} I(i,j) \tag{8.2}$$

8.3. The standard deviation of the intensity levels

The standard deviation of the intensity levels represents a measure of the contrast of an image (region of interest). It characterizes the dispersion (spreading) of the intensity levels with respect to the mean value. An image having a high contrast will have a large standard deviation (**Fig. 8.2**a – the histogram is spread on the entire range of intensity levels), and an image having a low contrast will be characterized by a small standard deviation (**Fig. 8.2**b – the histogram is restricted to some intensity levels located around the mean value).

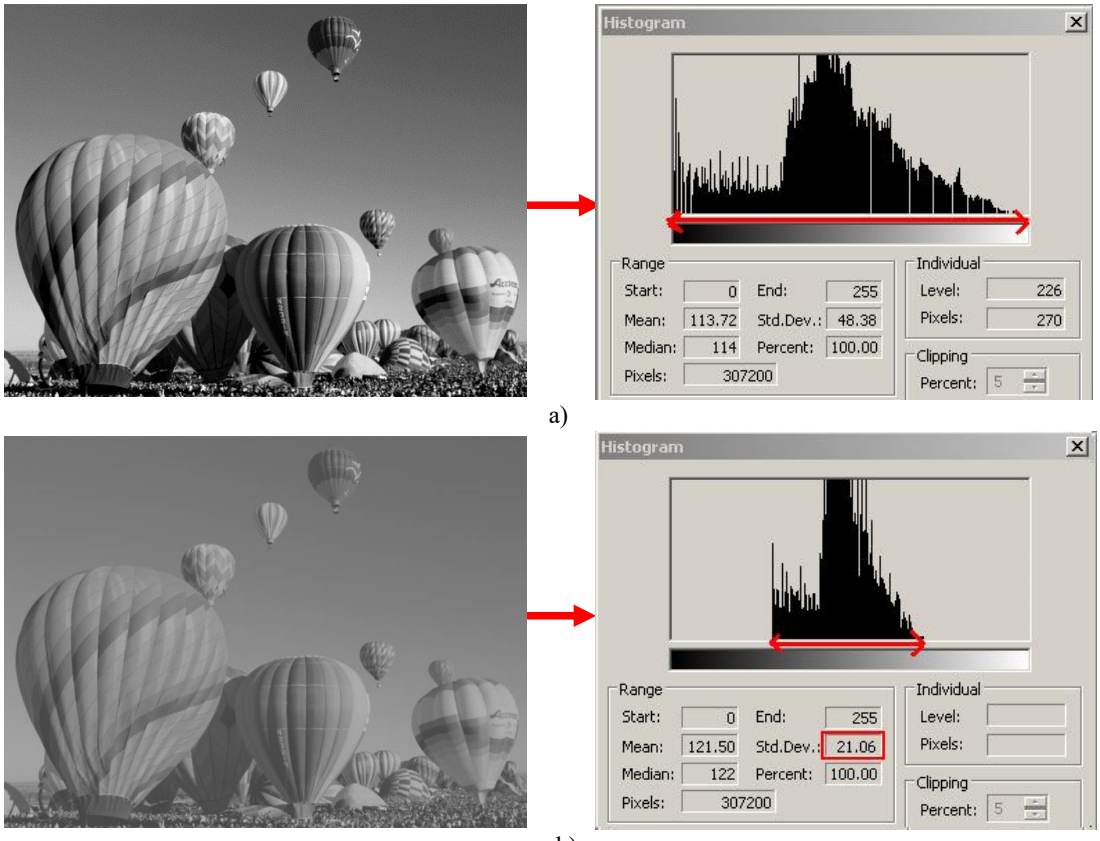


Fig. 8.2 The position of the histogram and of the standard deviation (2σ) of the intensity levels for an image of high contrast (a) and an image of low contrast (b).

The standard deviation of the intensity levels is given by:

$$\sigma = \sqrt{\sum_{g=0}^{L}(g-\mu)^2 \cdot p(g)} \tag{8.3}$$

$$\sigma = \sqrt{\frac{1}{M}\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}\left(I(i,j)-\mu\right)^2} \tag{8.4}$$

8.4. Cumulative Histogram

A cumulative histogram counts the cumulative number of pixel intensity values in all the bins up to the current bin.

$$C(g) = \sum_{j=0}^{g} h(j)$$

where h is the histogram of intensity levels and $g \in [0,255]$.

8.5. Basic global thresholding algorithm

This thresholding algorithm is suitable for grayscale images having a bimodal histogram. A bimodal histogram is characterized by two dominant modes, thus one threshold (T) is enough for image segmentation.

Algorithm

1. Initialization step:
   - Compute the image histogram h
   - Find the maximum intensity I_{max} and the minimum intensity I_{min} in the image
   - Take an initial value for threshold T:
     $T = (I_{min} + I_{max}) / 2$

2. Segment the image after threshold T by dividing the image pixels into 2 groups G_1 and G_2
   - Compute mean μ_{G_1} for the group of pixels which satisfy the condition $G_1 : I(i,j) \le T$
   - Compute mean μ_{G_2} for the group of pixels which satisfy the condition $G_2 : I(i,j) > T$

   Efficient implementation: compute the means μ_{G_1} and μ_{G_2} using the initial histogram

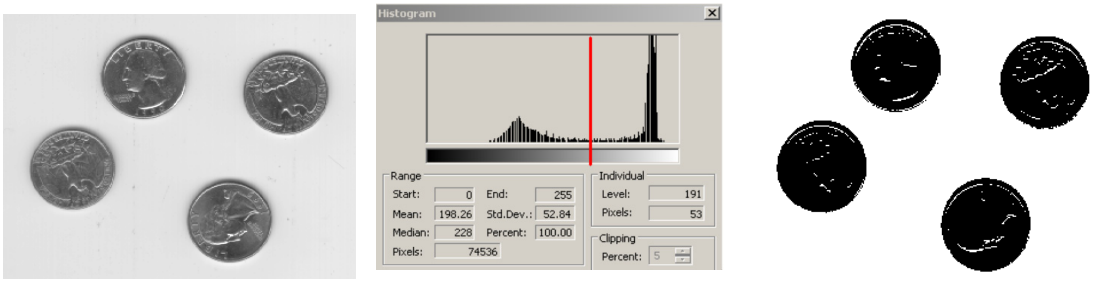
$$\mu_{G_1} = \frac{1}{N_1} \sum_{g=I_{min}}^{g=T} g \cdot h(g) \quad \text{where} \quad N_1 = \sum_{g=I_{min}}^{g=T} h(g)$$

$$\mu_{G_2} = \frac{1}{N_2} \sum_{g=T+1}^{g=I_{max}} g \cdot h(g) \quad \text{where} \quad N_2 = \sum_{g=T+1}^{g=I_{max}} h(g)$$

3. Update the threshold value: $T = (\mu_{G_1} + \mu_{G_2}) / 2$

4. Repeat 2-3 until $|T_k - T_{k-1}| < error$ (where *error* is a positive value)

5. Threshold the image using T



a. Original image b. Image histogram c. Binary image after thresholding with $T = 165$ ($error = 0.1$)

Fig. 8.3 Segmentation result using the computed threshold.

8.6. Analytical histogram transformation functions

In **Fig. 8.4** are shown some typical transformation functions of the intensity values, which can be expressed in an analytical form:

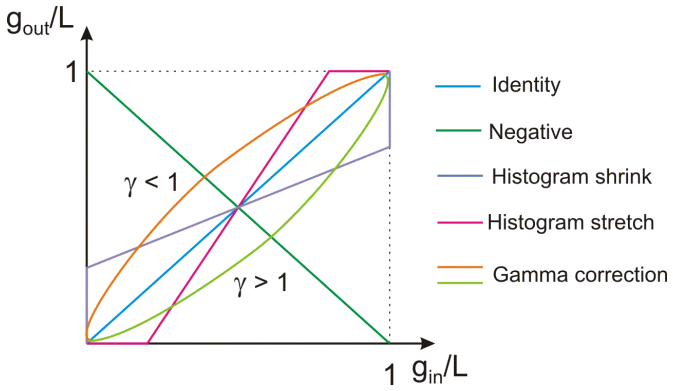


Fig. 8.4 Typical gray levels transformation functions.

