

Chapter 2

Low-Level Image Processing

As regards obstacles, the shortest distance between two points can be a curve.

Bertolt Brecht

Abstract This chapter covers some basic concepts of low-level image processing. It introduces fundamental methods for two primary image processing tasks, namely contrast enhancement, image smoothing, and edge detection. The chapter also introduces methods of function optimization for searching the optimal configuration of edge points.

2.1 Introduction

In the previous chapter, we have introduced the concept of low-level, medium-level, and high-level digital image processing. In low-level processing tasks a digital image is used as input and another digital image is obtained as output (e.g., an image improved for the visualization). In high-level processing the outcome is a description of the content of the input image. In the medium-level processing some features are obtained from the input image, such as edges or regions.

Different operators are adopted for low-level processing. Usually, we distinguish among the following operators:

1. *Point operators* that produce a single output pixel by processing each pixel independently of the other pixels.
2. *Local operators* that produce a single output pixel by processing a neighborhood of that pixel.
3. *Global operators* that produce a single output pixel by processing the entire image.

Generally the aim of low-level operators is to improve the visual quality of the input image by means of enhancement or noise removal processes. In this chapter we introduce two fundamental image processing tasks that are contrast enhancement and edge detection. To this aim, some notations are given in the following.

Let $f(x, y)$ a gray-level image of $M \times N$ pixels with L gray levels. We introduce the following definitions:

- the *level scale* or *dynamic range* of image $f(x, y)$ is the range $[a, b]$ of values such that $a \leq f(x, y) \leq b$ for each (x, y) ;
- the *histogram* $h(i)$, $i = 0, \dots, L - 1$ of image $f(x, y)$ denotes the occurrence frequency of each level.

In Listing 2.1, we provide a simple applet to compute the histogram of a gray-level image. The applet can be executed using the HTML code listed in 2.2. To run the applet it is necessary to create a Java project with the files `HistogramApplet.java` and `HistogramApplet.html`, then include in the same directory an image named `im.png` to be processed.

In the RGB color space, individual histograms of each component can be computed. In [5] we find a plugin to compute the histogram of each R, G, B component given an input color image.

Listing 2.1 `HistogramApplet.java`: a Java applet for histogram visualization.

```
import java.awt.*;
import java.awt.image.*;
import java.applet.Applet;
public class HistogramApplet extends Applet {
    private Image image;
    private ImageCanvas imageCanvas;
    private Panel panel;
    private TextArea text;

    public void init() {
        String image_file = getParameter("IMAGEFILE");
        image = getImage(getDocumentBase(), image_file);
        while(image.getWidth(this) < 0);
        Dimension imageSize = new Dimension(
            image.getWidth(this), image.getHeight(this));
        imageCanvas = new ImageCanvas(image, imageSize);
        int[] pixels = ImageCanvas.grabImage(image, imageSize);
        panel = new Panel(new GridLayout(1, 2, 10, 10));
        text = new TextArea(20, 5);
        panel.add(imageCanvas);
        panel.add(text);
        add(panel);
        text.setText((new Histogram(pixels)).toString());
    }
}

class Histogram {
    private int histo[] = new int[256];
    public String toString() {
        String text = "";
        for(int i=0; i<256; i++) {
            text += i+"_"+histo[i]+'\\n';
        }
    }
}
```

```

        return text;
    }
    public Histogram(int[] rgb) {
        for(int i=0; i<rgb.length; i++) {
            int tmp = (int) (
                (((rgb[i] & 0xff0000)>>16) * 0.299) +
                (((rgb[i] & 0x00ff00)>>8) * 0.587) +
                (((rgb[i] & 0x0000ff) * 0.114) );
            histo[tmp]++;
        }
    }
    public int getValueAt(int index) {
        return histo[index];
    }
}

class ImageCanvas extends Canvas {
    static final int MIN_WIDTH = 64;
    static final int MIN_HEIGHT = 64;
    private Image image;
    private Dimension size;

    public ImageCanvas(Image img, Dimension dim) {
        super();
        image = img;
        size = dim;
    }
    public Dimension getMinimumSize() {
        return new Dimension(MIN_WIDTH, MIN_HEIGHT);
    }
    public Dimension getPreferredSize() {
        return new Dimension(size);
    }

    public void paint(Graphics g) {
        g.drawImage(image, 0, 0, getBackground(), this);
    }
    static public int[] grabImage(Image image, Dimension size) {
        int[] data = new int[size.width * size.height];
        PixelGrabber pg = new PixelGrabber(
            image, 0, 0, size.width, size.height, data, 0, size.width);

        try {
            pg.grabPixels();
        }
        catch (InterruptedException e) {
            System.err.println(
                "ImageSampler:_interrupted_while_grabbing_pixels");
            return null;
        }
        if ((pg.status() & ImageObserver.ABORT) != 0) {
            System.err.println(
                "ImageSampler:_pixel_grab_aborted_or_errored");
            return null;
        }
        return data;
    }
}

```

Listing 2.2 HistogramApplet.html: Html code to run HistogramApplet.java.

```

<html>
<head><title>Histogram</title></head>
<body>
<H1>Histogram</H1>
<applet
    name="HistogramApplet"
    code="HistogramApplet.class"
    width="800"
    height="500"
    alt="If you have a Java-enabled browser,
    you would see an applet here.">
<param name="IMAGEFILE" value="im.png">
</applet>
</body>
</html>

```

Generally, to define each low-level operator, a mapping T that transforms an input image $f(x, y)$ into an output image $g(x, y)$ has to be defined over some neighborhood of each pixel. Namely

$$g(x, y) = T(f(x, y)) \quad (2.1)$$

where T is a linear or nonlinear function defined on the dynamic range $[a, b]$.

2.2 Contrast Enhancement

The contrast of an image refers to the range of gray levels used in the image—the dynamic range. It refers to the intensity variation of the pixels, defined by the minimum and maximum intensity value. Contrast resolution is the ability to distinguish between differences in intensity. For example, low contrast image values may be concentrated near a few values of the gray scale (e.g., mostly dark, or mostly bright, or mostly medium values). One definition of image contrast is the following:

$$C = \frac{S_A - S_B}{S_A + S_B}$$

where S_A and S_B are intensity average values computed on pixels of two different regions A and B (for example background and object).

