9

Machine Vision Algorithms

Carsten Steger

MVTec Software GmbH, Arnulfstraße 205, 80634 München, Germany

In the previous chapters, we have examined the different hardware components that are involved in delivering an image to the computer. Each of the components plays an essential role in the machine vision process. For example, illumination is often crucial to bring out the objects we are interested in. Triggered frame grabbers and cameras are essential if the image is to be captured at the right time with the right exposure. Lenses are important for acquiring a sharp and aberration-free image. Nevertheless, none of these components can "see," that is, extract the information we are interested in from the image. This is analogous to human vision. Without our eyes we cannot see. Yet, even with eyes we cannot see anything without our brain. The eye is merely a sensor that delivers data to the brain for interpretation. To this analogy a little further, even if we are myopic, we can still see – only worse. Hence, we can see that the processing of the images delivered to the computer by the sensors is truly the core of machine vision. Consequently, in this chapter, we will discuss the most important machine vision algorithms.

9.1 Fundamental Data Structures

Before we can delve into the study of the machine vision algorithms, we need to examine the fundamental data structures that are involved in machine vision applications. Therefore, in this section we will take a look at the data structures for images, regions, and subpixel-precise contours.

9.1.1 Images

An image is the basic data structure in machine vision, since this is the data that an image acquisition device typically delivers to the computer's memory. As we saw in Chapter 6, a pixel can be regarded as a sample of the energy that falls on the sensor element during the exposure, integrated over the spectral distribution of the light and the spectral response of the sensor. Depending on the camera type, typically the spectral response of the sensor will comprise the entire visible

Handbook of Machine and Computer Vision: The Guide for Developers and Users,
Second Edition. Edited by Alexander Hornberg.

spectrum and optionally a part of the near-infrared spectrum. In this case, the camera will return one sample of the energy per pixel, that is, a single-channel gray value image. RGB cameras, on the other hand, will return three samples per pixel, that is, a three-channel image. These are the two basic types of sensors that are encountered in machine vision applications. However, cameras capable of acquiring images with tens to hundreds of spectral samples per pixel are possible [1, 2]. Therefore, to handle all possible applications, an image can be considered as a set of an arbitrary number of channels.

Intuitively, an image channel can simply be regarded as a two-dimensional (2D) array of numbers. This is also the data structure that is used to represent images in a programming language. Hence, the gray value at the pixel (r, c) can be interpreted as an entry of a matrix: $g = f_{r,c}$. In a more formalized manner, we can regard an image channel f of width w and height h as a function from a rectangular subset $R = \{0, \dots, h-1\} \times \{0, \dots, w-1\}$ of the discrete 2D plane $\mathbb{Z}^2$ (i.e., $R \subset \mathbb{Z}^2$) to a real number, that is, $f : R \mapsto \mathbb{R}$, with the gray value g at the pixel position (r, c) defined by $g = f(r, c)$. Likewise, a multichannel image can be regarded as a function $f : R \mapsto \mathbb{R}^n$, where n is the number of channels.

In the above discussion, we have assumed that the gray values are given by real numbers. In almost all cases, the image acquisition device will discretize not only the image spatially but also the gray values to a fixed number of gray levels. In most cases, the gray values will be discretized to 8 bits (1 byte), that is, the set of possible gray values will be $\mathbb{G}_8 = \{0, \dots, 255\}$. In some cases, a higher bit depth will be used, for example, 10, 12, or even 16 bits. Consequently, to be perfectly accurate, a single-channel image should be regarded as a function $f : R \mapsto \mathbb{G}_b$, where $\mathbb{G}_b = \{0, \dots, 2^b - 1\}$ is the set of discrete gray values with b bits. However, in many cases this distinction is unimportant, so we will regard an image as a function to the set of real numbers.

Up to now, we have regarded an image as a function that is sampled spatially, because this is the manner in which we receive the image from an image acquisition device. For theoretical considerations, it is sometimes convenient to regard the image as a function in an infinite continuous domain, that is, $f : \mathbb{R}^2 \mapsto \mathbb{R}^n$. We will use this convention occasionally in this chapter. It will be obvious from the context which of the two conventions is used.

9.1.2 Regions

One of the tasks in machine vision is to identify regions in the image that have certain properties, for example, by performing a threshold operation (see Section 9.4). Therefore, at the minimum we need a representation for an arbitrary subset of the pixels in an image. Furthermore, for morphological operations, we will see in Section 9.6.1 that it will be essential that regions can also beyond the image borders to avoid artifacts. Therefore, we define a region as an arbitrary subset of the discrete plane: $R \subset \mathbb{Z}^2$.

The choice of the letter R is intentionally identical to the R that is used in the previous section to denote the rectangle of the image. In many cases, it is extremely useful to restrict the processing to a certain part of the image that is specified by a region of interest (ROI). In this context, we can regard an image as a

function from the ROI to a set of numbers, that is, $f : R \mapsto \mathbb{R}^n$. The ROI is sometimes also called the domain of the image because it is the domain of the image function f. We can even unify the two views: we can associate a rectangular ROI with every image that uses the full number of pixels. Therefore, from now on, we will silently assume that every image has an associated ROI, which will be denoted by R.

In Section 9.4.2, we will also see that often we will need to represent multiple objects in an image. Conceptually, this can simply be achieved by considering sets of regions.

From an abstract point of view, it is therefore simple to talk about regions in the image. It is not immediately clear, however, how best to represent regions. Mathematically, we can describe regions as sets, as in the above definition. An equivalent definition is to use the characteristic function of the region:

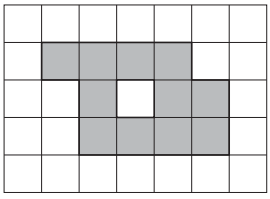
$$\chi_R(r,c) = \begin{cases} 1, & (r,c) \in R \\ 0, & (r,c) \notin R \end{cases} \tag{9.1}$$

This definition immediately suggests the use of binary images to represent regions. A binary image has a gray value of 0 for points that are not included in the region, and 1 (or any other number different from 0) for points that are included in the region. As an extension to this, we could represent multiple objects in the image as label images, that is, as images in which the gray value encodes the region to which the point belongs. Typically, a label of 0 would be used to represent points that are not included in any region, while numbers > 0 would be used to represent the different regions.

The representation of regions as binary images has one obvious drawback: it needs to store (sometimes very many) points that are not included in the region. Furthermore, the representation is not particularly efficient: we need to store at least 1 bit for every point in the image. Often, the representation actually uses 1 byte per point because it is much easier to access bytes than bits. This representation is also not particularly efficient for runtime purposes: to determine which points are included in the region, we need to perform a test for every point in the binary image. In addition, it is a little awkward to store regions that extend to negative coordinates as binary images, which also leads to cumbersome algorithms. Finally, the representation of multiple regions as label images leads to the fact that overlapping regions cannot be represented, which will cause problems if morphological operations are performed on the regions. Therefore, a representation that only stores the points included in a region in an efficient manner would be very useful.

Table 9.1 shows a small example region. We first note that, either horizontally or vertically, there are extended runs in which adjacent pixels belong to the region. This is typically the case for most regions. We can use this property and store only the necessary data for each run. Since images are typically stored line by line in memory, it is better to use horizontal runs. Therefore, the minimum amount of data for each run is the row coordinate of the run and the start and end columns of the run. This method of storing a region is called a run-length representation or run-length encoding. With this representation, the example region can be stored with just four runs, as shown in Table 9.1. Consequently, the region can also be

Table 9.1 Run-length representation of a region.



Run	Row	Start column	End column
1	1	1	4
2	2	2	2
3	2	4	5
4	3	2	5

regarded as the union of all of its runs:

$$R = \bigcup_{i=1}^{n} \mathbf{r}_i \tag{9.2}$$

Here, $\mathbf{r}_i$ denotes a single run, which can also be regarded as a region. Note that the runs are stored sorted in lexicographic order according to their row and start column coordinates. This means that there is an order of the runs $\mathbf{r}_i = (r_i, cs_i, ce_i)$ in R defined by: $\mathbf{r}_i \prec \mathbf{r}_j \Leftrightarrow r_i < r_j \vee r_i = r_j \wedge cs_i < cs_j$. This order is crucial for the execution speed of algorithms that use run-length encoded regions.

In the above example, the binary image could be stored with 35 bytes if 1 byte per pixel is used or with 5 bytes if 1 bit per pixel is used. If the coordinates of the region are stored as 2-byte integers, the region can be represented with 24 bytes in the run-length representation. This is already a saving, albeit a small one, compared to binary images stored with 1 byte per pixel, but no saving if the binary image is stored as compactly as possible with 1 bit per pixel. To get an impression of how much this representation really saves, we can note that we are roughly storing the boundary of the region in the run-length representation. On average, the number of points on the boundary of the region will be proportional to the square root of the area of the region. Therefore, we can typically expect a very significant saving from the run-length representation compared to binary images, which must at least store every pixel in the surrounding rectangle of the region. For example, a full rectangular ROI of a $w \times h$ image can be stored with h runs instead of $w \times h$ pixels in a binary image (i.e., wh or $\lceil w/8 \rceil h$ bytes, depending on whether 1 byte or 1 bit per pixel is used). Similarly, a circle with diameter d can be stored with d runs as opposed to at least $d \times d$ pixels. We can see that the run-length representation often leads to an enormous reduction in memory consumption. Furthermore, since this representation only stores the points actually contained in the region, we do not need to perform a test to see whether a point lies in the region or not. These two features can save a significant amount of execution time. Also, with this representation it is straightforward to have regions with negative coordinates. Finally, to represent multiple regions, lists or arrays of run-length encoded regions can be used. Since in this case each region is treated separately, overlapping regions do not pose any problems.

9.1.3 Subpixel-Precise Contours

The data structures we have considered so far are pixel-precise. Often, it is important to extract subpixel-precise data from an image because the application

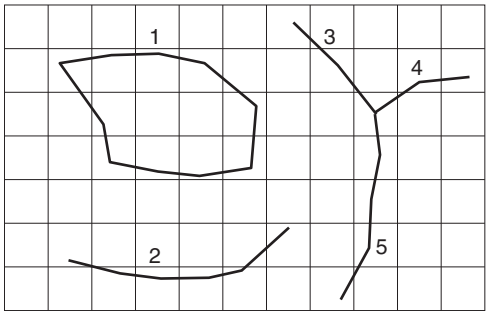


Figure 9.1 Different subpixel-precise contours. Contour 1 is a closed contour, while contours 2–5 are open contours. Contours 3–5 meet at a junction point.

requires an accuracy that is higher than the pixel resolution of the image. The subpixel data can, for example, be extracted with subpixel thresholding (see Section 9.4.3) or subpixel edge extraction (see Section 9.7.3). The results of these operations can be described with subpixel-precise contours. Figure 9.1 displays several example contours. As we can see, the contours can basically be represented as a polygon, that is, an ordered set of control points (r_i, c_i), where the ordering defines which control points are connected to each other. Since the extraction typically is based on the pixel grid, the distance between the control points of the contour is approximately 1 pixel on average. In the computer, the contours are simply represented as arrays of floating-point row and column coordinates. From Figure 9.1, we can also see that there is a rich topology associated with the contours. For example, contours can be closed (contour 1) or open (contours 2–5). Closed contours are usually represented by having the first contour point identical to the last contour point or by a special attribute that is stored with the contour. Furthermore, we can see that several contours can meet at a junction point, for example, contours 3–5. It is sometimes useful to explicitly store this topological information with the contours.

9.2 Image Enhancement

In the preceding chapters, we have seen that we have various means at our disposal to obtain a good image quality. The illumination, lenses, cameras, and frame grabbers (if used) all play a crucial role here. However, although we try very hard to select the best possible hardware setup, sometimes the image quality is not sufficient. Therefore, in this section we will take a look at several common techniques for image enhancement.

9.2.1 Gray Value Transformations

Despite our best efforts in controlling the illumination, in some cases it is necessary to modify the gray values of the image. One of the reasons for this may be a weak contrast. With controlled illumination, this problem usually only occurs locally. Therefore, we may only need to increase the contrast locally. Another possible reason for adjusting the gray values may be that the contrast or brightness of the image has changed from the settings that were in effect when we set up

our application. For example, illuminations typically age and produce a weaker contrast after some time.

A gray value transformation can be regarded as a point operation. This means that the transformed gray value $t_{r,c}$ depends only on the gray value $g_{r,c}$ in the input image at the same position: $t_{r,c} = f(g_{r,c})$. Here, $f(g)$ is a function that defines the gray value transformation to apply. Note that the domain and range of $f(g)$ typically are $\mathbb{G}_b$, that is, they are discrete. Therefore, to increase the transformation speed, gray value transformations can be implemented as a look-up table (LUT) by storing the output gray value for each possible input gray value in a table. If we denote the LUT as f_g, we have $t_{r,c} = f_g[g_{r,c}]$, where the [ ] operator denotes the table look-up.

The most important gray value transformation is a linear gray value scaling: $f(g) = ag + b$. If $g \in \mathbb{G}_b$, we need to ensure that the output value is also in $\mathbb{G}_b$. Hence, we must clip and round the output gray value as follows:

$$f(g) = \min(\max(\lfloor ag + b + 0.5 \rfloor, 0),\ 2^b - 1) \tag{9.3}$$

For $|a| > 1$, the contrast is increased, while for $|a| < 1$ the contrast is decreased. If $a < 0$, the gray values are inverted. For $b > 0$, the brightness is increased, while for $b < 0$ the brightness is decreased.

Figure 9.2a shows a small part of an image of a printed circuit board (PCB). The entire image was acquired such that the full range of gray values is used. Three components are visible in the image. As we can see, the contrast of the components is not as good as it could be. Figures 9.2b–e show the effect of applying

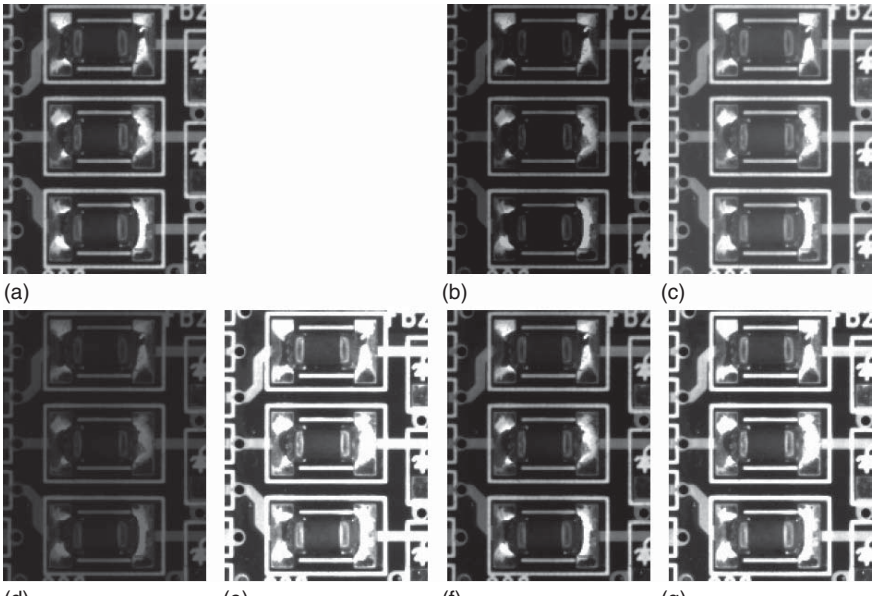


Figure 9.2 Examples of linear gray value transformations. (a) Original image. (b) Decreased brightness ($b = -50$). (c) Increased brightness ($b = 50$). (d) Decreased contrast ($a = 0.5$). (e) Increased contrast ($a = 2$). (f) Gray value normalization. (g) Robust gray value normalization ($p_l = 0, p_u = 0.8$).

a linear gray value transformation with different values for a and b. As we can see from Figure 9.2e, the component can be seen more clearly for $a = 2$.

The parameters of the linear gray value transformation must be selected appropriately for each application and adapted to changed illumination conditions. Since this can be quite cumbersome, ideally we would like to have a method that selects a and b automatically based on the conditions in the image. One obvious method to do this is to select the parameters such that the maximum range of the gray value space $\mathbb{G}_b$ is used. This can be done as follows: let $g_{\min}$ and $g_{\max}$ be the minimum and maximum gray value in the ROI under consideration. Then, the maximum range of gray values will be used if $a = (2^b - 1)/(g_{\max} - g_{\min})$ and $b = -ag_{\min}$. This transformation can be thought of as a normalization of the gray values. Figure 9.2f shows the effect of the gray value normalization of the image in Figure 9.2a. As we can see, the contrast is not much better than in the original image. This happens because there are specular reflections on the solder, which have the maximum gray value, and because there are very dark parts in the image with a gray value of almost 0. Hence, there is not much room to improve the contrast.

The problem with the gray value normalization is that a single pixel with a very bright or dark gray value can prevent us from using the desired gray value range. To get a better understanding of this point, we can take a look at the gray value histogram of the image. The gray value histogram is defined as the frequency with which a particular gray value occurs. Let n be the number of points in the ROI under consideration, and n_i be the number of pixels that have the gray value i. Then, the gray value histogram is a discrete function with domain $\mathbb{G}_b$ that has the values

$$h_i = \frac{n_i}{n} \tag{9.4}$$

In probabilistic terms, the gray value histogram can be regarded as the probability density of the occurrence of gray value i. We can also compute the cumulative histogram of the image as follows:

$$c_i = \sum_{j=0}^{i} h_j \tag{9.5}$$

This corresponds to the probability distribution of the gray values. Figure 9.3 shows the histogram and cumulative histogram of the image in Figure 9.2a. We can see that the specular reflections on the solder create a peak in the histogram at gray value 255. Furthermore, we can see that the smallest gray value in the image is 16. This explains why the gray value normalization did not increase the contrast significantly. We can also see that the dark part of the gray value range contains the most information about the components, while the bright part contains the information corresponding to the specular reflections as well as the printed rectangles on the board. Therefore, to get a more robust gray value normalization, we can simply ignore a part of the histogram that includes a fraction p_l of the darkest gray values and a fraction $1 - p_u$ of the brightest gray values. This can easily be done based on the cumulative histogram by selecting the smallest gray value for which $c_i \geq p_l$ and the largest gray value for which $c_i \leq p_u$. Conceptually,

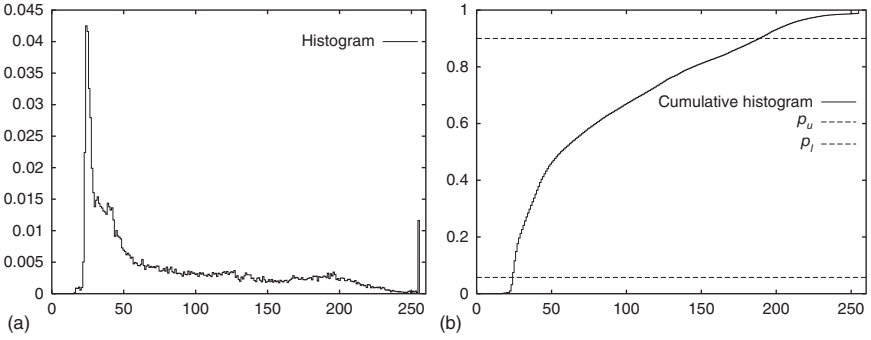


Figure 9.3 (a) Histogram of the image in Figure 9.2a. (b) Corresponding cumulative histogram with probability thresholds p_u and p_l superimposed.

this corresponds to intersecting the cumulative histogram with the lines $p = p_l$ and $p = p_u$. Figure 9.3b shows two example probability thresholds superimposed on the cumulative histogram. For the example image in Figure 9.2a, it is best to ignore only the bright gray values that correspond to the reflections and print on the board to get a robust gray value normalization. Figure 9.2g shows the result that is obtained with $p_l = 0$ and $p_u = 0.8$. As we can see, the contrast of the components is significantly improved.

The robust gray value normalization is an extremely powerful method that is used, for example, as a feature extraction method for optical character recognition (OCR; see Section 9.12), where it can be used to make the OCR features invariant to illumination changes. However, it requires transforming the gray values in the image, which is computationally expensive. If we want to make an algorithm robust to illumination changes, it is often possible to adapt the parameters to the changes in the illumination, for example, as described in Section 9.4.1 for the segmentation of images.

9.2.2 Radiometric Calibration

Many image processing algorithms rely on the fact that there is a linear correspondence between the energy that the sensor collects and the gray value in the image, namely $G = aE + b$, where E is the energy that falls on the sensor and G is the gray value in the image. Ideally, $b = 0$, which means that twice as much energy on the sensor leads to twice the gray value in the image. However, $b = 0$ is not necessary for measurement accuracy. The only requirement is that the correspondence is linear. If the correspondence is nonlinear, the accuracy of the results returned by these algorithms typically will degrade. Examples of this are the subpixel-precise threshold (see Section 9.4.3), the gray value features (see Section 9.5.2), and, most notably, subpixel-precise edge extraction (see Section 9.7, in particular Section 9.7.4). Unfortunately, sometimes the gray value correspondence is nonlinear, that is, either the camera or the analog frame grabber (if used) produces a nonlinear response to the energy. If this is the case, and we want to perform accurate measurements, we must determine the nonlinear response and invert it. If we apply the inverse response to the images,

(a) (b)

Figure 9.4 Examples of calibrated density targets that are traditionally used for radiometric calibration in laboratory settings. (a) Density step target (image acquired with a camera with linear response). (b) Twelve-patch ISO 14524 target (image simulated as if acquired with a camera with linear response).

the resulting images will have a linear response. The process of determining the inverse response function is known as *radiometric calibration*.

In laboratory settings, traditionally calibrated targets are used to perform the radiometric calibration. Figure 9.4 displays examples of target types that are commonly used. Consequently, the corresponding algorithms are called chart-based. The procedure is to measure the gray values in the different patches and to compare them to the known reflectance of the patches [3]. This yields a small number of measurements (e.g., 15 independent measurements in the target in Figure 9.4a and 12 in the target in Figure 9.4b), through which a function is fitted, for example, a gamma response function that includes gain and offset, given by

$$f(g) = (a + bg)^{\gamma} \tag{9.6}$$

There are several problems with this approach. First of all, it requires a very even illumination throughout the entire field of view in order to be able to determine the gray values of the patches correctly. For example, [3] requires less than 2% variation of the illuminance incident on the calibration target across the entire target. While this may be achievable in laboratory settings, it is much harder to achieve in a production environment, where the calibration often must be performed. Furthermore, effects like vignetting may lead to an apparent light drop-off toward the border, which also prevents the extraction of the correct gray values. This problem is always present, independent of the environment. Another problem is that there is a great variety of target layouts, and hence it is difficult to implement a general algorithm for finding the patches on the targets and to determine their correspondence to the true reflectances. In addition, the reflectances on the targets are often specified as a linear progression in density, which is related exponentially to the reflectance. For example, the targets in Figure 9.4 both have a linear density progression, that is, an exponential gray value progression. This means that the samples for the curve fitting are not evenly distributed, which can cause the fitted response to be less accurate in the parts of the curve that contain the samples with the larger spacing. Finally, the range of functions that can be modeled for the camera response is limited to the single function that is fitted through the data.

Because of the above problems, a radiometric calibration algorithm that does not require any calibration target is highly desirable. These algorithms are called chart-less radiometric calibration. They are based on taking several images of the same scene with different exposures. The exposure can be varied by changing the aperture stop of the lens or by varying the exposure time of the camera. Since the aperture stop can be set less accurately than the exposure time, and since the exposure time of most industrial cameras can be controlled very accurately in software, varying the exposure time is the preferred method of acquiring images with different exposures. The advantages of this approach are that no calibration targets are required and that they do not require an even illumination. Furthermore, the range of possible gray values can be covered with multiple images instead of a single image, as required by the algorithms that use calibration targets. The only requirement on the image content is that there should be no gaps in the histograms of the different images within the gray value range that each image covers. Furthermore, with a little extra effort, even overexposed (i.e., saturated) images can be handled.

To derive an algorithm for chart-less calibration, let us examine what two images with different exposures tell us about the response function. We know that the gray value G in the image is a nonlinear function r of the energy E that falls on the sensor during the exposure e [4]:

$$G = r(eE) \tag{9.7}$$

Note that e is proportional to the exposure time and also proportional to the area of the entrance pupil of the lens, that is, proportional to $(1/F)^2$, where F is the f-number of the lens. As described previously, in industrial applications we typically leave the aperture stop constant and vary the exposure time. Therefore, we can think of e as the exposure time.

The goal of the radiometric calibration is to determine the inverse response $q = r^{-1}$. The inverse response can be applied to an image via an LUT to achieve a linear response.

Now, let us assume that we have acquired two images with different exposures e_1 and e_2. Hence, we know that $G_1 = r(e_1E)$ and $G_2 = r(e_2E)$. By applying the inverse response q to both equations, we obtain $q(G_1) = e_1E$ and $q(G_2) = e_2E$. We can now divide the two equations to eliminate the unknown energy E, and obtain

$$\frac{q(G_1)}{q(G_2)} = \frac{e_1}{e_2} = e_{1,2} \tag{9.8}$$

As we can see, q depends only on the gray values in the images and on the ratio $e_{1,2}$ of the exposures, but not on the exposures e_1 and e_2 themselves. Equation (9.8) is the defining equation for all chart-less radiometric calibration algorithms.

One way to determine q based on equation (9.8) is to discretize q in a LUT. Thus, $q_i = q(G_i)$. To derive a linear algorithm to determine q, we can take logarithms on both sides of equation (9.8) to obtain $\log(q_1/q_2) = \log e_{1,2}$, that is, $\log(q_1) - \log(q_2) = \log e_{1,2}$ [4]. If we set $Q_i = \log(q_i)$ and $E_{1,2} = \log e_{1,2}$, each pixel in the image pair yields one linear equation for the inverse response function Q:

$$Q_1 - Q_2 = E_{1,2} \tag{9.9}$$

Hence, we obtain a linear equation system $AQ = E$, where Q is a vector of the LUT for the logarithmic inverse response function, while A is a matrix with 256 columns for byte images. Matrices A and E have as many rows as pixels in the image, for example, 307 200 for a 640×480 image. Therefore, this equation system is much too large to be solved in an acceptable time. To derive an algorithm that solves the equation system in an acceptable time, we can note that each row of the equation system has the following form:

$$(0 \quad \cdots \quad 0 \quad 1 \quad 0 \quad \cdots \quad 0 \quad -1 \quad 0 \quad \cdots \quad 0)\, Q = E_{1,2} \tag{9.10}$$

The indices of the 1 and -1 entries in the above equation are determined by the gray values in the first and second images. Note that each pair of gray values that occurs multiple times leads to several identical rows in A. Also note that $AQ = E$ is an overdetermined equation system, which can be solved through the normal equations $A^\top AQ = A^\top E$. This means that each row that occurs k times in A will have the weight k in the normal equations. The same behavior is obtained by multiplying the row (9.10) that corresponds to the gray value pair by $\sqrt{k}$ and to include that row only once in A. This typically reduces the number of rows in A from several hundred thousand to a few thousand, and thus makes the solution of the equation system feasible.

The simplest method to determine k is to compute the 2D histogram of the image pair. The 2D histogram determines how often gray value i occurs in the first image while gray value j occurs in the second image at the same position. Hence, for byte images, the 2D histogram is a 256×256 image in which the column coordinate indicates the gray value in the first image while the row coordinate indicates the gray value in the second image. It is obvious that the 2D histogram contains the required values of k. We will see examples of 2D histograms in the following.

Note that the discussion so far has assumed that the calibration is performed from a single image pair. It is, however, very simple to include multiple images in the calibration since additional images provide the same type of equations as in (9.10), and can thus simply be added to A. This makes it much easier to cover the entire range of gray values. Thus, we can start with a fully exposed image and successively reduce the exposure time until we reach an image in which the smallest possible gray values are assumed. We could even start with a slightly overexposed image to ensure that the highest gray values are assumed. However, in this case we have to take care that the overexposed (saturated) pixels are excluded from A because they violate the defining equation (9.8). This is a very tricky problem to solve in general, since some cameras exhibit a bizarre saturation behavior. Suffice it to say that for many cameras it is sufficient to exclude pixels with the maximum gray value from A.

Despite the fact that A has many more rows than columns, the solution Q is not uniquely determined because we cannot determine the absolute value of the energy E that falls onto the sensor. Hence, the rank of A is at most 255 for byte images. To solve this problem, we could arbitrarily require $q(255) = 255$, that is, scale the inverse response function such that the maximum gray value range is used. Since the equations are solved in a logarithmic space, it is slightly more convenient to require $q(255) = 1$ and to scale the inverse response to

the full gray value range later. With this, we obtain one additional equation of the form

$$(0 \quad \cdots \quad 0 \quad k)\, Q = 0 \tag{9.11}$$

To enforce the constraint $q(255) = 1$, the constant k must be chosen such that equation (9.11) has the same weight as the sum of all other equations (9.10), that is, $k = \sqrt{wh}$, where w and h are the width and height of the image.

Even with this normalization, we still face some practical problems. One problem is that, if the images contain very little noise, equation (9.10) can become decoupled, and hence do not provide a unique solution for Q. Another problem is that, if the possible range of gray values is not completely covered by the images, there are no equations for the range of gray values that are not covered. Hence, the equation system will become singular. Both problems can be solved by introducing smoothness constraints for Q, which couple the equations and enable an extrapolation of Q into the range of gray values that is not covered by the images. The smoothness constraints require that the second derivative of Q should be small. Hence, for byte images they lead to 254 equations of the form

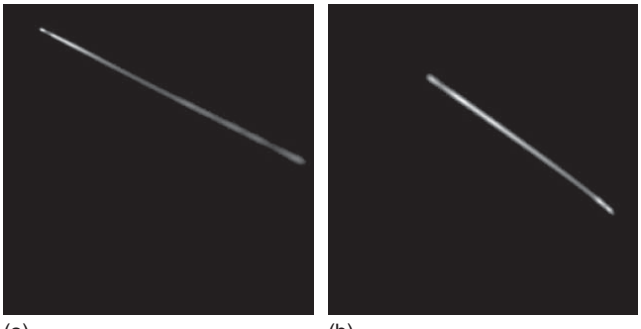
$$(0 \quad \cdots \quad 0 \quad s \quad -2s \quad s \quad 0 \quad \cdots \quad 0)\, Q = 0 \tag{9.12}$$

The parameter s determines the amount of smoothness that is required. Like for equation (9.11), s must be chosen such that equation (9.12) has the same weight as the sum of all the other equations, that is, $s = c\sqrt{wh}$, where c is a small number. Empirically, $c = 4$ works well for a wide range of cameras.

The approach of tabulating the inverse response q has two slight drawbacks. First of all, if the camera has a resolution of more than 8 bits, the equation system and 2D histograms will become very large. Second, the smoothness constraints will lead to straight lines in the logarithmic representation of q, that is, exponential curves in the normal representation of q in the range of gray values that is not covered by the images. Therefore, sometimes it may be preferable to model the inverse response as a polynomial, for example, as in [5]. This model also leads to linear equations for the coefficients of the polynomial. Since polynomials are not very robust in the extrapolation into areas in which no constraints exist, we also have to add smoothness constraints in this case by requiring that the second derivative of the polynomial is small. Because this is done in the original representation of q, the smoothness constraints will extrapolate straight lines into the gray value range that is not covered.

Let us now consider two cameras: one with a linear response, and one with a strong gamma response, that is, with a small γ in equation (9.6), and hence with a large γ in the inverse response q. Figure 9.5 displays the 2D histograms of two images taken with either camera with an exposure ratio of 0.5. Note that in both cases the values in the 2D histogram correspond to a line. The only difference is the slope of the line. A different slope, however, could also be caused by a different exposure ratio. Hence, we can see that it is quite important to know the exposure ratios precisely if we want to perform the radiometric calibration.

To conclude this section, we give two examples of the radiometric calibration. The first camera is a linear camera. Here, five images were acquired with exposure times of 32, 16, 8, 4, and 2 ms, as shown in Figure 9.6a. The calibrated



(a) (b)

Figure 9.5 (a) Two-dimensional histogram of two images taken with an exposure ratio of 0.5 with a linear camera. (b) Two-dimensional histogram of two images taken with an exposure ratio of 0.5 with a camera with a strong gamma response curve. For better visualization, the 2D histograms are displayed with a square root LUT. Note that in both cases the values in the 2D histogram correspond to a line. Hence, linear responses cannot be distinguished from gamma responses without knowing the exact exposure ratio.

inverse response curve is shown in Figure 9.6b. Note that the response is linear, but the camera has set a slight offset in the amplifier, which prevents very small gray values from being assumed. The second camera is a camera with a gamma response. In this case, six images were taken with exposure times of 30, 20, 10, 5, 2.5, and 1.25 ms, as shown in Figure 9.6c. The calibrated inverse response curve is shown in Figure 9.6d. Note the strong gamma response of the camera. The 2D histograms in Figure 9.5 were computed from the second and third brightest images in both sequences.

9.2.3 Image Smoothing

Every image contains some degree of noise. For the purposes of this chapter, noise can be regarded as random changes in the gray values, which occur for various reasons, for example, because of the randomness of the photon flux. In most cases, the noise in the image will need to be suppressed by using image smoothing operators.