8.6.1. **Identity function (no effect)**

$$g_{out} = g_{in} \tag{8.5}$$

8.6.2. **Image negative**

$$g_{out} = L - g_{in} = 255 - g_{in} \tag{8.6}$$

8.6.3.**Brightness changing (histogram slide)**

- A positive offset increases the brightness
- A negative offset decreases the brightness

$$g_{out} = g_{in} + offset \tag{8.7}$$

Attention: always the following checking will be done: 0 <= g_{out} <=255. If an overflow beyond these limits appears, output values will be truncated or scaled!!!

a. Original image b. Brightness *offset = 50* c. Brightness *offset = -50*

Fig. 8.5 Brightness change

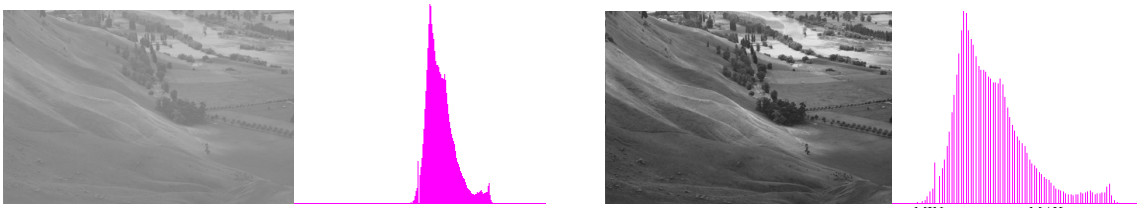
8.6.4. Histogram stretching / shrinking

- Remap pixel intensities from $[g_{in}^{MIN}, g_{in}^{MAX}]$ to $[g_{out}^{MIN}, g_{out}^{MAX}]$
- Histogram stretching increases the contrast
- Histogram shrinking decreases the contrast

$$g_{out} = g_{out}^{MIN} + (g_{in} - g_{in}^{MIN}) \frac{g_{out}^{MAX} - g_{out}^{MIN}}{g_{in}^{MAX} - g_{in}^{MIN}} \tag{8.8}$$

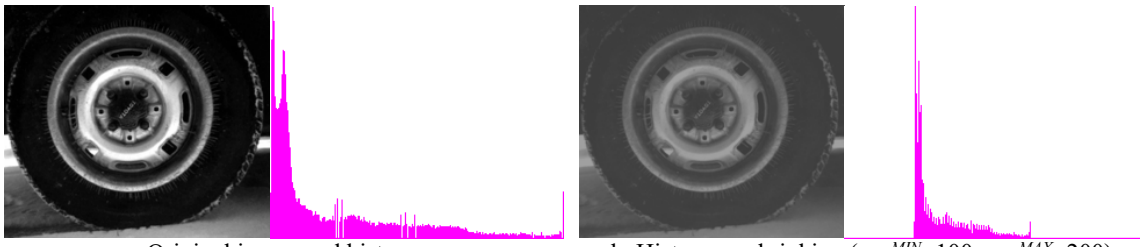
where:

$$\frac{g_{out}^{MAX} - g_{out}^{MIN}}{g_{in}^{MAX} - g_{in}^{MIN}} = \begin{cases} >1 \Rightarrow stretch \\ <1 \Rightarrow shrink \end{cases} \tag{8.9}$$



a. Original image and histogram b. Histogram stretching (g_{out}^{MIN}=10, g_{out}^{MAX}=250)

Fig. 8.6 Histogram stretching



a. Original image and histogram b. Histogram shrinking (g_{out}^{MIN}=100, g_{out}^{MAX}=200)

Fig. 8.7 Histogram shrinking

8.6.5. Gamma correction

- Can be used to correct the brightness of an image with a non-linear transformation

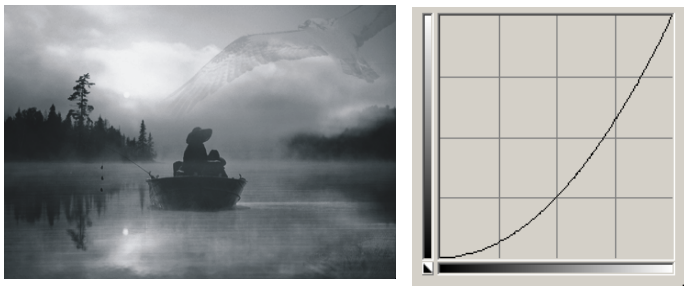
$$g_{out} = L\left(\frac{g_{in}}{L}\right)^{\gamma} \tag{8.10}$$

where: γ is a positive coefficient: < 1 (gamma encoding/compression) or > 1 (gamma decoding / decompression)

Attention: always check that: 0 <= g_{out} <= 255. If outside the domain, values should be saturated!!!

Input image

γ < 1: gamma encoding/compression



γ > 1: gamma decoding/expansion

Fig. 8.8 Results of gamma correction operations.

8.7. Histogram equalization

Histogram equalization is a transform which allows us to obtain an output image with a quasi-uniform histogram/PDF, regardless the shape of the histogram/PDF of the input image. For that purpose, the following transform will be used (see lecture notes for more details):

$$s_k = p_C(k) = \sum_{g=0}^{k} \frac{h(g)}{M}, \quad k = 0...L \tag{8.11}$$

where:

k – intensity level in input image,

s_k – corresponding normalized intensity level of the output image,

$p_C(k)$ – cumulative probability density function (CPDF) of the input image.

8.7.1. Histogram equalization algorithm

1. Compute the PDF of the input image as a vector p_r of 256 elements
2. Compute the CPDF of the input image (8.11), as a vector p_C of 256 elements.

3. Because the s_k values obtained from (8.11) are normalized intensity values, it is necessary to transform the normalized intensity values s_k back to un-normalized ones by multiplication with L (the highest intensity value: 255 for 8 bits/pixel images):

$$g_{out} = Ls_k = \frac{L}{M}\sum_{g=0}^{g_{in}} h(g) \quad , \quad k = g_{in} \tag{8.12}$$

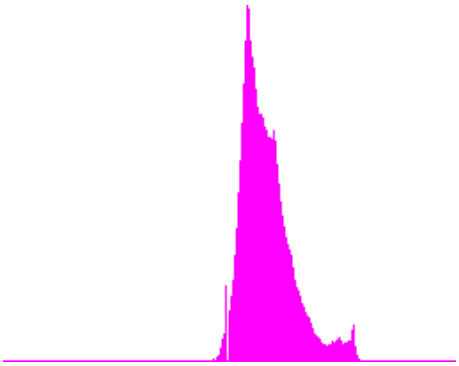
This transformation function can be written as an equivalence table (vector):

$$g_{out} = tab(g_{in}) = 255 \cdot p_C(g_{in}) \tag{8.13}$$

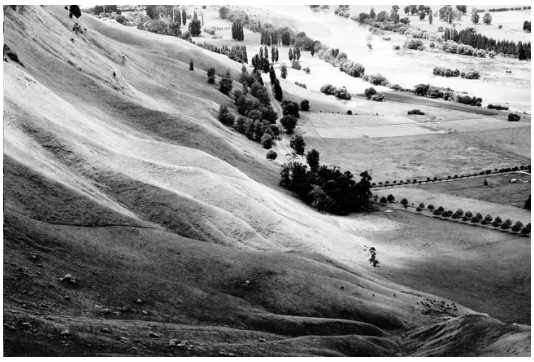
4. The intensity values of the output (equalized) image are computed using the equivalence table:

$$Dst(i,j) = tab\big(Src(i,j)\big) \tag{8.14}$$

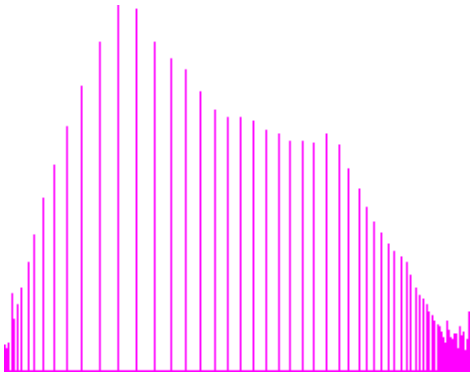
a. Original image



b. Original image histogram

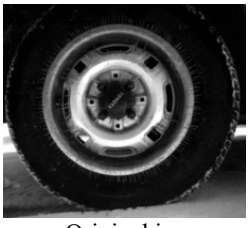


c. After histogram equalization



d. Equalized histogram

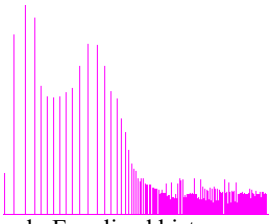
Fig. 8.9 Histogram equalization



e. Original image

f. After equalization

g. CPDF



h. Equalized histogram

Fig. 8.10 Histogram equalization

8.8. Practical work

1. Compute and display the mean and standard deviation, the histogram and the cumulative histogram of the image intensity levels. For the histogram use the *ShowHistogram* function from *OpenCVApplication* (see also Laboratory 3).
2. Implement the automatic threshold computation (section 8.5) and threshold the images according to this threshold. Display the threshold.
3. Implement the histogram transformation functions (section 8.6) for histogram stretching/shrinking, gamma correction, histogram slide. Input the limits $g_{out}^{MIN}, g_{out}^{MAX}$, the gamma coefficient and the brightness increase value from the console. After each processing display the histograms of the source and destination images.
4. Implement the histogram equalization algorithm (section 8.7). Display the histograms of the source and destination images.
5. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms.**

8.9. Bibliography

[1] R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.

9. Image filtering in the spatial and frequency domains

9.1. Introduction

In this laboratory, the convolution operator will be presented. This operator is used in the linear image filtering process applied in the spatial domain (in the image plane by directly manipulating the pixels) or in the frequency domain (applying a Fourier transform, filtering and then applying the inverse Fourier transform. Examples of such filters are low pass filters (for smoothing) and high pass filters (for edge enhancement).

9.2. The convolution process in the spatial domain

The convolution process implies the usage of a convolution mask/kernel H (usually with symmetric shape and size $w \times w$, with $w=2k+1$) which is applied on the source image according to (9.2).

$$I_D(i,j) = H * I_S \tag{9.1}$$

$$I_D(i,j) = \sum_{u=0}^{w-1} \sum_{v=0}^{w-1} H(u,v) \cdot I_S(i+u-k, j+v-k) \tag{9.2}$$

This implies the scanning of the source image I_S, pixel by pixel, ***ignoring the first and last k rows and columns*** (**Fig. 9.1**) and the computation of the intensity value in the current position (i, j) of the destination image I_D using (9.2). The convolution mask is positioned spatially with its central element over the current position (i, j).