Low contrast images can result from poor illumination, lack of dynamic range in the imaging sensor, or even wrong set up during image acquisition. A fundamental low-level task is to improve the contrast in an image, by means of *contrast enhancement operators*.

To improve the contrast it is necessary to transform the levels of the image into the range of all the levels available for visualization (typically the range $[0, 255]$). Specifically, a contrast stretching, that means highlighting a specific range of gray levels in an image, is performed. The idea behind contrast stretching is to increase

the dynamic range in the image being processed. Moreover, to enlarge the dynamic range it is necessary to interpolate between successive levels. Figure 2.1 shows an example.

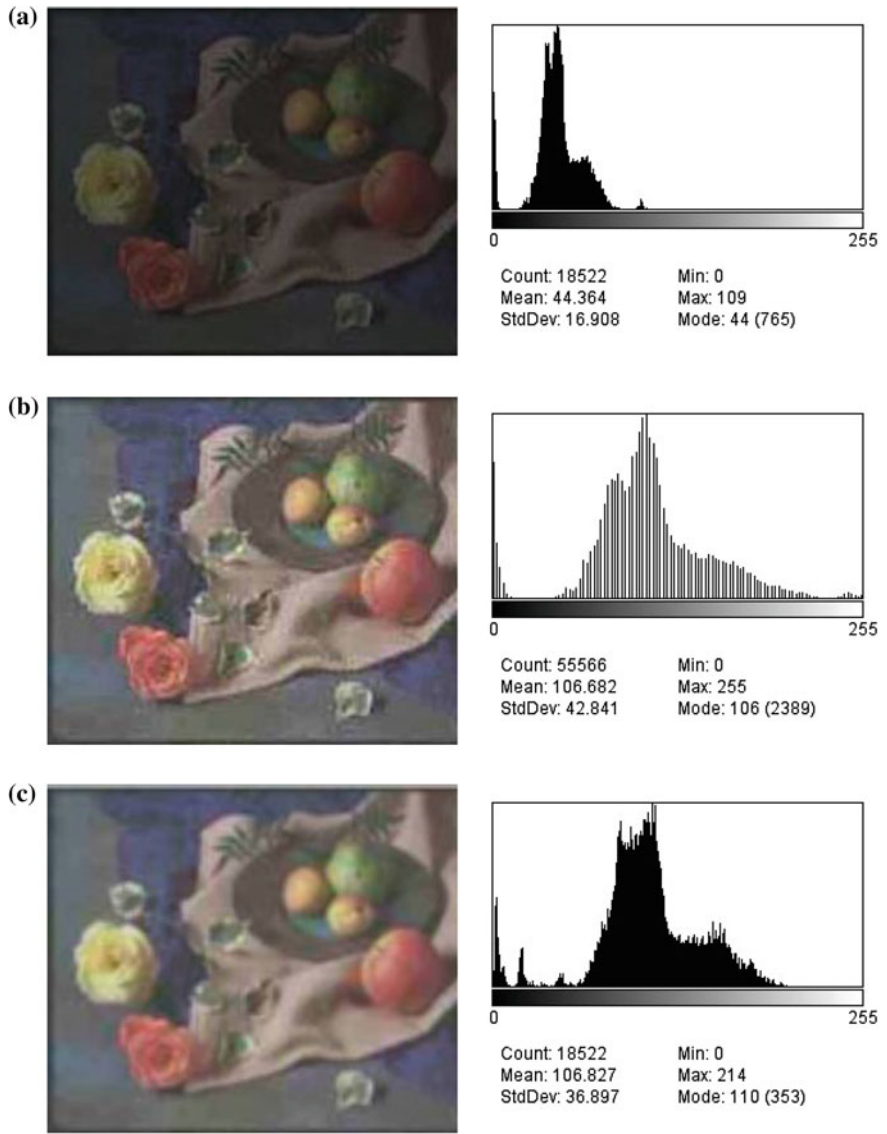


Fig. 2.1 **a** A RGB image and its brightness histogram. **b** The enhanced image and its histogram. **c** The enhanced and interpolated image and its histogram

2.2.1 Gray-Level Transformation

Some methods for contrast enhancement are based on gray-level transformation and histogram modification. These are point operators that are applied to a neighborhood reduced to (1×1) pixel. Hence Eq. (2.1) can be expressed in the form $l' = T(l)$ where l and l' denote the input pixel value and the output pixel value, respectively. Since the mapping $T(\cdot)$ denotes a point operator, it is independent on the pixel coordinates and it is the same for all the image pixels. Hence, each output pixel depends only on the input pixel having the same coordinates. These operators may be expressed by means of lookup tables.

Gray-level transformation operators can be divided into two main classes: linear operators and nonlinear operators. In the following we give some examples of both classes.

Linear Contrast Stretching

This transformation enhances the dynamic range by linearly stretching the original gray-level range $[a, b] \subset [0, 255]$ to the range $[0, 255]$. The transformation is defined as

$$l' = T(l) = 255 \frac{(l - a)}{(b - a)} \quad (2.2)$$

where $a \leq l \leq b$.

Generally the linear transformation from the range $[a, b]$ to the range $[a', b']$ is

$$l' = T(l) = (l - a) \frac{(b' - a')}{(b - a)} + a' \quad (2.3)$$

where $a \leq l \leq b$.

Linear Contrast Stretching with Clipping

This transformation is used when $[a, b] \supset [0, 255]$. If the number of levels outside the range $[0, 255]$ is small, these levels are clipped in the following manner: levels $l \leq 0$ are set to 0, levels $l \geq 255$ are set to 255. For all the other levels, the Eq. (2.2) is applied.

Logarithmic Transformation

This transformation is defined as

$$l' = c \log(1 + |l|) \quad (2.4)$$

with $c > 0$. This transformation is used to compress the dynamic range so as to enhance details related to low levels. For example, it is used to visualize the Fourier spectrum of an image [11].

Power-Law Transformation

This transformation is defined as

$$I' = I(c \cdot \exp^\gamma) \quad (2.5)$$

where c and γ are positive constant values. By varying the value of γ different functions can be obtained to compress or to expand the dynamic range of gray levels. Conventionally, the exponent in the power-law function is referred to as *gamma*. The power-law transformations are useful to perform gamma correction in the visualization of images on a monitor or generally they are useful for general-purpose contrast manipulation [6].