In a more formalized manner, noise can be regarded as a stationary stochastic process [6]. This means that the true gray value $g_{r,c}$ is disturbed by noise $n_{r,c}$ to get the observed gray value: $\hat{g}_{r,c} = g_{r,c} + n_{r,c}$. We can regard the noise $n_{r,c}$ as a random variable with mean 0 and variance σ^2 for every pixel. We can assume a mean of 0 for the noise because any mean different from 0 would constitute a systematic bias of the observed gray values, which we could not detect anyway. "Stationary" means that the noise does not depend on the position in the image, that is, it is identically distributed for each pixel. In particular, σ^2 is assumed constant throughout the image. The last assumption is a convenient abstraction that does not necessarily hold because the variance of the noise sometimes depends on the gray values in the image. However, we will assume that the noise is always stationary.

Figure 9.7 shows an image of an edge from a real application. The noise is clearly visible in the bright patch in Figure 9.7a and in the horizontal gray value profile

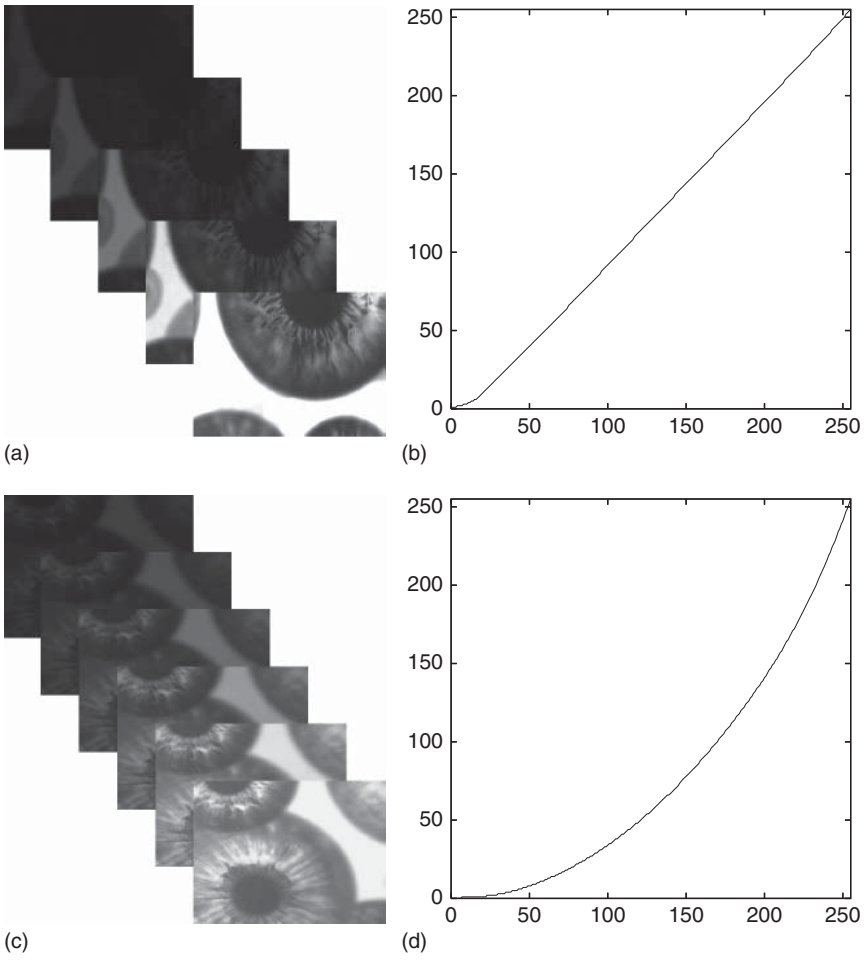


Figure 9.6 (a) Five images taken with a linear camera with exposure times of 32, 16, 8, 4, and 2 ms. (b) Calibrated inverse response curve. Note that the response is linear, but the camera has set a slight offset in the amplifier, which prevents very small gray values from being assumed. (c) Six images taken with a camera with a gamma response with exposure times of 30, 20, 10, 5, 2.5, and 1.25 ms. (d) Calibrated inverse response curve. Note the strong gamma response of the camera.

in Figure 9.7b. Figure 9.7c,d shows the actual noise in the image. How the noise has been calculated is explained below. It can be seen that there is slightly more noise in the dark patch of the image.

With the above discussion in mind, noise suppression can be regarded as a stochastic estimation problem, that is, given the observed noisy gray values $\hat{g}_{r,c}$, we want to estimate the true gray values $g_{r,c}$. An obvious method to reduce the noise is to acquire multiple images of the same scene and to simply average these images. Since the images are taken at different times, we will refer to this method as temporal averaging or the temporal mean. If we acquire n images, the temporal

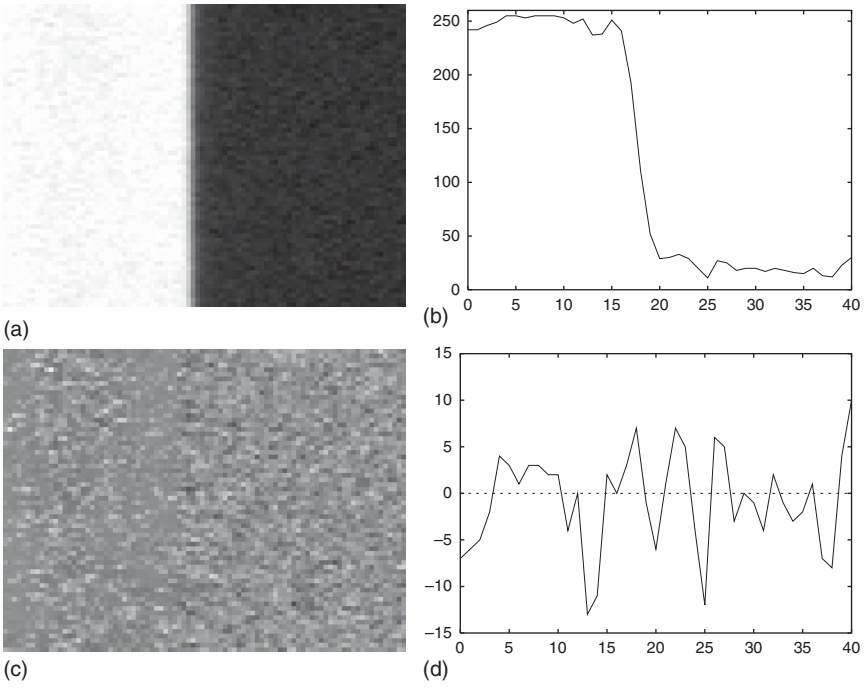


Figure 9.7 (a)Image of an edge. (b) Horizontal gray value profile through the center of the image. (c) Noise in (a) scaled by a factor of 5. (d) Horizontal gray value profile of the noise.

average is given by

$$g_{r,c} = \frac{1}{n} \sum_{i=1}^{n} \hat{g}_{r,c;i} \tag{9.13}$$

where $\hat{g}_{r,c;i}$ denotes the noisy gray value at position (r, c) in image i. This approach is frequently used in X-ray inspection systems, which inherently produce quite noisy images. From probability theory [6], we know that the variance of the noise is reduced by a factor of n by this estimation: $\sigma_m^2 = \sigma^2/n$. Consequently, the standard deviation of the noise is reduced by a factor of $\sqrt{n}$. Figure 9.8 shows the result of acquiring 20 images of an edge and computing the temporal average. Compared to Figure 9.7a, which shows one of the 20 images, the noise has been reduced by a factor of $\sqrt{20} \approx 4.5$, as can be seen from Figure 9.8b. Since this temporally averaged image is a very good estimate for the true gray values, we can subtract it from any of the images that were used in the averaging to obtain the noise in that image. This is how the image in Figure 9.7c was computed.

One of the drawbacks of the temporal averaging is that we have to acquire multiple images to reduce the noise. This is not very attractive if the speed of the application is important. Therefore, other means for reducing the noise are required in most cases. Ideally, we would like to use only one image to estimate the true gray value. If we turn to the theory of stochastic processes again, we see that the temporal averaging can be replaced with a spatial averaging if the

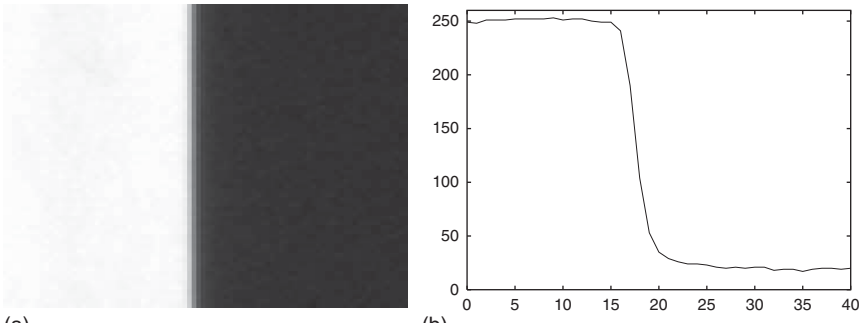


Figure 9.8 (a) Image of an edge obtained by averaging 20 images of the edge. (b) Horizontal gray value profile through the center of the image.

stochastic process, that is, the image, is ergodic [6]. This is precisely the definition of ergodicity, and we will assume for the moment that it holds for our images. Then, the spatial average or spatial mean can be computed over a window (also called mask) of $(2n + 1) \times (2m + 1)$ pixels as follows:

$$g_{r,c} = \frac{1}{(2n+1)(2m+1)} \sum_{i=-n}^{n} \sum_{j=-m}^{m} \hat{g}_{r-i,c-j} \tag{9.14}$$

This spatial averaging operation is also called a mean filter. Like in the case of temporal averaging, the noise variance is reduced by a factor that corresponds to the number of measurements that are used to calculate the average, that is, by $(2n + 1)(2m + 1)$. Figure 9.9 shows the result of smoothing the image of Figure 9.7 with a 5×5 mean filter. The standard deviation of the noise is reduced by a factor of 5, which is approximately the same as the temporal averaging in Figure 9.8. However, we can see that the edge is no longer as sharp as with temporal averaging. This happens, of course, because the images are not ergodic in general, but are only in areas of constant intensity. Therefore, in contrast to the temporal mean, the spatial mean filter blurs edges.

In equation (9.14), we have ignored the fact that the image has a finite extent. Therefore, if the mask is close to the image border, it will partially stick out of the

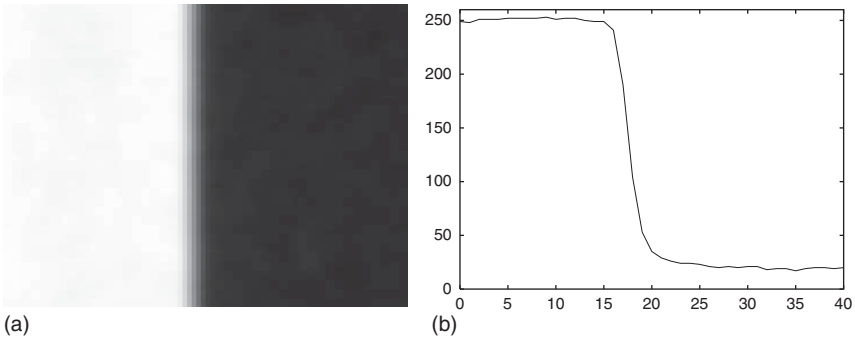


Figure 9.9 (a) Image of an edge obtained by smoothing the image of Figure 9.7a with a 5×5 mean filter. (b) Horizontal gray value profile through the center of the image.

image, and consequently will access undefined gray values. To solve this problem, several approaches are possible. A very simple approach is to calculate the filter only for pixels for which the mask lies completely within the image. This means that the output image is smaller than the input image, which is not very helpful if multiple filtering operations are applied in sequence. We could also define that the gray values outside the image are 0. For the mean filter, this would mean that the result of the filter would become progressively darker as the pixels get closer to the image border. This is also not desirable. Another approach would be to use the closest gray value on the image border for pixels outside the image. This approach would still create unwanted edges at the image border. Therefore, typically the gray values are mirrored at the image border. This creates the least amount of artifacts in the result.

As was mentioned earlier, noise reduction from a single image is preferable for reasons of speed. Therefore, let us take a look at the number of operations involved in the calculation of the mean filter. If the mean filter is implemented based on equation (9.14), the number of operations will be $(2n + 1)(2m + 1)$ for each pixel in the image, that is, the calculation will have the complexity $O(whmn)$, where w and h are the width and height of the image, respectively. For $w = 640$, $h = 480$, and $m = n = 5$ (i.e., an 11×11 filter), the algorithm will perform 37 171 200 additions and 307 200 divisions. This is quite a substantial number of operations, so we should try to reduce the operation count as much as possible. One way to do this is to use the associative law of addition of real numbers as follows:

$$g_{r,c} = \frac{1}{(2n+1)(2m+1)} \sum_{i=-n}^{n} \left(\sum_{j=-m}^{m} \hat{g}_{r-i,c-j} \right) \tag{9.15}$$

This may seem like a trivial observation, but if we look closer we can see that the term in parentheses only needs to be computed once and can be stored, for example, in a temporary image. Effectively, this means that we are first computing the sums in the column direction of the input image, save them in a temporary image, and then compute the sums in the row direction of the temporary image. Hence, the double sum in equation (9.14) of complexity $O(nm)$ is replaced by two sums of total complexity $O(n + m)$. Consequently, the complexity drops from $O(whmn)$ to $O(wh(m + n))$. With the above numbers, now only 6 758 400 additions are required. The above transformation is so important that it has its own name. Whenever a filter calculation allows a decomposition into separate row and column sums, the filter is called separable. It is obviously of great advantage if a filter is separable, and it is often the best speed improvement that can be achieved. In this case, however, it is not the best we can do. Let us take a look at the column sum, that is, the part in parentheses in equation (9.15), and let the result of the column sum be denoted by $t_{r,c}$. Then, we have

$$t_{r,c} = \sum_{j=-m}^{m} \hat{g}_{r,c-j} = t_{r,c-1} + \hat{g}_{r,c+m} - \hat{g}_{r,c-m-1} \tag{9.16}$$

That is, the sum at position (r, c) can be computed based on the already computed sum at position $(r, c - 1)$ with just two additions. The same holds, of course, also

for the row sums. The result of this is that we need to compute the complete sum only once for the first column or row, and can then update it very efficiently. With this, the total complexity is $O(wh)$. Note that the mask size does not influence the runtime in this implementation. Again, since this kind of transformation is so important, it has a special name. Whenever a filter can be implemented with this kind of updating scheme based on previously computed values, it is called a recursive filter. For the above example, the mean filter requires just 1 238 880 additions for the entire image. This is more than a factor of 30 faster for this example than the naive implementation based on equation (9.14). Of course, the advantage becomes even bigger for larger mask sizes.

In the above discussion, we have called the process of spatial averaging a mean filter without defining what is meant by the word "filter." We can define a filter as an operation that takes a function as input and produces a function as output. Since images can be regarded as functions (see Section 9.1.1), for our purposes a filter transforms an image into another image.

The mean filter is an instance of a linear filter. Linear filters are characterized by the following property: applying a filter to a linear combination of two input images yields the same result as applying the filter to the two images and then computing the linear combination. If we denote the linear filter by h, and the two images by f and g, we have

$$h\{af(p) + bg(p)\} = ah\{f(p)\} + bh\{g(p)\} \tag{9.17}$$

where $p = (r, c)$ denotes a point in the image and the $\{\ \}$ operator denotes the application of the filter. Linear filters can be computed by a convolution. For a one-dimensional (1D) function on a continuous domain, the convolution is given by

$$f * h = (f * h)(x) = \int_{-\infty}^{\infty} f(t)\ h(x - t)\ dt \tag{9.18}$$

Here, f is the image function and the filter h is specified by another function called the convolution kernel or the filter mask. Similarly, for 2D functions we have

$$f * h = (f * h)(r, c) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(u, v)\ h(r - u, c - v)\ du\ dv \tag{9.19}$$

For functions with discrete domains, the integrals are replaced by sums:

$$f * h = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f_{i,j}\ h_{r-i,c-j} \tag{9.20}$$

The integrals and sums are formally taken over an infinite domain. Of course, to be able to compute the convolution in a finite amount of time, the filter $h_{r,c}$ must be 0 for sufficiently large r and c. For example, the mean filter is given by

$$h_{r,c} = \begin{cases} \dfrac{1}{(2n+1)(2m+1)}, & |r| \le n \wedge |c| \le m \\ 0, & \text{otherwise} \end{cases} \tag{9.21}$$

The notion of separability can be extended for arbitrary, linear filters. If $h(r, c)$ can be decomposed as $h(r, c) = s(r)t(c)$ (or as $h_{r,c} = s_r t_c$), then h is called

separable. As for the mean filter, we can factor out s in this case to get a more efficient implementation:

$$\begin{aligned} f * h &= \sum_{i=-n}^{n} \sum_{j=-n}^{n} f_{i,j}\, h_{r-i,c-j} = \sum_{i=-n}^{n} \sum_{j=-n}^{n} f_{i,j}\, s_{r-i}\, t_{c-j} \\ &= \sum_{i=-n}^{n} s_{r-i} \left(\sum_{j=-n}^{n} f_{i,j}\, t_{c-j} \right) \end{aligned} \tag{9.22}$$

Obviously, separable filters have the same speed advantage as the separable implementation of the mean filter. Therefore, separable filters are preferred over nonseparable filters. There is also a definition for recursive linear filters, which we cannot cover in detail. The interested reader is referred to [7]. Recursive linear filters have the same speed advantage as the recursive implementation of the mean filter, that is, the runtime does not depend on the filter size. Unfortunately, many interesting filters cannot be implemented as recursive filters. Usually they can only be approximated by a recursive filter.

Although the mean filter produces good results, it is not the optimum smoothing filter. To see this, we can note that noise primarily manifests itself as high-frequency fluctuations of the gray values in the image. Ideally, we would like a smoothing filter to remove these high-frequency fluctuations. To see how well the mean filter performs this task, we can examine how the mean filter responds to certain frequencies in the image. The theory of how to do this is provided by the Fourier transform (see Section 9.2.4). Figure 9.10a shows the frequency response of a 3×3 mean filter. In this plot, the row and column coordinates of 0 correspond to a frequency of 0, which represents the medium gray value in the image, while the row and column coordinates of ± 0.5 represent the highest possible frequencies in the image. For example, the frequencies

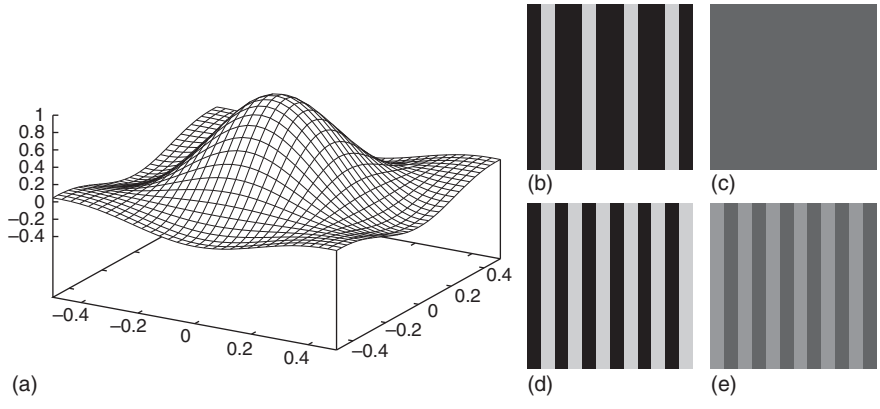


Figure 9.10 (a) Frequency response of the 3×3 mean filter. (b) Image with one-pixel-wide lines spaced three pixels apart. (c) Result of applying the 3×3 mean filter to the image in (b). Note that all the lines have been smoothed out. (d) Image with one-pixel-wide lines spaced two pixels apart. (e) Result of applying the 3×3 mean filter to the image in (d). Note that the lines have not been completely smoothed out, although they have a higher frequency than the lines in (b). Note also that the polarity of the lines has been reversed.

with column coordinate 0 and row coordinate ± 0.5 correspond to a grid with alternating one-pixel-wide vertical bright and dark lines. From Figure 9.10a, we can see that the 3×3 mean filter removes certain frequencies completely. These are the points for which the response has a value of 0. They occur for relatively high frequencies. However, we can also see that the highest frequencies are not removed completely. To illustrate this, Figure 9.10b shows an image with one-pixel-wide lines spaced three pixels apart. From Figure 9.10c, we can see that this frequency is completely removed by the 3×3 mean filter: the output image has a constant gray value. If we change the spacing of the lines to two pixels, as in Figure 9.10d, we can see from Figure 9.10e that this higher frequency is not removed completely. This is an undesirable behavior since it means that noise is not removed completely by the mean filter. Note also that the polarity of the lines has been reversed by the mean filter, which is also undesirable. This is caused by the negative parts of the frequency response. Furthermore, from Figure 9.10a we can see that the frequency response of the mean filter is not rotationally symmetric, that is, it is anisotropic. This means that diagonal structures are smoothed differently than horizontal or vertical structures.

Because the mean filter has the above drawbacks, the question of which smoothing filter is optimal arises. One way to approach this problem is to define certain natural criteria that the smoothing filter should fulfill, and then to search for the filters that fulfill the desired criteria. The first natural criterion is that the filter should be linear. This is natural because we can imagine an image being composed of multiple objects in an additive manner. Hence, the filter output should be a linear combination of the input. Furthermore, the filter should be position-invariant, that is, it should produce the same results no matter where an object is in the image. This is automatically fulfilled for linear filters. Also, we would like the filter to be rotation-invariant, that is, isotropic, so that it produces the same result independent of the orientation of the objects in the image. As we saw previously, the mean filter does not fulfill this criterion. We would also like to control the amount of smoothing (noise reduction) that is being performed. Therefore, the filter should have a parameter t that can be used to control the smoothing, where higher values of t indicate more smoothing. For the mean filter, this corresponds to the mask sizes m and n. We have already seen that the mean filter does not suppress all high frequencies, that is, noise, in the image. Therefore, a criterion that describes the noise suppression of the filter in the image should be added. One such criterion is that, the larger t gets, the more local maxima in the image should be eliminated. This is a very intuitive criterion, as can be seen in Figure 9.7a, where many local maxima due to noise can be detected. Note that, because of linearity, we only need to require maxima to be eliminated. This automatically implies that local minima are eliminated as well. Finally, sometimes we would like to execute the smoothing filter several times in succession. If we do this, we would also have a simple means to predict the result of the combined filtering. Therefore, first filtering with t and then with s should be identical to a single filter operation with $t + s$. It can be shown that, among all smoothing filters, the Gaussian filter is the only one that fulfills all of the above criteria [8]. Other natural criteria for a smoothing filter have been proposed [9–11], which also single out the Gaussian filter as the optimal smoothing filter.

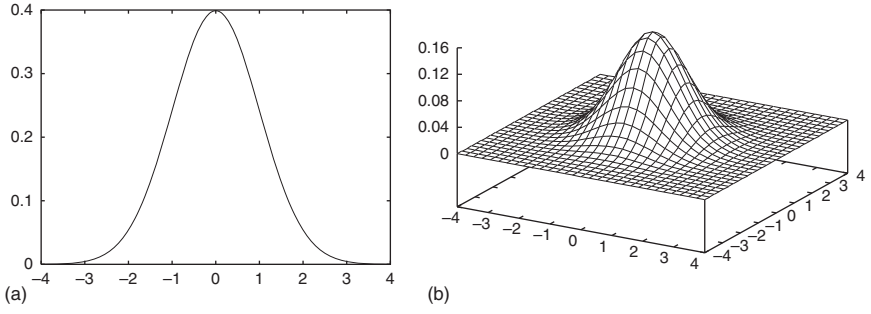


Figure 9.11 (a) One-dimensional Gaussian filter with $\sigma = 1$. (b) Two-dimensional Gaussian filter with $\sigma = 1$.

In one dimension, the Gaussian filter is given by

$$g_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma}\, e^{-x^2/(2\sigma^2)} \tag{9.23}$$

This is the function that also defines the probability density of a normally distributed random variable. In two dimensions, the Gaussian filter is given by

$$\begin{aligned} g_\sigma(r,c) &= \frac{1}{2\pi\sigma^2}\, e^{-(r^2+c^2)/(2\sigma^2)} = \frac{1}{\sqrt{2\pi}\sigma}\, e^{-r^2/(2\sigma^2)}\, \frac{1}{\sqrt{2\pi}\sigma}\, e^{-c^2/(2\sigma^2)} \\ &= g_\sigma(r) g_\sigma(c) \end{aligned} \tag{9.24}$$

Hence, the Gaussian filter is separable. Therefore, it can be computed very efficiently. In fact, it is the only isotropic, separable smoothing filter. Unfortunately, it cannot be implemented recursively. However, some recursive approximations have been proposed [12, 13]. Figure 9.11 shows plots of the 1D and 2D Gaussian filters with $\sigma = 1$. The frequency response of a Gaussian filter is also a Gaussian function, albeit with σ inverted. Therefore, Figure 9.11b also gives a qualitative impression of the frequency response of the Gaussian filter. It can be seen that the Gaussian filter suppresses high frequencies much better than the mean filter.

Like the mean filter, any linear filter will change the variance of the noise in the image. It can be shown that, for a linear filter $h(r,c)$ or $h_{r,c}$, the noise variance is multiplied by the following factor [6]:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(r,c)^2\, dr\, dc \quad \text{or} \quad \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} h_{r,c}^2 \tag{9.25}$$

For a Gaussian filter, this factor is $1/(4\pi\sigma^2)$. If we compare this to a mean filter with a square mask with parameter n, we see that, to get the same noise reduction with the Gaussian filter, we need to set $\sigma = (2n+1)/(2\sqrt{\pi})$. For example, a 5×5 mean filter has the same noise reduction effect as a Gaussian filter with $\sigma \approx 1.41$.

Figure 9.12 compares the results of the Gaussian filter with those of the mean filter of an equivalent size. For small filter sizes ($\sigma = 1.41$ and 5×5), there is hardly any noticeable difference between the results. However, if larger filter sizes are used, it becomes clear that the mean filter turns the edge into a ramp, leading to a badly defined edge that is also visually quite hard to locate, whereas the

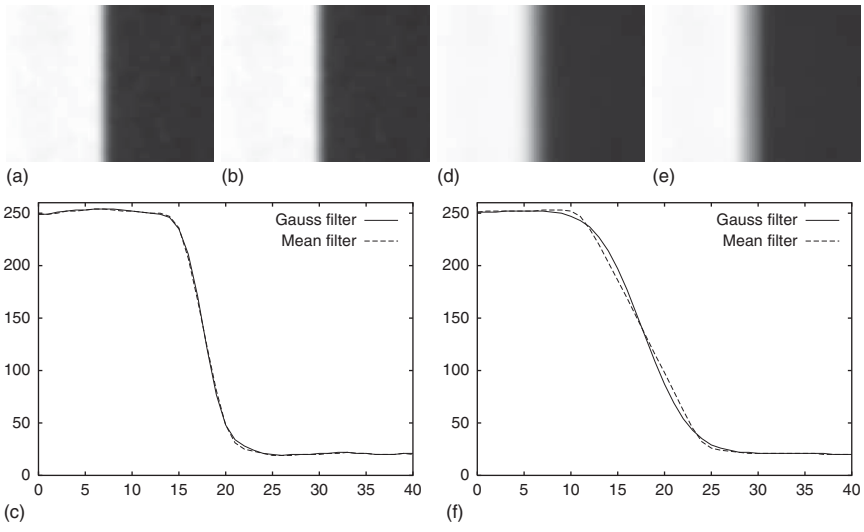


Figure 9.12 Images of an edge obtained by smoothing the image of Figure 9.7a. Results of (a) a Gaussian filter with $\sigma = 1.41$ and (b) a mean filter of size 5×5; and (c) the corresponding gray value profiles. Note that the two filters return very similar results in this example. Results of (d) a Gaussian filter with $\sigma = 3.67$ and (e) a 13×13 mean filter; and (f) the corresponding profiles. Note that the mean filter turns the edge into a ramp, leading to a badly defined edge, whereas the Gaussian filter produces a much sharper edge.

Gaussian filter produces a much sharper edge. Hence, we can see that the Gaussian filter produces better results, and consequently it is usually the preferred smoothing filter if the quality of the results is the primary concern. If speed is the primary concern, then the mean filter is preferable.

We close this section with a nonlinear filter that can also be used for noise suppression. The mean filter is a particular estimator for the mean value of a sample of random values. From probability theory, we know that other estimators are also possible, most notably the median of the samples. The median is defined as the value for which 50% of the values in the probability distribution of the samples are smaller than the median and 50% are larger. From a practical point of view, if the sample set contains n values g_i, $i = 0, \ldots, n-1$, we sort the values g_i in ascending order to get s_i, and then select the value median $(g_i) = s_{n/2}$. Hence, we can obtain a median filter by calculating the median instead of the mean inside a window around the current pixel. Let W denote the window, for example, a $(2n+1) \times (2m+1)$ rectangle as for the mean filter. Then the median filter is given by

$$g_{r,c} = \underset{(i,j) \in W}{\text{median}}\, \hat{g}_{r-i,c-j} \tag{9.26}$$

With sophisticated algorithms, it is possible to obtain a runtime complexity (even for arbitrary mask shapes) that is comparable to that of a separable linear filter: $O(whm)$, where m is the number of horizontal boundary pixels of the mask, that is, the pixels that are at the left or right border of a run of pixels in the

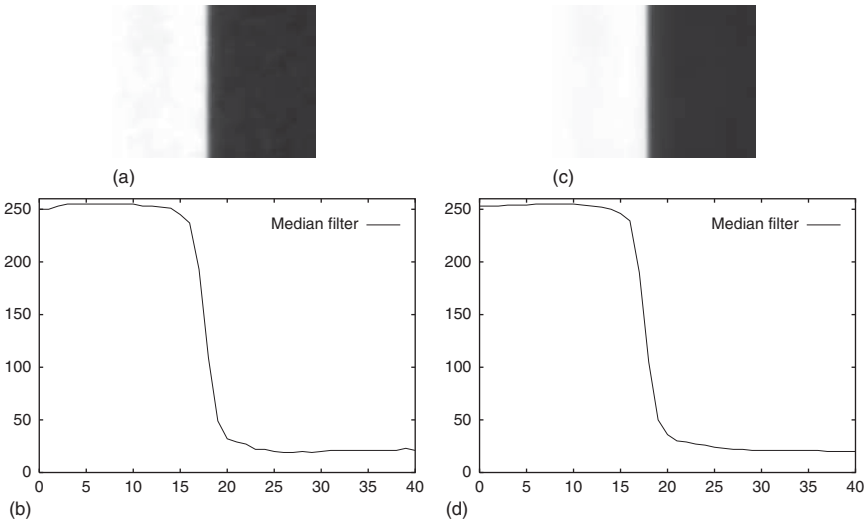


Figure 9.13 Images of an edge obtained by smoothing the image of Figure 9.7a. (a) Result with a median filter of size 5×5, and (b) the corresponding gray value profile. (c) Result of a 13×13 median filter, and (d) the corresponding profile. Note that the median filter preserves the sharpness of the edge to a great extent.

mask [14, 15]. For rectangular masks, it is possible to construct an algorithm with constant runtime per pixel (analogous to a recursive implementation of a linear filter) [16]. The properties of the median filter are quite difficult to analyze. We can note, however, that it performs no averaging of the input gray values, but simply selects one of them. This can lead to surprising results. For example, the result of applying a 3×3 median filter to the image in Figure 9.10b would be a completely black image. Hence, the median filter would remove the bright lines because they cover less than 50% of the window. This property can sometimes be used to remove objects completely from the image. On the other hand, applying a 3×3 median filter to the image in Figure 9.10d would swap the bright and dark lines. This result is as undesirable as the result of the mean filter on the same image. On the edge image of Figure 9.7a, the median filter produces quite good results, as can be seen from Figure 9.13. In particular, it should be noted that the median filter preserves the sharpness of the edge even for large filter sizes. However, it cannot be predicted if and how much the position of the edge is changed by the median filter, which is possible for the linear filters. Furthermore, we cannot estimate how much noise is removed by the median filter, in contrast to the linear filters. Therefore, for high-accuracy measurements the Gaussian filter should be used.

Finally, it should be mentioned that the median filter is a special case of the more general class of rank filters. Instead of selecting the median $s_{n/2}$ of the sorted gray values, the rank filter would select the sorted gray value at a particular rank r, that is, s_r. We will see other cases of rank operators in Section 9.6.2.

9.2.4 Fourier Transform

In the previous section, we considered the frequency responses of the mean and Gaussian filters. In this section, we will take a look at the theory that is used to derive the frequency response: the Fourier transform [17, 18]. The Fourier transform of a 1D function $h(x)$ is given by

$$H(f) = \int_{-\infty}^{\infty} h(x) e^{2\pi i f x}\, dx \tag{9.27}$$

It transforms the function $h(x)$ from the spatial domain into the frequency domain: that is, $h(x)$, a function of the position x, is transformed into $H(f)$, a function of the frequency f. Note that $H(f)$ is in general a complex number. Because of equation (9.27) and the identity $e^{ix} = \cos x + i \sin x$, we can think of $h(x)$ as being composed of sine and cosine waves of different frequencies and different amplitudes. Then $H(f)$ describes precisely which frequency occurs with which amplitude and with which phase (overlaying sine and cosine terms of the same frequency simply leads to a phase-shifted sine wave). The inverse Fourier transform from the frequency domain to the spatial domain is given by

$$h(x) = \int_{-\infty}^{\infty} H(f)\, e^{-2\pi i f x}\, df \tag{9.28}$$

Because the Fourier transform is invertible, it is best to think of $h(x)$ and $H(f)$ as being two different representations of the same function.