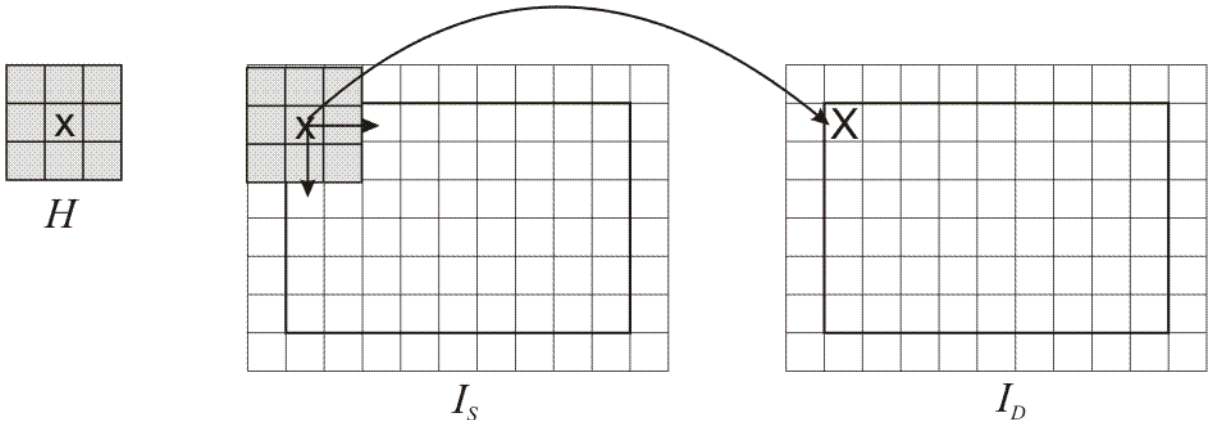


Fig. 9.1 Illustration of the convolution process.

The convolution kernels can have also non-symmetrical shapes (the central/reference element is not positioned in the center of symmetry). Convolution with such kernels is applied in a similar way, but such examples will not be presented in the current laboratory.

9.2.1. Low-pass filters

Low-pass filters are used for image smoothing and noise reduction (see the lecture material). Their effect is an averaging of the current pixel with the values of its neighbors, observable as a "blurring" of the output image (they allow to pass only the low frequencies of the image).

All elements of the kernels used for low-pass filtering have positive values. Therefore, a common practice used to scale the result in the intensity domain of the output image is to divide the result of the convolution with the sum of the elements of the kernel:

$$I_D(i,j) = \frac{1}{c}\sum_{u=0}^{w-1}\sum_{v=0}^{w-1} H(u,v)\cdot I_S(i+u-k, j+v-k) \tag{9.3}$$

where:

$$c = \sum_{u=0}^{w-1}\sum_{v=0}^{w-1} H(u,v) \tag{9.4}$$

Example kernel matrices:

Mean filter (3x3):

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \tag{9.5}$$

Gaussian filter (3x3):

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \tag{9.6}$$

a. **b.** **c.**

Fig. 9.2 a. Original image; b. Result obtained by applying a 3x3 mean filter. c. Result obtained by applying a 5x5 mean filter.

9.2.2. High-pass filters

These filters will highlight regions with step intensity variations, such as edges (will allow to pass the high frequencies).

The kernels used for edge detection have the sum of their elements equal to 0:

Laplace filters (edge detection) (3x3):

$$\begin{bmatrix}0 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 0\end{bmatrix} \tag{9.7}$$

or

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix} \tag{9.8}$$

High-pass filters (3x3):

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{9.9}$$

or

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{9.10}$$

a. b. c.

Fig. 9.3 a. The result of applying the Laplace edge detection filter (9.8) on the original image (Fig. 9.2a); b. The result of applying the Laplace edge detection filter (9.8) on the blurred image from Fig. 9.2b (previously filtered with the 3x3 mean filter); c. The result obtained by filtering the original image with the high-pass filter (9.10).

9.3. Image filtering in the frequency domain

The 1D discrete Fourier transform (DFT) of an array of N real or complex numbers is an array of N complex numbers, given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi jkn}{N}}, \quad k = \overline{0...N-1} \tag{9.11}$$

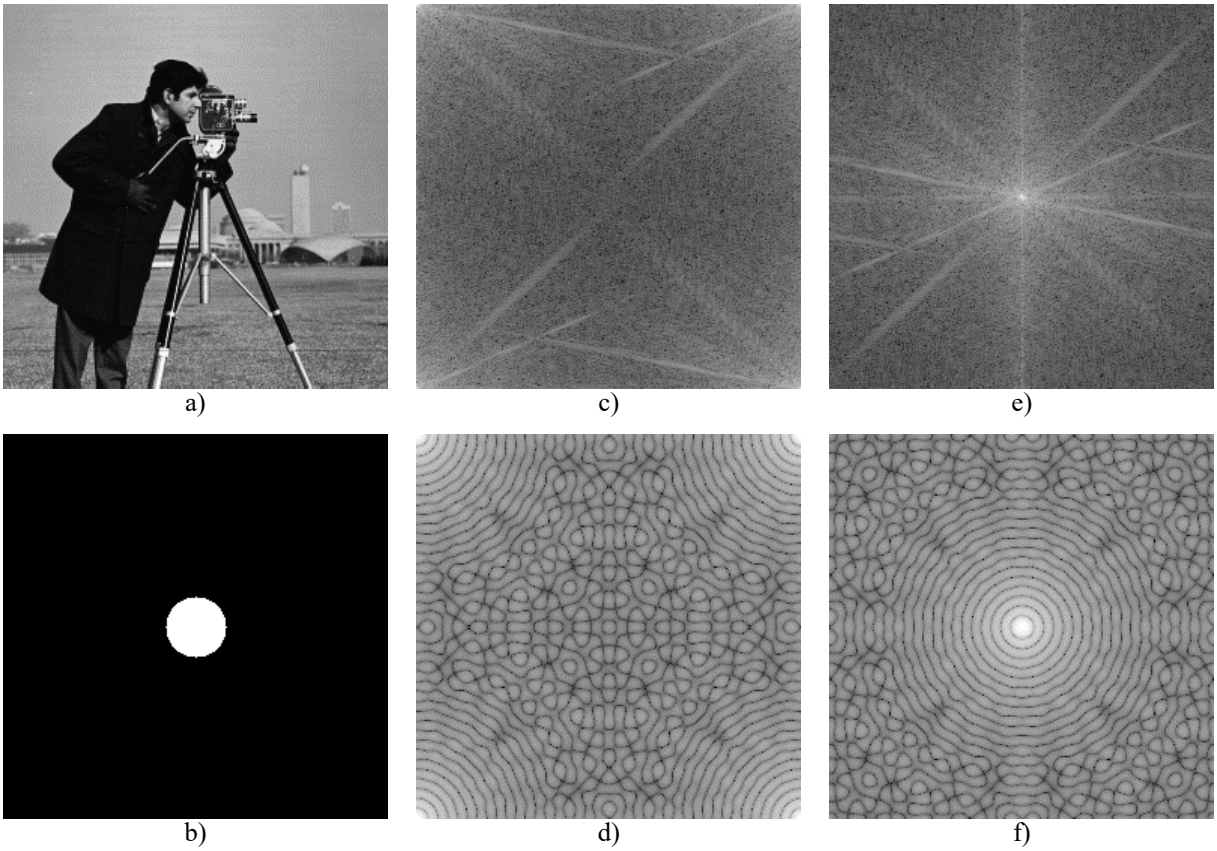
The inverse discrete Fourier transform (IDFT) is given by:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi jkn}{N}}, \quad n = \overline{0...N-1} \tag{9.12}$$

The 2D DFT is performed by applying the 1D DFT on each row of the input image and then on each column of the previous result. The 2D IDTF is performed by applying the 1D IDFT on each column of the DFT "image" and then on each row of the previous result. The set of complex numbers which are the result of the DFT may also be represented in polar coordinates (magnitude, phase). The set of (real) magnitudes represent the frequency power spectrum of the original array.

The DFT and its inverse are usually performed using the Fast Fourier Transform recursive approach, which reduces the computation time from $O(n^2)$ to $O(n \ln n)$, which represents a significant speed increase, especially in the case of 2D image processing, where a

$O(n^2m^2)$ complexity would be intractable for large images as opposed to the almost linear in number of pixels $O(nm\ln(nm))$complexity.



a) c) e)

b) d) f)

Fig. 9.4 a) and b) original images; c) and d) logarithm of magnitude spectra; e) and f) centered logarithm of magnitude spectra.