Sigmoid Transformation

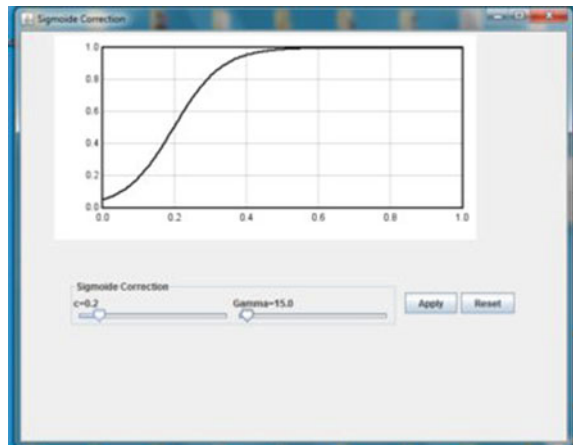
This transformation is defined as:

$$S(x) = 1/(1 + \exp^{-\gamma(x-c)}) \quad (2.6)$$

where the value of c indicates the abscissa of the inflection point of the function and the parameter γ controls the contrast (values greater than 5 results in an enhancement of the contrast). Figure 2.2 shows a plot of the S-function with $c = 0.2$ and $\gamma = 15$. By applying the S-function with different values of γ and c we obtain different contrast enhancement results.

In Fig. 2.3 we show some examples of contrast modification using the plugin given in Listing 2.3 that provides different S-functions.

Fig. 2.2 A plot of the S function with $c = 0.2$ and $\gamma = 15$



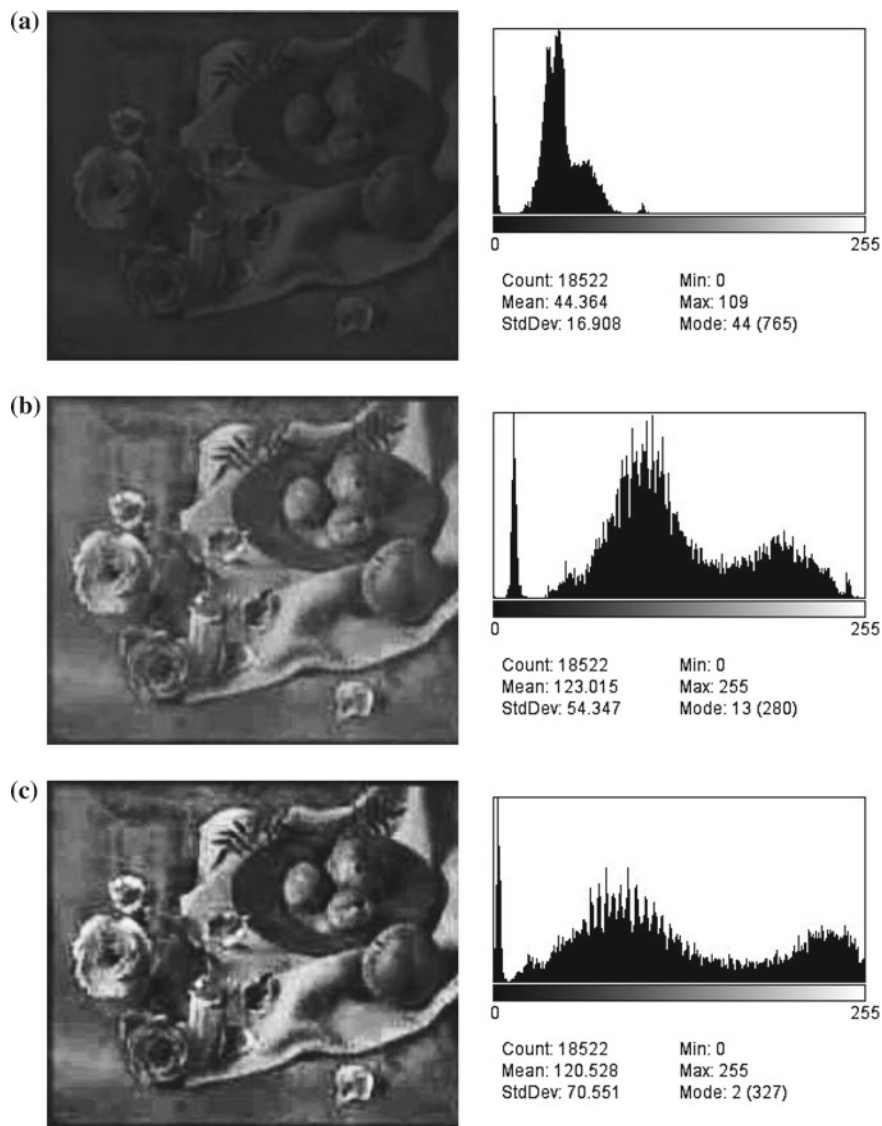


Fig. 2.3 Some examples of contrast modification using different S-functions. **a** Original image (brightness component of Fig. 2.1 and its histogram). **b** Contrasted image obtained using the S-function with $c = 0.2$ and $\gamma = 15$. **c** Contrasted image obtained by applying the S-function with $c = 0.2$ and $\gamma = 24$

Listing 2.3 S-function.java: Java plugin for contrast enhancement using the sigmoid function.

```

/**
 * Contrast enhancement by the following sigmoid function:
 * bb = 1/(1+Math.exp(GAMMA*(c-aa))).
 *
 * Different values for the parameters c and GAMMA
 * can be chosen
 *
 * Author: Ignazio Altomare
 * Date: 4/11/2010
 */
import ij.ImagePlus;
import ij.plugin.filter.PlugInFilter;
import ij.process.ImageProcessor;
import ij.gui.GenericDialog;
import ij.*;
import ij.gui.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import ij.text.*;

public class Sigmoid_Correction extends WindowAdapter
    implements PlugInFilter, ChangeListener, ActionListener {

    private int w;
    private int h;
    private ImagePlus im_sig;
    private ImageProcessor ip_orig, ip_sig;
    private byte[] im;
    private ImageWindow w_sig;

    // variables
    private int K = 256;
    private int aMax = K - 1;
    private float GAMMA_ini=15f;
    private float c_ini=0.5f;
    // window for visualizing the sigmoid function
    private JFrame windowSig;
    private PlotPanel graphicSig;
    // button for applySig e resetSig
    private JButton applySig;
    private JButton resetSig;
    //labels for C and Gamma
    private JLabel C_label;
    private JLabel Gamma_label;
    //slider for the values C and Gamma
    private JSlider C_slider;
    private JSlider Gamma_slider;

    public int setup(String arg, ImagePlus img) {
        return DOES_8G;
    }

    public void run(ImageProcessor ip) {
        w = ip.getWidth();
        h = ip.getHeight();