In two dimensions, the Fourier transform and its inverse are given by

$$\begin{aligned} H(u,v) &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(r,c)\, e^{2\pi i(ur+vc)}\, dr\, dc \\ h(r,c) &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} H(u,v)\, e^{-2\pi i(ur+vc)}\, du\, dv \end{aligned} \tag{9.29}$$

In image processing, $h(r,c)$ is an image, for which the position (r,c) is given in pixels. Consequently, the frequencies (u,v) are given in cycles per pixel.

Among the many interesting properties of the Fourier transform, probably the most interesting one is that a convolution in the spatial domain is transformed into a simple multiplication in the frequency domain: $(g * h)(r,c) \Leftrightarrow G(u,v)H(u,v)$, where the convolution is given by equation (9.19). Hence, a convolution can be performed by transforming the image and the filter into the frequency domain, multiplying the two results, and transforming the result back into the spatial domain.

Note that the convolution attenuates the frequency content $G(u,v)$ of the image $g(r,c)$ by the frequency response $H(u,v)$ of the filter. This justifies the analysis of the smoothing behavior of the mean and Gaussian filters that we have performed in Section 9.2.3. To make this analysis more precise, we can compute the Fourier transform of the mean filter in equation (9.21). It is given by

$$H(u,v) = \frac{1}{(2n+1)(2m+1)} \operatorname{sinc}((2n+1)u) \operatorname{sinc}((2m+1)v) \tag{9.30}$$

where $\operatorname{sinc} x = (\sin \pi x)/(\pi x)$. See Figure 9.10 for a plot of the response of the 3×3 mean filter. Similarly, the Fourier transform of the Gaussian filter in equation (9.24) is given by

$$H(u, v) = e^{-2\pi^2\sigma^2(u^2+v^2)} \tag{9.31}$$

Hence, the Fourier transform of the Gaussian filter is again a Gaussian function, albeit with σ inverted. Note that, in both cases, the frequency response becomes narrower if the filter size is increased. This is a relation that holds in general: $h(x/a) \Leftrightarrow |a|H(af)$.

Another interesting property of the Fourier transform is that it can be used to compute the correlation

$$g \star h = (g \star h)(r, c) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(r + u, c + v)\ h(u, v)\ du\ dv \tag{9.32}$$

Note that the correlation is very similar to the convolution in equation (9.19). The correlation is given in the frequency domain by $(g \star h)(r, c) \Leftrightarrow G(u, v)H(-u, -v)$. If $h(r, c)$ contains real numbers, which is the case for image processing, then $H(-u, -v) = \overline{H(u, v)}$, where the bar denotes complex conjugation. Hence, $(g \star h)(r, c) \Leftrightarrow G(u, v)\overline{H(u, v)}$.

Up to now, we have assumed that the images are continuous. Real images are, of course, discrete. This trivial observation has profound implications for the result of the Fourier transform. As noted above, the frequency variables u and v are given in cycles per pixel. If a discrete image $h(r, c)$ is transformed, the highest possible frequency for any sine or cosine wave is $1/2$, that is, one cycle per two pixels. The frequency $1/2$ is called the Nyquist critical frequency. Sine or cosine waves with higher frequencies look like sine or cosine waves with correspondingly lower frequencies. For example, a discrete cosine wave with frequency 0.75 looks exactly like a cosine wave with frequency 0.25, as shown in Figure 9.14. Effectively, values of $H(u, v)$ outside the square $[-0.5, 0.5] \times [-0.5, 0.5]$ are mapped to this square by repeated mirroring at the borders of the square. This effect is known as *aliasing*. To avoid aliasing, we must ensure that frequencies higher than the Nyquist critical frequency are removed before the image is

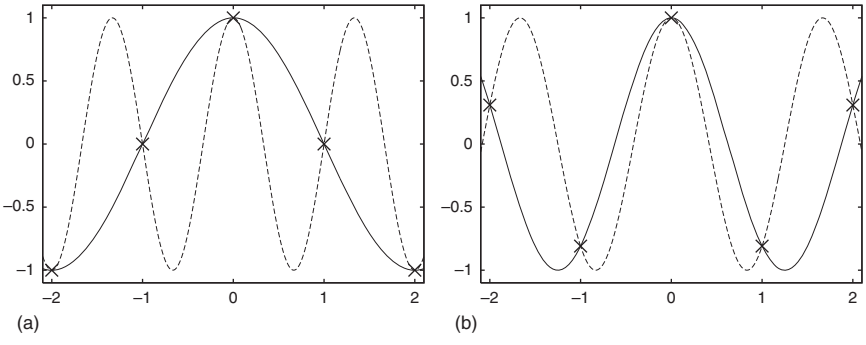


Figure 9.14 Example of aliasing. (a) Two cosine waves, one with a frequency of 0.25 and the other with a frequency of 0.75. (b) Two cosine waves, one with a frequency of 0.4 and the other with a frequency of 0.6. Note that if both functions are sampled at integer positions, denoted by the crosses, the discrete samples will be identical.

sampled. During the image acquisition, this can be achieved by optical low-pass filters in the camera. Aliasing, however, may also occur when an image is scaled down (see Section 9.3.3). Here, it is important to apply smoothing filters before the image is sampled at the lower resolution to ensure that frequencies above the Nyquist critical frequency are removed.

Real images are not only discrete, they are also only defined within a rectangle of dimension $w \times h$, where w is the image width and h is the image height. This means that the Fourier transform is no longer continuous, but can be sampled at discrete frequencies $u_k = k/h$ and $v_l = l/w$. As discussed previously, sampling the Fourier transform is useful only in the Nyquist interval $-1/2 \le u_k, v_l < 1/2$. With this, the discrete Fourier transform (DFT) is given by

$$\begin{aligned} H_{k,l} = H(u_k, v_l) &= \sum_{r=0}^{h-1} \sum_{c=0}^{w-1} h_{r,c}\, e^{2\pi i(u_k r + v_l c)} \\ &= \sum_{r=0}^{h-1} \sum_{c=0}^{w-1} h_{r,c}\, e^{2\pi i(kr/h + lc/w)} \end{aligned} \tag{9.33}$$

Analogously, the inverse DFT is given by

$$h_{r,c} = \frac{1}{wh} \sum_{k=0}^{h-1} \sum_{l=0}^{w-1} H_{k,l}\, e^{-2\pi i(kr/h + lc/w)} \tag{9.34}$$

As noted previously, conceptually, the frequencies u_k and v_l should be sampled from the interval $(-1/2, 1/2]$, that is, $k = -h/2 + 1, \ldots, h/2$ and $l = -w/2 + 1, \ldots, w/2$. Since we want to represent $H_{k,l}$ as an image, the negative coordinates are a little cumbersome. It is easy to see that equations (9.33) and (9.34) are periodic with period h and w. Therefore, we can map the negative frequencies to their positive counterparts: that is, $k = -h/2 + 1, \ldots, -1$ is mapped to $k = h/2 + 1, \ldots, h - 1$; and likewise for l.

We noted previously that for real images, $H(-u, -v) = \overline{H(u, v)}$. This property still holds for the DFT, with the appropriate change of coordinates as defined above, that is, $H_{h-k,w-l} = \overline{H_{k,l}}$ (for $k, l > 0$). In practice, this means that we do not need to compute and store the complete Fourier transform $H_{k,l}$ because it contains redundant information. It is sufficient to compute and store one half of $H_{k,l}$, for example, the left half. This saves a considerable amount of processing time and memory. This type of Fourier transform is called the real-valued Fourier transform.

To compute the Fourier transform from equations (9.33) and (9.34), it might seem that $O((wh)^2)$ operations are required. This would prevent the Fourier transform from being useful in image processing applications. Fortunately, the Fourier transform can be computed in $O(wh \log (wh))$ operations for $w = 2^n$ and $h = 2^m$ [18] as well as for arbitrary w and h [19]. This fast computation algorithm for the Fourier transform is aptly called the fast Fourier transform (FFT). With self-tuning algorithms [19], the FFT can be computed in real time on standard processors.

As discussed previously, the Fourier transform can be used to compute the convolution with any linear filter in the frequency domain. While this can be used to

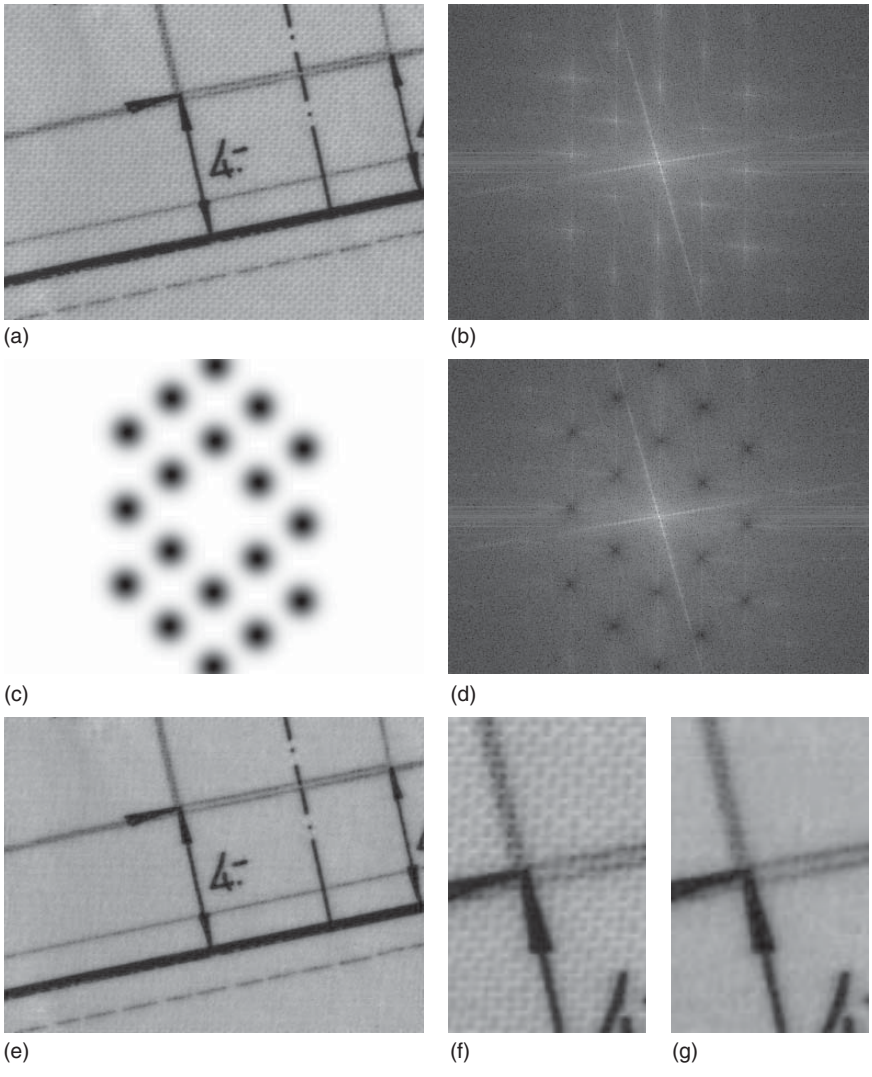


Figure 9.15 (a) Image of a map showing the texture of the paper. (b) Fourier transform of (a). Because of the high dynamic range of the result, $H_{u,v}^{1/16}$ is displayed. Note the distinct peaks in $H_{u,v}$, which correspond to the texture of the paper. (c) Filter $G_{u,v}$ used to remove the frequencies that correspond to the texture. (d) Convolution $H_{u,v}G_{u,v}$. (e) Inverse Fourier transform of (d). (f, g) Detail of (a) and (e), respectively. Note that the texture has been removed.

perform filtering with standard filter masks, for example, the mean or Gaussian filter, typically there is a speed advantage only for relatively large filter masks. The real advantage of using the Fourier transform for filtering lies in the fact that filters can be customized to remove specific frequencies from the image, which occur, for example, for repetitive textures.

Figure 9.15a shows an image of a map. The map is drawn on a highly structured paper that exhibits significant texture. The texture makes the extraction of the

data in the map difficult. Figure 9.15b displays the Fourier transform $H_{u,v}$. Note that the Fourier transform is cyclically shifted so that the zero frequency is in the center of the image to show the structure of the data more clearly. Hence, Figure 9.15b displays the frequencies u_k and v_l for $k = -h/2 + 1, \ldots, h/2$ and $l = -w/2 + 1, \ldots, w/2$. Because of the high dynamic range of the result, $H_{u,v}^{1/16}$ is displayed. It can be seen that $H_{u,v}$ contains several highly significant peaks that correspond to the characteristic frequencies of the texture. Furthermore, there are two significant orthogonal lines that correspond to the lines in the map. A filter $G_{u,v}$ that removes the characteristic frequencies of the texture is shown in Figure 9.15c, while the result of the convolution $H_{u,v}G_{u,v}$ in the frequency domain is shown in Figure 9.15d. The result of the inverse Fourier transform, that is, the convolution in the spatial domain, is shown in Figure 9.15e. Figure 9.15f,g shows details of the input and result images, which show that the texture of the paper has been removed. Thus, it is now very easy to extract the map data from the image.

9.3 Geometric Transformations

In many applications, one cannot ensure that the objects to be inspected are always in the same position and orientation in the image. Therefore, the inspection algorithm must be able to cope with these position changes. Hence, one of the problems is to detect the position and orientation, also called the pose, of the objects to be examined. This will be the subject of later sections of this chapter. For the purposes of this section, we assume that we know the pose already. In this case, the simplest procedure to adapt the inspection to a particular pose is to align the ROIs appropriately. For example, if we know that an object is rotated by 45°, we could simply rotate the ROI by 45° before performing the inspection. In some cases, however, the image must be transformed (aligned) to a standard pose before the inspection can be performed. For example, the segmentation of the OCR is much easier if the text is either horizontal or vertical. Another example is the inspection of objects for defects based on a reference image. Here, we also need to align the image of the object to the pose in the reference image, or vice versa. Therefore, in this section we will examine different geometric image transformations that are useful in practice.

9.3.1 Affine Transformations

If the position and rotation of the objects cannot be kept constant with the mechanical setup, we need to correct the rotation and translation of the object. Sometimes the distance of the object to the camera changes, leading to an apparent change in size of the object. These transformations are part of a very useful class of transformations called affine transformations, which are transformations that can be described by the following equation:

$$\begin{pmatrix} \tilde{r} \\ \tilde{c} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} r \\ c \end{pmatrix} + \begin{pmatrix} t_r \\ t_c \end{pmatrix} \tag{9.35}$$

Hence, an affine transformation consists of a linear part given by a 2×2 matrix and a translation. The above notation is a little cumbersome, however, since we

always have to list the translation separately. To circumvent this, we can use a representation where we extend the coordinates with a third coordinate of 1, which enables us to write the transformation as a simple matrix multiplication:

$$\begin{pmatrix} \tilde{r} \\ \tilde{c} \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r \\ c \\ 1 \end{pmatrix} \tag{9.36}$$

Note that the translation is represented by the elements a_{13} and a_{23} of the matrix A. This representation with an added redundant third coordinate is called homogeneous coordinates. Similarly, the representation with two coordinates in equation (9.35) is called inhomogeneous coordinates. We will see the true power of the homogeneous representation below. Any affine transformation can be constructed from the following basic transformations, where the last row of the matrix has been omitted:

$$\begin{pmatrix} 1 & 0 & t_r \\ 0 & 1 & t_c \end{pmatrix} \quad \text{Translation} \tag{9.37}$$

$$\begin{pmatrix} s_r & 0 & 0 \\ 0 & s_c & 0 \end{pmatrix} \quad \text{Scaling in row and column direction} \tag{9.38}$$

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \end{pmatrix} \quad \text{Rotation by an angle of } \alpha \tag{9.39}$$

$$\begin{pmatrix} \cos\theta & 0 & 0 \\ \sin\theta & 1 & 0 \end{pmatrix} \quad \text{Skew of row axis by an angle of } \theta \tag{9.40}$$

The first three basic transformations need no further explanation. The skew (or slant) is a rotation of only one axis, in this case the row axis. It is quite useful to rectify slanted characters in the OCR.

9.3.2 Projective Transformations

An affine transformation enables us to correct almost all relevant pose variations that an object may undergo. However, sometimes affine transformations are not general enough. If the object in question is able to rotate in three dimensions, it will undergo a general perspective transformation, which is quite hard to correct because of the occlusions that may occur. However, if the object is planar, we can model the transformation of the object by a 2D perspective transformation, which is a special 2D projective transformation [20, 21]. Projective transformations are given by

$$\begin{pmatrix} \tilde{r} \\ \tilde{c} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} r \\ c \\ w \end{pmatrix} \tag{9.41}$$

Note the similarity to the affine transformation in equation (9.36). The only changes that were made are that the transformation is now described by a full 3×3 matrix and that we have replaced the 1 in the third coordinate with a variable w. This representation is actually the true representation in homogeneous

coordinates. It can also be used for affine transformations, which are special projective transformations. With this third coordinate, it is not obvious how we are able to obtain a transformed 2D coordinate, that is, how to compute the corresponding inhomogeneous point. First of all, it must be noted that in homogeneous coordinates all points $p = (r, c, w)^\top$ are only defined up to a scale factor, that is, the vectors p and λp ($\lambda \neq 0$) represent the same 2D point [20, 21]. Consequently, the projective transformation given by the matrix H is also defined only up to a scale factor, and hence has only eight independent parameters. To obtain an inhomogeneous 2D point from the homogeneous representation, we must divide the homogeneous vector by w. This requires $w \neq 0$. Such points are called finite points. Conversely, points with $w = 0$ are called points at infinity because they can be regarded as lying infinitely far away in a certain direction [20, 21].

Since a projective transformation has eight independent parameters, it can be uniquely determined from four corresponding points [20, 21]. This is how the projective transformations will usually be determined in machine vision applications. We will extract four points in an image, which typically represent a rectangle, and will rectify the image so that the four extracted points will be transformed to the four corners of the rectangle, that is, to their corresponding points. Unfortunately, because of space limitations we cannot give the details of how the transformation is computed from the point correspondences. The interested reader is referred to [20, 21].

9.3.3 Image Transformations

After having taken a look at how coordinates can be transformed with affine and projective transformations, we can consider how an image should be transformed. Our first idea might be to go through all the pixels in the input image, to transform their coordinates, and to set the gray value of the transformed point in the output image. Unfortunately, this simple strategy does not work. This can be seen by checking what happens if an image is scaled by a factor of 2: only one-quarter of the pixels in the output image would be set. The correct way to transform an image is to loop through all the pixels in the output image and to calculate the position of the corresponding point in the input image. This is the simplest way to ensure that all relevant pixels in the output image are set. Fortunately, calculating the positions in the original image is simple: we only need to invert the matrix that describes the affine or projective transformation, which results again in an affine or projective transformation.

When the image coordinates are transformed from the output image to the input image, typically not all pixels in the output image transform back to coordinates that lie in the input image. This can be taken into account by computing a suitable ROI for the output image. Furthermore, we see that the resulting coordinates in the input image will typically not be integer coordinates. An example of this is given in Figure 9.16, where the input image is transformed by an affine transformation consisting of a translation, rotation, and scaling. Therefore, the gray values in the output image must be interpolated.

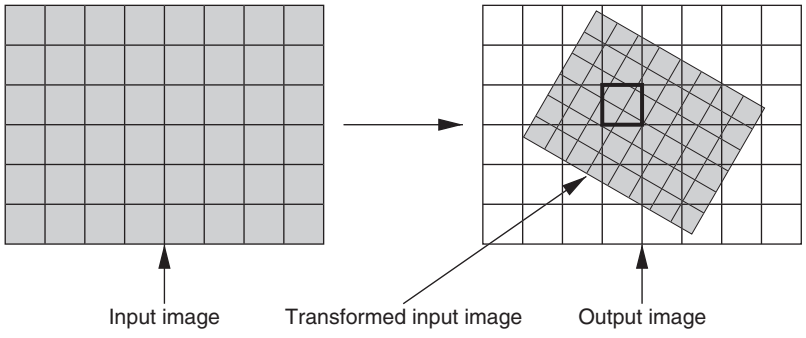


Figure 9.16 Affine transformation of an image. Note that integer coordinates in the output image transform to non-integer coordinates in the original image, and hence must be interpolated.

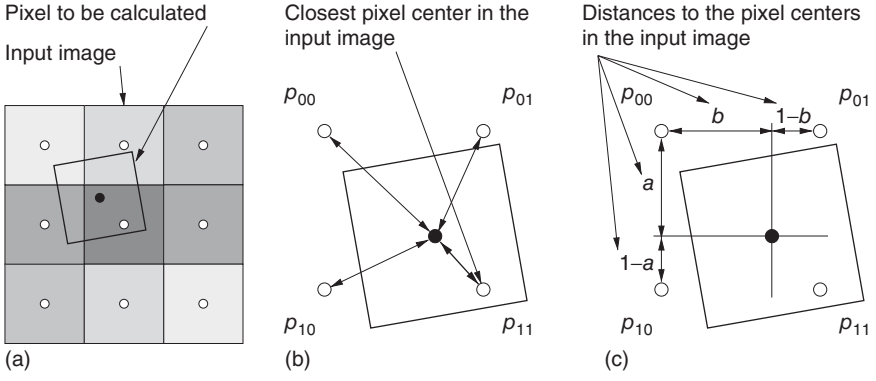


Figure 9.17 (a) A pixel in the output image is transformed back to the input image. Note that the transformed pixel center lies at a non-integer position between four adjacent pixel centers. (b) Nearest-neighbor interpolation determines the closest pixel center in the input image and uses its gray value in the output image. (c) Bilinear interpolation determines the distances to the four adjacent pixel centers and weights their gray values using the distances.

The interpolation can be done in several ways. Figure 9.17a displays a pixel in the output image that has been transformed back to the input image. Note that the transformed pixel center lies on a non-integer position between four adjacent pixel centers. The simplest and fastest interpolation method is to calculate the closest of the four adjacent pixel centers, which only involves rounding the floating-point coordinates of the transformed pixel center, and to use the gray value of the closest pixel in the input image as the gray value of the pixel in the output image, as shown in Figure 9.17b. This interpolation method is called nearest-neighbor interpolation. To see the effect of this interpolation, Figure 9.18a displays an image of a serial number of a bank note, where the characters are not horizontal. Figure 9.18c,d displays the result of rotating the image such that the serial number is horizontal using this interpolation. Note that, because the gray value is taken from the closest pixel center in the input image, the edges of the characters have a jagged appearance, which is undesirable.

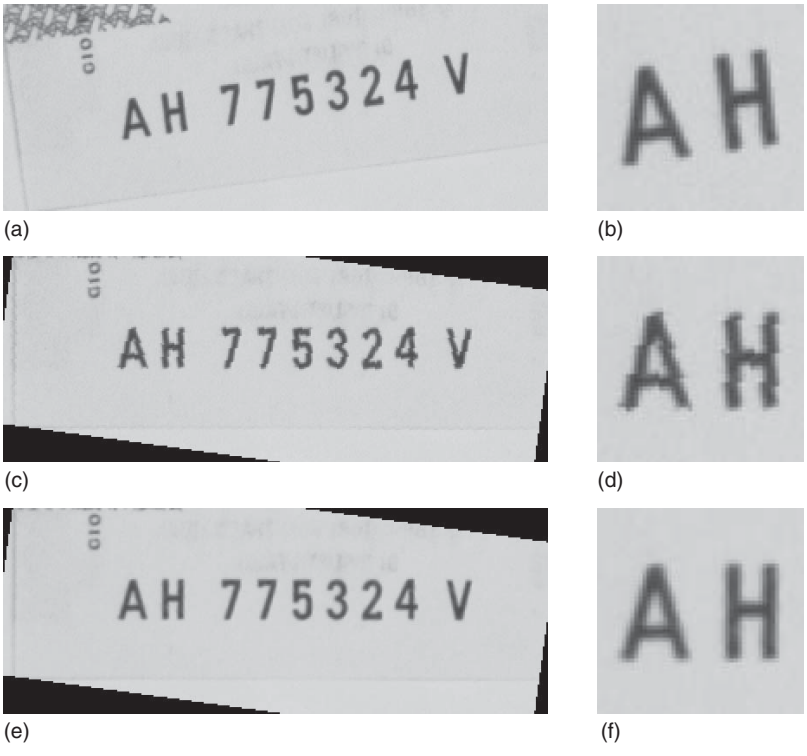


Figure 9.18 (a) Image showing a serial number of a bank note. (b) Detail of (a). (c) Image rotated such that the serial number is horizontal using nearest-neighbor interpolation. (d) Detail of (c). Note the jagged edges of the characters. (e) Image rotated using bilinear interpolation. (f) Detail of (e). Note the smooth edges of the characters.

The reason for the jagged appearance in the result of the nearest-neighbor interpolation is that essentially we are regarding the image as a piecewise constant function: every coordinate that falls within a rectangle of extent ± 0.5 in each direction is assigned the same gray value. This leads to discontinuities in the result, which cause the jagged edges. This behavior is especially noticeable if the image is scaled by a factor > 1. To get a better interpolation, we can use more information than the gray value of the closest pixel. From Figure 9.17a, we can see that the transformed pixel center lies in a square of four adjacent pixel centers. Therefore, we can use the four corresponding gray values and weight them appropriately. One way to do this is to use bilinear interpolation, as shown in Figure 9.17c. First, we compute the horizontal and vertical distances of the transformed coordinate to the adjacent pixel centers. Note that these are numbers between 0 and 1. Then, we weight the gray values according to their distances to get the bilinear interpolation:

$$\tilde{g} = b(ag_{11} + (1-a)g_{01}) + (1-b)(ag_{10} + (1-a)g_{00}) \tag{9.42}$$

Figure 9.18e,f displays the result of rotating the image of Figure 9.18a using bilinear interpolation. Note that the edges of the characters now have a very smooth

appearance. This much better result more than justifies the longer computation time (typically a factor of around 2).

To conclude the discussion on interpolation, we discuss the effects of scaling an image down. In the bilinear interpolation scheme, we would interpolate from the closest four pixel centers. However, if the image is scaled down, adjacent pixel centers in the output image will not necessarily be close in the input image. Imagine a larger version of the image of Figure 9.10b (one-pixel-wide vertical lines spaced three pixels apart) being scaled down by a factor of 4 using nearest-neighbor interpolation: we would get an image with one-pixel-wide lines that are four pixels apart. This is certainly not what we would expect. For bilinear interpolation, we would get similar unexpected results. If we scale down an image, we are essentially subsampling it. As a consequence, we may obtain aliasing effects (see Section 9.2.4). An example of aliasing can be seen in Figure 9.19. The image in Figure 9.19a is scaled down by a factor of 3 in Figure 9.19c using bilinear interpolation. Note that the stroke widths of the vertical strokes of the letter H, which are equally wide in Figure 9.19a, now appear to be substantially different. This is undesirable. To improve the image transformation, the image

Figure 9.19 (a) Image showing a serial number of a bank note. (b) Detail of (a). (c) The image of (a) scaled down by a factor of 3 using bilinear interpolation. (d) Detail of (c). Note the different stroke widths of the vertical strokes of the letter H. This is caused by aliasing. (e) Result of scaling the image down by integrating a smoothing filter (in this case a mean filter) into the image transformation. (f) Detail of (e).

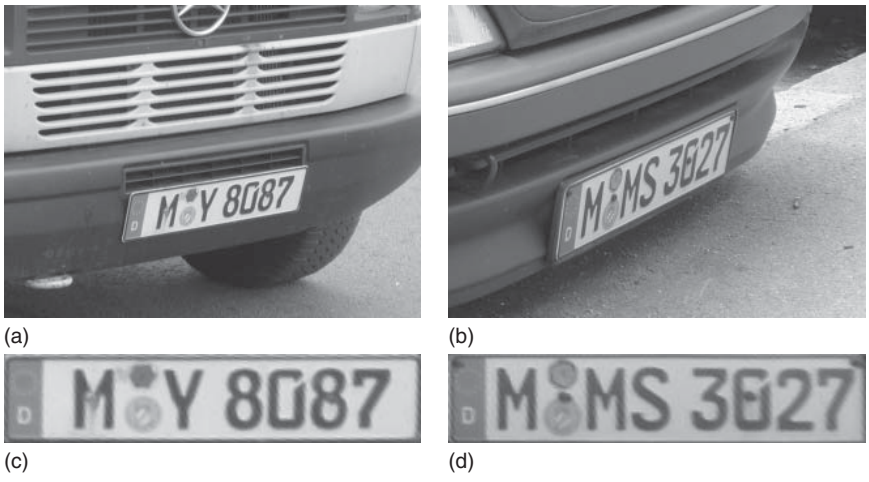


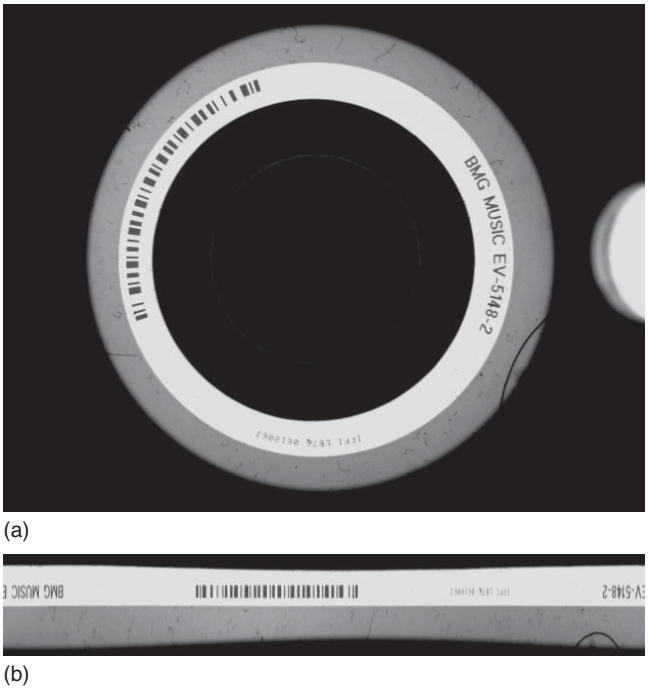
Figure 9.20 (a,b) Images of license plates. (c,d) Result of a projective transformation that rectifies the perspective distortion of the license plates.

must be smoothed before it is scaled down, for example, using a mean or a Gaussian filter. Alternatively, the smoothing can be integrated into the gray value interpolation. Figure 9.19e shows the result of integrating a mean filter into the image transformation. Because of the smoothing, the strokes of the H now have the same width again.

In the above examples, we have seen the usefulness of the affine transformations for rectifying text. Sometimes, an affine transformation is not sufficient for this purpose. Figure 9.20a,b shows two images of license plates on cars. Because the position of the camera with respect to the car could not be controlled in this example, the images of the license plates show severe perspective distortions. Figure 9.20c,d shows the result of applying projective transformations to the images that cut out the license plates and rectify them. Hence, the images in Figure 9.20c,d would result if we had looked at the license plates perpendicularly from in front of the car. Obviously, it is now much easier to segment and read the characters on the license plates.

9.3.4 Polar Transformations

We conclude this section with another very useful geometric transformation: the polar transformation. This transformation is typically used to rectify parts of images that show objects that are circular or that are contained in circular rings in the image. An example is shown in Figure 9.21a. Here, we can see the inner part of a CD that contains a ring with a bar code and some text. To read the bar code, one solution is to rectify the part of the image that contains the bar code. For this purpose, the polar transformation can be used, which converts the image into polar coordinates (d, ϕ), that is, into the distance d to the center of the transformation and the angle ϕ of the vector to the center of the transformation. Let the center of the transformation be given by (m_r, m_c). Then, the polar coordinates



(a)

(b)

Figure 9.21 (a) Image of the center of a CD showing a circular bar code. (b) Polar transformation of the ring that contains the bar code. Note that the bar code is now straight and horizontal.

of a point (r, c) are given by

$$\begin{aligned} d &= \sqrt{(r - m_r)^2 + (c - m_c)^2} \\ \phi &= \arctan\left(-\frac{r - m_r}{c - m_c}\right) \end{aligned} \tag{9.43}$$

In the calculation of the arc tangent function, the correct quadrant must be used, based on the sign of the two terms in the fraction in the argument of arctan. Note that the transformation of a point into polar coordinates is quite expensive to compute because of the square root and the arc tangent. Fortunately, to transform an image, like for affine and projective transformations, the inverse of the polar transformation is used, which is given by

$$\begin{aligned} r &= m_r - d \sin \phi \\ c &= m_c + d \cos \phi \end{aligned} \tag{9.44}$$

Here, the sines and cosines can be tabulated because they only occur for a finite number of discrete values, and hence only need to be computed once. Therefore, the polar transformation of an image can be computed efficiently. Note that by restricting the ranges of d and ϕ, we can transform arbitrary circular sectors. Figure 9.21b shows the result of transforming a circular ring that contains the bar code in Figure 9.21a. Note that because of the polar transformation the bar code is straight and horizontal, and consequently can be read easily.

9.4 Image Segmentation

In the preceding sections, we have looked at operations that transform an image into another image. These operations do not give us information about the objects in the image. For this purpose, we need to segment the image, that is, extract regions from the image that correspond to the objects we are interested in. More formally, segmentation is an operation that takes an image as input and returns one or more regions or subpixel-precise contours as output.