9.3.1. Aliasing

The aliasing phenomenon is a consequence of the Nyquist frequency limit (a sampled signal cannot represent frequencies higher than half the sampling frequency). This means that the higher half of the frequency domain representation is redundant. This fact can also be seen from the identity:

$$X_k = X^*_{N-k} \tag{9.13}$$

(where the asterisk denotes complex conjugation) which is true if the input numbers x_k are real. Therefore, the typical 1D Fourier spectrum will contain the low frequency components in both the lower and upper part, with high frequency located symmetrically about the middle. In 2D, the low frequency components will be located near the image corners and the high frequency components in the middle (see **Fig. 9.4**c, d). This makes the spectrum hard to read and interpret. In order to center the low frequency components spectrum about the middle of the spectrum, one should first perform the transformation on the input data:

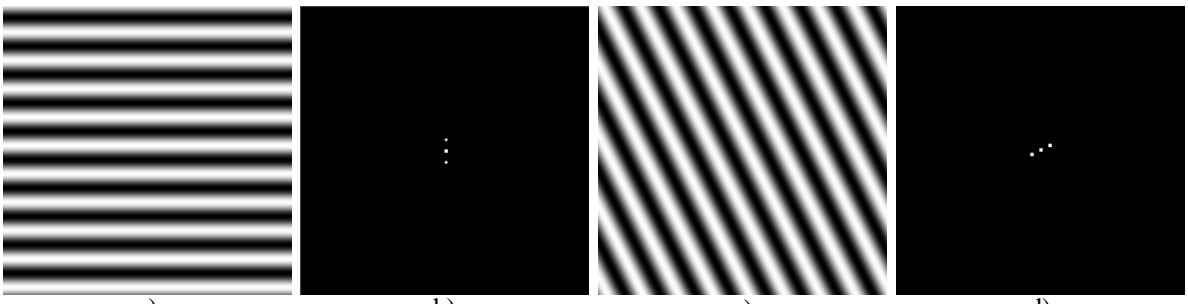
$$x_k \leftarrow (-1)^k x_k \tag{9.14}$$

In 2D the centering transformation becomes:

$$x_{uv} \leftarrow (-1)^{u+v} x_{uv} \tag{9.15}$$

After applying this centering transform, in 1D the spectrum will contain the low frequency components in the center, and the high frequency components will be located symmetrically toward the left and right ends of the spectrum. In 2D, the low frequency components will be located in the middle of the image, while various high frequency components will be located toward the edges.

The magnitudes located on any line passing through the DFT image center represent the 1D frequency spectrum components of the original image, along the direction of the line. Every such line is therefore symmetrical about its middle (the image center).



a) b) c) d)

Fig. 9.5 Fourier transforms of sine image waves a) and c). The center point in b) and d) represent the DC component, the other two symmetrical points are due to the sine wave frequency.

9.3.2. Ideal low-pass and high-pass filters in frequency domain

The convolution in spatial domain is equivalent to scalar multiplication in frequency domain. Therefore, especially for large convolution kernels, it is computationally convenient to perform convolution in the frequency domain.

The algorithm for filtering in the frequency domain is:

a) Perform the image centering transform on the original image (9.15).
b) Perform the DFT transform.
c) Alter the Fourier coefficients according to the required filtering.
d) Perform the IDFT transform.
e) Perform the image centering transform again (this undoes the first centering transform).

An ideal low pass filter will alter all the Fourier coefficients that are further away from the image center *(W/2, H/2)* than a given distance *R*, by turning them to zero (*W* is the image width and *H* is the image height):

$$X'_{uv} = \begin{cases} X_{uv}, & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 \le R^2 \\ 0, & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 > R^2 \end{cases} \tag{9.16}$$

An ideal high-pass filter will alter all Fourier coefficients that are at a distance less than R from the image center *(W/2, H/2)*, by turning them to 0.

$$X'_{uv} = \begin{cases} X_{uv}, & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 > R^2 \\ 0, & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 \le R^2 \end{cases} \tag{9.17}$$

The results of filtering with ideal low- and high-pass filtering are presented in **Fig. 9.6** b) and c). Unfortunately, the corresponding spatial filters **Fig. 9.6** e) and d) are not FIR (they have an infinite support) and keep oscillating away from their centers. Because of this, the low-pass and high-pass filtered images have a disturbing ringing wavy aspect. In order to correct this, the cutoff in the frequency domain must be smoother, as presented in the next section.

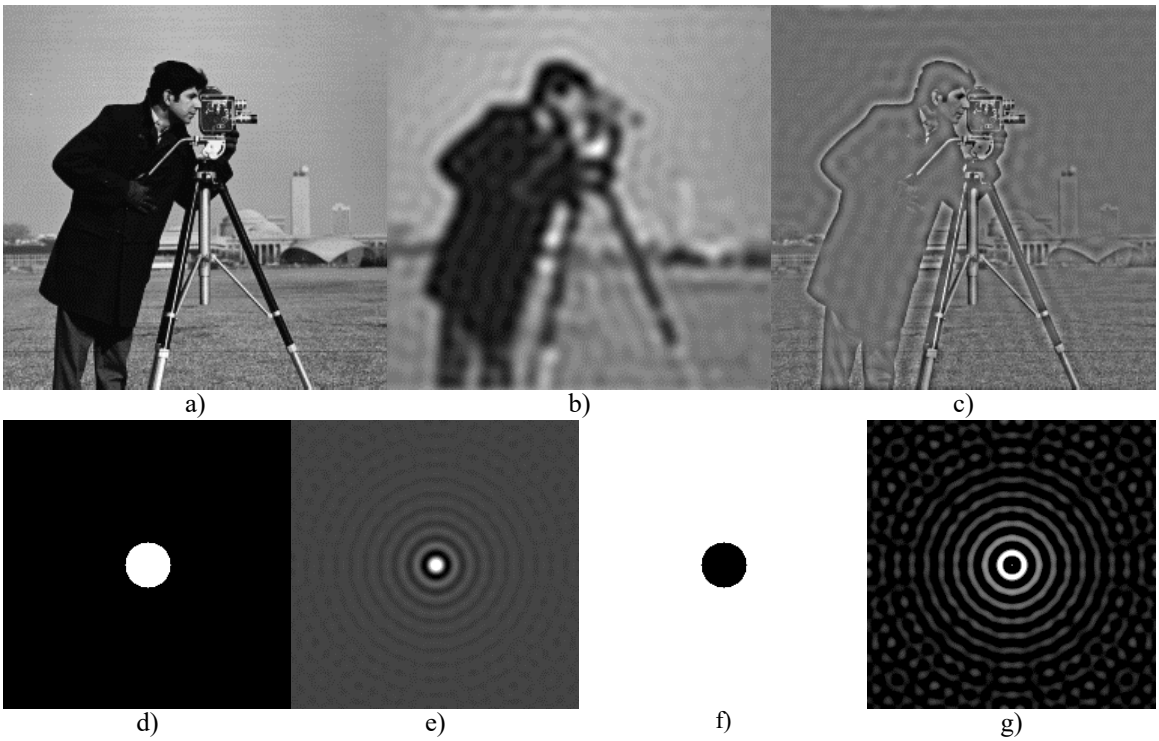


Fig. 9.6 a) original image; b) result of ideal low-pass filtering; c) result of ideal high-pass filtering; d) ideal low-pass filter in the frequency domain; e) corresponding ideal low-pass filter in spatial domain; f) ideal high-pass filter in frequency domain; g) corresponding ideal high-pass filter in the spatial domain. (R=20)

9.3.3. Gaussian low-pass and high-pass filtering in the frequency domain

In the case of Gaussian filtering, the frequency coefficients are not cut abruptly, but smoother cutoff process is used instead. This also takes advantage of the fact that the DFT of a Gaussian function is also a Gaussian function (**Fig. 9.7**d-g).

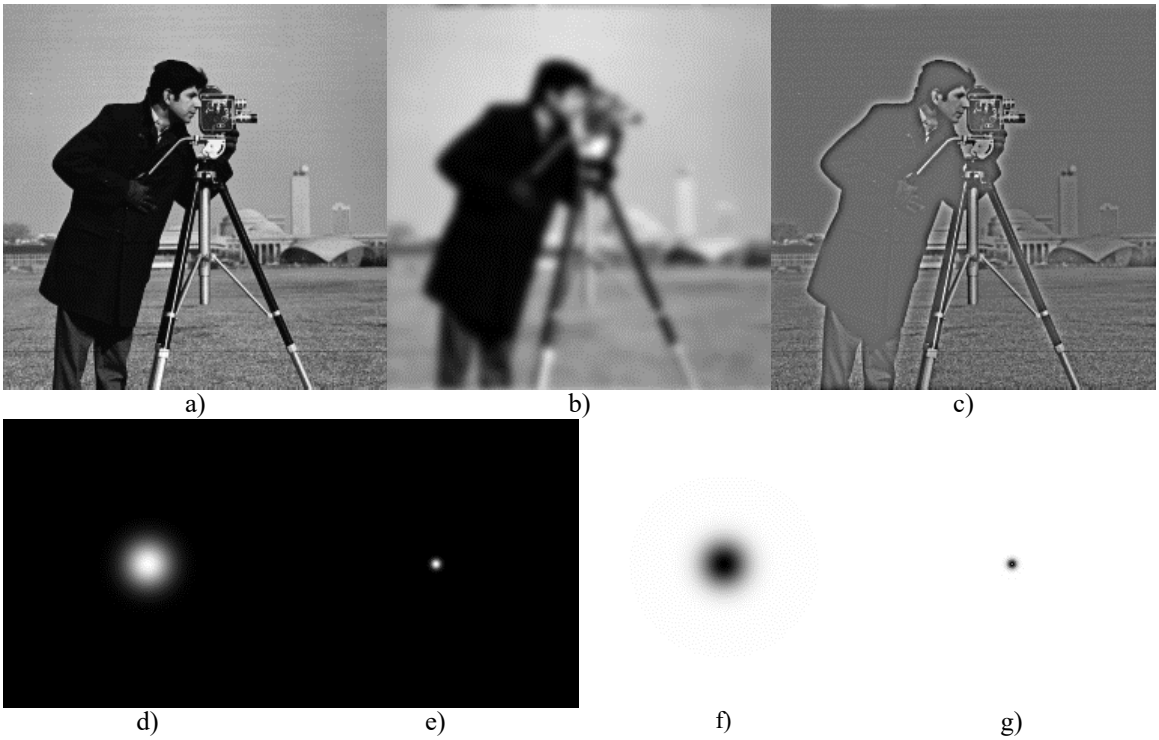
The Gaussian low-pass filter attenuates frequency components that are further away from the image center *(W/2, H/2)*. $A \sim \frac{1}{\sigma}$ where σ is the standard deviation of the equivalent spatial domain Gaussian filter.