```

```

    ip_orig=ip;

    //create a copy of the image
    im_sig = NewImage.createByteImage("Sigmoid Correction",w,h,1,
        NewImage.FILL_BLACK);
    ip_sig = (im_sig.getProcessor()).convertToByte(true);
    ip_sig.copyBits(ip,0,0,Blitter.COPY);

    //get pixel values
    im = (byte[]) ip_sig.getPixels();

    //process
    this.process();

    //show the sigmoid window
    this.showSig();

    w_sig = new ImageWindow(im_sig);
    w_sig.addWindowListener(this);
    im_sig.updateAndDraw();
}

private void process(){
    // create a lookup table for the mapping function
    int[] Fgc = new int[K];
    for (int a = 0; a < K; a++) {
        double aa = (double) a / (double) aMax;    // scale to [0,1]
        double bb = 1/(1+Math.exp(GAMMA_ini*(c_ini-aa)));

        // scale back to [0,255]
        int b = (int) Math.round(bb * aMax);
        Fgc[a] = b;
    }

    ip_sig.applyTable(Fgc);    // modify the image
}

private ImagePlus plotSig(){
    float[] x = new float[256];
    float[] y = new float[256];

    for(int i=0; i<256; i++){
        x[i]=(float)i/(float)aMax;
        y[i]=(float)(1/(1+Math.exp(GAMMA_ini*(float)(c_ini-x[i]))));
    }

    Plot p = new Plot("Sigmoid Correction","", "",x,y);
    p.setLimits(0.0,1.0,0.0,1.0);
    p.setLineWidth(2);

    return p.getImagePlus();
}

private void showSig(){
    //create buttons
    applySig=new JButton("Apply");
    applySig.addActionListener(this);
    resetSig=new JButton("Reset");

```

```

        resetSig.addActionListener(this);

        //create panels
        JPanel panelSigmoid=new JPanel(new GridLayout(2,2));
        JPanel panelApply_Reset=new JPanel();
        graphicSig = new PlotPanel(this.plotSig().getImage());

        //set borders of panel
        panelSigmoid.setBorder(BorderFactory.createTitledBorder
            ("Sigmoid Correction"));

        //create labels
        C_label=new JLabel();
        Gamma_label=new JLabel();

        this.setLabelSig();

        //add labels to panel
        panelSigmoid.add(C_label);
        panelSigmoid.add(Gamma_label);

        //create sliders
        C_slider=new JSlider(JSlider.HORIZONTAL);
        Gamma_slider=new JSlider(JSlider.HORIZONTAL);

        this.setSliderSig();

        C_slider.addChangeListener(this);
        Gamma_slider.addChangeListener(this);

        //add slider to panel
        panelSigmoid.add(C_slider);
        panelSigmoid.add(Gamma_slider);

        //add Apply and Reset buttons
        panelApply_Reset.add(applySig);
        panelApply_Reset.add(resetSig);

        //create window for the Sigmoid function
        windowSig = new JFrame("Sigmoid Correction");
        windowSig.setSize(700,550);
        windowSig.setLocation(300,200);
        windowSig.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        windowSig.setLayout(new FlowLayout());
        Container contentPane=windowSig.getContentPane();
        contentPane.add(graphicSig);
        contentPane.add(panelSigmoid);
        contentPane.add(panelApply_Reset);
        windowSig.setVisible(true);
    }

    private void setLabelSig(){
        C_label.setText("c="+c_ini);
        Gamma_label.setText("Gamma="+GAMMA_ini);
    }

    private void setSliderSig(){
        C_slider.setMinimum(1);
        C_slider.setMaximum(10);
    }

```

```

        C_slider.setValue((int)(c_ini*10));
        Gamma_slider.setMinimum(1);
        Gamma_slider.setMaximum(255);
        Gamma_slider.setValue((int) GAMMA_ini);
    }

    private void resetSig(){
        c_ini=0.5f;
        GAMMA_ini=15;
        C_slider.setMinimum(1);
        C_slider.setMaximum(10);
        C_slider.setValue((int)(c_ini*10));
        Gamma_slider.setMinimum(1);
        Gamma_slider.setMaximum(255);
        Gamma_slider.setValue((int) GAMMA_ini);
    }

    public void actionPerformed(ActionEvent e){

        Object source=e.getSource();

        if(source==applySig){
            ip_sig.copyBits(ip_orig,0,0,Blitter.COPY);
            im = (byte[]) ip_sig.getPixels();
            this.process();
            im_sig.updateAndDraw();
        }

        if(source==resetSig){
            this.resetSig();
            setSliderSig();
            setLabelSig();
            graphicSig.updateImage(plotSig().getImage());
        }
    }

    public void stateChanged(ChangeEvent e){
        Object source=e.getSource();

        if(source==C_slider){

            c_ini = (float)C_slider.getValue()/((float)10);

            setSliderSig();
            setLabelSig();
            graphicSig.updateImage(plotSig().getImage());
        }

        if(source==Gamma_slider){

            GAMMA_ini = Gamma_slider.getValue();

            setSliderSig();
            setLabelSig();
            graphicSig.updateImage(plotSig().getImage());
        }
    }
}

```

```

public void windowClosing(WindowEvent e){
    windowSig.setVisible(false);
}
}

```

2.2.2 Thresholding

Another way to perform contrast stretching is thresholding. A threshold t is defined so that levels $l \leq t$ are set to 0, while levels $l > t$ are set to 255. In this way a binary image is obtained having values $\{0, 255\}$ or $\{0, 1\}$ (Fig. 2.4).