9.4.1 Thresholding

The simplest segmentation algorithm is to threshold the image. The threshold operation is defined by

$$S = \{(r, c) \in R \mid g_{\min} \leq f_{r,c} \leq g_{\max}\} \tag{9.45}$$

Hence, the threshold operation selects all points in the ROI R of the image that lie within a specified range of gray values into the output region S. Often, $g_{\min} = 0$ or $g_{\max} = 2^b - 1$ is used. If the illumination can be kept constant, the thresholds $g_{\min}$ and $g_{\max}$ are selected when the system is set up and are never modified. Since the threshold operation is based on the gray values themselves, it can be used whenever the object to be segmented and the background have significantly different gray values.

Figure 9.22a,b shows two images of integrated circuits (ICs) on a PCB with a rectangular ROI overlaid in light gray. The result of thresholding the two images with $g_{\min} = 90$ and $g_{\max} = 255$ is shown in Figure 9.22c,d. Since the illumination is kept constant, the same threshold works for both images. Note also that there are some noisy pixels in the segmented regions. They can be removed, for example,

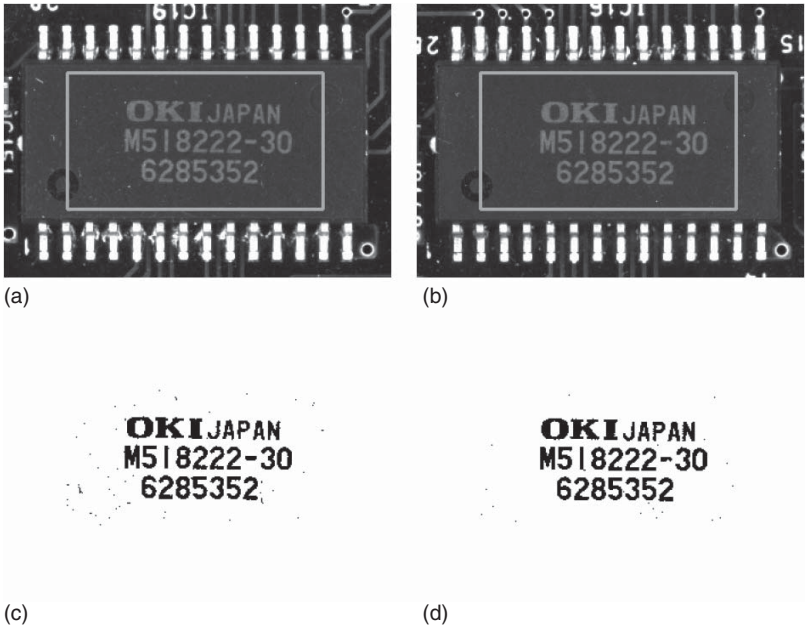


Figure 9.22 (a,b) Images of prints on ICs with a rectangular ROI overlaid in light gray. (c,d) Result of thresholding the images in (a),(b) with $g_{\min} = 90$ and $g_{\max} = 255$.

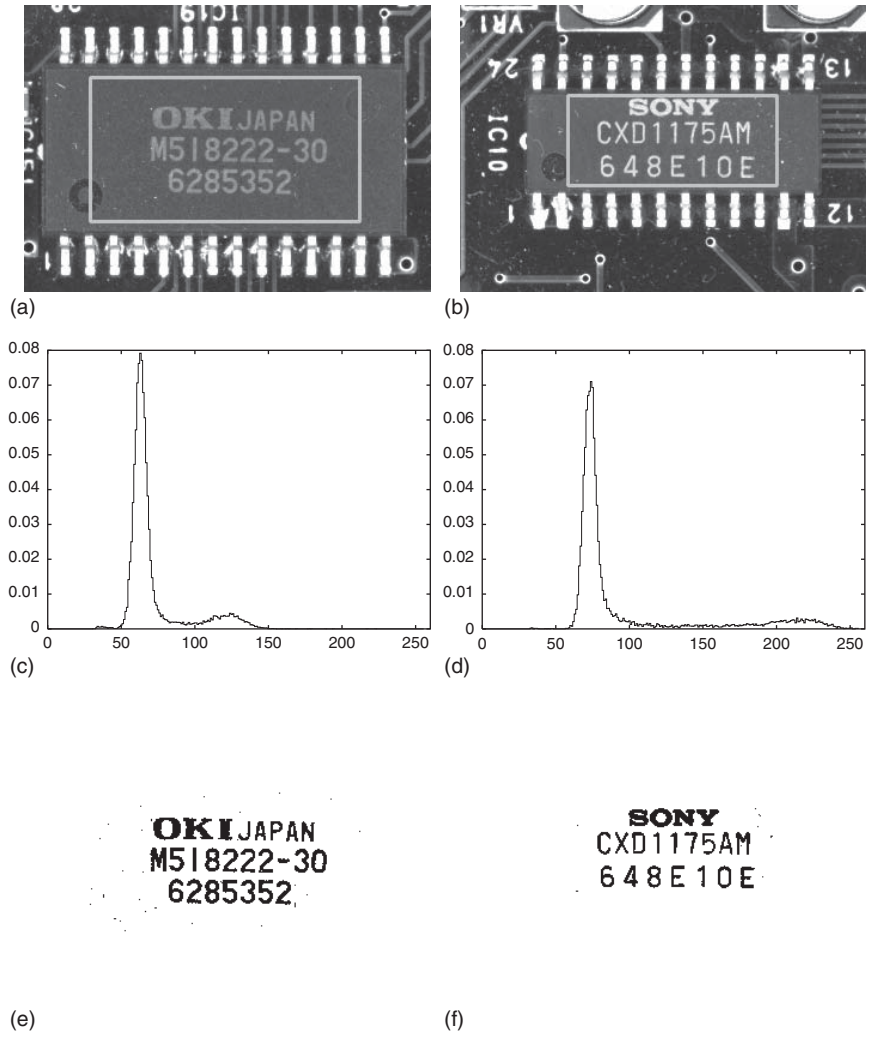


Figure 9.23 (a,b) Images of prints on ICs with a rectangular ROI overlaid in light gray. (c,d) Gray value histogram of the images in (a) and (b) within the respective ROI. (e,f) Result of thresholding the images in (a) and (b) with a threshold selected automatically based on the gray value histogram.

based on their area (see Section 9.5) or based on morphological operations (see Section 9.6).

The constant threshold works well only as long as the gray values of the object and the background do not change. Unfortunately, this occurs less frequently than one would wish, for example, because of changing illumination. Even if the illumination is kept constant, different gray value distributions on similar objects may prevent us from using a constant threshold. Figure 9.23 shows an example of this. In Figure 9.23a,b, two different ICs on the same PCB are shown. Despite the identical illumination, the prints have a substantially different gray value distribution, which will not allow us to use the same threshold for both images. Nevertheless, the print and the background can be separated easily in both cases. Therefore, ideally, we would like to have a method that is able to determine the

thresholds automatically. This can be done based on the gray value histogram of the image. Figure 9.23c,d shows the histograms of the images in Figure 9.23a,b. It is obvious that there are two relevant peaks (maxima) in the histograms in both images. The one with the smaller gray value corresponds to the background, while the one with the higher gray value corresponds to the print. Intuitively, a good threshold corresponds to the minimum between the two peaks in the histogram. Unfortunately, neither the two maxima nor the minimum is well defined because of random fluctuations in the gray value histogram. Therefore, to robustly select the threshold that corresponds to the minimum, the histogram must be smoothed, for example, by convolving it with a 1D Gaussian filter. Since it is not clear which σ to use, a good strategy is to smooth the histogram with progressively larger values of σ until two unique maxima with a unique minimum in between are obtained. The result of using this approach to select the threshold automatically is shown in Figure 9.23e,f. As can be seen, for both images suitable thresholds have been selected. This approach of selecting the thresholds is not the only approach. Other approaches are described, for example, in [22, 23]. All these approaches have in common that they are based on the gray value histogram of the image. One example of such a different approach is to assume that the gray values in the foreground and background each have a normal (Gaussian) probability distribution, and to jointly fit two Gaussian densities to the histogram. The threshold is then defined as the gray value for which the two Gaussian densities have equal probabilities. Another approach is to select the threshold by maximizing a measure of separability in gray values of the region and the background [24].

While calculating the thresholds from the histogram often works extremely well, it fails whenever the assumption that there are two peaks in the histogram is violated. One such example is shown in Figure 9.24. Here, the print is so noisy that the gray values of the print are extremely spread out, and consequently there is no discernible peak for the print in the histogram. Another reason for the failure of the desired peak to appear is an inhomogeneous illumination. This typically destroys the relevant peaks or moves them so that they are in the wrong location. An uneven illumination often even prevents us from using a threshold operation altogether because there are no fixed thresholds that work throughout the entire image. Fortunately, often the objects of interest can be characterized by being

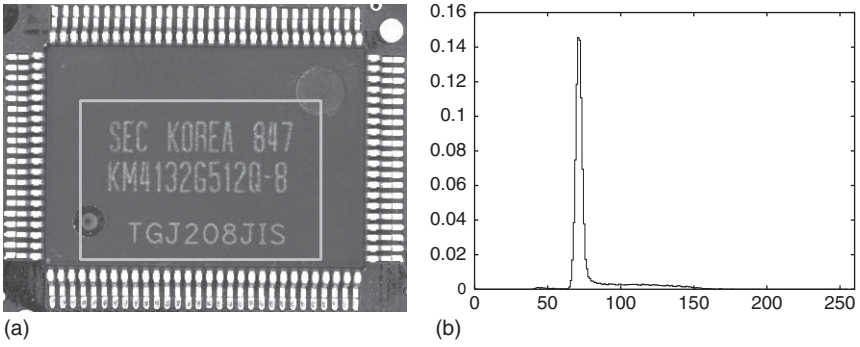


Figure 9.24 (a) Image of a print on an IC with a rectangular ROI overlaid in light gray. (b) Gray value histogram of the image in (a) within the ROI. Note that there are no significant minima and only one significant maximum in the histogram.

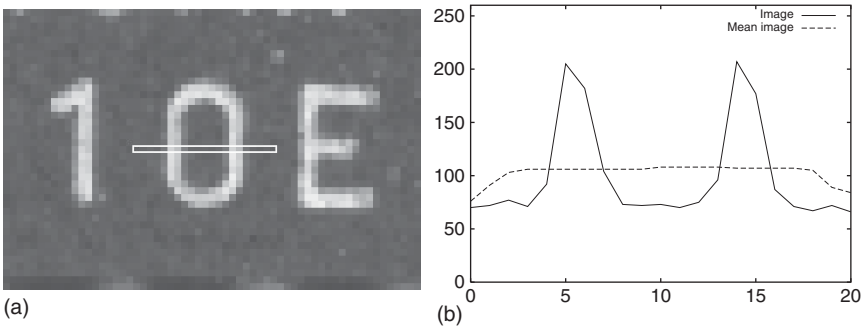


Figure 9.25 (a) Image showing a small part of a print on an IC with a one-pixel-wide horizontal ROI. (b) Gray value profiles of the image and the image smoothed with a 9 × 9 mean filter. Note that the text is substantially brighter than the local background estimated by the mean filter.

locally brighter or darker than their local background. The prints on the ICs we have examined so far are a good example of this. Therefore, instead of specifying global thresholds, we would like to specify by how much a pixel must be brighter or darker than its local background. The only problem we have is how to determine the gray value of the local background. Since a smoothing operation, for example, the mean, Gaussian, or median filter (see Section 9.2.3), calculates an average gray value in a window around the current pixel, we can simply use the filter output as an estimate of the gray value of the local background. The operation of comparing the image to its local background is called a dynamic thresholding operation. Let the image be denoted by $f_{r,c}$ and the smoothed image be denoted by $g_{r,c}$. Then, the dynamic thresholding operation for bright objects is given by

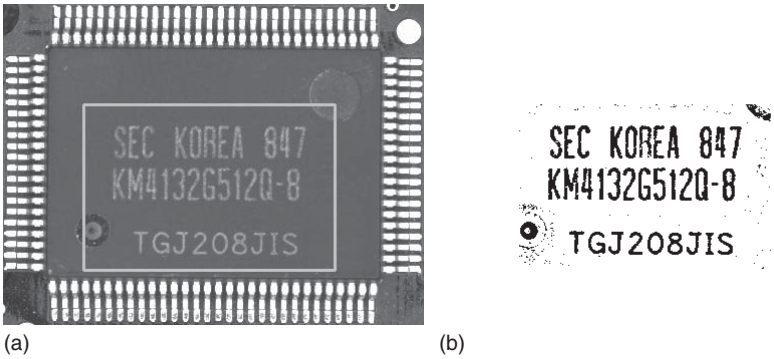
$$S = \{(r, c) \in R \mid f_{r,c} - g_{r,c} \geq g_{\text{diff}}\} \tag{9.46}$$

while the dynamic thresholding operation for dark objects is given by

$$S = \{(r, c) \in R \mid f_{r,c} - g_{r,c} \leq -g_{\text{diff}}\} \tag{9.47}$$

Figure 9.25 gives an example of how the dynamic thresholding works. In Figure 9.25a, a small part of a print on an IC with a one-pixel-wide horizontal ROI is shown. Figure 9.25b displays the gray value profiles of the image and the image smoothed with a 9×9 mean filter. It can be seen that the text is substantially brighter than the local background estimated by the mean filter. Therefore, the characters can be segmented easily with the dynamic thresholding operation.

In the dynamic thresholding operation, the size of the smoothing filter determines the size of the objects that can be segmented. If the filter size is too small, the local background will not be estimated well in the center of the objects. As a rule of thumb, the diameter of the mean filter must be larger than the diameter of the objects to be recognized. The same holds for the median filter. An analogous relation exists for the Gaussian filter. Furthermore, in general, if larger filter sizes are chosen for the mean and Gaussian filters, the filter output will be more representative of the local background. For example, for light objects the filter output will become darker within the light objects. For the median filter, this is not true since it will completely eliminate the objects if the filter mask is larger than the diameter of the objects. Hence, the gray values will be representative of the local



(a) (b)

Figure 9.26 (a) Image of a print on an IC with a rectangular ROI overlaid in light gray. (b) Result of segmenting the image in (a) with a dynamic thresholding operation with $g_{\text{diff}} = 5$ and a 31×31 mean filter.

background if the filter is sufficiently large. If the gray values in the smoothed image are more representative of the local background, we can typically select a larger threshold g_{diff}, and hence can suppress noise in the segmentation better. However, the filter mask cannot be chosen arbitrarily large because neighboring objects might adversely influence the filter output. Finally, it should be noted that the dynamic thresholding operation returns a segmentation result not only for objects that are brighter or darker than their local background but also at the bright or dark region around edges.

Figure 9.26a again shows the image of Figure 9.24a, which could not be segmented with an automatic threshold. In Figure 9.26b, the result of segmenting the image with a dynamic thresholding operation with $g_{\text{diff}} = 5$ is shown. The local background was obtained with a 31×31 mean filter. Note that the difficult print is segmented very well with the dynamic thresholding.

As described so far, the dynamic thresholding operation can be used to compare the image with its local background, which is obtained by smoothing the image. With a slight modification, the dynamic thresholding operation can also be used to detect errors in an object, for example, for print inspection. Here, the image $g_{r,c}$ is an image of the ideal object, that is, the object without errors; $g_{r,c}$ is called the reference image. To detect deviations from the ideal object, we can simply look for too bright or too dark pixels in the image $f_{r,c}$ by using equation (9.46) or (9.47). Often, we are not interested in whether the pixels are too bright or too dark, but simply in whether they deviate too much from the reference image, that is, the union of equations (9.46) and (9.47), which is given by

$$S = \{(r, c) \in R \mid |f_{r,c} - g_{r,c}| > g_{\text{abs}}\} \tag{9.48}$$

Note that this pixel-by-pixel comparison requires that the image $f_{r,c}$ of the object to check and the reference image $g_{r,c}$ are aligned very accurately to avoid spurious gray value differences that would be interpreted as errors. This can be ensured either by the mechanical setup or by finding the pose of the object in the current image, for example, using template matching (see Section 9.11), and then transforming the image to the pose of the object in the ideal image (see Section 9.3).

This kind of dynamic thresholding operation is very strict on the shape of the objects. For example, if the size of the object increases by half a pixel and the gray value difference between the object and the background is 200, the gray value difference between the current image and the model image will be 100 at the object's edges. This is a significant gray value difference, which would surely be larger than any reasonable g_{abs}. In real applications, however, small variations of the object's shape typically should be tolerated. On the other hand, small gray value changes in the areas where the object's shape does not change should still be recognized as an error. To achieve this behavior, we can introduce a thresholding operation that takes the expected gray value variations in the image into account. Let us denote the permissible variations in the image by $v_{r,c}$. Ideally, we would like to segment the pixels that differ from the reference image by more than the permissible variations:

$$S = \{(r,c) \in R \mid |f_{r,c} - g_{r,c}| > v_{r,c}\} \tag{9.49}$$

The permissible variations can be determined by learning them from a set of training images. For example, if we use n training images of objects with permissible variations, the standard deviation of the gray values of each pixel can be used to derive $v_{r,c}$. If we use n images to define the variations of the ideal object, we might as well use the mean of each pixel to define the reference image $g_{r,c}$ to reduce noise. Of course, the n training images must be aligned with sufficient accuracy. The mean and standard deviation of the n training images are given by

$$\begin{aligned} m_{r,c} &= \frac{1}{n}\sum_{i=1}^{n} g_{r,c;i} \\ s_{r,c} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n} (g_{r,c;i} - m_{r,c})^2} \end{aligned} \tag{9.50}$$

The images $m_{r,c}$ and $s_{r,c}$ model the reference image and the allowed variations of the reference image. Hence, we can call this approach a variation model. Note that $m_{r,c}$ is identical to the temporal average of equation (9.13). To define $v_{r,c}$, ideally we could simply set $v_{r,c}$ to a small multiple c of the standard deviation, that is, $v_{r,c} = cs_{r,c}$, where, for example, $c = 3$. Unfortunately, this approach does not work well if the variations in the training images are extremely small, for example, because the noise in the training images is significantly smaller than in the test images, or because parts of the object are near the saturation limit of the camera. In these cases, it is useful to introduce an absolute threshold a for the variation images, which is used whenever the variations in the training images are very small: $v_{r,c} = \max(a, cs_{r,c})$. As a further generalization, it is sometimes useful to have different thresholds for too bright and too dark pixels. With this, the variation threshold is no longer symmetric with respect to $m_{r,c}$, and we need to introduce two threshold images for the too bright and too dark pixels. If we denote the threshold images by $u_{r,c}$ and $l_{r,c}$, the absolute thresholds by a and b, and the factors for the standard deviations by c and d, the variation model segmentation is given by

$$S = \{(r,c) \in R \mid f_{r,c} < l_{r,c} \vee f_{r,c} > u_{r,c}\} \tag{9.51}$$

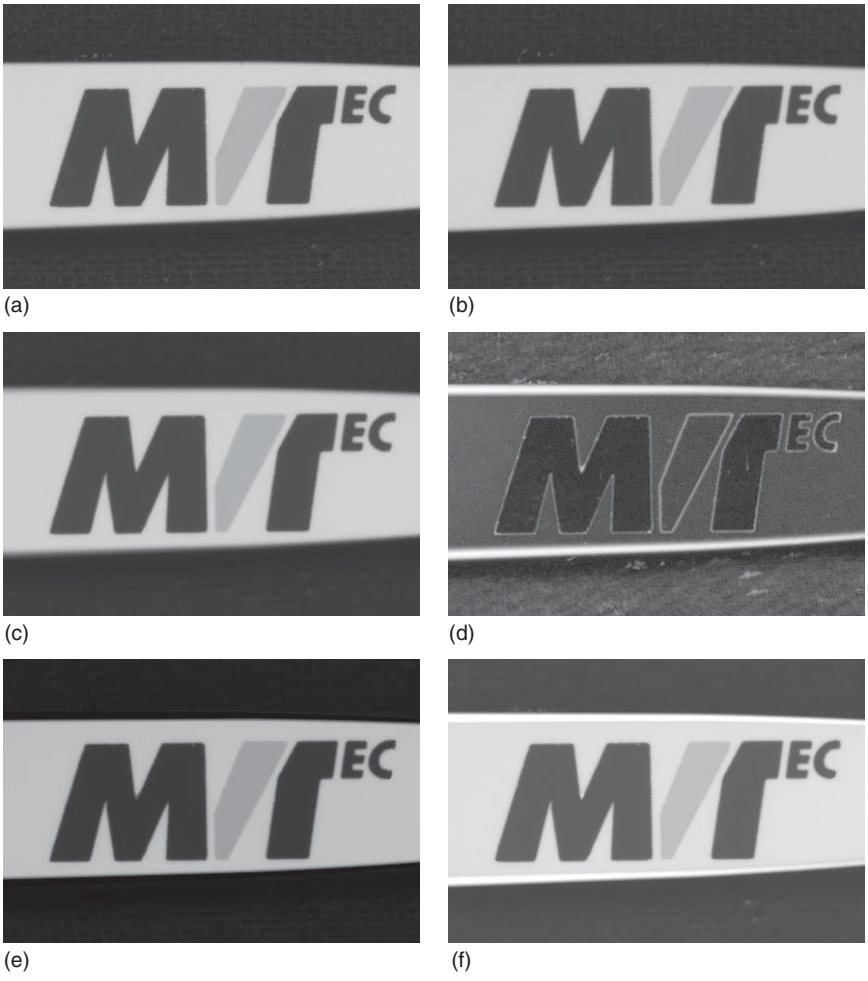


Figure 9.27 (a,b) Two images of a sequence of 15 showing a print on the clip of a pen. Note that the letter V in the MVTec logo moves slightly with respect to the rest of the logo. (c) Reference image $m_{r,c}$ of the variation model computed from the 15 training images. (d) Standard deviation image $s_{r,c}$. For better visibility, $s_{r,c}^{1/4}$ is displayed. (e,f) Minimum and maximum threshold images $u_{r,c}$ and $l_{r,c}$ computed with $a = b = 20$ and $c = d = 3$.

where

$$\begin{aligned} u_{r,c} &= m_{r,c} + \max(a, cs_{r,c}) \\ l_{r,c} &= m_{r,c} - \max(b, ds_{r,c}) \end{aligned} \tag{9.52}$$

Figure 9.27a,b displays two images of a sequence of 15 showing a print on the clip of a pen. All images are aligned such that the MVTec logo is in the center of the image. Note that the letter V in the MVTec logo moves with respect to the rest of the logo and that the corners of the letters may change their shape slightly. This happens because of the pad printing technology used to print the logo. The two colors of the logo are printed with two different pads, which can

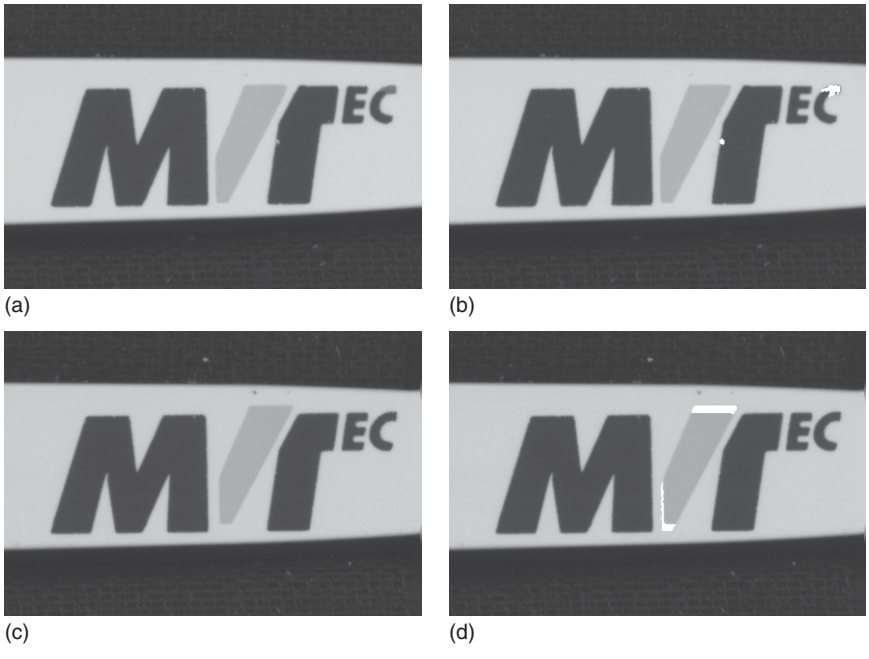


Figure 9.28 (a) Image showing a logo with errors in the letters T (small hole) and C (too little ink). (b) Errors displayed in white, segmented with the variation model of Figure 9.27. (c) Image showing a logo in which the letter V has moved too high and to the right. (d) Segmented errors.

move with respect to each other. Furthermore, the size of the letters may vary because of slightly different pressures with which the pads are pressed onto the clip. To ensure that the logo has been printed correctly, the variation model can be used to determine the mean and variation images shown in Figure 9.27c,d, and from them the threshold images shown in Figure 9.27e,f. Note that the variation is large at the letter V of the logo because this letter may move with respect to the rest of the logo. Also note the large variation at the edges of the clip, which occurs because the logo's position varies on the clip.

Figure 9.28a shows a logo with errors in the letters T (small hole) and C (too little ink). From Figure 9.28b, it can be seen that the two errors can be detected reliably. Figure 9.28c shows a different kind of error: the letter V has moved too high and to the right. This kind of error can also be detected easily, as shown in Figure 9.28d.

As described so far, the variation model requires n training images to construct the reference and variation images. In some applications, however, it is possible to acquire only a single reference image. In these cases, there are two options to create the variation model. The first option is to create artificial variations of the model, for example, by creating translated versions of the reference image. Another option can be derived by noting that the variations are necessarily large at the edges of the object if we allow small size and position tolerances. This can be clearly seen in Figure 9.27d. Consequently, in the absence of training images

that show the real variations of the object, a reasonable approximation for $s_{r,c}$ is given by computing the edge amplitude image of the reference image using one of the edge filters described in Section 9.7.3.

9.4.2 Extraction of Connected Components

The segmentation algorithms in the previous section return one region as the segmentation result (recall the definitions in equations (9.45)–(9.47)). Typically, the segmented region contains multiple objects that should be returned individually. For example, in the examples in Figures 9.22–9.26 we are interested in obtaining each character as a separate region. Typically, the objects we are interested in are characterized by forming a connected set of pixels. Hence, to obtain the individual regions we must compute the connected components of the segmented region.

To be able to compute the connected components, we must define when two pixels should be considered connected. On a rectangular pixel grid, there are only two natural options to define the connectivity. The first possibility is to define two pixels as being connected if they have an edge in common, that is, if the pixel is directly above, below, left, or right of the current pixel, as shown in Figure 9.29a. Since each pixel has four connected pixels, this definition is called the 4-connectivity or 4-neighborhood. Alternatively, the definition can be extended to also include the diagonally adjacent pixels, as shown in Figure 9.29b. This definition is called the 8-connectivity or 8-neighborhood.

While these definitions are easy to understand, they cause problematic behavior if the same definition is used on both foreground and background. Figure 9.30 shows some of the problems that occur if 8-connectivity is used for foreground and background. In Figure 9.30a, there is clearly a single line in the foreground, which divides the background into two connected components. This is what we would intuitively expect. However, as Figure 9.30b shows, if the line is slightly rotated, we still obtain a single connected component in the foreground. However, now the background is also a single component. This is quite counterintuitive. Figure 9.30c shows another peculiarity. Again, the foreground region consists of a single connected component. Intuitively, we would say that the region contains a hole. However, the background also is a single connected component, indicating that the region contains no hole. The only remedy for this problem is to use opposite connectivities on the foreground and background. If, for example, 4-connectivity is used for the background in the examples in Figure 9.30, all of the above problems are solved. Likewise, if 4-connectivity is used for the foreground and 8-connectivity for the background, the inconsistencies are avoided.

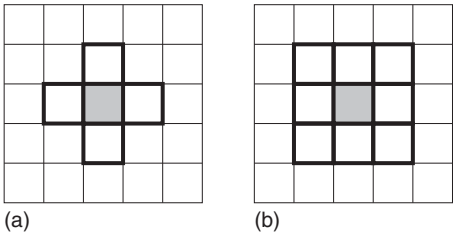


Figure 9.29 Two possible definitions of connectivity on rectangular pixel grids: (a) 4-connectivity and (b) 8-connectivity.

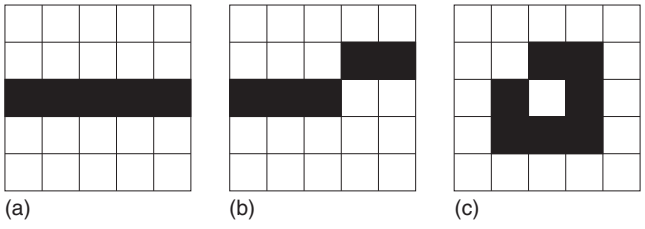


Figure 9.30 Some peculiarities occur when the same connectivity, in this case 8-connectivity, is used for the foreground and background. (a) The single line in the foreground clearly divides the background into two connected components. (b) If the line is very slightly rotated, there is still a single line, but now the background is a single component, which is counterintuitive. (c) The single region in the foreground intuitively contains one hole. However, the background is also a single connected component, indicating that the region has no hole, which is also counterintuitive.

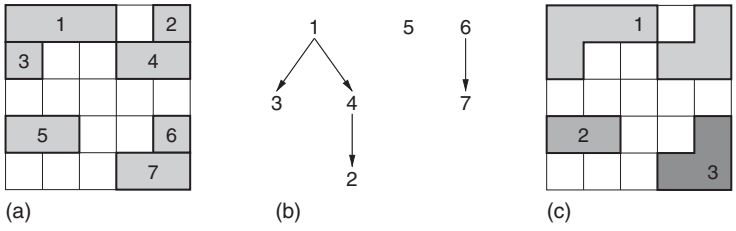


Figure 9.31 (a) Run-length representation of a region containing seven runs. (b) Search tree when performing a depth-first search for the connected components of the region in (a) using 8-connectivity. The numbers indicate the runs. (c) Resulting connected components.

To compute the connected components on the run-length representation of a region, a classical depth-first search can be performed [25]. We can repeatedly search for the first unprocessed run, and then search for overlapping runs in the adjacent rows of the image. The used connectivity determines whether two runs overlap. For 4-connectivity, the runs must at least have one pixel in the same column, while for the 8-connectivity the runs must at least touch diagonally. An example of this procedure is shown in Figure 9.31. The run-length representation of the input region is shown in Figure 9.31a, the search tree for the depth-first search using the 8-connectivity is shown in Figure 9.31b, and the resulting connected components are shown in Figure 9.31c. For the 8-connectivity, three connected components result. If 4-connectivity had been used, four connected components would result.

It should be noted that the connected components can also be computed from the representation of a region as a binary image. The output of this operation is a label image. Therefore, this operation is also called labeling or component labeling. For a description of algorithms that compute the connected components from a binary image, see [22, 23].

To conclude this section, Figure 9.32a,b shows the result of computing the connected components of the regions in Figure 9.23e,f. As can be seen, each character is a connected component. Furthermore, the noisy segmentation results are also returned as separate components. Thus, it is easy to remove them from the segmentation, for example, based on their area.

(a) (b)

Figure 9.32 (a,b) Result of computing the connected components of the regions in Figure 9.23e,f. The connected components are visualized by using eight different gray values cyclically.

9.4.3 Subpixel-Precise Thresholding

All the thresholding operations we have discussed so far have been pixel-precise. In most cases, this precision is sufficient. However, some applications require a higher accuracy than the pixel grid. Therefore, an algorithm that returns a result with subpixel precision is sometimes required. Obviously, the result of this subpixel-precise thresholding operation cannot be a region, which is only pixel-precise. The appropriate data structure for this purpose therefore is a subpixel-precise contour (see Section 9.1.3). This contour will represent the boundary between regions in the image that have gray values above the gray value threshold g_{sub} and regions that have gray values below g_{sub}. To obtain this boundary, we must convert the discrete representation of the image into a continuous function. This can be done, for example, with bilinear interpolation (see equation (9.42) in Section 9.3.3). Once we have obtained a continuous representation of the image, the subpixel-precise thresholding operation conceptually consists of intersecting the image function $f(r, c)$ with the constant function $g(r, c) = g_{\text{sub}}$. Figure 9.33 shows the bilinearly interpolated image $f(r, c)$ in a 2×2 block of the four closest pixel centers. The closest pixel centers lie at the corners of the graph. The bottom of the graph shows the intersection curve of the image $f(r, c)$ in this 2×2 block with the constant gray value $g_{\text{sub}} = 100$. Note that this curve is part of a hyperbola. Since this hyperbolic curve would be quite cumbersome to represent, we can simply substitute it with a straight line segment between the two points where the hyperbola leaves the 2×2 block. This line segment constitutes one segment of the subpixel contour we are interested in. Each 2×2 block in the image typically contains between zero and two of these line segments. If the 2×2 block contains an intersection of two contours, four line segments may occur. To obtain meaningful contours, these segments need to be linked. This can be done by repeatedly selecting the first unprocessed line segment in the image as the first segment of the contour and then tracing the adjacent line segments until the contour closes, reaches the image border, or reaches an intersection point. The result of this linking step typically is closed contours that enclose a region in the image in which the gray values are either larger or smaller than the threshold. Note that, if such a region contains holes, one contour will be created for the outer boundary of the region and one for each hole.