$$X'_{uv} = X_{uv} e^{-\frac{\left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2}{A^2}} \tag{9.18}$$

The Gaussian high-pass filter attenuates frequency components that are near to the image center *(W/2, H/2)*:

$$X'_{uv} = X_{uv}\left(1-e^{-\frac{\left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2}{A^2}}\right) \tag{9.19}$$

Fig. 9.7 shows the results of Gaussian filter. Notice that the ringing (wavy) effect visible in **Fig. 9.6** disappeared.



a) b) c)

d) e) f) g)

Fig. 9.7 a) original image; b) result of Gaussian low-pass filtering; c) result of Gaussian high-pass Filtering; d) Gaussian low-pass filter in the frequency domain; e) corresponding Gaussian low-pass filter in the spatial domain; f) Gaussian high-pass filter in the frequency domain; g) corresponding Gaussian high-pass filter in the spatial domain. (A=20)

9.4. Implementation details

9.4.1. Spatial domain filters

Low-pass filters will always have positive coefficients, and therefore, the resulting filtered image will have positive values. You must ensure that the resulting image fits in the desired range (0-255 in our case). In order to ensure this, you must ensure that the coefficients of a low-pass filter sum to 1. If you are using integer operations pay attention to the order of operations! Usually, the division should be the last operation performed in order to minimize the rounding errors!

High-pass filters will have both positive and negative coefficients. You must ensure that the final result is an integer between 0 and 255! There are three possibilities to ensure that the resulting image fits the destination range. The first one is to compute:

$$S_+ = \sum_{F_k>0} F_k, \quad S_- = \sum_{F_k<0} -F_k,$$
$$S = \frac{1}{2\max\{S_+, S_-\}} \tag{9.20}$$
$$I_D(u,v) = S(F * I_S)(u,v) + \left\lfloor \frac{L}{2} \right\rfloor$$

In the formula above S_+ represents the sum of positive filter coefficients and S_- the sum of negative filter coefficients magnitudes. This result of the convolution operation with the high-pass filter $F * I_s$ always lies in the interval $[-LS_-, LS_+]$ where L is the maximum image gray level (255). By multiplying with S, the results will be scaled to the interval [*-L/2, L/2*]. By adding L/2 the interval [*-L/2, L/2*] will be translated to [*0, L*].

Another approach is to perform the convolution operation using signed integers, determine the global minimum (min) and maximum (max) from the result and then linearly transform the resulting values with:

$$I_D(u,v) = L \cdot \frac{(F * I_S)(u,v) - \min}{\max - \min} \tag{9.21}$$

Where min = min($F * I_s$) and max = max($F * I_s$) are computed globally from the entire convoluted image.

The third approach is to compute the magnitude of the result and saturate everything that exceeds the domain [0, *L*].

9.4.2. Frequency domain filters

It is common practice for visualization and for processing purposes to consider a representation of the frequency space which has the (0,0) coefficient in the image center. This can be achieved by cross-swapping the four quadrants of the Fourier image channels. Equivalently, we can preprocess the source image using 9.15. The generic filter presented below uses the following helper function, which performs the centering operation.

```
void centering_transform(Mat img){
	//expects floating point image
	for (int i = 0; i < img.rows; i++){
		for (int j = 0; j < img.cols; j++){
			img.at<float>(i, j) = ((i + j) & 1) ? -img.at<float>(i, j) : img.at<float>(i, j);
		}
	}
}
```

The OpenCV library provides an implementation for performing Discrete Fourier Transform. The following template code performs both the direct and the inverse transformation. Processing should be done on the magnitude channel of the Fourier transform. Since DFT works best if the input image has dimensions equal to powers of two, use *cameraman.bmp* as your input.

```
Mat generic_frequency_domain_filter(Mat src){
	//convert input image to float image
	Mat srcf;
	src.convertTo(srcf, CV_32FC1);

	//centering transformation
	centering_transform(srcf);

	//perform forward transform with complex image output
	Mat fourier;
	dft(srcf, fourier, DFT_COMPLEX_OUTPUT);

	//split into real and imaginary channels
	Mat channels[] = { Mat::zeros(src.size(), CV_32F), Mat::zeros(src.size(), CV_32F) };
	split(fourier, channels);  // channels[0] = Re(DFT(I)), channels[1] = Im(DFT(I))

	//calculate magnitude and phase in floating point images mag and phi
	Mat mag, phi;
	magnitude(channels[0], channels[1], mag);
	phase(channels[0], channels[1], phi);

	//display the phase and magnitude images here
	// ......

	//insert filtering operations on Fourier coefficients here
	// ......

	//store in real part in channels[0] and imaginary part in channels[1]
	// ......

	//perform inverse transform and put results in dstf
	Mat dst, dstf;
	merge(channels, 2, fourier);
	dft(fourier, dstf, DFT_INVERSE | DFT_REAL_OUTPUT | DFT_SCALE);

	//inverse centering transformation
	centering_transform(dstf);

	//normalize the result and put in the destination image
	normalize(dstf, dst, 0, 255,  NORM_MINMAX, CV_8UC1);
	//Note: normalizing distorts the resut while enhancing the image display in the range [0,255].
	//For exact results (see Practical work 3) the normalization should be replaced with convertion:
	//dstf.convertTo(dst, CV_8UC1);

	return dst;
}
```

9.5. Practical work

1. Implement a general filter, which performs the convolution operator with a custom kernel matrix. The scaling coefficient should be computed automatically as either the reciprocal of the sum of filter coefficients for low pass filters or according to equation (9.20) for high-pass filters.
2. Test the filter with the kernels from equations (9.5) .... (9.10)
3. Study the provided generic function for processing in the frequency domain. Perform the conversion of a source image from spatial domain to frequency domain by using the Fourier transform (DFT), then apply the inverse Fourier transform (IDFT) on the obtained Fourier spectrum coefficients and check if the destination is the same as the source image.
4. Add a processing function that computes and displays the logarithm of the magnitude of the Fourier transform of an input image. Add 1 to the magnitude to avoid *log*(0).
5. Add processing functions that perform low- and high-pass filtering in the frequency domain using the ideal and Gaussian filters from equations (9.16)...(9.19).

6. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

9.6. References

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8.
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002.

10. Noise modeling and digital image filtering

10.1. Introduction

Noise represents unwanted information which deteriorates image quality. Noise is defined as a process (n) which affects the acquired image (f) and is not part of the scene (initial signal – s). Using the additive noise model, this process can be written as:

$$f(i, j) = s(i, j) + n(i, j) \tag{10.1}$$

Digital image noise may come from various sources. The acquisition process for digital images converts optical signals into electrical signals and then into digital signals and is one processes by which the noise is introduced in digital images. Each step in the conversion process experiences fluctuations, caused by natural phenomena, and each of these steps adds a random value to the resulting intensity of a given pixel.

10.2. Noise modeling

Noise (n) may be modeled either by a histogram or a probability density function which is superimposed on the probability density function of the original image (s). In the following, the models for the most common types of noise will be presented: salt and pepper noise and Gaussian noise. Other types of noise, such as negative exponential model, gamma/Erlang model, Rayleigh model are also presented in the literature (see the course notes!).

10.2.1. The salt & pepper noise

In the *salt&pepper* noise model only two possible values are possible, a and b, and the probability of obtaining each of them is less than 0.1 (otherwise, the noise would vastly dominate the image). For an 8 bit/pixel image, the typical intensity value for *pepper* noise is close to 0 and for *salt* noise is close to 255.

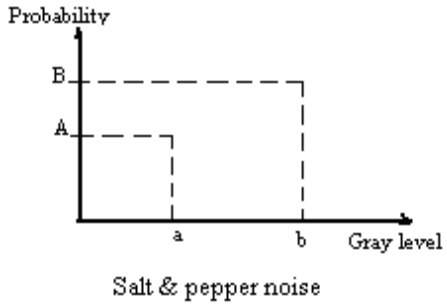


Fig. 10.1 Probability density function for the salt & pepper noise model.

$$PDF_{salt\&pepper} = \begin{cases} A & for\ g = a\ (\text{"}pepper\text{"}) \\ B & for\ g = b\ (\text{"}salt\text{"}) \end{cases} \tag{10.2}$$

The *salt&pepper* noise is generally caused by malfunctioning of camera's sensor cells, by memory cell failure or by synchronization errors in the image digitizing or transmission.