2.2.3 Histogram Transformation

Histograms are the basis for numerous spatial domain processing techniques. Histograms are simple to calculate and also lend themselves to economic hardware implementations, thus making them a popular tool for real-time image processing.

The histogram represents a global information for an image: all the pixels having a particular value i contribute to populate the i -th bin of the histogram. That is $h(i) = n_i$ being n_i the number of pixels having value i . The presence of peaks in an histogram may represent bright or dark regions, or regions having low or high contrast. The modification of the histogram produces a different distribution of gray levels. Hence histogram manipulation can be used effectively for image enhancement. The modification of an histogram is defined by means of a point transformation $T : j \rightarrow k$ such that if the level j has frequency $h(j)$ then the transformed level k has frequency $g(k)$ where $h(\cdot)$ and $g(\cdot)$ are the initial histogram and the transformed one, respectively.



Fig. 2.4 **a** LENA image. **b** Image obtained by thresholding only the brightness channel

Equalization

Equalization or normalization is the transformation of the level distribution into a uniform distribution in which the frequency of each transformed level is approximately constant, i.e., $g(i) \simeq \text{constant}$ for $i = 0, \dots, L_{\max}$ where L_{\max} is the maximum gray level. Since equalization provides an histogram that is almost uniform, it improves the image by augmenting the contrast and removing regions that are too bright or too dark.

Let $f(x, y)$ be a gray-level image of $n = M \times N$ pixels with $L_{\max} + 1$ gray levels and let $h(i)$ be its histogram. We define the cumulative histogram of f as

$$h_c(i) = \sum_{j=0}^i h(j) = \sum_{j=0}^i n_j$$

for $i = 0, \dots, L_{\max}$. If the histogram h is uniform then its cumulative function h_c is a line. Hence the equalization of the histogram h can be computed by imposing the cumulative histogram h_c to be linear. To this aim we consider the following equation:

$$\frac{i}{h_c(i)} = \frac{L_{\max}}{n} \quad i = 0, \dots, L_{\max}$$

from which we obtain

$$i = L_{\max} \frac{h_c(i)}{n} = L_{\max} \frac{\sum_{j=0}^i h(j)}{n} = L_{\max} \frac{\sum_{j=0}^i n_j}{n}$$

where n is the number of pixels and n_j is the occurrence number of the level j .

Appendix A provides reference to a Java plugin [5] useful to evaluate the histogram of a color image.

2.3 Image Smoothing

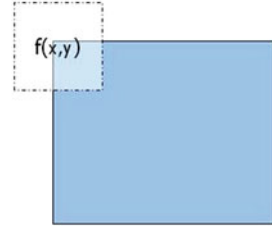
Smoothing, also called blurring, is a simple and frequently image processing operation, used to ‘blur’ images and remove detail and noise. A blurring process can attenuate the abrupt transitions of the gray levels between a pixel and its neighbor (random noise) or the irrelevant details associated to a small number of pixels.

Generally to perform a smoothing operation we apply a filter to the image, by means of a local operator. A local operator produces a value for each pixel (x, y) of the output image g computed in a neighbor or window w (Fig. 2.5) centered in the pixel (x, y) of the input image f by the following equation:

$$g = T(f, w)$$

In Fig. 2.5 a filter is visualized as a window of coefficients sliding across the image, that is the image is explored in a fixed sequence (for example, from left to

Fig. 2.5 Window or mask centered in the pixel $f(x, y)$



right and from top to down). The function T can be linear or not linear. The most common types of filter are linear: the output value $g(x, y)$ is determined as a weighted sum of input pixel values $f(x + i, y + j)$. Local operators based on the convolution operation can be formally described by means of the theory of linear systems, namely the Fourier transform and the convolution product theorem.

Linear image smoothing is a local operator based on a convolution of the image with a matrix h of proper dimension $L \times L$, called *mask* or *kernel*, where L is an odd value. More formally given an image $f(x, y)$ of $M \times N$ pixels and $h(x, y)$ a $L \times L$ spatial mask, we define $l = \lfloor L/2 \rfloor$ and the following equation describes the convolution product in the spatial domain between the image f and the mask h , with origin in the center of the mask.

$$g = h \otimes f$$

$$g(x, y) = \sum_{i=-l}^l \sum_{j=-l}^l h(i, j) f(x + i, y + j) \text{ for } x = 0, \dots, M - 1, y = 0, \dots, N - 1$$

There are many kind of filters depending on the mask used in the convolution equation. In the following we will mention the most used.

Mean Filter

A simple process for image smoothing consists in locally computing the mean value for each pixel. This can be obtained by means of the convolution of the input image with the mask in Fig. 2.6 (lowpass spatial filtering). The mask is a square matrix of coefficients used to compute a new value starting from the neighbor of the examined pixel. It is also called filter in the spatial domain. The multiplying factor is needed to normalize the weights to 1. In this way the range of the output results the same as the input. The effect of the convolution operator is to compute each output pixel as mean value of the pixels in its $L \times L$ neighborhood.

Fig. 2.6 3×3 mean filter

$$h = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

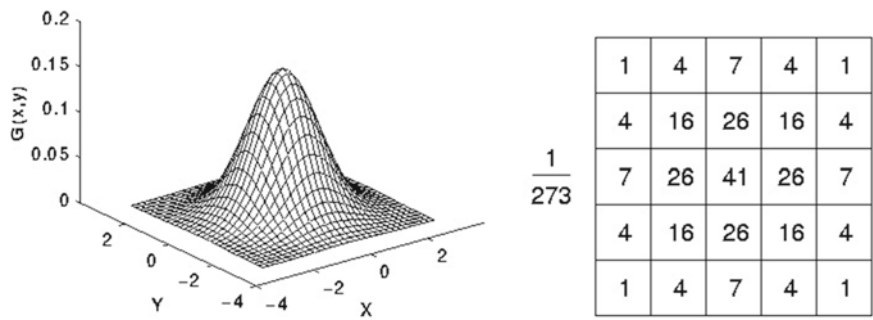


Fig. 2.7 A two-dimensional Gaussian function with $\sigma = 1$ and the corresponding 5×5 mask

Gaussian Filter

The Gaussian smoothing operator is a two-dimensional convolution operator that is used to blur images and remove details and noise. It uses kernels having the shape of a two-dimensional Gaussian function:

$$G(x, y, \sigma) = \left[\frac{1}{2\pi\sigma^2} \right] \exp \left(- \frac{x^2 + y^2}{2\sigma^2} \right)$$

where σ is the standard deviation and $r = x^2 + y^2$ is the ray from the center. The degree of smoothing is determined by the value of σ . Larger values of σ require larger convolution kernels in order to be accurately represented. Figure 2.7 shows a two-dimensional Gaussian function and the corresponding mask.