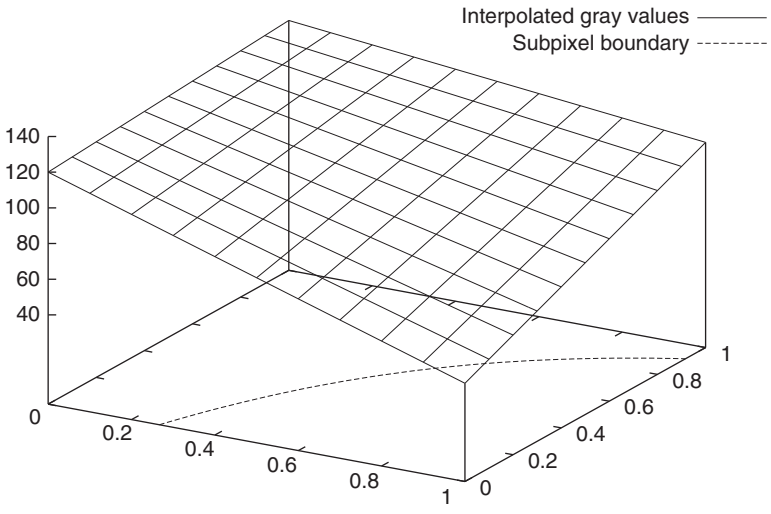


Figure 9.33 The graph shows gray values that are interpolated bilinearly between four pixel centers, lying at the corners of the graph, and the intersection curve with the gray value $g_{sub} = 100$ at the bottom of the graph. This curve (part of a hyperbola) is the boundary between the region with gray values >100 and gray values <100.

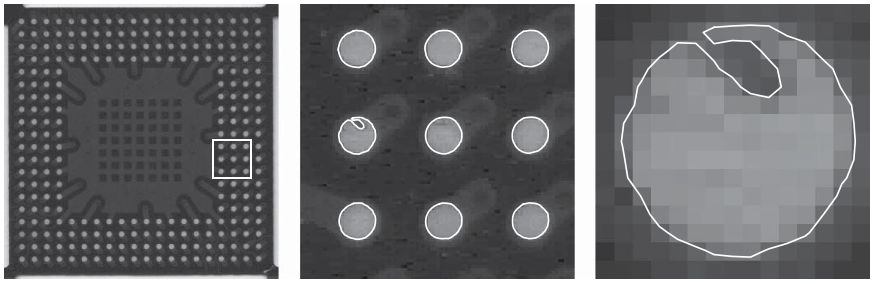


Figure 9.34 (a) Image of a PCB with BGA solder pads. (b) Result of applying a subpixel-precise threshold to the image in (a). The part that is being displayed corresponds to the white rectangle in (a). (c) Detail of the left pad in the center row of (b).

Figure 9.34a shows an image of a PCB that contains a ball grid array (BGA) of solder pads. To ensure good electrical contact, it must be ensured that the pads have the correct shape and position. This requires high accuracy, and, since in this application typically the resolution of the image is small compared to the size of the balls and pads, the segmentation must be performed with subpixel accuracy. Figure 9.34b shows the result of performing a subpixel-precise thresholding operation on the image in Figure 9.34a. To see enough details of the results, the part that corresponds to the white rectangle in Figure 9.34a is displayed. The boundary of the pads is extracted with very good accuracy. Figure 9.34c shows even more detail: the left pad in the center row of Figure 9.34b, which contains an error that must be detected. As can be seen, the subpixel-precise contour correctly captures the erroneous region of the pad. We can also easily see the individual line segments in the subpixel-precise contour and how they are contained in the

2×2 pixel blocks. Note that each block lies between four pixel centers. Therefore, the contour's line segments end at the lines that connect the pixel centers. Note also that in this part of the image there is only one block in which two line segments are contained: at the position where the contour enters the error on the pad. All the other blocks contain one or no line segments.

9.5 Feature Extraction

In the previous sections, we have seen how to extract regions or subpixel-precise contours from an image. While the regions and contours are very useful, they may not be sufficient because they contain the raw description of the segmented data. Often, we must select certain regions or contours from the segmentation result, for example, to remove unwanted parts of the segmentation. Furthermore, often we are interested in gauging the objects. In other applications, we might want to classify the objects, for example, in the OCR, to determine the type of the object. All these applications require that we determine one or more characteristic quantities from the regions or contours. The quantities we determine are called features. Typically they are real numbers. The process of determining the features is called feature extraction. There are different kinds of features. Region features are features that can be extracted from the regions themselves. In contrast, gray value features also use the gray values in the image within the region. Finally, contour features are based on the coordinates of the contour.

9.5.1 Region Features

By far the simplest region feature is the area of the region:

$$a = |R| = \sum_{(r,c) \in R} 1 = \sum_{i=1}^{n} ce_i - cs_i + 1 \tag{9.53}$$

Hence, the area a of the region is simply the number of points $|R|$ in the region. If the region is represented as a binary image, the first sum has to be used to compute the area; whereas if a run-length representation is used, the second sum can be used. Recall from equation (9.2) that a region can be regarded as the union of its runs, and the area of a run is extremely simple to compute. Note that the second sum contains many fewer terms than the first sum, as discussed in Section 9.1.2. Hence, the run-length representation of a region will lead to a much faster computation of the area. This is true for almost all region features.

Figure 9.35 shows the result of selecting all regions with an area ≥ 20 from the regions in Figure 9.32a,b. Note that all the characters have been selected, while all noisy segmentation results have been removed. These regions could now be used as input for OCR.

The area is a special case of a more general class of features called the moments of the region. The moment of order (p, q), with $p \geq 0$ and $q \geq 0$, is defined as

$$m_{p,q} = \sum_{(r,c) \in R} r^p c^q \tag{9.54}$$

Note that $m_{0,0}$ is the area of the region. As for the area, simple formulas to compute the moments solely based on the runs can be derived. Hence, the moments can be computed very efficiently in the run-length representation.

(a) (b)

Figure 9.35 (a,b) Result of selecting regions with an area ≥20 from the regions in Figure 9.32a,b. The connected components are visualized by using eight different gray values cyclically.

The moments in equation (9.54) depend on the size of the region. Often, it is desirable to have features that are invariant to the size of the objects. To obtain such features, we can simply divide the moments by the area of the region if $p + q \geq 1$ to get normalized moments:

$$n_{p,q} = \frac{1}{a} \sum_{(r,c) \in R} r^p c^q \tag{9.55}$$

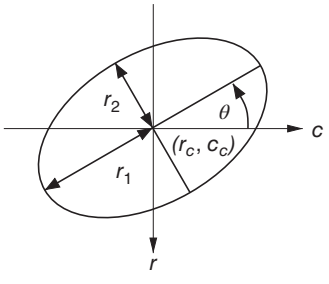
The most interesting feature that can be derived from the normalized moments is the center of gravity of the region, which is given by $(n_{1,0}, n_{0,1})$. It can be used to describe the position of the region. Note that the center of gravity is a subpixel-precise feature, even though it is computed from pixel-precise data.

The normalized moments depend on the position in the image. Often, it is useful to make the features invariant to the position of the region in the image. This can be done by calculating the moments relative to the center of gravity of the region. These central moments are given by $(p + q \geq 2)$

$$\mu_{p,q} = \frac{1}{a} \sum_{(r,c) \in R} (r - n_{1,0})^p (c - n_{0,1})^q \tag{9.56}$$

Note that they are also normalized. The second central moments $(p + q = 2)$ are particularly interesting. They enable us to define an orientation and an extent for the region. This is done by assuming that the moments of order 1 and 2 of the region were obtained from an ellipse. Then, from these five moments, the five geometric parameters of the ellipse can be derived. Figure 9.36 displays the ellipse's parameters graphically. The center of the ellipse is identical to the center of gravity of the region. The major and minor axes r_1 and r_2 and the angle of the

Figure 9.36 Geometric parameters of an ellipse.

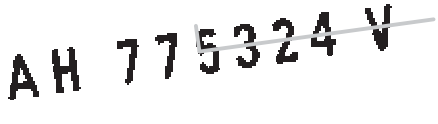


ellipse with respect to the column axis are given by

$$r_1 = \sqrt{2\left(\mu_{2,0} + \mu_{0,2} + \sqrt{(\mu_{2,0} - \mu_{0,2})^2 + 4\mu_{1,1}^2}\right)}$$
$$r_2 = \sqrt{2\left(\mu_{2,0} + \mu_{0,2} - \sqrt{(\mu_{2,0} - \mu_{0,2})^2 + 4\mu_{1,1}^2}\right)} \tag{9.57}$$
$$\theta = -\frac{1}{2}\arctan\frac{2\mu_{1,1}}{\mu_{0,2} - \mu_{2,0}}$$

For a derivation of these results, see [22] (note that there the diameters are used instead of the radii). From the ellipse parameters, we can derive another very useful feature: the anisometry r_1/r_2. This is scale-invariant and describes how elongated a region is.

The ellipse parameters are extremely useful to determine the orientations and sizes of regions. For example, the angle θ can be used to rectify rotated text. Figure 9.37a shows the result of thresholding the image in Figure 9.18a. The segmentation result is treated as a single region, that is, the connected components have not been computed. Figure 9.37a also displays the ellipse parameters by overlaying the major and minor axes of the equivalent ellipse. Note that the major axis is slightly longer than the region because the equivalent ellipse does not need to have the same area as the region. It only needs to have the same moments of orders 1 and 2. The angle of the major axis is a very good estimate for the rotation of the text. In fact, it has been used to rectify the images in Figure 9.18b,c. Figure 9.37b shows the axes of the characters after the connected components have been computed. Note how well the orientation of the regions corresponds with our intuition.



(a)

(b)

Figure 9.37 Result of thresholding the image in Figure 9.18a overlaid with a visualization of the ellipse parameters. The light gray lines represent the major and minor axes of the regions. Their intersection is the center of gravity of the regions. (a) The segmentation is treated as a single region. (b) The connected components of the region are used. The angle of the major axis in (a) has been used to rotate the images in Figure 9.18b,c.

While the ellipse parameters are extremely useful, they have two minor shortcomings. First of all, the orientation can be determined only if $r_1 \neq r_2$. Our first thought might be that this applies only to circles, which have no meaningful orientation anyway. Unfortunately, this is not true. There is a much larger class of objects for which $r_1 = r_2$. All objects that have a fourfold rotational symmetry, such as squares, have $r_1 = r_2$. Hence, their orientation cannot be determined with the ellipse parameters. The second slight problem is that, since the underlying model is an ellipse, the orientation θ can only be determined modulo π (180°). This problem can be solved by determining the point in the region that has the largest distance from the center of gravity and use it to select θ or $\theta + \pi$ as the correct orientation.

In the above discussion, we have used various transformations to make the moment-based features invariant to certain transformations, for example, translation and scaling. Several approaches have been proposed to create moment-based features that are invariant to a larger class of transformations, for example, translation, rotation, and scaling [26] or even general affine transformations [27, 28]. They are primarily used to classify objects.

Apart from the moment-based features, there are several other useful features that are based on the idea of finding an enclosing geometric primitive for the region. Figure 9.38a displays the smallest axis-parallel enclosing rectangle of a region. This rectangle is often also called the bounding box of the region. It can be calculated very easily based on the minimum and maximum row and column coordinates of the region. Based on the parameters of the rectangle, other useful quantities like the width and height of the region and their ratio can be calculated. The parameters of the bounding box are particularly useful if we want to quickly find out whether two regions can intersect. Since the smallest axis-parallel enclosing rectangle sometimes is not very tight, we can also define a smallest enclosing rectangle of arbitrary orientation, as shown in Figure 9.38b. Its computation is much more complicated than the computation of the bounding box, however, so we cannot give details here. An efficient implementation can be found in [29]. Note that an arbitrarily oriented rectangle has the same parameters as an ellipse. Hence, it also enables us to define the position, size, and orientation of a region. Note that, in contrast to the ellipse parameters, a useful orientation for squares is returned. The final useful enclosing primitive is an enclosing circle, as shown in Figure 9.38c. Its computation is also quite complex [30]. It also enables us to define the position and size of a region.

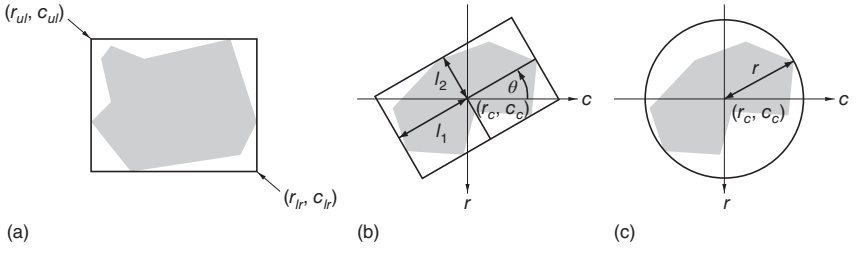


Figure 9.38 (a) Smallest axis-parallel enclosing rectangle of a region. (b) Smallest enclosing rectangle of arbitrary orientation. (c) Smallest enclosing circle.

The computation of the smallest enclosing rectangle of arbitrary orientation and the smallest enclosing circle is based on first computing the convex hull of the region. The convex hull of a set of points, and in particular a region, is the smallest convex set that contains all the points. A set is convex if, for any two points in the set, the straight line between them is completely contained in the set. The convex hull of a set of points can be computed efficiently [31, 32]. The convex hull of a region is often useful to construct ROIs from regions that have been extracted from the image. Based on the convex hull of the region, another useful feature can be defined: the convexity, which is defined as the ratio of the area of the region to the area of its convex hull. It is a feature between 0 and 1 that measures how compact the region is. A convex region has a convexity of 1. The convexity can, for example, be used to remove unwanted segmentation results, which often are highly nonconvex.

Another useful feature of a region is its contour length. To compute it, we need to trace the boundary of the region to get a linked contour of the boundary pixels [22]. Once the contour has been computed, we simply need to sum the Euclidean distances of the contour segments, which are 1 for horizontal and vertical segments and $\sqrt{2}$ for diagonal segments. Based on the contour length l and the area a of the region, we can define another measure for the compactness of a region: $c = l^2/(4\pi a)$. For circular regions, this feature is 1, while all other regions have larger values. The compactness has similar uses as the convexity.

9.5.2 Gray Value Features

We have already seen some gray value features in Section 9.2.1, namely the minimum and maximum gray values within the region:

$$g_{\min} = \min_{(r,c)\in R} g_{r,c}, g_{\max} = \max_{(r,c)\in R} g_{r,c} \tag{9.58}$$

They are used for the gray value normalization in Section 9.2.1. Another obvious feature is the mean gray value within the region:

$$\overline{g} = \frac{1}{a} \sum_{(r,c)\in R} g_{r,c} \tag{9.59}$$

Here, a is the area of the region, given by equation (9.53). The mean gray value is a measure of the brightness of the region. A single measurement within a reference region can be used to measure additive brightness changes with respect to the conditions when the system was set up. Two measurements within different reference regions can be used to measure linear brightness changes and thereby to compute a linear gray value transformation (see Section 9.2.1) that compensates the brightness change, or to adapt segmentation thresholds. The mean gray value is a statistical feature. Another statistical feature is the variance of the gray values:

$$s^2 = \frac{1}{a-1} \sum_{(r,c)\in R} (g_{r,c} - \overline{g})^2 \tag{9.60}$$

and the standard deviation $s = \sqrt{s^2}$. Measuring the mean and standard deviation within a reference region can also be used to construct a linear gray value transformation that compensates brightness changes. The standard deviation can be used to adapt segmentation thresholds. Furthermore, the standard deviation is a measure of the amount of texture that is present within the region.

The gray value histogram (9.4) and the cumulative histogram (9.5), which we have already encountered in Section 9.2.1, are also gray value features. From the histogram, we have already used a feature for the robust contrast normalization: the α-quantile

$$g_\alpha = \min\{g : c_g \geq \alpha\} \tag{9.61}$$

where c_g is defined in equation (9.5). It was used to obtain the robust minimum and maximum gray values in Section 9.2.1. The quantiles were called p_l and p_u there. Note that, for $\alpha = 0.5$, we obtain the median gray value. It has similar uses as the mean gray value.

In the previous section, we have seen that the region's moments are extremely useful features. They can be extended to gray value features in a natural manner. The gray value moment of order (p, q), with $p \geq 0$ and $q \geq 0$, is defined as

$$m_{p,q} = \sum_{(r,c)\in R} g_{r,c}\, r^p c^q \tag{9.62}$$

This is the natural generalization of the region moments because we obtain the region moments from the gray value moments by using the characteristic function χ_R (Equation (9.1)) of the region as the gray values. Like for the region moments, the moment $a = m_{0,0}$ can be regarded as the gray value area of the region. It is actually the "volume" of the gray value function $g_{r,c}$ within the region. Like for the region moments, normalized moments can be defined by

$$n_{p,q} = \frac{1}{a} \sum_{(r,c)\in R} g_{r,c}\, r^p c^q \tag{9.63}$$

The moments $(n_{1,0}, n_{0,1})$ define the gray value center of gravity of the region. With this, central gray value moments can be defined by

$$\mu_{p,q} = \frac{1}{a} \sum_{(r,c)\in R} g_{r,c}\, (r - n_{1,0})^p (c - n_{0,1})^q \tag{9.64}$$

Like for the region moments, based on the second central moments we can define the ellipse parameters, major and minor axes, and the orientation. The formulas are identical to equation (9.57). Furthermore, the anisometry can also be defined identically as for the regions.

All the moment-based gray value features are very similar to their region-based counterparts. Therefore, it is interesting to look at their differences. Like we saw, the gray value moments reduce to the region moments if the characteristic function of the region is used as the gray values. The characteristic function can be interpreted as the membership of a pixel to the region. A membership of 1 means that the pixel belongs to the region, while 0 means that the pixel does not belong to the region. This notion of belonging to the region is crisp: that is, for every pixel a hard decision must be made. Suppose now that, instead of making a hard decision for every pixel, we could make a "soft" or "fuzzy" decision about whether a pixel belongs to the region, and that we encode the degree of belonging to the region by a number $\in [0, 1]$. We can interpret the degree of belonging as a fuzzy membership value, as opposed to the crisp binary membership value. With this, the gray value image can be regarded as a fuzzy set [33]. The advantage of regarding the image as a fuzzy set is that we do not have to make a hard decision about whether a pixel belongs to the object or not. Instead, the fuzzy membership value determines what percentage of the pixel belongs to the object. This

Figure 9.39 Subpixel-precise circle position and area using the gray value and region moments. The image represents a fuzzy membership, scaled to values between 0 and 200. The solid line is the result of segmenting with a membership of 100. The dotted line is a circle that has the same center of gravity and area as the segmented region. The dashed line is a circle that has the same gray value center of gravity and gray value area as the image. (a) Shift: 0; error in the area for the region moments: 13.2%; for the gray value moments: −0.05%. (b) Shift: 5/32 pixel; error in the row coordinate for the region moments: −0.129; for the gray value moments: 0.003. (c) Shift: 1/2 pixel; error in the area for the region moments: −8.0%; for the gray value moments: −0.015%. Note that the gray value moments yield a significantly better accuracy for this small object.

enables us to measure the position and size of the objects much more accurately, especially for small objects, because in the transition zone between the foreground and background there will be some mixed pixels that allow us to capture the geometry of the object more accurately. An example of this is shown in Figure 9.39. Here, a synthetically generated subpixel-precise ideal circle of radius 3 is shifted in subpixel increments. The gray values represent a fuzzy membership, scaled to values between 0 and 200 for display purposes. The figure displays a pixel-precise region, thresholded with a value of 100, which corresponds to a membership above 0.5, as well as two circles that have a center of gravity and area that were obtained from the region and gray value moments. The gray value moments were computed in the entire image. It can be seen that the area and center of gravity are computed much more accurately by the gray value moments because the decision about whether a pixel belongs to the foreground or not has been avoided. In this example, the gray value moments result in an area error that is always smaller than 0.25% and a position error smaller than 1/200 of a pixel. In contrast, the area error for the region moments can be up to 13.2% and the position error can be up to 1/6 of a pixel. Note that both types of moments yield subpixel-accurate measurements. We can see that on ideal data it is possible to obtain an extremely high accuracy with the gray value moments, even for very small objects. On real data, the accuracy will necessarily be somewhat lower. It should also be noted that the accuracy advantage of the gray value moments primarily occurs for small objects. Because the gray value moments must access every pixel within the region, whereas the region moments can be computed solely based on the run-length representation of the region, the region moments can be computed much faster. Hence, the gray moments are typically used only for relatively small regions.

The only question we need to answer is how to define the fuzzy membership value of a pixel. If we assume that the camera has a fill factor of 100% and the

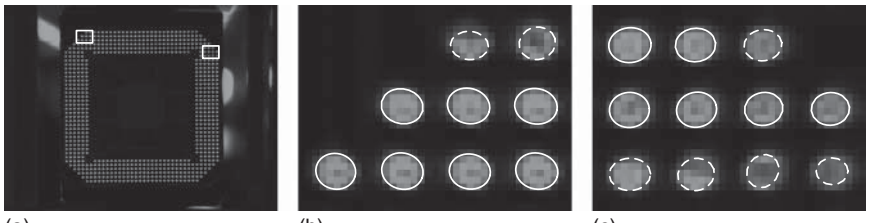


Figure 9.40 (a) Image of a BGA device. The two rectangles correspond to the image parts shown in (b) and (c). The results of inspecting the balls for correct size (gray value area ≥20) and correct gray value anisometry (≤1.25) are visualized in (b) and (c). Correct balls are displayed as solid ellipses, while defective balls are displayed as dashed ellipses.

gray value response of the camera (and analog frame grabber, if used) is linear, the gray value difference of a pixel from the background is proportional to the portion of the object that is covered by the pixel. Consequently, we can define a fuzzy membership relation as follows: every pixel that has a gray value below the background gray value $g_{\min}$ has a membership value of 0. Conversely, every pixel that has a gray value above the foreground gray value $g_{\max}$ has a membership value of 1. In between, the membership values are interpolated linearly. Since this procedure would require floating-point images, the membership is scaled to an integer image with b bits, typically 8 bits. Consequently, the fuzzy membership relation is a simple linear gray value scaling, as defined in Section 9.2.1. If we scale the fuzzy membership image in this manner, the gray value area needs to be divided by the maximum gray value, for example, 255, to obtain the true area. The normalized and central gray value moments do not need to be modified in this manner since they are, by definition, invariant to a scaling of the gray values.

Figure 9.40 displays a real application where the above principles are used. In Figure 9.40a, a BGA device with solder balls is displayed, along with two rectangles that indicate the image parts shown in Figure 9.40b,c. The image in Figure 9.40a is first transformed into a fuzzy membership image using $g_{\min} = 40$ and $g_{\max} = 120$ with 8-bit resolution. The individual balls are segmented and then inspected for correct size and shape by using the gray value area and the gray value anisometry. The erroneous balls are displayed with dashed lines. To aid visual interpretation, the ellipses representing the segmented balls are scaled such that they have the same area as the gray value area. This is done because the gray value ellipse parameters typically return an ellipse with a different area than the gray value area, analogous to the region ellipse parameters (see the discussion following equation (9.57) in Section 9.5.1). As can be seen, all the balls that have an erroneous size or shape, indicating partially missing solder, have been correctly detected.

9.5.3 Contour Features

Many of the region features we have discussed in Section 9.5.1 can be transferred to subpixel-precise contour features in a straightforward manner. For example, the length of the subpixel-precise contour is even easier to compute because the contour is already represented explicitly by its control points (r_i, c_i), for $i = 1, \ldots, n$. It is also simple to compute the smallest enclosing axis-parallel rectangle

(the bounding box) of the contour. Furthermore, the convex hull of the contour can be computed like for regions [31, 32]. From the convex hull, we can also derive the smallest enclosing circles [30] and the smallest enclosing rectangles of arbitrary orientation [29].

In the previous two sections, we have seen that the moments are extremely useful features. An interesting question, therefore, is whether they can be defined for contours. In particular, it is interesting to see whether a contour has an area. Obviously, for this to be true, the contour must enclose a region, that is, it must be closed and must not intersect itself. To simplify the formulas, let us assume that a closed contour is specified by $(r_1, c_1) = (r_n, c_n)$. Let the subpixel-precise region that the contour encloses be denoted by R. Then, the moment of order (p, q) is defined as

$$m_{p,q} = \iint_{(r,c)\in R} r^p c^q \, dr \, dc \tag{9.65}$$

Like for regions, we can define normalized and central moments. The formulas are identical to equation (9.55) and (9.56) with the sums being replaced by integrals. It can be shown that these moments can be computed solely based on the control points of the contour [34]. For example, the area and center of gravity of the contour are given by

$$\begin{aligned} a &= \frac{1}{2}\sum_{i=1}^{n} r_{i-1}c_i - r_i c_{i-1} \\ n_{1,0} &= \frac{1}{6a}\sum_{i=1}^{n}(r_{i-1}c_i - r_i c_{i-1})(r_{i-1} + r_i) \\ n_{0,1} &= \frac{1}{6a}\sum_{i=1}^{n}(r_{i-1}c_i - r_i c_{i-1})(c_{i-1} + c_i) \end{aligned} \tag{9.66}$$

Analogous formulas can be derived for the second-order moments. Based on them, we can again compute the ellipse parameters, major axis, minor axis, and orientation. The formulas are identical to equations (9.57). The moment-based contour features can be used for the same purposes as the corresponding region and gray value features. By performing an evaluation similar to that in Figure 9.39, it can be seen that the contour center of gravity and the ellipse parameters are equally as accurate as the gray value center of gravity. The accuracy of the contour area is slightly worse than that of the gray value area because we have approximated the hyperbolic segments with line segments. Since the true contour is a circle, the line segments always lie inside the true circle. Nevertheless, subpixel-thresholding and the contour moments could also have been used to detect the erroneous balls in Figure 9.40.

9.6 Morphology

In Section 9.4, we discussed how to segment regions. We have already seen that segmentation results often contain unwanted noisy parts. Furthermore, sometimes the segmentation will contain parts in which the shape of the object we

are interested in has been disturbed, for example, because of reflections. Therefore, often we need to modify the shape of the segmented regions to obtain the desired results. This is the subject of the field of mathematical morphology, which can be defined as a theory for the analysis of spatial structures [35]. For our purposes, mathematical morphology provides a set of extremely useful operations that enable us to modify or describe the shape of objects. Morphological operations can be defined on regions and gray value images. We will discuss both types of operations in this section.

9.6.1 Region Morphology

All region morphology operations can be defined in terms of six very simple operations: union, intersection, difference, complement, translation, and transposition. We will take a brief look at these operations first.

The union of two regions R and S is the set of points that lie in R or in S:

$$R \cup S = \{p \mid p \in R \vee p \in S\} \tag{9.67}$$

One important property of the union is that it is commutative: $R \cup S = S \cup R$. Furthermore, it is associative: $(R \cup S) \cup T = R \cup (S \cup T)$. While this may seem like a trivial observation, it will enable us to derive very efficient implementations for the morphological operations below. The algorithm to compute the union of two binary images is obvious: we simply need to compute the logical OR of the two images. The runtime complexity of this algorithm obviously is $O(wh)$, where w and h are the width and height of the binary image. In the run-length representation, the union can be computed with a lower complexity: $O(n + m)$, where n and m are the number of runs in R and S. The principle of the algorithm is to merge the runs of the two regions while observing the order of the runs (see Section 9.1.2) and then to pack overlapping runs into single runs.

The intersection of two regions R and S is the set of points that lie in R and in S:

$$R \cap S = \{p \mid p \in R \wedge p \in S\} \tag{9.68}$$

Like the union, the intersection is commutative and associative. Again, the algorithm on binary images is obvious: we compute the logical AND of the two images. For the run-length representation, again an algorithm that has complexity $O(n + m)$ can be found.

The difference of two regions R and S is the set of points that lie in R but not in S:

$$R \backslash S = \{p \mid p \in R \wedge p \notin S\} = R \cap \overline{S} \tag{9.69}$$

The difference is neither commutative nor associative. Note that it can be defined in terms of the intersection and the complement of a region R, which is defined as all the points that do not lie in R:

$$\overline{R} = \{p \mid p \notin R\} \tag{9.70}$$

Since the complement of a finite region is infinite, it is impossible to represent it as a binary image. Therefore, for the representation of regions as binary images, it is important to define the operations without the complement. It is, however,

possible to represent it as a run-length-encoded region by adding a flag that indicates whether the region or its complement is being stored. This can be used to define a more general set of morphological operations. There is an interesting relation between the number of connected components of the background $|C(\overline{R})|$ and the number of holes of the foreground $|H(R)|$: we have $|C(\overline{R})| = 1 + |H(R)|$. As discussed in Section 9.4.2, complementary connectivities must be used for the foreground and the background for this relation to hold.

Apart from the set operations, two basic geometric transformations are used in morphological operations. The translation of a region by a vector t is defined as

$$R_t = \{p \mid p - t \in R\} = \{q \mid q = p + t \text{ for } p \in R\} \tag{9.71}$$

Finally, the transposition of a region is defined as a mirroring about the origin:

$$\check{R} = \{-p \mid p \in R\} \tag{9.72}$$

Note that this is the only operation where a special point (the origin) is singled out. All the other operations do not depend on the origin of the coordinate system, that is, they are translation-invariant.

With these building blocks, we can now take a look at the morphological operations. They typically involve two regions. One of these is the region we want to process, which will be denoted by R in the following. The other region has a special meaning. It is called the structuring element, and will be denoted by S. The structuring element is the means by which we can describe the shapes we are interested in.

The first morphological operation we consider is the Minkowski addition, which is defined by

$$R \oplus S = \{r + s \mid r \in R, s \in S\} = \bigcup_{s \in S} R_s = \bigcup_{r \in R} S_r = \{t \mid R \cap (\check{S})_t \neq \emptyset\} \tag{9.73}$$

It is interesting to interpret the formulas. The first formula says that, to get the Minkowski addition of R with S, we take every point in R and every point in S and compute the vector sum of the points. The result of the Minkowski addition is the set of all points thus obtained. If we single out S, this can also be interpreted as taking all points in S, translating the region R by the vector corresponding to the point s from S, and computing the union of all the translated regions. Thus, we obtain the second formula. By symmetry, we can also translate S by all points in R to obtain the third formula. Another way to look at the Minkowski addition is the fourth formula. It tells us that we move the transposed structuring element around in the plane. Whenever the translated, transposed structuring element and the region have at least one point in common, we copy the translated reference point into the output. Figure 9.41 shows an example of the Minkowski addition.

While the Minkowski addition has a simple formula, it has one small drawback. Its geometric criterion is that the transposed structuring element has at least one point in common with the region. Ideally, we would like to have an operation that returns all translated reference points for which the structuring element itself has

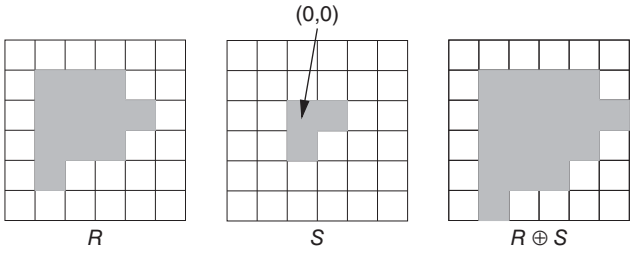


Figure 9.41 Example of the Minkowski addition $R \oplus S$.

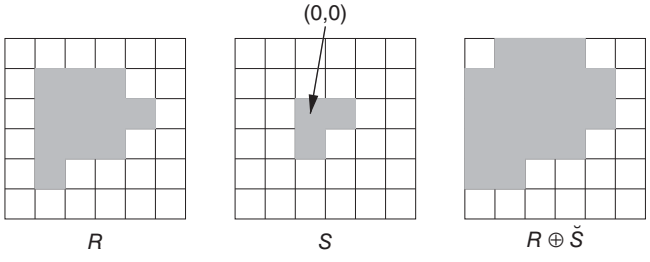


Figure 9.42 Example of the dilation $R \oplus \check{S}$.

at least one point in common with the region. To achieve this, we only need to use the transposed structuring element in the Minkowski addition. This operation is called a dilation, and is defined by

$$R \oplus \check{S} = \{t \mid R \cap S_t \neq \emptyset\} = \bigcup_{s \in S} R_{-s} \tag{9.74}$$

Figure 9.42 shows an example of the dilation. Note that the results of the Minkowski addition and dilation are different. This is true whenever the structuring element is not symmetric with respect to the origin. If the structuring element is symmetric, the results of the Minkowski addition and dilation are identical. Be aware that this is assumed in many discussions about and implementations of the morphology. Therefore, the dilation is often defined without the transposition, which is technically incorrect.