10.2.2. Gaussian noise

The Gaussian noise has a normal (Gaussian) probability density function:

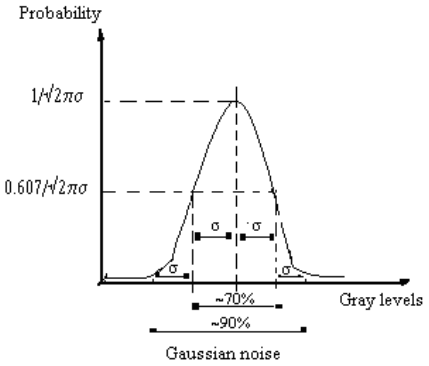


Fig. 10.2 Probability density function for the Gaussian noise model.

$$PDF_{Gaussian} = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(g-\mu)^2}{2\sigma^2}} \tag{10.3}$$

where:

g = gray level;
μ = mean;
σ = standard deviation;

Approximately 70% of the values are contained between $\mu \pm \sigma$ and 90% of the values are contained between $\mu \pm 2\sigma$. Although, theoretically speaking, the PDF is non-zero everywhere between -∞ and +∞, it is customary to consider the function 0 beyond $\mu \pm 3\sigma$.

Gaussian noise is useful for modeling natural processes which introduce noise (e.g. noise caused by the discrete nature of radiation and the conversion of the optical signal into an electrical one – detector/shot noise, the electrical noise during acquisition – sensor electrical signal amplification, etc.).

10.3. Noise removal using spatial filters

10.3.1. Ordered filters (non-linear)

Ordered filters are based on a specific image statistic, called ordered statistic. They are called non-linear, because they cannot be applied as a linear operator (such as a convolution kernel). These filters operate on small windows, and replace the value of the central pixel (similarly to convolution). The ordered statistic is a technique which arranges all the pixels in sequential order, based on their gray-level value. The position of an element in this ordered set can be characterized by its rank. Given a *N*x*N* window *W*, the pixel values can be sorted in ascending order:

$$I_1 \le I_2 \le I_3 \le \ldots \le I_{N^2} \tag{10.4}$$

Where:

$\{ I_1, I_2, I_3, \ldots, I_{N^2} \}$ represent the intensity values of the pixels located within the *N*x*N* window *W*.

For example: given a 3x3 window:

$$\begin{bmatrix} 110 & 110 & 114 \\ 100 & 106 & 104 \\ 95 & 88 & 85 \end{bmatrix}$$

The result of applying the ordered statistic will be:

{85, 88, 95, 100, 104, 106, 110, 110, 114}

The median filter: selects the middle value from the ordered statistic and replaces the destination pixel with it. In the example above, the selected value would be 104. The median filter allows the elimination of *salt&pepper* noise.

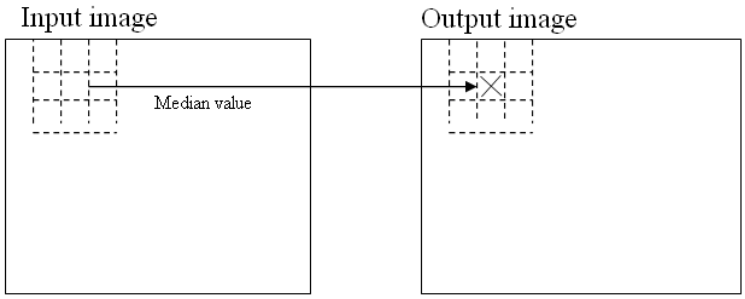


Fig. 10.3 Applying the median filter.

The maximum filter: selects the largest value amongst the ordered values of pixels from the window. In the above example, the value selected is 114. This filter can be used to eliminate the *pepper noise*, but it amplifies the *salt* noise if applied to a *salt&pepper* noise image.

The minimum filter: selects the smallest value amongst the ordered values of pixels from the window. In the above example, the value selected is 85. This filter can be used to eliminate the *salt noise*, but it amplifies the *pepper* noise if applied to a *salt&pepper* noise image.

10.3.2. Linear filters

These filters are applied by convolution (a linear operation) with a low-pass filter convolution kernel. In the following, the computation of the elements of a convolution kernel for Gaussian noise elimination will be presented.

10.3.3. Designing a variable size Gaussian convolution kernel

Gaussian noise removal must be performed using a filter with adequate shape and size, correlated to the amount of the Gaussian noise that corrupts the image (see Fig. 10.2). The filter size w of such a filter is usually 6σ (for example, for a Gaussian noise with σ=0.8 $\Rightarrow$ w = 4.8 $\approx$ 5).

Constructing the elements of such a kernel/Gaussian filter G will be performed using the following equations:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}} \qquad (10.5)$$

Where:

(x_0, y_0) – are the coordinates of the central row and column of the kernel (see Fig. 10.4).

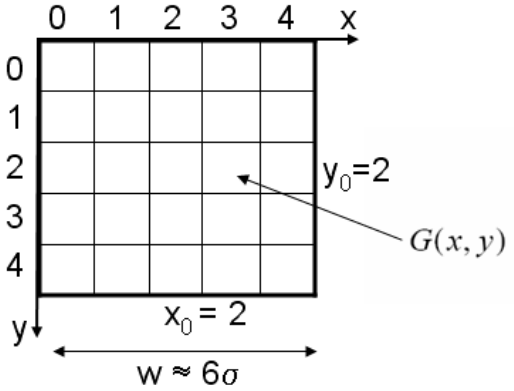


Fig. 10.4 Design example of a Gaussian kernel/filter G having a 5x5 size.

10.3.4. Image filtering/restoration

It is accomplished by the convolution of the source image with a Gaussian kernel/filter computed previously:

$$I_D = G * I_S \tag{10.6}$$

When the filter size w is large, the convolution may be time consuming (w x w multiplications for each pixel). In this case, the Gaussian decomposition may be used:

$$G(x, y) = G(x)G(y) \tag{10.7}$$

and replacing the convolution of a 2D nucleus G with two convolutions of a 1D nucleus G_x and G_y:

$$I_D = \left(G_x G_y\right) * I_S = G_x * \left(G_y * I_S\right) \tag{10.8}$$

Where:

G_x and G_y are 1D vectors (Fig. 10.5):

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-x_0)^2}{2\sigma^2}} \tag{10.9}$$

$$G(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-y_0)^2}{2\sigma^2}} \tag{10.10}$$

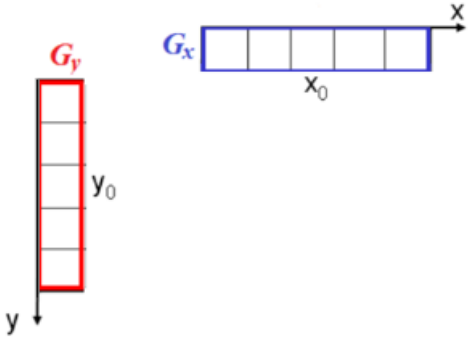


Fig. 10.5 The two vectors G_x and G_y into which a 2D Gaussian kernel may be separated.

In this case, the number of multiplications needed for each pixel is w for each of the two convolutions.

10.4. Processing time computation

```
double t = (double)getTickCount(); // Get the current time [ms]
// … Actual processing …
// Get the current time again and compute the time difference [ms]
t = ((double)getTickCount() - t) / getTickFrequency();
// Display (in the console window) the processing time in [ms]
printf("Time = %.3f [ms]\n", t * 1000);
```

10.5. Practical work

1. Implement a median filter with a variable dimension (w = 3, 5 or 7) specified by the user. Display the processing time.
2. Implement the filtering operation with a 2D Gaussian filter, with variable size w (w = 3, 5 or 7), specified by the user. The values of the kernel's components will be automatically computed as a function of σ ($\sigma = w/6$), as in equation (10.5). Display the processing time. Compare the processing times against different values of w.
3. Implement Gaussian filtering by using a Gaussian kernel separated into 2 vector components G_x and G_y having a variable size w (w = 3, 5 or 7), specified by the user. The vector components values G_x and G_y will be computed automatically as a function of σ ($\sigma = w/6$), as in equations (10.9) and (10.10). Display the processing time. Compare the processing times between the 2D and 1D Gaussian filters.
4. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms.**

10.6. References

[1] R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.

11. Edge detection

11.1. Introduction

This laboratory presents the edge detection problem in digital images. Edge points are found where the image intensity encounters a steep variation along a specific direction 'x' (Fig. 11.1). This intensity variation can be detected and quantified by finding the local maxima of the first order derivative of the image intensity (the gradient: $\nabla f=f'$) or by finding the zero crossings of the second order derivative of the image intensity (the laplacian: $\nabla^2 f=f''$).

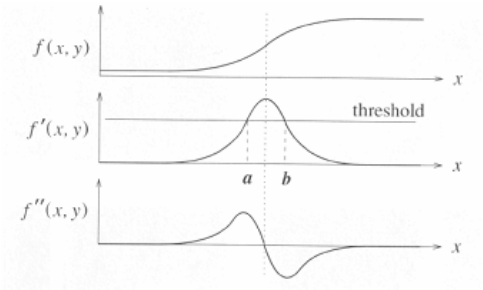


Fig. 11.1 Detection methods of the edge points (points were the image intensity suffers a steep variation).