About 99.7 of values drawn from a Gaussian function are within three standard deviations (σ away from the mean). This fact is known as the three-sigma rule. For this reason, the Gaussian smoothing eliminates the influence of those points that are away from 3σ with respect to the current pixel in the mono-dimensional case and $6\sqrt{2}\sigma$ in the bi-dimensional case (the central lobe of the two-dimensional Gaussian function has the value $2\sqrt{2}\sigma$). A Java plugin for the two-dimensional Gaussian filter is available at [7]. As an example, Fig. 2.8 shows the effect of the Gaussian smoothing on the Lena image.

2.4 Edge Detection

Edges represent abrupt changes or discontinuities in an amplitude attribute of an image such as luminance, surface orientation, color and so on. Edges characterize object boundaries and are usually defined as curves separating two regions having different average values of their characteristics. The causes of the region dissimilarity may be due to some factors such as the geometry of the scene, radiometric characteristics of the surface, illumination, and so on. If the regions are sufficiently



Fig. 2.8 The Lena image and the result of Gaussian smoothing with $\sigma = 3$

homogeneous, the transition between two adjacent regions may be detected by analyzing the discontinuities along gray levels.

Edge detection is a fundamental problem in image analysis and computer vision. It is the process to locate and identify sharp discontinuities in an image giving boundaries between different regions. This boundary detection is the first step in many computer vision edge-based tasks such as face recognition, obstacle detection, target recognition, image compression, and so on. Edge detection is a local operator based on a convolution of the image with a matrix h of dimension $L \times L$, called *mask* or *kernel*, where L is an odd value.

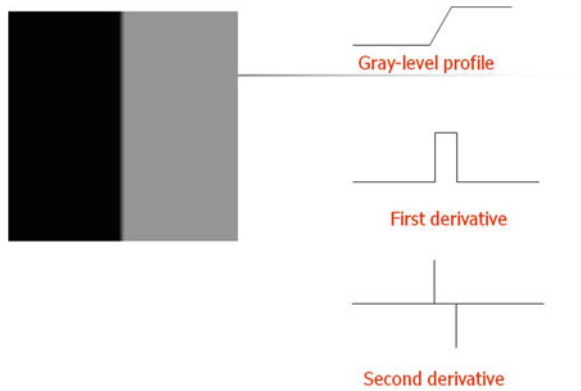
An edge is characterized by the following features:

- *Edge normal*: the unit vector in the direction of maximum intensity change.
- *Edge direction*: the unit vector along the edge (perpendicular to the edge normal).
- *Edge position or center*: the image position at which the edge is located.
- *Edge strength or magnitude*: local image contrast along the normal. Generally a pixel is an edge pixel if its strength overcomes a predefined threshold value.

An edge detection method detects the pixels candidate to be points of the boundary of an object or a region. To derive a boundary of an object all the edge pixels of that boundary should be grouped. This can be done by *border following* algorithms or *grouping* algorithms.

Since edges may not be represented by perfect discontinuities, the quality of detected edges is highly dependent on noise, lighting conditions, objects of same intensities, and density of edges in the scene. Regarding this problem it should be noted that even though noise is not visible in the original image, noise is highlighted in derivatives, especially in second derivatives. Hence edge detection, being based on derivatives, is highly affected by noise. Some noise effect can be reduced by thresholding. For example, we could define a point in an image as an edge point if its first derivative is greater than a specified threshold. By doing so, we automatically assess which significant gray-level transitions can be considered as edge. Another problem arises when the edge is located on a soft discontinuity. A solution to this problem is proposed in Chap. 8.

Fig. 2.9 Edge profile and derivatives



The most common edge detection methods are the Gradient operator based on first derivatives, the Laplacian and the LoG (Laplacian of a Gaussian) operators based on second derivatives. Figure 2.9 shows how the magnitude of the first derivative can be used to detect the presence of an edge in an image. The sign of the second derivative can be used to determine whether an edge pixel lies on the dark or the light side of an edge. The zero crossings of the second derivative provide a powerful way of locating edges in an image.

Generally an edge detection method involves three steps

1. Smoothing to reduce the noise;
2. Applying edge enhancement, that is a local operation that extracts all image pixels candidate to be edge points;
3. Applying edge localization (thresholding) to select the edge points among all the candidate points.

Steps 1. and 2. can be implemented by convolving the input image with a proper mask so as to obtain the gradient image. In the third step the edge points are detected, for example by looking for maximum and minimum magnitude values for the first derivative operators. These operators analyze the distribution of the gradient values in the neighborhood of a given pixel and determine if the pixel has to be classified as an edge point on the basis of threshold values. The results of these edge detectors are very sensitive to the threshold value. These operators require high computational time and hence cannot be used in real-time applications.

Gradient Operator

Given an image $f(x, y)$, its gradient is defined by

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial f(x, y)}{\partial x} \\ \frac{\partial f(x, y)}{\partial y} \end{bmatrix}$$

The magnitude of the gradient is given by

$$m(x, y) = |\nabla f(x, y)| = \left(\left(\frac{\partial f(x, y)}{\partial x} \right)^2 + \left(\frac{\partial f(x, y)}{\partial y} \right)^2 \right)^{\frac{1}{2}}$$

The direction of the gradient is given by

$$\alpha(x, y) = \tan^{-1} \left(\frac{\frac{\partial f(x, y)}{\partial x}}{\frac{\partial f(x, y)}{\partial y}} \right)$$

The Gradient operator may be implemented as a convolution with the masks shown in Figs. 2.10 and 2.11.