The implementation of the Minkowski addition for binary images is straightforward. As suggested by the second formula in equation (9.73), it can be implemented as a nonlinear filter with logical OR operations. The runtime complexity is proportional to the size of the image times the number of pixels in the structuring element. The second factor can be reduced to roughly the number of boundary pixels in the structuring element [15]. Also, for binary images represented with 1 bit per pixel, very efficient algorithms can be developed for special structuring elements [36]. Nevertheless, in both cases the runtime complexity is proportional to the number of pixels in the image. To derive an implementation for the run-length representation of the regions, we first need to examine some algebraic properties of the Minkowski addition. It is commutative: $R \oplus S = S \oplus R$. Furthermore, it is distributive with respect to the union: $(R \cup S) \oplus$

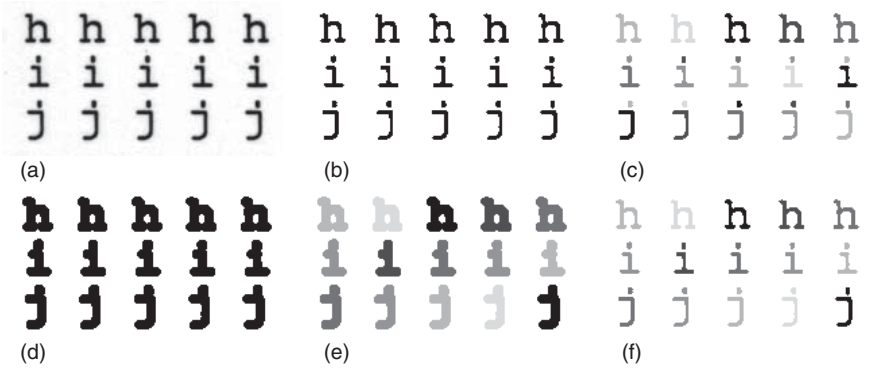


Figure 9.43 (a) Image of a print of several characters. (b) Result of thresholding (a). (c) Connected components of (b) displayed with six different gray values. Note that the characters and their dots are separate connected components, which is undesirable. (d) Result of dilating the region in (b) with a circle of diameter 5. (e) Connected components of (d). Note that each character is now a single connected component. (f) Result of intersecting the connected components in (e) with the original segmentation in (b). This transforms the connected components into the correct shape.

$T = R \oplus T \cup S \oplus T$. Since a region can be regarded as the union of its runs, we can use the commutativity and distributivity to transform the Minkowski addition as follows:

$$R \oplus S = \left(\bigcup_{i=1}^{n} \mathbf{r}_i\right) \oplus \left(\bigcup_{j=1}^{m} \mathbf{s}_j\right) = \bigcup_{j=1}^{m} \left(\left(\bigcup_{i=1}^{n} \mathbf{r}_i\right) \oplus \mathbf{s}_j\right) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \mathbf{r}_i \oplus \mathbf{s}_j \tag{9.75}$$

Thus, the Minkowski addition can be implemented as the union of nm dilations of single runs, which are trivial to compute. Since the union of the runs can be computed easily, the runtime complexity is $O(mn)$, which is better than for binary images.

As we have seen, the dilation and Minkowski addition enlarge the input region. This can be used, for example, to merge separate parts of a region into a single part, and thus to obtain the correct connected components of objects. One example of this is shown in Figure 9.43. Here, we want to segment each character as a separate connected component. If we compute the connected components of the thresholded region in Figure 9.43b, we can see that the characters and their dots are separate components (Figure 9.43c). To solve this problem, we first need to connect the dots with their characters. This can be achieved using a dilation with a circle of diameter 5 (Figure 9.43d). With this, the correct connected components are obtained (Figure 9.43e). Unfortunately, they have the wrong shape because of the dilation. This can be corrected by intersecting the components with the originally segmented region. Figure 9.43f shows that with these simple steps we have obtained one component with the correct shape for each character.

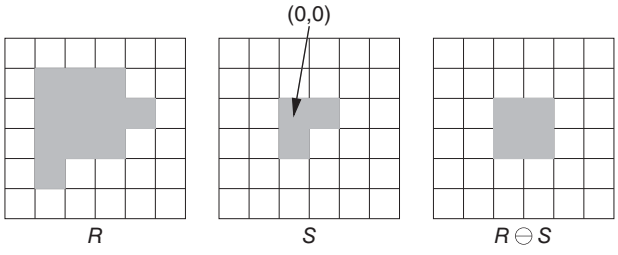


Figure 9.44 Example of the Minkowski subtraction $R \ominus S$.

The dilation is also very useful for constructing ROIs based on regions that were extracted from the image. We will see an example of this in Section 9.7.3.

The second type of morphological operation is the Minkowski subtraction. It is defined by

$$R \ominus S = \bigcap_{s \in S} R_s = \{r \mid \forall \quad s \in S : r - s \in R\} = \{t \mid (\check{S})_t \subseteq R\} \tag{9.76}$$

The first formula is similar to the second formula in equation (9.73) with the union having been replaced by an intersection. Hence, we can still think about moving the region R by all vectors s from S. However, now the points must be contained in all translated regions (instead of at least one translated region). This is what the second formula in equation (9.76) expresses. Finally, if we look at the third formula, we see that we can also move the transposed structuring element around in the plane. If it is completely contained in the region R, we add its reference point to the output. Again, note the similarity to the Minkowski addition, where the structuring element had to have at least one point in common with the region. For the Minkowski subtraction, it must lie completely within the region. Figure 9.44 shows an example of the Minkowski subtraction.

The Minkowski subtraction has the same small drawback as the Minkowski addition: its geometric criterion is that the transposed structuring element must lie completely within the region. As for the dilation, we can use the transposed structuring element. This operation is called an erosion and is defined by

$$R \ominus \check{S} = \bigcap_{s \in S} R_{-s} = \{t \mid S_t \subseteq R\} \tag{9.77}$$

Figure 9.45 shows an example of the erosion. Again, note that the Minkowski subtraction and erosion produce identical results only if the structuring element is symmetric with respect to the origin. Be aware that this is often silently assumed, and the erosion is defined as a Minkowski subtraction, which is technically incorrect.

As we have seen from the small examples, the Minkowski subtraction and erosion shrink the input region. This can, for example, be used to separate objects that are attached to each other. Figure 9.46 shows an example of this. Here, the goal is to segment the individual globular objects. The result of thresholding the image is shown in Figure 9.46b. If we compute the connected components of

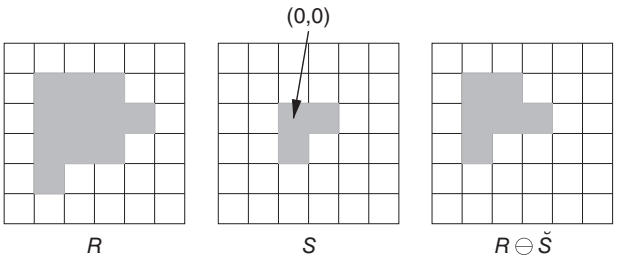


Figure 9.45 Example of the erosion $R \ominus \check{S}$.

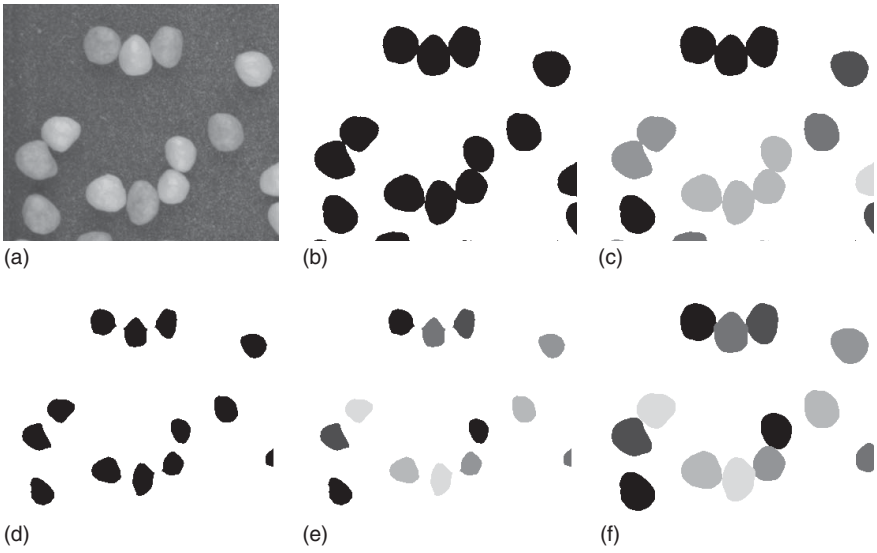


Figure 9.46 (a) Image of several globular objects. (b) Result of thresholding (a). (c) Connected components of (b) displayed with six different gray values. Note that several objects touch each other and hence are in the same connected component. (d) Result of eroding the region in (b) with a circle of diameter 15. (e) Connected components of (d). Note that each object is now a single connected component. (f) Result of dilating the connected components in (e) with a circle of diameter 15. This transforms the correct connected components into approximately the correct shape.

this region, an incorrect result is obtained because several objects touch each other (Figure 9.46c). The solution is to erode the region with a circle of diameter 15 (Figure 9.46d) before computing the connected components (Figure 9.46e). Unfortunately, the connected components have the wrong shape. Here, we cannot use the same strategy that we used for the dilation (intersecting the connected components with the original segmentation) because the erosion has shrunk the region. To approximately get the original shape back, we can dilate the connected components with the same structuring element that we used for the erosion (Figure 9.46f).

We can see another use of the erosion if we remember its definition: it returns the translated reference point of the structuring element S for every translation

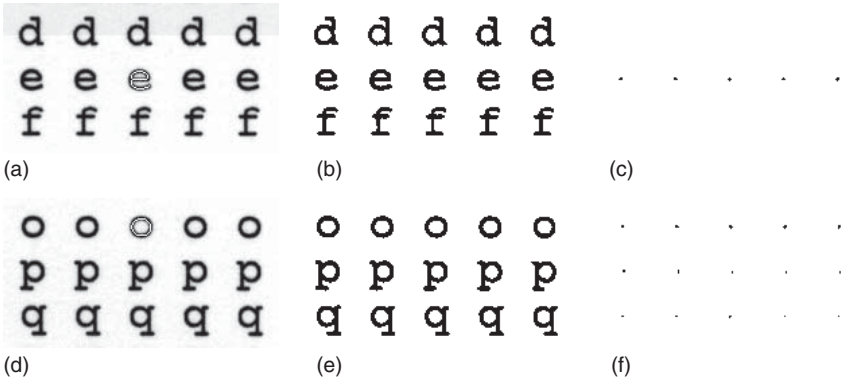


Figure 9.47 (a) Image of a print of several characters with the structuring element used for the erosion overlaid in white. (b) Result of thresholding (a). (c) Result of the erosion of (b) with the structuring element in (a). Note that the reference point of all letters "e" has been found. (d) A different set of characters with the structuring element used for the erosion overlaid in white. (e) Result of thresholding (d). (f) Result of the erosion of (e) with the structuring element in (d). Note that the reference point of the letter "o" has been identified correctly. In addition, the circular parts of the letters "p" and "q" have been extracted.

for which S_t completely fits into the region R. Hence, the erosion acts like a template matching operation. An example of this use of the erosion is shown in Figure 9.47. In Figure 9.47a, we can see an image of a print of several letters, with the structuring element used for the erosion overlaid in white. The structuring element corresponds to the center line of the letter "e." The reference point of the structuring element is its center of gravity. The result of eroding the thresholded letters (Figure 9.47b) with the structuring element is shown in Figure 9.47c. Note that all letters "e" have been correctly identified. In Figure 9.47d–f, the experiment is repeated with another set of letters. The structuring element is the center line of the letter "o." Note that the erosion correctly finds the letters "o." However, additionally the circular parts of the letters "p" and "q" are found, since the structuring element completely fits into them.

An interesting property of the Minkowski addition and subtraction as well as the dilation and erosion is that they are dual to each other with respect to the complement operation. For the Minkowski addition and subtraction, we have

$$R \oplus S = \overline{\overline{R} \ominus S} \quad \text{and} \quad R \ominus S = \overline{\overline{R} \oplus S} \tag{9.78}$$

The same identities hold for the dilation and erosion. Hence, a dilation of the foreground is identical to an erosion of the background, and vice versa. We can make use of the duality whenever we want to avoid computing the complement explicitly, and hence to speed up some operations. Note that the duality holds only if the complement can be infinite. Hence, it does not hold for binary images, where the complemented region needs to be clipped to a certain image size.

One extremely useful application of the erosion and dilation is the calculation of the boundary of a region. The algorithm to compute the true boundary as a linked list of contour points is quite complicated [22]. However, an approximation to the boundary can be computed very easily. If we want to compute the inner

boundary, we simply need to erode the region appropriately and to subtract the eroded region from the original region:

$$\partial R = R \backslash (R \ominus S) \tag{9.79}$$

By duality, the outer boundary (the inner boundary of the background) can be computed with a dilation:

$$\partial R = (R \oplus S) \backslash R \tag{9.80}$$

To get a suitable boundary, the structuring element S must be chosen appropriately. If we want to obtain an 8-connected boundary, we must use the structuring element S_8 in Figure 9.48. If we want a 4-connected boundary, we must use S_4.

Figure 9.49 displays an example of the computation of the inner boundary of a region. A small part of the input region is shown in Figure 9.49a. The boundary of the region computed by equation (9.79) with S_8 is shown in Figure 9.49b, while the result with S_4 is shown in Figure 9.49c. Note that the boundary is only approximately 8- or 4-connected. For example, in the 8-connected boundary there are occasional 4-connected pixels. Finally, the boundary of the region as computed by an algorithm that traces around the boundary of the region and links the boundary points into contours is shown in Figure 9.49d. Note that this is the true boundary of the region. Also note that, since only part of the region is displayed, there is no boundary at the bottom of the displayed part.

As we have seen, the erosion can be used as a template matching operation. However, sometimes it is not selective enough and returns too many matches. The reason for this is that the erosion does not take into account the background. For this reason, an operation that explicitly models the background is needed. This operation is called the hit-or-miss transform. Since the foreground and background should be taken into account, it uses a structuring element that consists of two parts: $S = (S^f, S^b)$ with $S^f \cap S^b = \emptyset$. With this, the hit-or-miss transform is defined as

$$R \otimes S = (R \ominus \check{S}^f) \cap (\overline{R} \ominus \check{S}^b) = (R \ominus \check{S}^f) \backslash (R \oplus \check{S}^b) \tag{9.81}$$

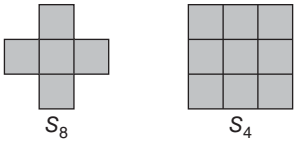


Figure 9.48 Structuring elements for computing the boundary of a region with 8-connectivity (S_8) and 4-connectivity (S_4).

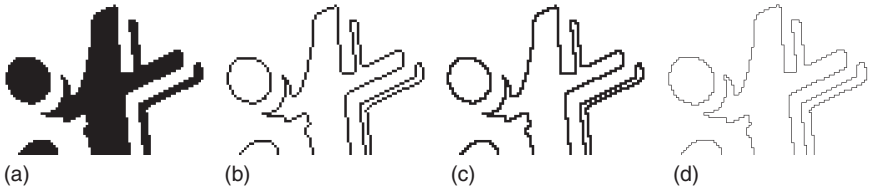


Figure 9.49 (a) Detail of a larger region. (b) The 8-connected boundary of (a) computed by equation (9.79). (c) The 4-connected boundary of (a). (d) Linked contour of the boundary of (a).

Figure 9.50 (a) Image of a print of several characters. (b) The structuring element used for the hit-or-miss transform. The black part is the foreground structuring element and the light gray part is the background structuring element. (c) Result of the hit-or-miss transform of the thresholded image (see Figure 9.47e) with the structuring element in (b). Note that only the reference point of the letter "o" has been identified, in contrast to the erosion (see Figure 9.47f).

Hence, the hit-or-miss transform returns those translated reference points for which the foreground structuring element S^f completely lies within the foreground and the background structuring element S^b completely lies within the background. The second equation is especially useful from an implementation point of view since it avoids having to compute the complement. The hit-or-miss transform is dual to itself if the foreground and background structuring elements are exchanged: $R \otimes S = \overline{R} \otimes S'$, where $S' = (S^b, S^f)$.

Figure 9.50 shows the same image as Figure 9.47d. The goal here is to match only the letters "o" in the image. To do so, we can define a structuring element that crosses the vertical strokes of the letters "p" and "q" (and also "b" and "d"). One possible structuring element for this purpose is shown in Figure 9.50b. With the hit-or-miss transform, we are able to remove the found matches for the letters "p" and "q" from the result, as can be seen from Figure 9.50c.

We now turn our attention to operations in which the basic operations we have discussed so far are executed in succession. The first such operation is the opening:

$$R \circ S = (R \ominus \check{S}) \oplus S = \bigcup_{S_t \subseteq R} S_t \tag{9.82}$$

Hence, the opening is an erosion followed by a Minkowski addition with the same structuring element. The second equation tells us that we can visualize the opening by moving the structuring element around the plane. Whenever the structuring element completely lies within the region, we add the entire translated structuring element to the output region (and not just the translated reference point as in the erosion). The opening's definition causes the location of the reference point to cancel out, which can be seen from the second equation. Therefore, the opening is translation-invariant with respect to the structuring element. In contrast to the erosion and dilation, the opening is idempotent, that is, applying it multiple times has the same effect as applying it once: $(R \circ S) \circ S = R \circ S$.

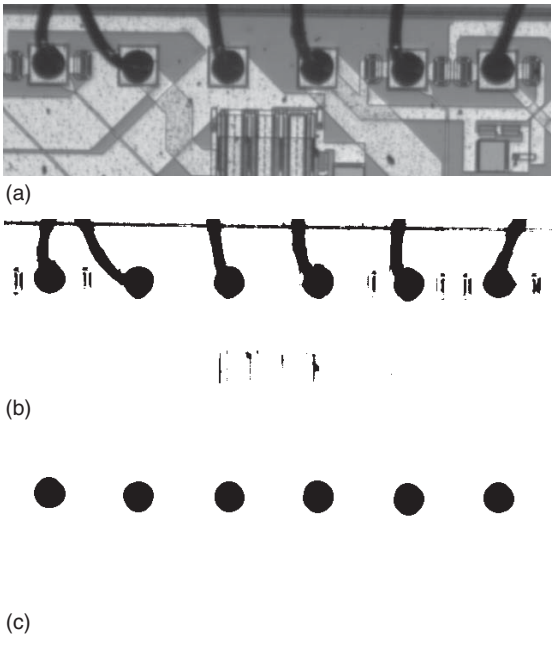
Like the erosion, the opening can be used as a template matching operation. In contrast to the erosion and hit-or-miss transform, it returns all points of the input region into which the structuring element fits. Hence it preserves the shape of the object to find. An example of this is shown in Figure 9.51, where the same input images and structuring elements as in Figure 9.47 are used. Note that the opening has found the same instances of the structuring elements as the erosion but has

(a) (b)

Figure 9.51 (a) Result of applying an opening with the structuring element in Figure 9.47a to the segmented region in Figure 9.47b. (b) Result of applying an opening with the structuring element in Figure 9.47d to the segmented region in Figure 9.47e. The result of the opening is overlaid in light gray onto the input region, displayed in black. Note that the opening finds the same instances of the structuring elements as the erosion but preserves the shape of the matched structuring elements.

preserved the shape of the matched structuring elements. Hence, in this example it also finds the letters "p" and "q." To find only the letters "o," we could combine the hit-or-miss transformation with a Minkowski addition to get a hit-or-miss opening: $R \odot S = (R \otimes S) \oplus S^f$.

Another very useful property of the opening results if structuring elements like circles or rectangles are used. If an opening with these structuring elements is performed, parts of the region that are smaller than the structuring element are removed from the region. This can be used to remove unwanted appendages from the region and to smooth the boundary of the region by removing small protrusions. Furthermore, small bridges between object parts can be removed, which can be used to separate objects. Finally, the opening can be used to suppress small objects. Figure 9.52 shows an example of using the opening to remove unwanted appendages and small objects from the segmentation. In Figure 9.52a, an image of a ball-bonded die is shown. The goal is to segment the balls on the pads. If the



(a)

(b)

(c)

Figure 9.52 (a) Image of a ball-bonded die. The goal is to segment the balls. (b) Result of thresholding (a). The segmentation includes the wires that are bonded to the pads. (c) Result of performing an opening with a circle of diameter 31. The wires and the other extraneous segmentation results have been removed by the opening, and only the balls remain.

image is thresholded (Figure 9.52b), the wires that are attached to the balls are also extracted. Furthermore, there are extraneous small objects in the segmentation. By performing an opening with a circle of diameter 31, the wires and small objects are removed, and only smooth region parts that correspond to the balls are retained.

The second interesting operation in which the basic morphological operations are executed in succession is the closing:

$$R \bullet S = (R \oplus \check{S}) \ominus S = \overline{\bigcup_{S_t \subseteq \overline{R}} S_t} \tag{9.83}$$

Hence, the closing is a dilation followed by a Minkowski subtraction with the same structuring element. There is, unfortunately, no simple formula that tells us how the closing can be visualized. The second formula is actually defined by the duality of the opening and the closing, namely a closing on the foreground is identical to an opening on the background, and vice versa:

$$R \bullet S = \overline{\overline{R} \circ S} \quad \text{and} \quad R \circ S = \overline{\overline{R} \bullet S} \tag{9.84}$$

Like the opening, the closing is translation-invariant with respect to the structuring element. Furthermore, it is also idempotent.

Since the closing is dual to the opening, it can be used to merge objects that are separated by gaps that are smaller than the structuring element. If structuring elements like circles or rectangles are used, the closing can be used to close holes and to remove indentations that are smaller than the structuring element. The second property enables us to smooth the boundary of the region.

Figure 9.53 shows how the closing can be used to remove indentations in a region. In Figure 9.53a, a molded plastic part with a protrusion is shown. The goal is to detect the protrusion because it is a production error. Since the actual object is circular, if the entire part were visible, the protrusion could be detected by performing an opening with a circle that is almost as large as the object and then subtracting the opened region from the original segmentation. However, only a part of the object is visible, so the erosion in the opening would create artifacts or remove the object entirely. Therefore, by duality we can pursue the opposite approach: we can segment the background and perform a closing on it. Figure 9.53b shows the result of thresholding the background. The protrusion is now an indentation in the background. The result of performing a closing with a circle of diameter 801 is shown in Figure 9.53c. The diameter of the circle was set to 801 because it is large enough to completely fill the indentation and to recover the circular shape of the object. If much smaller circles were used, for example, with a diameter of 401, the indentation would not be filled completely. To detect the error itself, we can compute the difference between the closing and the original segmentation. To remove some noisy pixels that result because the boundary of the original segmentation is not as smooth as the closed region, the difference can be postprocessed with an opening, for example, with a 5×5 rectangle, to remove the noisy pixels. The resulting error region is shown in Figure 9.53d.

The operations we have discussed so far have been mostly concerned with the region as a 2D object. The only exception has been the calculation of the boundary

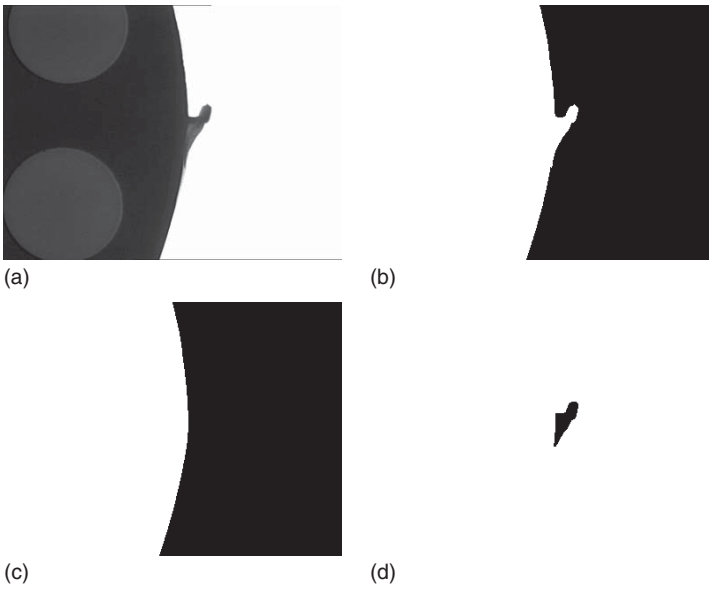


Figure 9.53 (a) Image of size 768 × 576 showing a molded plastic part with a protrusion. (b) Result of thresholding the background of (a). (c) Result of a closing on (b) with a circle of diameter 801. Note that the protrusion (the indentation in the background) has been filled in and the circular shape of the plastic part has been recovered. (d) Result of computing the difference between (c) and (b) and performing an opening with a 5 × 5 rectangle on the difference to remove small parts. The result is the erroneous protrusion of the mold.

of a region, which reduces a region to its 1D outline, and hence gives a more condensed description of the region. If the objects are mostly linear, that is, are regions that have a much greater length than width, a more salient description of the object would be obtained if we could somehow capture its one-pixel-wide center line. This center line is called the skeleton or medial axis of the region. Several definitions of a skeleton can be given [35]. One intuitive definition can be obtained if we imagine that we try to fit circles that are as large as possible into the region. More precisely, a circle C is maximal in the region R if there is no other circle in R that is a superset of C. The skeleton then is defined as the set of the centers of the maximal circles. Consequently, a point on the skeleton has at least two different points on the boundary of the region to which it has the same shortest distance. Algorithms to compute the skeleton are given in [35, 37]. They basically can be regarded as sequential hit-or-miss transforms that find points on the boundary of the region that cannot belong to the skeleton and delete them. The goal of the skeletonization is to preserve the homotopy of the region, that is, the number of connected components and holes. One set of structuring elements for computing an 8-connected skeleton is shown in Figure 9.54 [35]. These structuring elements are used sequentially in all four possible orientations to find pixels with the hit-or-miss transform that can be deleted from the region. The iteration is continued until no changes occur. It should be noted

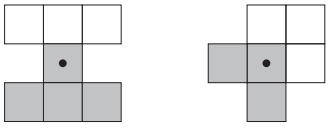


Figure 9.54 Structuring elements for computing an 8-connected skeleton of a region. These structuring elements are used sequentially in all four possible orientations to find pixels that can be deleted.

Figure 9.55 (a) Image showing a part of a PCB with several tracks. (b) Result of thresholding (a). (c) The 8-connected skeleton computed with an algorithm that uses the structuring elements in Figure 9.54 [35]. (d) Result of computing the skeleton with an algorithm that produces fewer skeleton branches [38].

that skeletonization is an example of an algorithm that can be implemented more efficiently on binary images than on the run-length representation.

Figure 9.55a shows a part of an image of a PCB with several tracks. The image is thresholded (Figure 9.55b), and the skeleton of the thresholded region is computed with the above algorithm (Figure 9.55c). Note that the skeleton contains several undesirable branches on the upper two tracks. For this reason, many different skeletonization algorithms have been proposed. One algorithm that produces relatively few unwanted branches is described in [38]. The result of this algorithm is shown in Figure 9.55d. Note that there are no undesirable branches in this case.

The final region morphology operation that we will discuss is the distance transform, which returns an image instead of a region. This image contains, for each point in the region R, the shortest distance to a point outside the region (i.e., to $\overline{R}$). Consequently, all points on the inner boundary of the region have a distance of 1. Typically, the distance of the other points is obtained by considering paths that must be contained in the pixel grid. Thus, the chosen connectivity defines which paths are allowed. If the 4-connectivity is used, the corresponding distance is called the city-block distance. Let (r_1, c_1) and (r_2, c_2) be two points. Then the city-block distance is given by

$$d_4 = |r_2 - r_1| + |c_2 - c_1| \tag{9.85}$$

Figure 9.56a shows the city-block distance between two points. In the example, the city-block distance is 5. On the other hand, if 8-connectivity is used, the

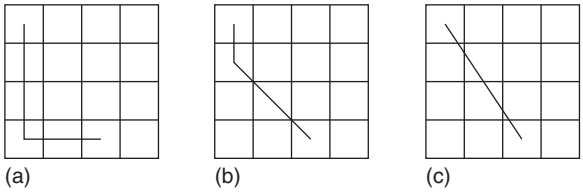


Figure 9.56 (a) City-block distance between two points. (b) Chessboard distance. (c) Euclidean distance.

corresponding distance is called the chessboard distance. It is given by

$$d_8 = \max\{|r_2 - r_1|, |c_2 - c_1|\} \tag{9.86}$$

In the example in Figure 9.56b, the chessboard distance between the two points is 3. Both these distances are approximations to the Euclidean distance, given by

$$d_e = \sqrt{(r_2 - r_1)^2 + (c_2 - c_1)^2} \tag{9.87}$$

For the example in Figure 9.56c, the Euclidean distance is $\sqrt{13}$.

Algorithms to compute the distance transform are described in [39]. They work by initializing the distance image outside the region with 0 and within the region with a suitably chosen maximum distance, that is, $2^b - 1$, where b is the number of bits in the distance image, for example, $2^{16} - 1$. Then, two sequential line-by-line scans through the image are performed, one from the top left to the bottom right corner, and the second in the opposite direction. In each case, a small mask is placed at the current pixel, and the minimum over the elements in the mask of the already computed distances plus the elements in the mask is computed. The two masks are shown in Figure 9.57. If $d_1 = 1$ and $d_2 = \infty$ are used (i.e., d_2 is ignored), the city-block distance is computed. For $d_1 = 1$ and $d_2 = 1$, the chessboard distance results. Interestingly, if $d_1 = 3$ and $d_2 = 4$ is used and the distance image is divided by 3, a very good approximation to the Euclidean distance results, which can be computed solely with integer operations. This distance is called the chamfer-$3-4$ distance [39]. With slight modifications, the true Euclidean distance can be computed [40]. The principle is to compute the number of horizontal and vertical steps to reach the boundary using masks similar to the ones in Figure 9.57, and then to compute the Euclidean distance from the number of steps.

The skeleton and the distance transform can be combined to compute the width of linear objects very efficiently. In Figure 9.58a, a PCB with tracks that have several errors is shown. The protrusions on the tracks are called spurs, while the indentations are called mouse bites [41]. They are deviations from the correct track width. Figure 9.58b shows the result of computing the distance transform with the chamfer-$3-4$ distance on the segmented tracks. The errors are clearly visible in the distance transform. To extract the width of the tracks, we need to calculate the skeleton of the segmented tracks (Figure 9.58c). If the skeleton is used as the ROI for the distance image, each point on the skeleton will have the corresponding distance to the border of the track. Since the skeleton is the center line of the track, this distance is the width of the track. Hence, to detect errors, we simply need to threshold the distance image within the skeleton. Note that in this example it is extremely useful that we have defined that images can have an

d_2	d_1	d_2
d_1	0	

	0	d_1
d_2	d_1	d_2

Figure 9.57 Masks used in the two sequential scans to compute the distance transform. The left mask is used in the left-to-right, top-to-bottom scan. The right mask is used in the scan in the opposite direction.

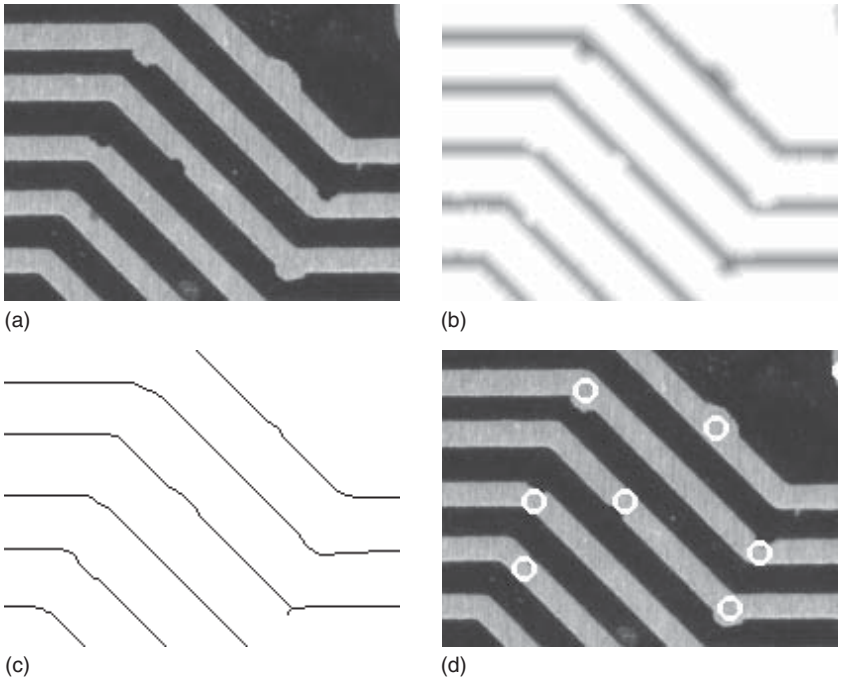


Figure 9.58 (a) Image showing a part of a PCB with several tracks that have spurs and mouse bites. (b) Distance transform of the result of thresholding (a). The distance image is visualized inverted (dark gray values correspond to large distances). (c) Skeleton of the segmented region. (d) Result of extracting too narrow or too wide parts of the tracks by using (c) as the ROI for (b) and thresholding the distances. The errors are visualized by drawing circles at the centers of gravity of the connected components of the error region.

arbitrary ROI. Figure 9.58d shows the result of drawing circles at the centers of gravity of the connected components of the error region. All major errors have been detected correctly.