Further on only the gradient based methods will be approached.

11.2. Computing the image gradient

The gradient in an image point is a vector heading the direction of the intensity variation around this point (Fig. 11.2). Its module is proportional with the speed of this variation (11.1). If the edge points are part of a contour (as in Fig. 11.2) the gradient will be perpendicular on the tangent to the contour at that point.

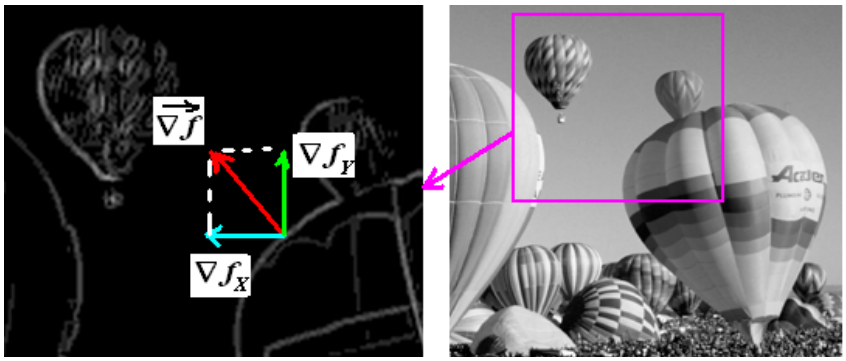


Fig. 11.2 Left: illustration of the image gradient (in an edge point) on the image of the gradient module.

The gradient of a two variables continuous function f is defined as:

$$\nabla f(x,y)=\begin{bmatrix}\nabla f_x\\ \nabla f_y\end{bmatrix}=\begin{bmatrix}\frac{\partial f}{\partial x}\\ \frac{\partial f}{\partial y}\end{bmatrix}=\begin{bmatrix}\lim\limits_{\Delta x\to 0}\frac{f(x+\Delta x,y)-f(x,y)}{\Delta x}\\ \lim\limits_{\Delta y\to 0}\frac{f(x,y+\Delta y)-f(x,y)}{\Delta y}\end{bmatrix} \tag{11.1}$$

For digital images, the gradient can be approximated by making Δx and Δy equal to 1 in (11.1):

$$\nabla f(x,y) = \begin{bmatrix} \nabla f_x \\ \nabla f_y \end{bmatrix} = \begin{bmatrix} f[x+1,y] - f[x,y] \\ f[x,y+1] - f[x,y] \end{bmatrix} \tag{11.2}$$

Other approximations of the two components of the gradient can be computed through the convolution of the image with the following kernels:

Prewitt:

$$\begin{aligned} \nabla f_x &= f(x,y) * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \\ \nabla f_y &= f(x,y) * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \end{aligned} \tag{11.3}$$

Sobel:

$$\begin{aligned} \nabla f_x &= f(x,y) * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ \nabla f_y &= f(x,y) * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \end{aligned} \tag{11.4}$$

Roberts (cross):

$$\begin{aligned} \nabla f_x &= f(x,y) * \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ \nabla f_y &= f(x,y) * \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \end{aligned} \tag{11.5}$$

In the case of Roberts (cross), that applies convolution with a 2 x 2 filter, the result of the convolution will be stored in the position of the top left corner of the window.

As a vector, the gradient can be quantified by a magnitude (11.6) and a direction (11.7).

$$\text{Magnitude: } \left|\nabla f(x,y)\right| = \sqrt{\left(\nabla f_x(x,y)\right)^2 + \left(\nabla f_y(x,y)\right)^2} \tag{11.6}$$

$$\text{Direction: } \theta(x,y) = arctg\left(\frac{\nabla f_y(x,y)}{\nabla f_x(x,y)}\right) \tag{11.7}$$

Direction must be incremented with 135^0 when applying Roberts (cross).

11.3. The Canny edge detection method

The edge detection method proposed by Canny is based on the image gradient computation but in addition tries to:

- maximize the signal-to-noise ratio for a proper detection;
- find a good localization of the edge points;
- minimize the number of positive responses around a single edge (suppression of the gradient module non-maxims)

The steps of the Canny edge detection method are given bellow:

1. Noise filtering through a Gaussian kernel
2. Computing the gradient's module and direction
3. Non-maxima suppression of the gradient's module
4. Edge linking through adaptive hysteresis thresholding.

11.3.1. Noise filtering through a Gaussian kernel

The noise in the image is high frequency information which overlaps the original image signal. This introduces false edge points. The noise intrinsic to the image acquisition process can be modeled by a Gaussian distribution and can be suppressed by a Gaussian filter (see laboratory 10).

11.3.2. Computing the gradient's magnitude and direction

Computing the gradient's module and direction requires the allocation of two temporary image buffers (with the same size as the image) and the initialization of their elements according to equations (11.6) and (11.7) respectively, where the horizontal $\nabla f_x(x,y)$ and the vertical $\nabla f_y(x,y)$ components of the image gradient can be computed using the Prewitt operator (11.3) or the Sobel operator (11.4).

11.3.3. Non-maxima suppression of the gradient's module

Its purpose is the thinning of the edges by retaining only the edge points with the highest gradient module along the direction of the image intensity variation (along the direction of the gradient vector).

The first step consists in the quantization of the gradient directions, computed using (11.7), in 4 regions shown in Fig. 11.3:

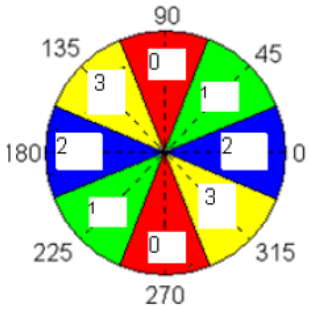


Fig. 11.3 Quantization of the gradient directions in the non-maxima suppression step.

Supposing that, for example, the direction of the gradient in an image point is "1" (Fig. 11.4), the module of the gradient in point P is a local maximum if: $|\nabla P| > |\nabla I_1|$ and $|\nabla P| > |\nabla I_1'|$. If these conditions are fulfilled, the point P is retained as an edge point, otherwise is rejected.

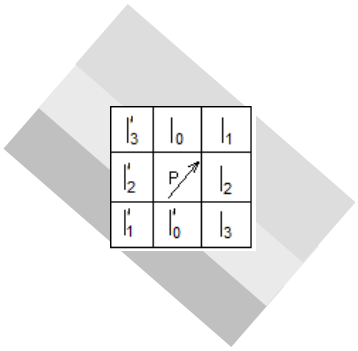


Fig. 11.4 Example for the non-maxima suppression.

11.3.4. Edge linking through adaptive hysteresis thresholding

After computing the image gradient and performing the non-maxima suppression procedure, an "image" is obtained in which the pixel values are equal with the gradient's modules in that pixel. Moreover, the thickness of the edge pixels (with non-zero module) has an ideal value of one pixel. In the following, the steps required to obtain the final edges are described:

11.3.4.1. Adaptive thresholding

Adaptive thresholding tries to extract a quite constant number edge points for a given image size. In this way, lighting and contrast variations are compensated (fixed threshold would extract either too much or too few edge points).

The parameter which is given to the threshold detection procedure is the ratio between the number of edge points and the number of points with non-zero gradient module:

$$NoEdgePixels = p \bullet (NoPixels - ZeroGradientModulePixels) \quad (11.8)$$

Parameter p has usually values between 0.01 and 0.1.

The algorithm is the following:

1. The histogram of the gradient's magnitude image (values obtained after non-maxima suppression) is computed. These values will be scaled to fit within [0..255] range (by division with $4\sqrt{2}$ if the gradient was computed using the Sobel operator). The result is a histogram $Hist[0..255]$:

$$Hist[i] = No\ of\ pixels\ having\ the\ scaled\ gradient\ magnitude\ value\ i \quad (11.9)$$

2. The number of pixels with non-zero values which would not be edge points is computed:

$$NoNonEdge = (1\text{-}p) \bullet (Height \bullet Width - Hist[0]) \quad (11.10)$$

3. Starting with position 1 the values of the histogram are summed. When the sum exceeds the value *NoNonEdge*, then the index *i* reached in the counting process is the searched threshold. This technique, intuitively, will find the gradient magnitude value (*AdaptiveThresholding*) bellow which *NoNonEdge* pixels are found.

Pay attention to the pixels located at the image margins (where the image gradient was not computed)! Their values should be zero or should not be taken into account, because they can modify the value of the threshold.

11.3.4.2. Edge extension through hysteresis

Adaptive thresholding does not guarantees the completeness of the edges (shadowed parts of the objects or presence of noise can affect the edge detection process). The result will be an image with many fragmented edges.

Therefore an edge extension technique is needed. The edges obtained by adaptive thresholding are considered STRONG EDGES and we try to extend them with weaker edge points, which have not passed the thresholding with the initial value, but could be detected with a lower threshold.

Formally, two thresholds are defined:

$$Threshold\_high = AdaptiveThresholding \quad (11.11)$$

$$Threshold\_low = k \bullet Threshold\_high \quad (11.12)$$

Where k<1 (for example, $k = 0.4$).

The image of the gradient module is scanned pixel by pixel. In the destination image the pixels with the gradient magnitude higher then *Threshold_high* are labeled as STRONG_EDGES (e.g. with the value 255). The pixels with the gradient magnitude between *Threshold_low* and *Threshold_high* are labeled as WEAK_EDGES (e.g. with the value 128).