Laplacian Operator

In many applications it is of particular interest to construct derivative operators, which are isotropic, i.e., rotation invariant. This means that rotating the image f and applying the operator gives the same result as applying the operator on f and then rotating the result. In other words, if the operator is isotropic then equally sharpened edges are enhanced in any direction. One of these isotropic operators is the Laplacian operator, defined as

$$\nabla^2 f(x, y) = \frac{\partial^2 f(x, y)}{\partial^2 x} + \frac{\partial^2 f(x, y)}{\partial^2 y}$$

This can be implemented using the mask of Fig. 2.12. If $f(x, y)$ is not constant or it does not vary linearly then the Laplacian of f has a zero crossing, i.e., a sign change crossing the x axis.

Laplacian of a Gaussian

Using second-order derivatives, the edge localization step is based on the extraction of zero-crossing points which indicate a sign change crossing the x -axis. Since the second-order derivative is very sensitive to noise, a filtering function is required. In [8] a Gaussian function is used to smooth the image hence deriving the operator called Laplacian of a Gaussian (LoG). The Gaussian smoothing operator is a

Fig. 2.10 Sobel masks to detect vertical edges

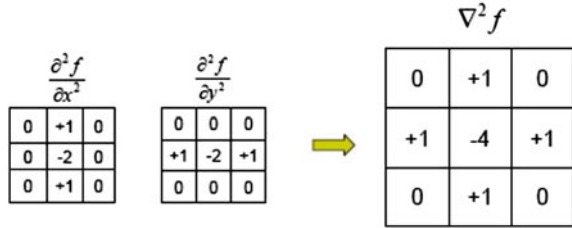
-1	-2	-1
0	0	0
1	2	1

1	2	1
0	0	2
-1	-2	-1

Fig. 2.11 Sobel masks to detect horizontal edges

-1	0	1
-2	0	2
-1	1	1

1	0	-1
2	0	-2
1	0	-1

Fig. 2.12 Laplacian mask

two-dimensional convolution operator that is used to blur images and remove details and noise. It uses kernels that represent the shape of a two-dimensional Gaussian function

$$G(x, y, \sigma) = \left[\frac{1}{2\pi\sigma^2} \right] \exp\left(\frac{-x^2 + y^2}{2\sigma^2}\right)$$

The LoG operator based on this Gaussian function is defined as

$$LoG(x, y, \sigma) = c \left[\frac{(x^2 + y^2)}{\sigma^2} - 1 \right] \exp\left(\frac{-x^2 + y^2}{2\sigma^2}\right)$$

where c is a factor that normalizes to 1 and the value of σ determines for each pixel (x, y) the number of points that influence the evaluation of the Laplacian in (x, y) . A significant problem of the LoG operator is that the localization of edges with an asymmetric profile by zero-crossing points introduces a bias which increases with the smoothing effect of filtering [1].

2.4.1 Canny Operator

An interesting solution to avoid the dependence of detected edges on noise was proposed by J. Canny in [2], who defined an optimal operator for edge detection including three criteria: good detection, good localization, identification of single edge point (a given edge in the image should be marked only once).

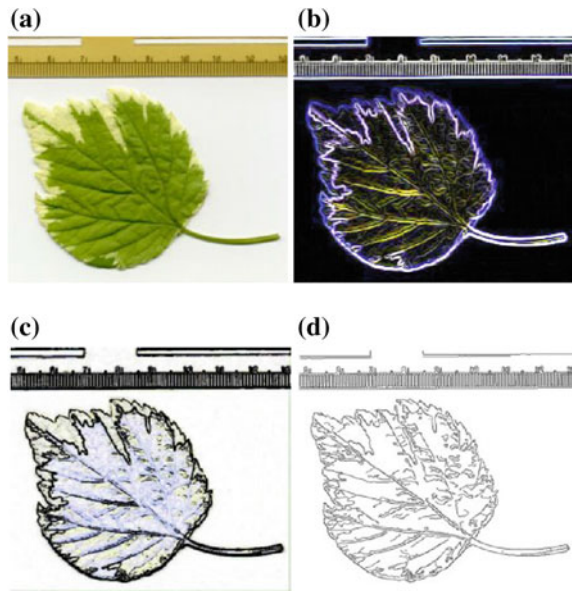
Let $f(x, y)$ a gray-level image of $M \times N$ pixels and $G(x, y)$ a Gaussian filter. The Canny operator performs the following steps:

1. First the noise is filtered out from the image. A suitable Gaussian filter is used for this task. Gaussian smoothing can be performed using a convolution product $f_s(x, y) = g(x, y) \otimes f(x, y)$. Some parameters have to be fixed for this operator, such as the standard deviation σ of the Gaussian filter. The width of the mask must be chosen carefully since it is directly proportional to the localization error. Since the Gaussian smoothing eliminates the influence of the points far more than $3(2\sqrt{2}\sigma)$ with respect to the current pixel, the mask size must be equal to $6\sqrt{2}\sigma$ for a fixed value of σ .

2. The second step consists of computing the gradient of f by means of the Sobel masks along x (columns) and y (rows) directions. Edge strength is found out by taking the gradient of the image.
3. The third step finds the edge direction using the gradient in x and y directions. For each pixel (x, y) we evaluate the gradient strength $m(x, y)$ and the gradient direction $\alpha(x, y)$, where m and α are matrices having the same size of the image $f(x, y)$. A non-maxima suppression algorithm is applied that follows the edge direction and suppresses any pixel value that is not to be considered as edge point. That is, for each pixel (x, y) we consider the gradient direction $\alpha(x, y)$ and check if $m(x, y)$ has a local maximum in that direction. Usually a small number of directions is considered. For example, the four directions $(0^\circ, 90^\circ, 45^\circ, -45^\circ)$ of a (3×3) window centered in the pixel (x, y) may be considered to produce an initial edge map $S(x, y)$.
4. The last step uses double thresholding to eliminate false edges. Two thresholds $t_1 < t_2$ are selected, with a ratio of 2 or 3. Pixels of $edge(x, y)$ having a gradient magnitude $m(x, y)$ greater than t_2 are definitively labeled as edge pixels. If a point (x, y) has $m(x, y) < t_2$ and is also connected to points yet labeled as edge points then $m(x, y)$ is compared with t_1 . If $m(x, y) > t_1$ then the point (x, y) is definitively labeled as an edge point. All the other points are not labeled as edge points.