9.6.2 Gray Value Morphology

Because morphological operations are very versatile and useful, the question of whether they can be extended to gray value images arises quite naturally. This can indeed be done. In analogy to the region morphology, let $g(r, c)$ denote the image that should be processed, and let $s(r, c)$ be an image with ROI S. Like in the region morphology, the image s is called the structuring element. The gray value Minkowski addition is then defined as

$$g \oplus s = (g \oplus s)_{r,c} = \max_{(i,j) \in S} \{g_{r-i,c-j} + s_{i,j}\} \tag{9.88}$$

This is a natural generalization because the Minkowski addition for regions is obtained as a special case if the characteristic function of the region is used as

the gray value image. If, additionally, an image with gray value 0 within the ROI S is used as the structuring element, the Minkowski addition becomes

$$g \oplus s = \max_{(i,j)\in S} \{g_{r-i,c-j}\} \tag{9.89}$$

For characteristic functions, the maximum operation corresponds to the union. Furthermore, $g_{r-i,c-j}$ corresponds to the translation of the image by the vector (i, j). Hence, equation (9.89) is equivalent to the second formula in equation (9.73).

Like in the region morphology, the dilation can be obtained by transposing the structuring element. This results in the following definition:

$$g \oplus \check{s} = (g \oplus \check{s})_{r,c} = \max_{(i,j)\in S} \{g_{r+i,c+j} + s_{i,j}\} \tag{9.90}$$

The typical choice for the structuring element in the gray value morphology is the flat structuring element that was already used previously: $s(r, c) = 0$ for $(r, c) \in S$. With this, the gray value dilation has a similar effect as the region dilation: it enlarges the foreground, that is, parts in the image that are brighter than their surroundings, and shrinks the background, that is, parts in the image that are darker than their surroundings. Hence, it can be used to connect disjoint parts of a bright object in the gray value image. This is sometimes useful if the object cannot be segmented easily using region operations alone. Conversely, the dilation can be used to split dark objects.

The Minkowski subtraction for gray value images is given by

$$g \ominus s = (g \ominus s)_{r,c} = \min_{(i,j)\in S} \{g_{r-i,c-j} - s_{i,j}\} \tag{9.91}$$

As above, by transposing the structuring element we obtain the gray value erosion:

$$g \ominus \check{s} = (g \ominus \check{s})_{r,c} = \min_{(i,j)\in S} \{g_{r+i,c+j} - s_{i,j}\} \tag{9.92}$$

Like the region erosion, the gray value erosion shrinks the foreground and enlarges the background. Hence, the erosion can be used to split touching bright objects and to connect disjoint dark objects. In fact, the dilation and erosion, as well as the Minkowski addition and subtraction, are dual to each other, like for regions. For the duality, we need to define what the complement of an image should be. If the images are stored with b bits, the natural definition for the complement operation is $\overline{g}_{r,c} = 2^b - 1 - g_{r,c}$. With this, it can be easily shown that the erosion and dilation are dual:

$$g \oplus s = \overline{\overline{g} \ominus s} \quad \text{and} \quad g \ominus s = \overline{\overline{g} \oplus s} \tag{9.93}$$

Therefore, all the properties that hold for one operation for bright objects hold for the other operation for dark objects, and vice versa.

Note that the dilation and erosion can also be regarded as two special rank filters (see Section 9.2.3) if flat structuring elements are used. They select the minimum and maximum gray values within the domain of the structuring element, which can be regarded as the filter mask. Therefore, the dilation and erosion are sometimes referred to as the maximum and minimum filters (or max and min filters).

Efficient algorithms to compute the dilation and erosion are given in [15]. Their runtime complexity is $O(whn)$, where w and h are the dimensions of the image, while n is roughly the number of points on the boundary of the domain of the structuring element for flat structuring elements. For rectangular structuring elements, algorithms with a runtime complexity of $O(wh)$, that is, with a constant number of operations per pixel, can be found [42]. This is similar to the recursive implementation of a linear filter.

With these building blocks, we can define a gray value opening like for regions as an erosion followed by a Minkowski addition: that is

$$g \circ s = (g \ominus \check{s}) \oplus s \tag{9.94}$$

and the closing as a dilation followed by a Minkowski subtraction: that is

$$g \bullet s = (g \oplus \check{s}) \ominus s \tag{9.95}$$

The gray value opening and closing have properties similar to those of their region counterparts. In particular, with the above definition of the complement for images, they are dual to each other:

$$g \circ s = \overline{\overline{g} \bullet s} \quad \text{and} \quad g \bullet s = \overline{\overline{g} \circ s} \tag{9.96}$$

Like the region operations, they can be used to fill in small holes or, by duality, to remove small objects. Furthermore, they can be used to join or separate objects and to smooth the inner and outer boundaries of objects in the gray value image.

Figure 9.59 shows how the gray value opening and closing can be used to detect errors in the tracks on a PCB. We have already seen in Figure 9.58 that some of these errors can be detected by looking at the width of the tracks with the distance transform and the skeleton. This technique is very useful because it enables us to detect relatively large areas with errors. However, small errors are harder to detect with this technique because the distance transform and skeleton are only pixel-precise, and consequently the width of the track can be determined reliably with a precision of only two pixels. Smaller errors can be detected more reliably with the gray value morphology. Figure 9.59a shows a part of a PCB with several tracks that have spurs, mouse bites, pinholes, spurious copper, and open and short circuits [41]. The results of performing a gray value opening and closing with an octagon of diameter 11 are shown in Figure 9.59b,c. Because of the horizontal, vertical, and diagonal layout of the tracks, using an octagon as the structuring element is preferable. It can be seen that the opening smooths out the spurs, while the closing smooths out the mouse bites. Furthermore, the short circuit and spurious copper are removed by the opening, while the pinhole and open circuit are removed by the closing. To detect these errors, we can require that the opened and closed images should not differ too much. If there were no errors, the differences would solely be caused by the texture on the tracks. Since the gray values of the opened image are always smaller than those of the closed image, we can use the dynamic threshold operation for bright objects (Equation (9.46)) to perform the required segmentation. Every pixel that has a gray value difference greater than g_{diff} can be considered as an error. Figure 9.59d shows the result of segmenting the errors using a dynamic threshold $g_{\text{diff}} = 60$. This detects all the errors on the board.

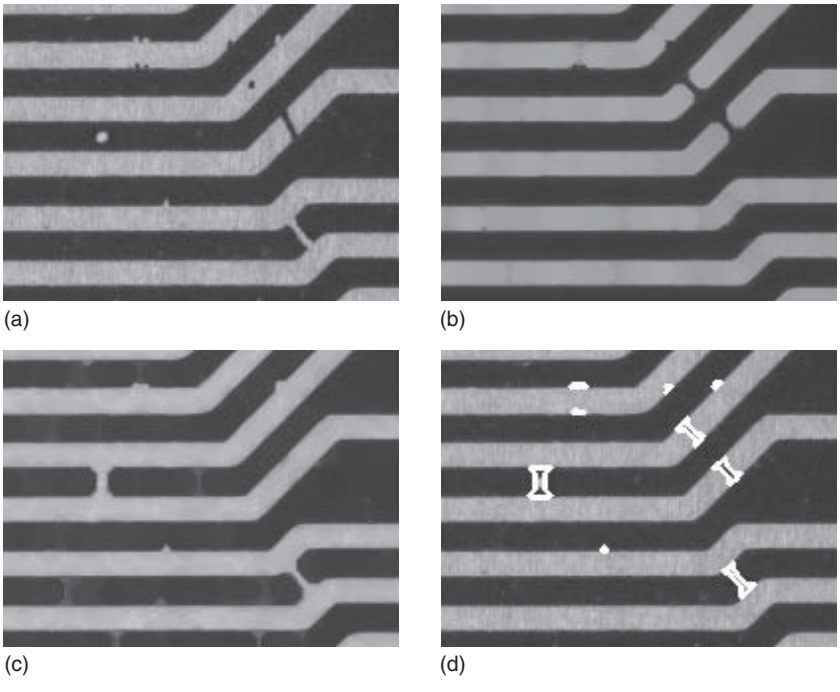


Figure 9.59 (a) Image showing a part of a PCB with several tracks that have spurs, mouse bites, pinholes, spurious copper, and open and short circuits. (b) Result of performing a gray value opening with an octagon of diameter 11 on (a). (c) Result of performing a gray value closing with an octagon of diameter 11 on (a). (d) Result of segmenting the errors in (a) by using a dynamic threshold operation with the images of (b) and (c).

We conclude this section with an operator that computes the range of gray values that occur within the structuring element. This can be obtained easily by calculating the difference between the dilation and erosion:

$$g \diamond s = (g \oplus \check{s}) - (g \ominus \check{s}) \tag{9.97}$$

Since this operator produces results similar to those of a gradient filter (see Section 9.7), it is sometimes called the morphological gradient.

Figure 9.60 shows how the gray range operator can be used to segment punched serial numbers. Because of the scratches, texture, and illumination, it is difficult to segment the characters in Figure 9.60a directly. In particular, the scratch next to the upper left part of the "2" cannot be separated from the "2" without splitting several of the other numbers. The result of computing the gray range within a 9×9 rectangle is shown in Figure 9.60b. With this, it is easy to segment the numbers (Figure 9.60c) and to separate them from other segmentation results (Figure 9.60d).

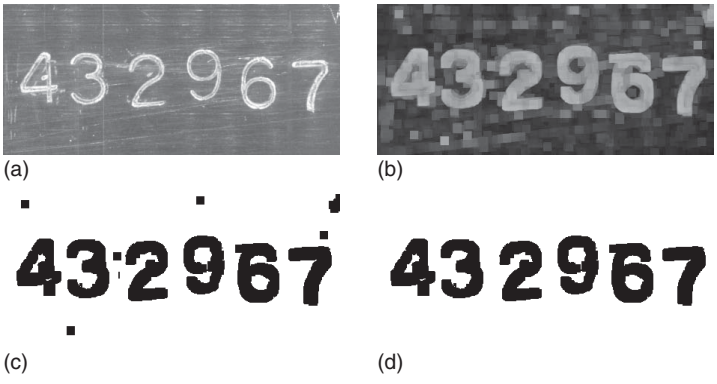


Figure 9.60 (a) Image showing a punched serial number. Because of the scratches, texture, and illumination, it is difficult to segment the characters directly. (b) Result of computing the gray range within a 9×9 rectangle. (c) Result of thresholding (b). (d) Result of computing the connected components of (c) and selecting the characters based on their size.

9.7 Edge Extraction

In Section 9.4, we discussed several segmentation algorithms. They have in common that they are based on thresholding the image, with either pixel or subpixel accuracy. It is possible to achieve very good accuracies with these approaches, as we saw in Section 9.5. However, in most cases the accuracy of the measurements that we can derive from the segmentation result critically depends on choosing the correct threshold for the segmentation. If the threshold is chosen incorrectly, the extracted objects typically become larger or smaller because of the smooth transition from the foreground to the background gray value. This problem is especially grave if the illumination can change, since in this case the adaptation of the thresholds to the changed illumination must be very accurate. Therefore, a segmentation algorithm that is robust with respect to illumination changes is extremely desirable. From the above discussion, we see that the boundary of the segmented region or subpixel-precise contour moves if the illumination changes or the thresholds are chosen inappropriately. Therefore, the goal of a robust segmentation algorithm must be to find the boundary of the objects as robustly and accurately as possible. The best way to describe the boundaries of the objects robustly is by regarding them as edges in the image. Therefore, in this section we will examine methods to extract edges.

9.7.1 Definition of Edges in One and Two Dimensions

To derive an edge extraction algorithm, we need to define what edges actually are. For the moment, let us make the simplifying assumption that the gray values in the object and in the background are constant. In particular, we assume that

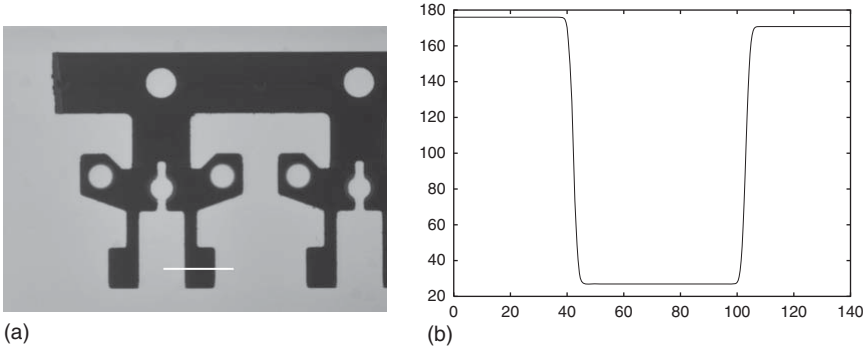


Figure 9.61 (a) Image of a back-lit workpiece with a horizontal line that indicates the location of the idealized gray value profile in (b).

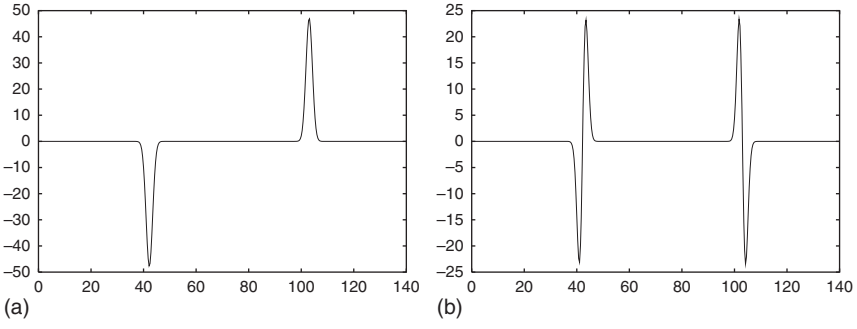


Figure 9.62 (a) First derivative $f'(x)$ of the ideal gray value profile in Figure 9.61b. (b) Second derivative $f'(x)$.

the image contains no noise. Furthermore, let us assume that the image is not discretized, that is, it is continuous. To illustrate this, Figure 9.61b shows an idealized gray value profile across the part of a workpiece that is indicated in Figure 9.61a.

From the above example, we can see that edges are areas in the image in which the gray values change significantly. To formalize this, let us regard the image for the moment as a 1D function $f(x)$. From elementary calculus we know that the gray values change significantly if the first derivative of $f(x)$ differs significantly from 0, that is, $|f'(x)| \gg 0$. Unfortunately, this alone is insufficient to define a unique edge location because there are typically many connected points for which this condition is true since the transition between the background and foreground gray values is smooth. This can be seen in Figure 9.62a, where the first derivative $f'(x)$ of the ideal gray value profile in Figure 9.61b is displayed. Note, for example, that there is an extended range of points for which $|f'(x)| \geq 20$. Therefore, to obtain a unique edge position, we must additionally require that the absolute value of the first derivative $|f'(x)|$ is locally maximal. This is called non-maximum suppression.

From elementary calculus we know that, at the points where $|f'(x)|$ is locally maximal, the second derivative vanishes: $f''(x) = 0$. Hence, edges are given by the locations of inflection points of $f(x)$. To remove flat inflection points, we

would additionally have to require that $f'(x)f''(x) < 0$. However, this restriction is seldom observed. Therefore, in one dimension, an alternative and equivalent definition to the maxima of the absolute value of the first derivative is to define edges as the locations of the zero-crossings of the second derivative. Figure 9.62b displays the second derivative $f''(x) = 0$ of the ideal gray value profile in Figure 9.61b. Clearly, the zero-crossings are in the same positions as the maxima of the absolute value of the first derivative in Figure 9.62a.

From Figure 9.62a, we can also see that in one dimension we can easily associate a polarity with an edge based on the sign of $f'(x)$. We speak of a positive edge if $f'(x) > 0$ and of a negative edge if $f'(x) < 0$.

We now turn to edges in continuous 2D images. Here, the edge itself is a curve $s(t) = (r(t), c(t))$, which is parameterized by a parameter t, for example, its arc length. At each point of the edge curve, the gray value profile perpendicular to the curve is a 1D edge profile. With this, we can adapt the first 1D edge definition above for the 2D case: we define an edge as the points in the image where the directional derivative in the direction perpendicular to the edge is locally maximal. From differential geometry, we know that the direction $n(t)$ perpendicular to the edge curve $s(t)$ is given by $n(t) = s'(t)^{\perp} \parallel s''(t)$. Unfortunately, the edge definition seemingly requires us to know the edge position $s(t)$ already to obtain the direction perpendicular to the edge, and hence looks like a circular definition. Fortunately, the direction $n(t)$ perpendicular to the edge can be determined easily from the image itself. It is given by the gradient vector of the image, which points in the direction of steepest ascent of the image function $f(r, c)$. The gradient of the image is given by the vector of its first partial derivatives:

$$\nabla f = \nabla f(r, c) = \left(\frac{\partial f(r, c)}{\partial r}, \frac{\partial f(r, c)}{\partial c} \right) = (f_r, f_c) \tag{9.98}$$

In the last equation, we have used a subscript to denote the partial derivative with respect to the subscripted variable. We will use this convention throughout this section. The Euclidean length

$$\| \nabla f \|_2 = \sqrt{f_r^2 + f_c^2} \tag{9.99}$$

of the gradient vector is the equivalent of the absolute value of the first derivative $|f'(x)|$ in one dimension. We will also call the length of the gradient vector its magnitude. It is also often called the amplitude. The gradient direction is, of course, directly given by the gradient vector. We can also convert it to an angle by calculating $\phi = -\arctan(f_r/f_c)$. Note that ϕ increases in the mathematically positive direction (counterclockwise), starting at the column axis. This is the usual convention. With the above definitions, we can define edges in two dimensions as the points in the image where the gradient magnitude is locally maximal in the direction of the gradient. To illustrate this definition, Figure 9.63a shows a plot of the gray values of an idealized corner. The corresponding gradient magnitude is shown in Figure 9.63b. The edges are the points at the top of the ridge in the gradient magnitude.

In one dimension, we have seen that the second edge definition (the zero-crossings of the second derivative) is equivalent to the first definition.

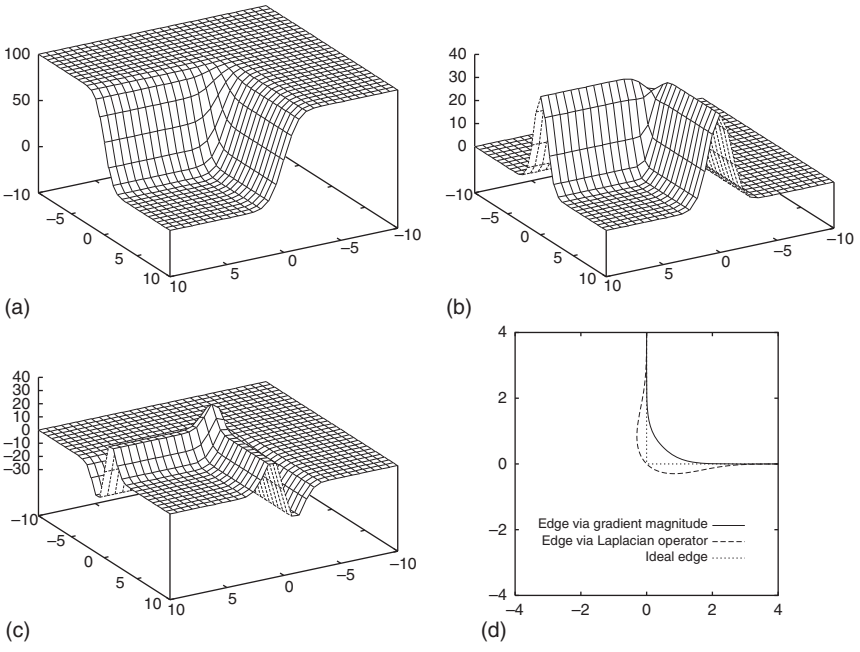


Figure 9.63 (a) Image of an idealized corner, for example, one of the corners at the bottom of the workpiece in Figure 9.61a. (b) Gradient magnitude of (a). (c) Laplacian of (a). (d) Comparison of the edges that result from the two definitions in two dimensions.

Therefore, it is natural to ask whether this definition can be adapted for the 2D case. Unfortunately, there is no direct equivalent for the second derivative in two dimensions, since there are three partial derivatives of order two. A suitable definition for the second derivative in two dimensions is the Laplacian operator (Laplacian for short), defined by

$$\Delta f = \Delta f(r,c) = \frac{\partial^2 f(r,c)}{\partial r^2} + \frac{\partial^2 f(r,c)}{\partial c^2} = f_{rr} + f_{cc} \tag{9.100}$$

With this, the edges can be defined as the zero-crossings of the Laplacian: $\Delta f(r,c) = 0$. Figure 9.63c shows the Laplacian of the idealized corner in Figure 9.63a. The results of the two edge definitions are shown in Figure 9.63d. It can be seen that, unlike for the 1D edges, the two definitions do not result in the same edge positions. The edge positions are identical only for straight edges. Whenever the edge is significantly curved, the two definitions return different results. It can be seen that the definition via the maxima of the gradient magnitude always lies inside the ideal corner, whereas the definition via the zero-crossings of the Laplacian always lies outside the corner and passes directly through the ideal corner. The Laplacian edge also is in a different position from the true edge for a larger part of the edge. Therefore, in two dimensions the definition via the maxima of the gradient magnitude is usually preferred. However, in some applications the fact that the Laplacian edge passes through the corner can be used to measure objects with sharp corners more accurately.

9.7.2 1D Edge Extraction

We now turn our attention to edges in real images, which are discrete and contain noise. In this section, we will discuss how to extract edges from 1D gray value profiles. This is a very useful operation that is used frequently in machine vision applications because it is extremely fast. It is typically used to determine the position or diameter of an object.

The first problem we have to address is how to compute the derivatives of the discrete 1D gray value profile. Our first idea might be to use the differences of consecutive gray values on the profile: $f_i' = f_i - f_{i-1}$. Unfortunately, this definition is not symmetric. It would compute the derivative at the "half-pixel" positions $f_{i-(1/2)}$. A symmetric way to compute the first derivative is given by

$$f_i' = \frac{1}{2}(f_{i+1} - f_{i-1}) \tag{9.101}$$

This formula is obtained by fitting a parabola through three consecutive points of the profile and computing the derivative of the parabola at the center point. The parabola is uniquely defined by the three points. With the same mechanism, we can also derive a formula for the second derivative:

$$f_i'' = \frac{1}{2}(f_{i+1} - 2f_i + f_{i-1}) \tag{9.102}$$

Note that the above methods to compute the first and second derivatives are linear filters, and hence can be regarded as the following two convolution masks:

$$\frac{1}{2} \cdot (1 \;\; 0 \;\; -1) \quad \text{and} \quad \frac{1}{2} \cdot (1 \;\; -2 \;\; 1) \tag{9.103}$$

Note that the -1 is the last element in the first derivative mask because the elements of the mask are mirrored in the convolution (see equation (9.18)).

Figure 9.64a displays the true gray value profile taken from the horizontal line in the image in Figure 9.61a. Its first derivative, computed with equation (9.101), is shown in Figure 9.64b. We can see that the noise in the image causes a very large number of local maxima in the absolute value of the first derivative, and consequently also a large number of zero-crossings in the second derivative. The salient edges can easily be selected by thresholding the absolute value of the first derivative: $|f_i'| \geq t$. For the second derivative, the edges cannot be selected

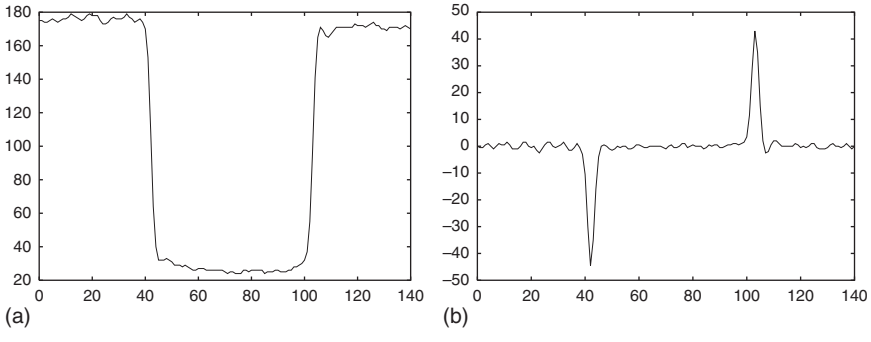


Figure 9.64 (a) Gray value profile taken from the horizontal line in the image in Figure 9.61a. (b) First derivative f_i' of the gray value profile.

as easily. In fact, we have to resort to calculating the first derivative as well to be able to select the relevant edges. Hence, the edge definition via the first derivative is preferable because it can be done with one filter operation instead of two, and consequently the edges can be extracted much faster.

The gray value profile in Figure 9.64a already contains relatively little noise. Nevertheless, in most cases it is desirable to suppress the noise even further. If the object we are measuring has straight edges in the part in which we are performing the measurement, we can use the gray values perpendicular to the line along which we are extracting the gray value profile and average them in a suitable manner. The simplest way to do this is to compute the mean of the gray values perpendicular to the line. If, for example, the line along which we are extracting the gray value profile is horizontal, we can calculate the mean in the vertical direction as follows:

$$f_i = \frac{1}{2m+1} \sum_{j=-m}^{m} f_{r+j,c+i} \tag{9.104}$$

This acts like a mean filter in one direction. Hence, the noise variance is reduced by a factor of $2m + 1$. Of course, we could also use a 1D Gaussian filter to average the gray values. However, since this would require larger filter masks for the same noise reduction, and consequently would lead to longer execution times, in this case the mean filter is preferable.

If the line along which we want to extract the gray value profile is horizontal or vertical, the calculation of the profile is simple. If we want to extract the profile from inclined lines or from circles or ellipses, the computation is slightly more difficult. To enable meaningful measurements for distances, we must sample the line with a fixed distance, typically one pixel. Then, we need to generate lines perpendicular to the curve along which we want to extract the profile. This procedure is shown for an inclined line in Figure 9.65. Because of this, the points from which we must extract the gray values typically do not lie on pixel centers. Therefore, we will have to interpolate them. This can be done with the techniques discussed in Section 9.3.3, that is, with nearest-neighbor or bilinear interpolation.

Figure 9.66 shows a gray value profile and its first derivative obtained by vertically averaging the gray values along the line shown in Figure 9.61a. The size of the 1D mean filter was 21 pixels in this case. If we compare this with Figure 9.64, which shows the profile obtained from the same line without averaging, we can see that the noise in the profile has been reduced significantly. Because of this, the salient edges are even easier to select than without the averaging.

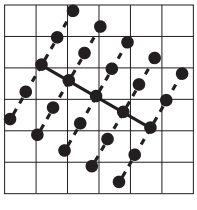


Figure 9.65 Creation of the gray value profile from an inclined line. The line is shown by the heavy solid line. The circles indicate the points that are used to compute the profile. Note that they do not lie on the pixel centers. The direction in which the 1D mean is computed is shown by the dashed lines.

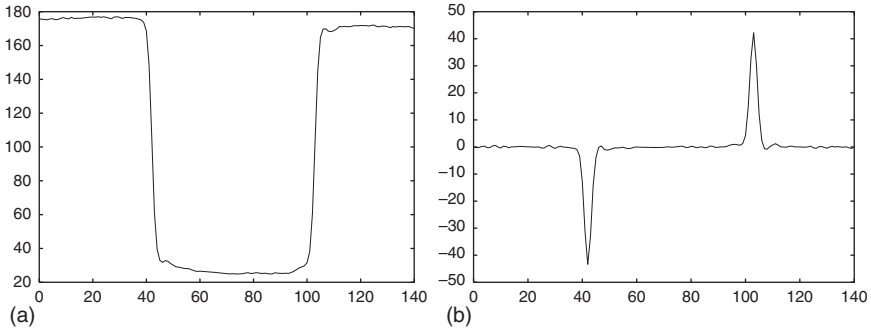
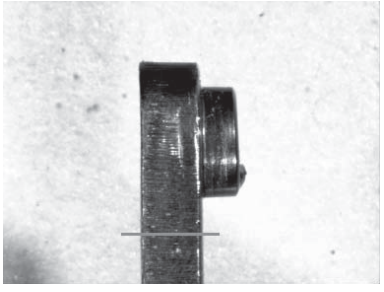


Figure 9.66 (a) Gray value profile taken from the horizontal line in the image in Figure 9.61a and averaged vertically over 21 pixels. (b) First derivative f_i' of the gray value profile.



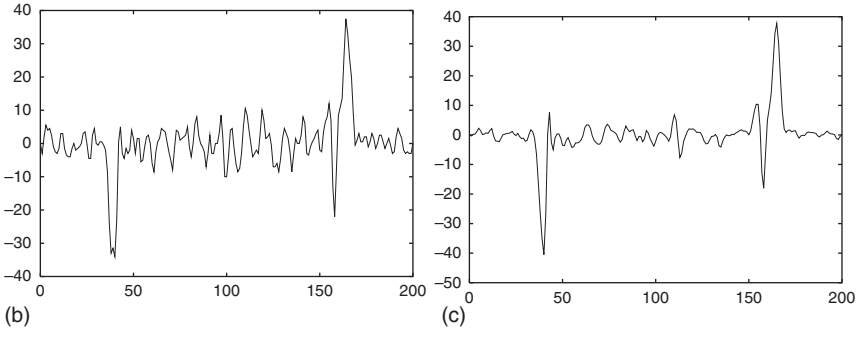


Figure 9.67 (a) Image of a relay with a horizontal line that indicates the location of the gray value profile. (b) First derivative of the gray value profile without averaging. (c) First derivative of the gray value profile with vertical averaging over 21 pixels.

Unfortunately, the averaging perpendicular to the curve along which the gray value profile is extracted is sometimes insufficient to smooth the profiles enough to enable us to extract the relevant edges easily. One example is shown in Figure 9.67. Here, the object to be measured has a significant amount of texture, which is not as random as noise and consequently does not average out completely. Note that, on the right side of the profile, there is a negative edge with an amplitude almost as large as the edges we want to extract. Another reason for the noise not to cancel out completely may be that we cannot choose the

size of the averaging large enough, for example, because the object's boundary is curved.

To solve these problems, we must smooth the gray value profile itself to suppress the noise even further. This is done by convolving the profile with a smoothing filter: $f_s = f * h$. We can then extract the edges from the smoothed profile via its first derivative. This would involve two convolutions: one for the smoothing filter, and the other for the derivative filter. Fortunately, the convolution has a very interesting property that we can use to save one convolution. The derivative of the smoothed function is identical to the convolution of the function with the derivative of the smoothing filter: $(f * h)' = f * h'$. We can regard h' as an edge filter.

Like for the smoothing filters, the natural question to ask is which edge filter is optimal. This problem was addressed by Canny [43]. He proposed three criteria that an edge detector should fulfill. First of all, it should have a good detection quality, that is, it should have a low probability of falsely detecting an edge point and also a low probability of erroneously missing an edge point. This criterion can be formalized as maximizing the signal-to-noise ratio of the output of the edge filter. Second, the edge detector should have good localization quality, that is, the extracted edges should be as close as possible to the true edges. This can be formalized by minimizing the variance of the extracted edge positions. Finally, the edge detector should return only a single edge for each true edge, that is, it should avoid multiple responses. This criterion can be formalized by maximizing the distance between the extracted edge positions. Canny then combined these three criteria into one optimization criterion and solved it using the calculus of variations. To do so, he assumed that the edge filter has a finite extent (mask size). Since adapting the filter to a particular mask size involves solving a relatively complex optimization problem, Canny looked for a simple filter that could be written in closed form. He found that the optimal edge filter can be approximated very well with the first derivative of the Gaussian filter:

$$g'_\sigma(x) = \frac{-x}{\sqrt{2\pi}\ \sigma^3}\ e^{-x^2/(2\sigma^2)} \tag{9.105}$$

One drawback of using the true derivative of the Gaussian filter is that the edge amplitudes become progressively smaller as σ is increased. Ideally, the edge filter should return the true edge amplitude independent of the smoothing. To achieve this for an idealized step edge, the output of the filter must be multiplied with $\sqrt{2\pi}\ \sigma$.

Note that the optimal smoothing filter would be the integral of the optimal edge filter, that is, the Gaussian smoothing filter. It is interesting to note that, like the criteria in Section 9.2.3, Canny's formulation indicates that the Gaussian filter is the optimal smoothing filter.

Since the Gaussian filter and its derivatives cannot be implemented recursively (see Section 9.2.3), Deriche used Canny's approach to find optimal edge filters that can be implemented recursively [44]. He derived the following two filters:

$$\begin{aligned} d'_\alpha(x) &= -\alpha^2 x\ e^{-\alpha|x|} \\ e'_\alpha(x) &= -2\alpha \sin(\alpha x)\ e^{-\alpha|x|} \end{aligned} \tag{9.106}$$

The corresponding smoothing filters are

$$\begin{aligned} d_\alpha(x) &= \tfrac{1}{4}\alpha(\alpha|x| + 1)\, e^{-\alpha|x|} \\ e_\alpha(x) &= \tfrac{1}{2}\alpha[\sin(\alpha|x|) + \cos(\alpha|x|)]\, e^{-\alpha|x|} \end{aligned} \tag{9.107}$$

Note that, in contrast to the Gaussian filter, where larger values for σ indicate more smoothing, in the Deriche filters smaller values for α indicate more smoothing. The Gaussian filter has similar effects to the first Deriche filter for $\sigma = \sqrt{\pi}/\alpha$. For the second Deriche filter, the relation is $\sigma = \sqrt{\pi}/(2\alpha)$. Note that the Deriche filters are significantly different from the Canny filter. This can also be seen from Figure 9.68, which compares the Canny and Deriche smoothing and edge filters with equivalent filter parameters.

Figure 9.69 shows the result of using the Canny edge detector with $\sigma = 1.5$ to compute the smoothed first derivative of the gray value profile in Figure 9.67a. Like in Figure 9.67c, the profile was obtained by averaging over 21 pixels vertically. Note that the amplitude of the unwanted edge on the right side of the profile has been reduced significantly. This enables us to select the salient edges more easily.