The pixels with the gradient magnitude bellow *Threshold_low* are considered NON-EDGES and are rejected. The inverted result (negative) of this labeling is shown in Fig. 11.5-left:

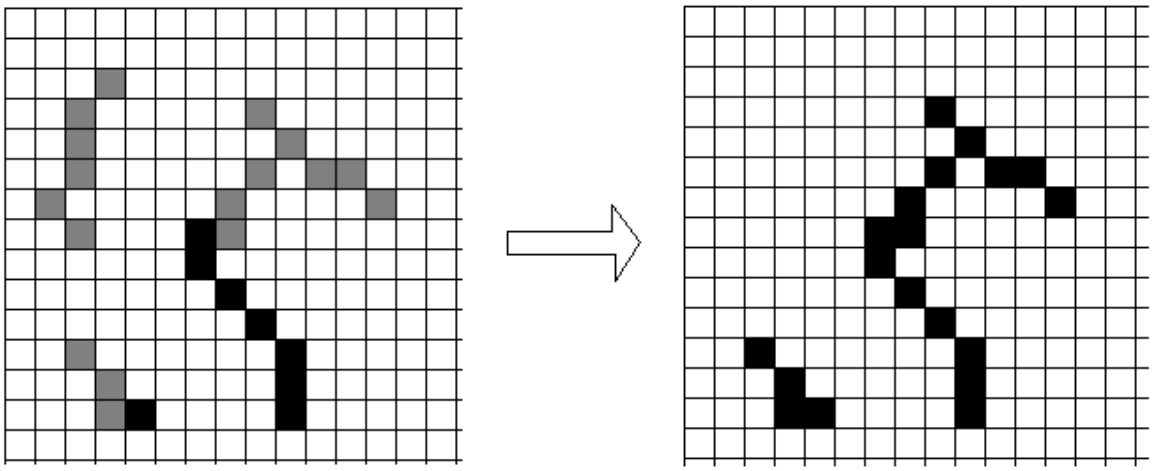


Fig. 11.5 Left: the image of the labeled strong and weak edges; Right: the result of the extension of the strong edges with connected WEAK edges.

Next step consists in the extension of the STRONG_EDGE points with neighboring WEAK_EDGE points, if they are parts of a connected component (see laboratory and lecture related to "Labeling") – as in Fig. 11.5. If a STRONG_EDGE point has WEAK_EDGE neighbor, the WEAK_EDGE neighbor is labeled as a STRONG_EDGE point. This STRONG_EDGE becomes a new source of edge extension. The process is repeated until the STRONG_EDGE points cannot be extended further by joining them with WEAK_EDGE points.

An efficient implementation of this step uses a queue to perform a breadth first search through WEAK_EDGE points connected to STRONG_EDGE points and mark them as STRONG_EDGE points. The algorithm would look like this:

1. Scan the image, top left to bottom right, pick the first STRONG_EDGE point encountered and push its coordinates in the queue.
2. While (queue is not empty)
    a) Extracts the first point from the queue
    b) Find all the WEAK_EDGE neighbors of the current point
    c) Label in the image all these neighbors as STRONG_EDGE points
    d) Push the image coordinates of these neighbors into the queue
    e) Continue to the next STRONG_EDGE point
3. Go to step 1 considering the next STRONG_EDGE point.
4. Eliminate the remaining WEAK_EDGE points from the image by turning them to NON_EDGE (0)

Final consideration: regarding the definition of the neighborhood used in the above algorithm, the common 4-type or 8-type neighborhood can be used, or a tolerance of 1 to 2 pixels can be considered. The reason is noise may cause edge interruptions by small gaps.

11.4. Practical work

The practical activities related to this laboratory will be split in two:

11.4.1. Laboratory 1(first WEEK)

1. The horizontal ∇f_x and vertical ∇f_y components of the gradient through convolution with the kernels given in (11.3) ... (11.5) will be computed and the results will be shown in destination windows (the convolution operation was already implemented in lab. 9).
2. The gradient magnitude (11.6) and direction (11.7) will be computed using the three operators (Sobel, Prewitt and Roberts) and the results of the gradient magnitude will be shown in a destination window.
3. The thresholding with an arbitrary and fixed threshold of the results obtained at point 2 will be shown in a destination window.
4. The steps 1 – 3 of the Canny edge detection algorithm will be implemented (step 1 – was already implemented in laboratory 10; step 2 – is the implementation with Sobel filters; step 3 – implement the non-maxima suppression operation). The results obtained after step 3 will be shown in a destination window. The results will be compared with the one obtained at point 2 after the simple use of the Sobel operator.
5. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

11.4.2. Laboratory 2 (second WEEK)

1. Edge linking through adaptive hysteresis thresholding algorithm (step 4 of the Canny method) will be implemented. The intermediate results of the STRONG_EDGE and WEAK_EDGE points (after the hysteresis thresholding and before the edge extension step) and the final results (with the final edges) will be shown destination windows. The implementation will be experimented for different values of the parameters *p, k* and neighborhood types.
2. The final results of the implemented Canny edge detection method will be tested/experimented on different image types.
3. **Save your work. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

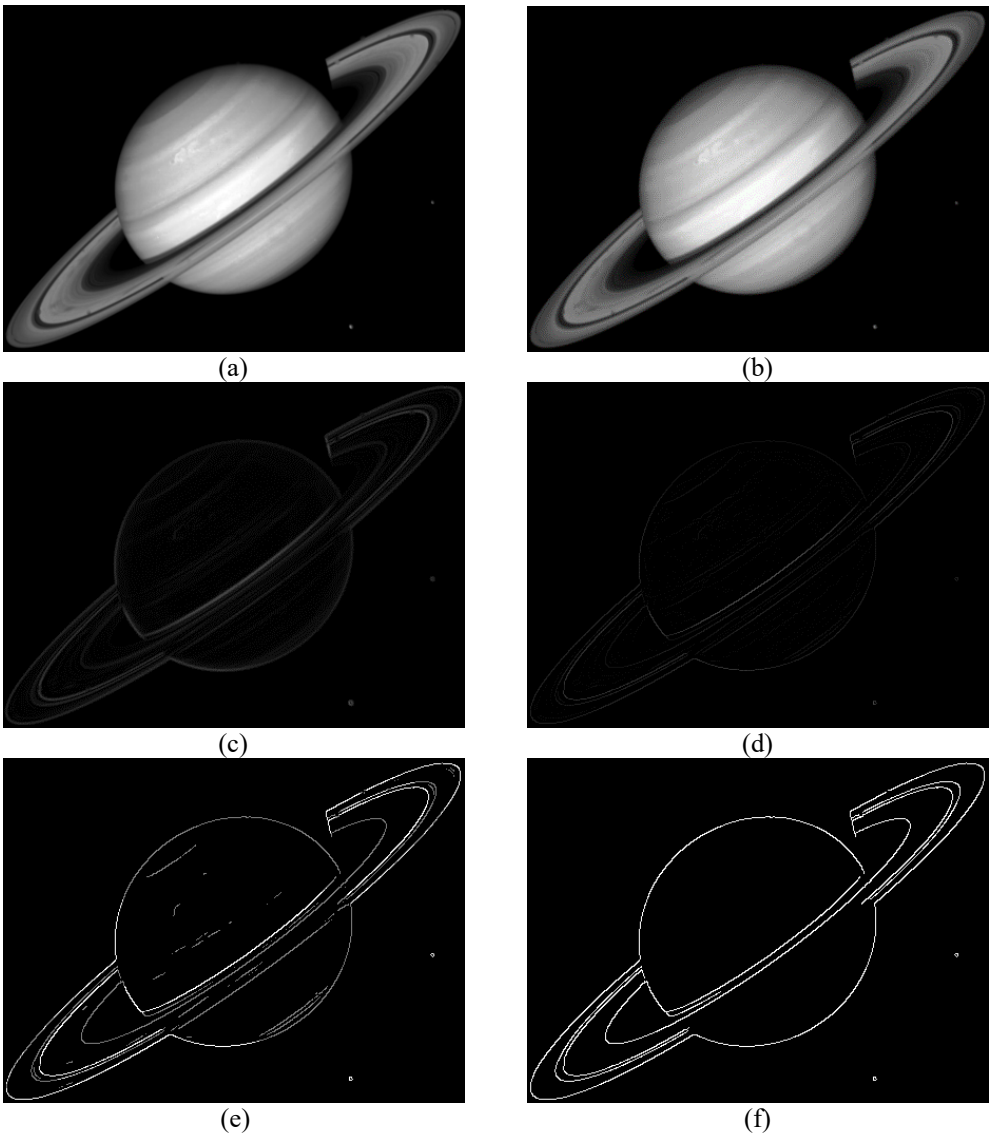


Fig. 11.6 Canny edge detection - sample results: a) Initial image; b) After Gaussian filtering ($\sigma = 0.5$); c) Normalized gradient magnitude (using Sobel operators); d) After non-maxima suppression; e) After adaptive thresholding ($p = 0.1$); f) Final edges after edge extension with N8 and weak-edge removal.

11.5. References

[1] E.Trucco, A.Verri, *Introductory Techniques for 3-D Computer Vision,* Prentice Hall, 2001.
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002.