An *Imagej* plugin that implements the Canny algorithm can be found at [4]. An application example is shown in Fig. 2.13. We can observe how an appropriate choice of the parameters of the Canny operator may produce very thin edges.

Fig. 2.13 **a** Original image. **b** Image obtained by applying the gradient in (x, y) directions. **c** Image after thresholding of the brightness of the image **b**. **d** Image obtained from **a** after applying Canny operator with $\sigma = 1$, $t_1 = 2.5$, $t_2 = 7.5$



2.4.2 Optimization-Based Operators

In the previous sections, we have seen that detection of edges usually involves two stages. The first one is an edge enhancement process that requires the evaluation of derivatives of the image making use of gradient or Laplacian operators. Methods such as thresholding or zero crossing produce an edge map that contains pixels candidates to be labeled as edge points. In the second stage, pixels of the edge map are selected and combined in contours using processes such as boundary detection, edge linking, and grouping of local edges [11, 13].

This last phase can be viewed as a search of the optimal configuration of those pixels that better approximate the edges. More precisely let us consider an image $F = \{f(x, y); 0 \leq x \leq M - 1, 0 \leq y \leq N - 1\}$ and an edge configuration $S = \{s(x, y); 0 \leq x \leq M - 1, 0 \leq y \leq N - 1\}$ where $s(x, y) = 1$ if (x, y) is an edge pixel, $s(x, y) = 0$ otherwise. Therefore an edge could be considered as one of the possible paths in the universe of the pixels of the image F . If we define a function $T(S)$ evaluating the edge S , the searching of the best edge configuration can be accomplished by means of an optimization method that minimizes/maximizes the function $T(S)$.

In other words, the edge detection problem can be formulated as one of optimization where the evaluation function depends on the local edge structure. Since the search space for the optimal solution is extremely large due to the number of possible configurations of the $M \times N$ pixels of the image, a ‘blind’ search would be fully inefficient. Then optimization methods are necessary which take into account the geometric and topological constraints of the problem. In this sense some methods have been introduced such as graph searching, relaxation, and simulated annealing [9, 10, 12].

In [3] optimization techniques known as Genetic Algorithms are proposed for the search of the optimal edge. The peculiarities of these algorithms are the robustness in the application to different classes of problems and the natural parallel implementation. When using a genetic algorithm for optimization, a solution is encoded as a string of genes to form a chromosome representing an individual. In [3] an individual is an edge configuration S represented by a string of $M \times N$ bits. Each bit encodes the presence (or not) of an edge pixel in the image F . The approach consists essentially of two phases: evaluation of the likelihood of a pixel to be an edge pixel and boundary detection by means of genetic algorithms. An objective function T is supplied which assigns a fitness value to each edge configuration S . This function evaluates the cost of S as the sum of the costs of each pixel (x, y) in S . The assumptions are that the edges should tend to be continuous, thin and of sufficient length; moreover the edges should be perpendicular to the gradient at each pixel. The cost function T evaluates at each point the deviation from the previous assumptions by computing a linear combination of five weighted factors: fragmentation, thickness, local length, region similarity, and curvature. These factors capture the local nature of the edges and are evaluated in a $(w \times w)$ window centered on each pixel (x, y) using the values of the configuration S and a likelihood map L based on the gradient (amplitude and

direction). The pixels in this window constitute the neighbor of the central pixel. The genetic algorithm, starting from an initial population (i.e., a collection of possible solutions) iteratively produces new generations of individuals (i.e., potential solutions) using the operators of reproduction, crossover, and mutation. Since the problem is the minimization of the objective function $T(S)$, each individual S of the population must reproduce itself in proportion to the inverse of its function $T(S)$. The iterative optimization process ends when the mean value of the objective function T does not change, within a tolerance value, between two consecutive generations.

References

1. Bhardwaja, S., Mittal, A.: A survey on various edge detector techniques. *Procedia Technol.* **4**, 220–226 (2012)
2. Canny, J.F.: A computational approach to edge detection. *IEEE Trans. PAMI* **8**(6), 679–698 (1986)
3. Caponetti, L., Abbattista, N., Carapella, G.: A genetic approach to edge detection. In: *Proceedings of the IEEE International Conference on Image Processing*, 1994. ICIP-94, pp. 318–322. IEEE, New York (1994)
4. Gibara, T.: Canny edge detector. ImageJ plugin available at: <http://rsbweb.nih.gov/ij/plugins/canny/index.html>
5. Gibara, T.: Color histogram. ImageJ plugin available at: <http://rsb.info.nih.gov/ij/plugins/color-histogram.html>
6. Huang, S.C., Cheng, F.C., Chiu, Y.S.: Efficient contrast enhancement using adaptive gamma correction with weighting distribution. *IEEE Trans. Image Process.* **22**(3), 1032–1041 (2013)
7. Lieng, E.: 2D Gaussian filter. Java plugin available at: <https://imagej.nih.gov/ij/plugins/gaussian-filter.html>
8. Marr, Y.D., Hildreth, E.: Theory of edge detection. *Proc. R. Soc. Lond. B: Biol. Sci.* **207**(1167), 187–217 (1980)
9. Martelli, A.: An application of heuristic search methods to edge and contour detection. *Commun. ACM* **19**(2), 73–83 (1976)
10. Mumford, D., Shah, J.: Boundary detection by minimizing functionals. In: *Proceedings of the IEEE Computer Vision Pattern Recognition (San Francisco)*, pp. 22–26 (1985)
11. Rosenfeld, A., Kak, A.C.: *Digital Picture Processing*. Academic Press, Cambridge (1982)
12. Tan, H.L., Gelfand, S.B., Delp, E.J.: A cost minimization approach to edge detection using simulated annealing. *IEEE Trans. PAMI* **14**(1), 3–18 (1991)
13. Torre, V., Poggio, T.A.: On edge detection. *IEEE Trans. PAMI* **8**(2), 147–163 (1986)

Fuzzy Logic for Image Processing

A Gentle Introduction Using Java

Caponetti, L.; Castellano, G.

2017, XIV, 138 p. 61 illus., 33 illus. in color., Softcover

ISBN: 978-3-319-44128-3