To extract the edge position, we need to perform the non-maximum suppression. If we are only interested in the edge positions with pixel accuracy, we can

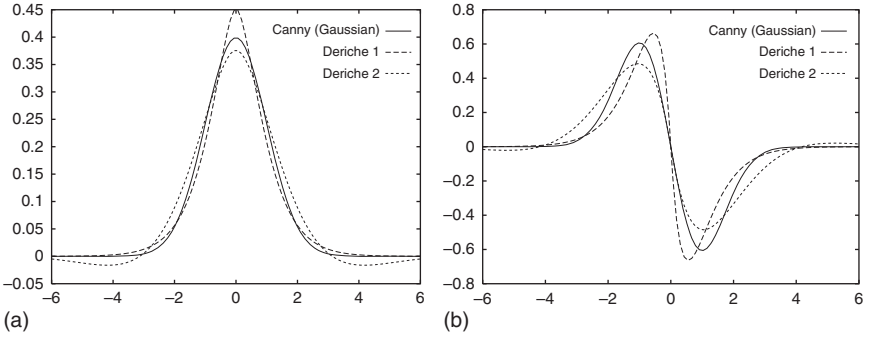
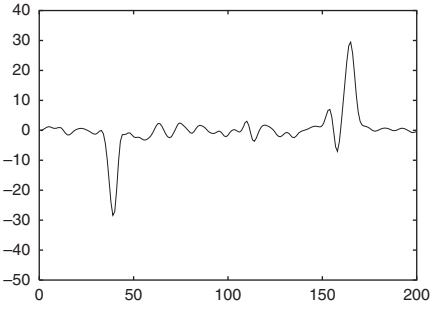


Figure 9.68 Comparison of the Canny and Deriche filters. (a) Smoothing filters. (b) Edge filters.

Figure 9.69 Result of applying the Canny edge filter with $\sigma = 1.5$ to the gray value profile in Figure 9.67a with vertical averaging over 21 pixels.



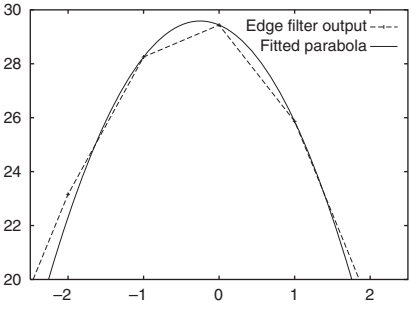


Figure 9.70 Principle of extracting edge points with subpixel accuracy. The local maximum of the edge amplitude is detected. Then, a parabola is fitted through the three points around the maximum. The maximum of the parabola is the subpixel-accurate edge location. The edge amplitude was taken from the right edge in Figure 9.69.

proceed as follows. Let the output of the edge filter be denoted by $e_i = |f * h'|_i$, where h' denotes one of the above edge filters. Then, the local maxima of the edge amplitude are given by the points for which $e_i > e_{i-1} \wedge e_i > e_{i+1} \wedge e_i \geq t$, where t is the threshold to select the relevant edges.

Unfortunately, extracting the edges with pixel accuracy is often not accurate enough. To extract edges with subpixel accuracy, we can note that around the maximum the edge amplitude can be approximated well with a parabola. Figure 9.70 illustrates this by showing a zoomed part of the edge amplitude around the right edge in Figure 9.69. If we fit a parabola through three points around the maximum edge amplitude and calculate the maximum of the parabola, we can obtain the edge position with subpixel accuracy. If an ideal camera system is assumed, this algorithm is as accurate as the precision with which the floating-point numbers are stored in the computer [45].

We conclude the discussion of the 1D edge extraction by showing the results of edge extraction on the two examples we have used so far. Figure 9.71a shows the edges that have been extracted along the line shown in Figure 9.61a with the Canny filter with $\sigma = 1.0$. From the two zoomed parts around the extracted edge

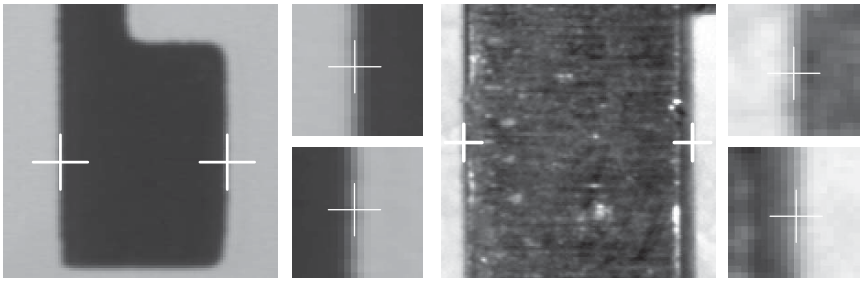


Figure 9.71 (a) Result of extracting 1D edges along the line shown in Figure 9.61a. The two small images show a zoomed part around the edge positions. In this case, they both lie very close to the pixel centers. The distance between the two edges is 60.95 pixels. (b) Result of extracting 1D edges along the line shown in Figure 9.67a. Note that the left edge, shown in detail in the upper right image, is almost exactly in the middle between two pixel centers. The distance between the two edges is 125.37 pixels.

positions, we can see that, by coincidence, both edges lie very close to the pixel centers. Figure 9.71b displays the result of extracting edges along the line shown in Figure 9.67a with the Canny filter with $\sigma = 1.5$. In this case, the left edge is almost exactly in the middle of two pixel centers. Hence, we can see that the algorithm is successful in extracting the edges with subpixel precision.

9.7.3 2D Edge Extraction

As discussed in Section 9.7.1, there are two possible definitions for edges in two dimensions, which are not equivalent. Like in the 1D case, the selection of salient edges will require us to perform a thresholding based on the gradient magnitude. Therefore, the definition via the zero-crossings of the Laplacian requires us to compute more partial derivatives than the definition via the maxima of the gradient magnitude. Consequently, we will concentrate on the maxima of the gradient magnitude for the 2D case. We will add some comments on the zero-crossings of the Laplacian at the end of this section.

As in the 1D case, the first question we need to answer is how to compute the partial derivatives of the image that are required to calculate the gradient. Similar to equation (9.101), we could use finite differences to calculate the partial derivatives. In two dimensions, they would be

$$f_{r;i,j} = \tfrac{1}{2}(f_{i+1,j} - f_{i-1,j}) \quad \text{and} \quad f_{c;i,j} = \tfrac{1}{2}(f_{i,j+1} - f_{i,j-1}) \tag{9.108}$$

However, as we have seen previously, typically the image must be smoothed to obtain good results. For time-critical applications, the filter masks should be as small as possible, that is, 3×3. All 3×3 edge filters can be brought into the following form by scaling the coefficients appropriately (note that the filter masks are mirrored in the convolution):

$$\begin{pmatrix} 1 & 0 & -1 \\ a & 0 & -a \\ 1 & 0 & -1 \end{pmatrix} \qquad \begin{pmatrix} 1 & a & 1 \\ 0 & 0 & 0 \\ -1 & -a & -1 \end{pmatrix} \tag{9.109}$$

If we use $a = 1$, we obtain the Prewitt filter. Note that it performs as a mean filter perpendicular to the derivative direction. For $a = \sqrt{2}$, the Frei filter is obtained, and for $a = 2$ we obtain the Sobel filter, which performs an approximation to a Gaussian smoothing perpendicular to the derivative direction. Of the above three filters, the Sobel filter returns the best results because it uses the best smoothing filter.

Ando [46] has proposed a 3×3 edge filter that tries to minimize the artifacts that invariably are obtained with small filter masks. In our notation, his filter would correspond to $a = 2.435\,101$. Unfortunately, like the Frei filter, it requires floating-point calculations, which makes it unattractive for time-critical applications.

The 3×3 edge filters are primarily used to quickly find edges with moderate accuracy in images of relatively good quality. Since speed is important and the calculation of the gradient magnitude via the Euclidean length (the 2-norm) of the gradient vector ($\|\nabla f\|_2 = \sqrt{f_r^2 + f_c^2}$) requires an expensive

square root calculation, the gradient magnitude is typically computed by one of the following norms: the 1-norm $\| \nabla f \|_1 = |f_r| + |f_c|$ or the maximum norm $\| \nabla f \|_\infty = \max(|f_r|, |f_c|)$. Note that the first norm corresponds to the city-block distance in the distance transform, while the second norm corresponds to the chessboard distance (see Section 9.6.1). Furthermore, the non-maximum suppression also is relatively expensive and is often omitted. Instead, the gradient magnitude is simply thresholded. Because this results in edges that are wider than one pixel, the thresholded edge regions are skeletonized. Note that this implicitly assumes that the edges are symmetric.

Figure 9.72 shows an example where this simple approach works quite well because the image is of good quality. Figure 9.72a displays the edge amplitude around the leftmost hole of the workpiece in Figure 9.61a computed with the Sobel filter and the 1-norm. The edge amplitude is thresholded (Figure 9.72b) and the skeleton of the resulting region is computed (Figure 9.72c). Since the assumption that the edges are symmetric is fulfilled in this example, the resulting edges are in the correct location.

This approach fails to produce good results on the more difficult image of the relay in Figure 9.67a. As can be seen from Figure 9.73a, the texture on the relay causes many areas with high gradient magnitude, which are also present in the segmentation (Figure 9.73b) and the skeleton (Figure 9.73c). Another interesting thing to note is that the vertical edge at the right corner of the top edge of the relay is quite blurred and asymmetric. This produces holes in the segmented edge region, which are exacerbated by the skeletonization.

Figure 9.72 (a) Edge amplitude around the leftmost hole of the workpiece in Figure 9.61a computed with the Sobel filter and the 1-norm. (b) Thresholded edge region. (c) Skeleton of (b).

Figure 9.73 (a) Edge amplitude around the top part of the relay in Figure 9.67a computed with the Sobel filter and the 1-norm. (b) Thresholded edge region. (c) Skeleton of (b).

Because the 3×3 filters are not robust against noise and other disturbances, for example, textures, we need to adapt the approach to optimal 1D edge extraction described in the previous section to the 2D case. In two dimensions, we can derive the optimal edge filters by calculating the partial derivatives of the optimal smoothing filters, since the properties of the convolution again allow us to move the derivative calculation into the filter. Consequently, Canny's optimal edge filters in two dimensions are given by the partial derivatives of the Gaussian filter. Because the Gaussian filter is separable, so are its derivatives:

$$g_r = \sqrt{2\pi}\sigma g'_\sigma(r) g_\sigma(c) \quad \text{and} \quad g_c = \sqrt{2\pi}\sigma g_\sigma(r) g'_\sigma(c) \tag{9.110}$$

(see the discussion following equation (9.105) for the factors of $\sqrt{2\pi}\sigma$). To adapt the Deriche filters to the 2D case, the separability of the filters is postulated. Hence, the optimal 2D Deriche filters are given by $d'_\alpha(r)d_\alpha(c)$ and $d_\alpha(r)d'_\alpha(c)$ for the first Deriche filter, and by $e'_\alpha(r)e_\alpha(c)$ and $e_\alpha(r)e'_\alpha(c)$ for the second Deriche filter (see equation (9.106)).

The advantage of the Canny filter is that it is isotropic, that is, rotation-invariant (see Section 9.2.3). Its disadvantage is that it cannot be implemented recursively. Therefore, the execution time depends on the amount of smoothing specified by σ. The Deriche filters, on the other hand, can be implemented recursively, and hence their runtime is independent of the smoothing parameter α. However, they are anisotropic, that is, the edge amplitude they calculate depends on the angle of the edge in the image. This is undesirable because it makes the selection of the relevant edges harder. Lanser has shown that the anisotropy of the Deriche filters can be corrected [47]. We will refer to the isotropic versions of the Deriche filters as the Lanser filters.

Figure 9.74 displays the result of computing the edge amplitude with the second Lanser filter with $\alpha = 0.5$. Compared to the the Sobel filter, the Lanser filter was able to suppress the noise and texture significantly better. This can be seen from the edge amplitude image (Figure 9.74a) as well as the thresholded edge region (Figure 9.74b). Note, however, that the edge region still contains a hole for the vertical edge that starts at the right corner of the topmost edge of the relay. This happens because the edge amplitude only has been thresholded and the important step of the non-maximum suppression has been omitted to compare the results of the Sobel and Lanser filters.

As we saw in the above examples, thresholding the edge amplitude and then skeletonizing the region sometimes does not yield the desired results. To obtain

Figure 9.74 (a) Edge amplitude around the top part of the relay in Figure 9.67a computed with the second Lanser filter with $\alpha = 0.5$. (b) Thresholded edge region. (c) Skeleton of (b).

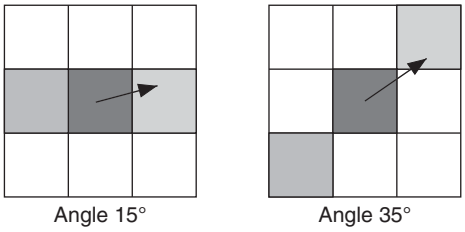


Figure 9.75 Examples of the pixels that are examined in the non-maximum suppression for different gradient directions.

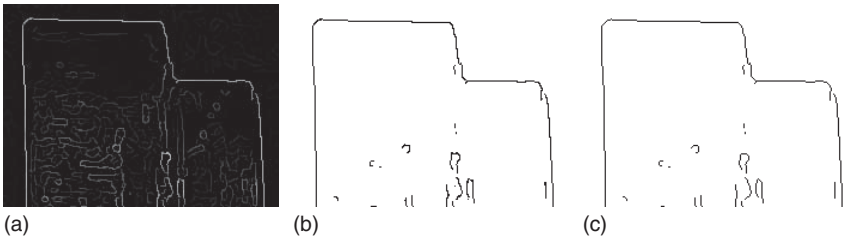


Figure 9.76 (a) Result of applying the non-maximum suppression to the edge amplitude image in Figure 9.74a. (b) Thresholded edge region. (c) Skeleton of (b).

the correct edge locations, we must perform the non-maximum suppression (see Section 9.7.1). In the 2D case, this can be done by examining the two neighboring pixels that lie closest to the gradient direction. Conceptually, we can think of transforming the gradient vector into an angle. Then, we divide the angle range into eight sectors. Figure 9.75 shows two examples of this. Unfortunately, with this approach, diagonal edges often are still two pixels wide. Consequently, the output of the non-maximum suppression must still be skeletonized.

Figure 9.76 shows the result of applying the non-maximum suppression to the edge amplitude image in Figure 9.74a. From the thresholded edge region in Figure 9.76b, it can be seen that the edges are now in the correct locations. In particular, the incorrect hole in Figure 9.74 is no longer present. We can also see that the few diagonal edges are sometimes two pixels wide. Therefore, their skeleton is computed and displayed in Figure 9.76c.

Up to now, we have been using simple thresholding to select the salient edges. This works well as long as the edges we are interested in have roughly the same contrast or have a contrast that is significantly different from the contrast of noise, texture, or other irrelevant objects in the image. In many applications, however, we face the problem that, if we select the threshold so high that only the relevant edges are selected, they are often fragmented. If, on the other hand, we set the threshold so low that the edges we are interested in are not fragmented, we end up with many irrelevant edges. These two situations are illustrated in Figure 9.77a,b. A solution to this problem was proposed by Canny [43]. He devised a special thresholding algorithm for segmenting edges: hysteresis thresholding. Instead of a single threshold, it uses two thresholds. Points with an edge amplitude greater than the higher threshold are immediately accepted as safe edge points. Points with an edge amplitude smaller than the lower threshold are immediately rejected. Points with an edge amplitude between the two thresholds are accepted only if they are connected to safe edge points via a path

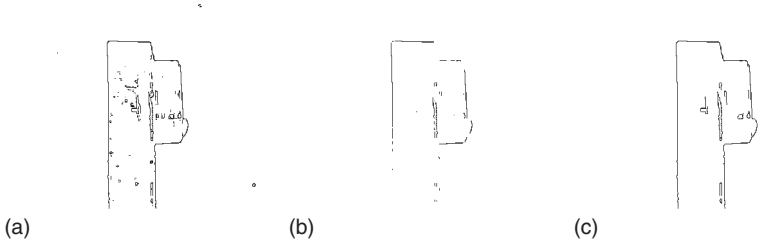


Figure 9.77 (a) Result of thresholding the edge amplitude for the entire relay image in Figure 9.67a with a threshold of 60. This causes many irrelevant texture edges to be selected. (b) Result of thresholding the edge amplitude with a threshold of 140. This selects only the relevant edges. However, they are severely fragmented and incomplete. (c) Result of hysteresis thresholding with a low threshold of 60 and a high threshold of 140. Only the relevant edges are selected, and they are complete.

in which all points have an edge amplitude above the lower threshold. We can also think of this operation as first selecting the edge points with an amplitude above the upper threshold, and then extending the edges as far as possible while remaining above the lower threshold. Figure 9.77c shows that hysteresis thresholding enables us to select only the relevant edges without fragmenting them or missing edge points.

Like in the 1D case, the pixel-accurate edges we have extracted so far are often not accurate enough. We can use a similar approach as for 1D edges to extract edges with subpixel accuracy: we can fit a 2D polynomial to the edge amplitude and extract its maximum in the direction of the gradient vector [45, 48]. The fitting of the polynomial can be done with convolutions with special filter masks (the so-called facet model masks) [22, 49]. To illustrate this, Figure 9.78a shows a 7×7 part of an edge amplitude image. The fitted 2D polynomial obtained from the central 3×3 amplitudes is shown in Figure 9.78b, along with an arrow that indicates the gradient direction. Furthermore, contour lines of the polynomial are shown. They indicate that the edge point is offset by approximately a quarter of a pixel in the direction of the arrow.

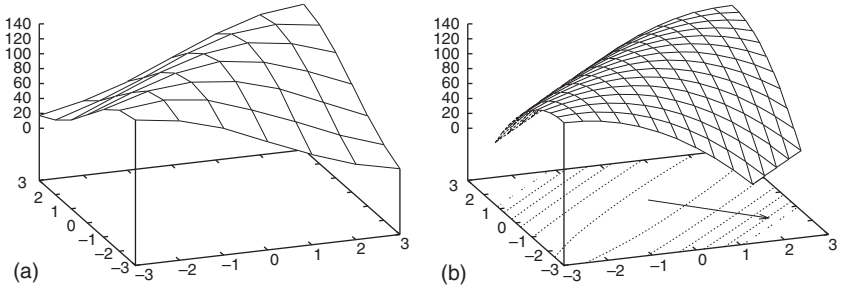


Figure 9.78 (a) A 7×7 part of an edge amplitude image. (b) Fitted 2D polynomial obtained from the central 3×3 amplitudes in (a). The arrow indicates the gradient direction. The contour lines in the plot indicate that the edge point is offset by approximately a quarter of a pixel in the direction of the arrow.

The above procedure gives us one subpixel-accurate edge point per non-maximum suppressed pixel. These individual edge points must be linked into subpixel-precise contours. This can be done by repeatedly selecting the first unprocessed edge point to start the contour and then successively finding adjacent edge points until the contour closes, reaches the image border, or reaches an intersection point.

Figure 9.79 illustrates the subpixel edge extraction along with a very useful strategy to increase the processing speed. The image in Figure 9.79a is the same workpiece as in Figure 9.61a. Because the subpixel edge extraction is relatively costly, we want to reduce the search space as much as possible. Since the workpiece is back-lit, we can threshold it easily (Figure 9.79b). If we calculate the inner boundary of the region with equation (9.79), the resulting points are close to the edge points we want to extract. We only need to dilate the boundary slightly, for example, with a circle of diameter 5 (Figure 9.79c), to obtain an ROI for the edge extraction. Note that the ROI is only a small fraction of the entire image. Consequently, the edge extraction can be done an order of magnitude faster than in the entire image, without any loss of information. The resulting subpixel-accurate edges are shown in Figure 9.79d for the part of the image indicated by the rectangle in Figure 9.79a. Note how well they capture the shape of the hole.

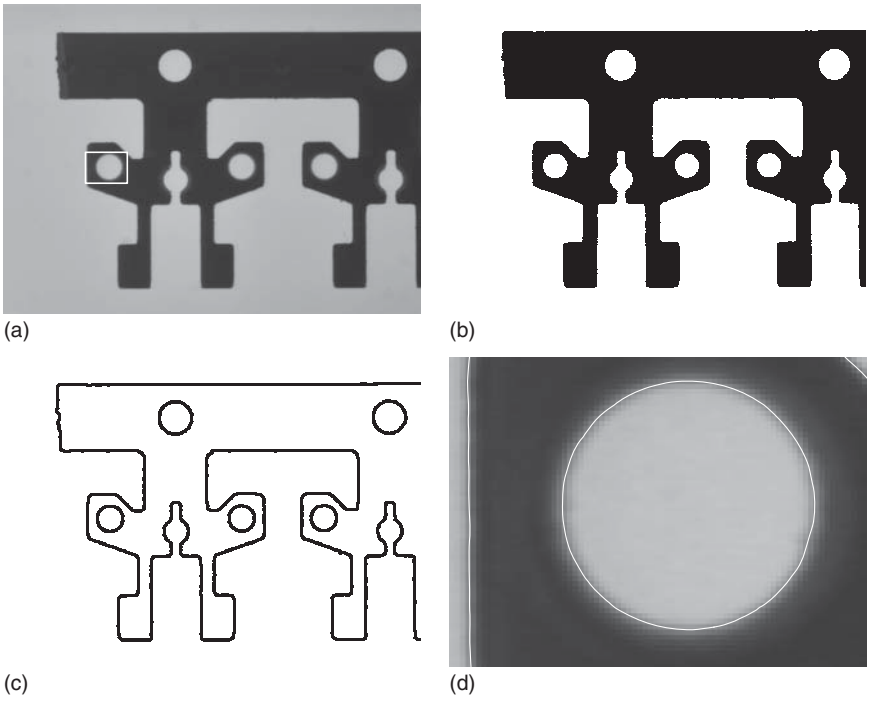


Figure 9.79 (a) Image of the workpiece in Figure 9.61a with a rectangle that indicates the image part shown in (d). (b) Thresholded workpiece. (c) Dilation of the boundary of (b) with a circle of diameter 5. This is used as the ROI for the subpixel edge extraction. (d) Subpixel-accurate edges of the workpiece extracted with the Canny filter with $\sigma = 1$.

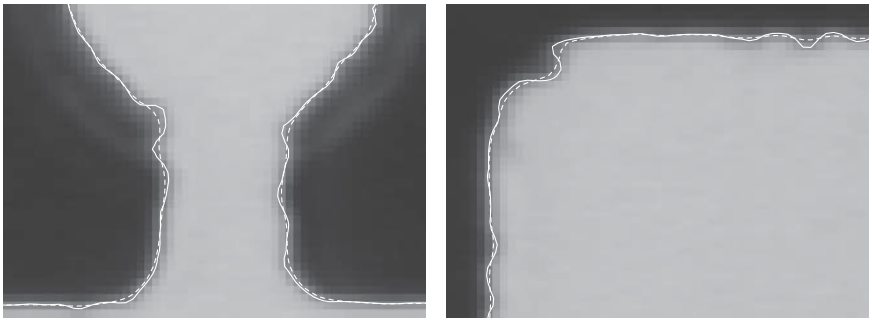


Figure 9.80 Comparison of the subpixel-accurate edges extracted via the maxima of the gradient magnitude in the gradient direction (dashed lines) and the edges extracted via the subpixel-accurate zero-crossings of the Laplacian. In both cases, a Gaussian filter with $\sigma = 1$ was used. Note that, since the Laplacian edges must follow the corners, they are much more curved than the gradient magnitude edges.

We conclude this section with a look at the second edge definition via the zero crossings of the Laplacian. Since the zero-crossings are just a special threshold, we can use the subpixel-precise thresholding operation, defined in Section 9.4.3, to extract edges with subpixel accuracy. To make this as efficient as possible, we must first compute the edge amplitude in the entire ROI of the image. Then, we threshold the edge amplitude and use the resulting region as the ROI for the computation of the Laplacian and for the subpixel-precise thresholding. The resulting edges for two parts of the workpiece image are compared to the gradient magnitude edge in Figure 9.80. Note that, since the Laplacian edges must follow the corners, they are much more curved than the gradient magnitude edges, and hence are more difficult to process further. This is another reason why the edge definition via the gradient magnitude is usually preferred.

Despite the above arguments, the property that the Laplacian edge exactly passes through corners in the image can be used advantageously in some applications. Figure 9.81a shows an image of a bolt for which the depth of the thread must be measured. Figure 9.81b–d display the results of extracting the border of the bolt with subpixel-precise thresholding, the gradient magnitude

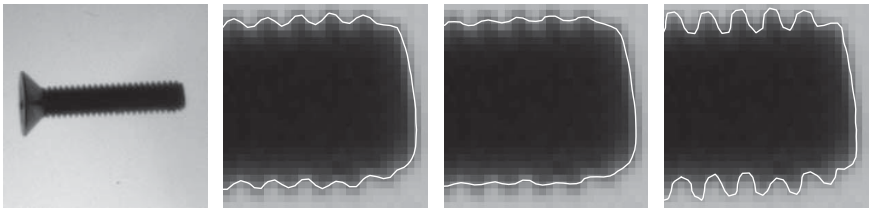


Figure 9.81 (a) Image of a bolt for which the depth of the thread must be measured. (b) Result of performing a subpixel-precise thresholding operation. (c) Result of extracting the gradient magnitude edges with a Canny filter with $\sigma = 0.7$. (d) Result of extracting the Laplacian edges with a Gaussian filter with $\sigma = 0.7$. Note that for this application the Laplacian edges return the most suitable result.

edges with a Canny filter with $\sigma = 0.7$, and the Laplacian edges with a Gaussian filter with $\sigma = 0.7$. Note that in this case the most suitable results are obtained with the Laplacian edges.

9.7.4 Accuracy of Edges

In the previous two sections, we have seen that edges can be extracted with subpixel resolution. We have used the terms "subpixel-accurate" and "subpixel-precise" to describe these extraction mechanisms without actually justifying the use of the words "accurate" and "precise." Therefore, in this section we will examine whether the edges we can extract are actually subpixel-accurate and subpixel-precise.

Since the words "accuracy" and "precision" are often confused or used interchangeably, let us first define what we mean by them. By precision, we denote how close on average an extracted value is to its mean value [50, 51]. Hence, precision measures how repeatably we can extract the value. By accuracy, on the other hand, we denote how close on average the extracted value is to its true value [50, 51]. Note that the precision does not tell us anything about the accuracy of the extracted value. The measurements could, for example, be offset by a systematic bias, but still be very precise. Conversely, the accuracy does not necessarily tell us how precise the extracted value is. The measurement could be quite accurate, but not very repeatable. Figure 9.82 shows the different situations that can occur. Also note that accuracy and precision are statements about the average distribution of the extracted values. From a single value, we cannot tell whether the measurements are accurate or precise.

If we adopt a statistical point of view, the extracted values can be regarded as random variables. With this, the precision of the values is given by the variance of the values: $V[x] = \sigma_x^2$. If the extracted values are precise, they have a small variance. On the other hand, the accuracy can be described by the difference of the expected value $E[x]$ from the true value T: $|E[x] - T|$. Since we typically do not know anything about the true probability distribution of the extracted values, and consequently cannot determine $E[x]$ and $V[x]$, we must estimate them with the empirical mean and variance of the extracted values.

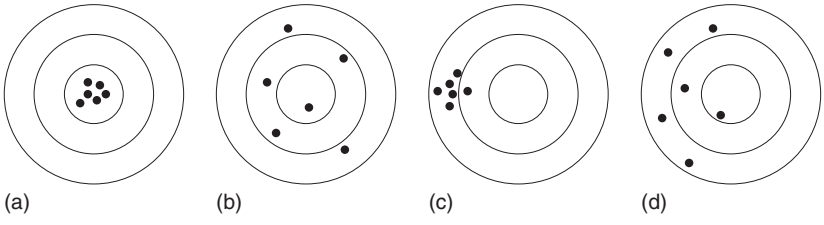


Figure 9.82 Comparison of accuracy and precision. The center of the circles indicates the true value of the feature. The dots indicate the outcome of the measurements of the feature. (a) Accurate and precise. (b) Accurate but not precise. (c) Not accurate but precise. (d) Neither accurate nor precise.

The accuracy and precision of edges is analyzed extensively in [45, 52]. The precision of ideal step edges extracted with the Canny filter can be derived analytically. If we denote the true edge amplitude by a and the noise variance in the image by σ_n^2, it can be shown that the variance of the edge positions σ_e^2 is given by

$$\sigma_e^2 = \frac{3}{8}\frac{\sigma_n^2}{a^2} \tag{9.111}$$

Even though this result was derived analytically for continuous images, it also holds in the discrete case. This result has also been verified empirically in [45, 52]. Note that it is quite intuitive: the larger the noise in the image, the less precisely the edges can be located; furthermore, the larger the edge amplitude, the higher is the precision of the edges. Note also that, possibly contrary to our intuition, increasing the smoothing does not increase the precision. This happens because the noise reduction achieved by the larger smoothing cancels out exactly with the weaker edge amplitude that results from the smoothing. From equation (9.111), we can see that the Canny filter is subpixel-precise ($\sigma_e \leq 1/2$) if the signal-to-noise ratio $a^2/\sigma_n^2 \geq 3/2$. This can, of course, be easily achieved in practice. Consequently, we see that we were justified in calling the Canny filter subpixel-precise.

The same derivation can also be performed for the Deriche and Lanser filters. For continuous images, the following variances result:

$$\sigma_e^2 = \frac{5}{64}\frac{\sigma_n^2}{a^2} \quad \text{and} \quad \sigma_e^2 = \frac{3}{16}\frac{\sigma_n^2}{a^2} \tag{9.112}$$

Note that the Deriche and Lanser filters are more precise than the Canny filter. Like for the Canny filter, the smoothing parameter α has no influence on the precision. In the discrete case, this is, unfortunately, no longer true because of the discretization of the filter. Here, less smoothing (larger values of α) leads to slightly worse precision than predicted by equation (9.112). However, for practical purposes, we can assume that the smoothing for all the edge filters that we have discussed has no influence on the precision of the edges. Consequently, if we want to control the precision of the edges, we must maximize the signal-to-noise ratio by using proper lighting and cameras. Furthermore, if analog cameras are used, the frame grabber should have a high signal-to-noise ratio and a line jitter that is as small as possible.

For ideal step edges, it is also easy to convince oneself that the expected position of the edge under noise corresponds to its true position. This happens because both the ideal step edge and the above filters are symmetric with respect to the true edge positions. Therefore, the edges that are extracted from noisy, ideal step edges must be distributed symmetrically around the true edge position. Consequently, their mean value is the true edge position. This is also verified empirically for the Canny filter in [45, 52]. Of course, it can also be verified for the Deriche and Lanser filters.

While it is easy to show that edges are very accurate for ideal step edges, we must also perform experiments on real images to test the accuracy on real data.

This is important because some of the assumptions that are used in the edge extraction algorithms may not hold in practice. Because these assumptions are seldom stated explicitly, we should examine them carefully here. Let us focus on straight edges because, as we have seen from the discussion in Section 9.7.1, especially Figure 9.63, sharply curved edges will necessarily lie in incorrect positions. See also [53] for a thorough discussion on the positional errors of the Laplacian edge detector for ideal corners of two straight edges with varying angles. Because we concentrate on straight edges, we can reduce the edge detection to the 1D case, which is simpler to analyze. From Section 9.7.1, we know that 1D edges are given by the inflection points of the gray value profiles. This implicitly assumes that the gray value profile and, consequently, its derivatives are symmetric with respect to the true edge. Furthermore, to obtain subpixel positions, the edge detection implicitly assumes that the gray values at the edge change smoothly and continuously as the edge moves in subpixel increments through a pixel. For example, if an edge covers 25% of a pixel, we would assume that the gray value in the pixel is a mixture of 25% of the foreground gray value and 75% of the background gray value. We will see whether these assumptions hold in real images.

To test the accuracy of the edge extraction on real images, it is instructive to repeat the experiments in [45, 52] with a different camera. In [45, 52], a print of an edge is mounted on an xy-stage and shifted in 50 µm increments, which corresponds to approximately 1/10 of a pixel, for a total of 1mm. The goals are to determine whether the shifts of 1/10 of a pixel can be detected reliably and to obtain information about the absolute accuracy of the edges. Figure 9.83a shows an image used in this experiment. We are not going to repeat the test to see whether the subpixel shifts can be detected reliably here. The 1/10 pixel shifts can be detected with a very high confidence (more than 99.99 999%). What is more interesting is to look at the absolute accuracy. Since we do not know the true edge position, we must get an estimate for it. Because the edge was shifted in linear increments in the test images, such an estimate can be obtained by fitting a straight line through the extracted edge positions and subtracting the line from the measured edge positions.

Figure 9.83b displays the result of extracting the edge in Figure 9.83a along a horizontal line with the Canny filter with $\sigma = 1$. The edge position error is shown in Figure 9.83c. We can see that there are errors of up to $\approx 1/22$ pixel. What causes these errors? As we discussed previously, for ideal cameras no error occurs, so one of the assumptions must be violated. In this case, the assumption that is violated is that the gray value is a mixture of the foreground and background gray values that is proportional to the area of the pixel covered by the object. This happens because the camera did not have a fill factor of 100%, that is, the light-sensitive area of a pixel on the sensor was much smaller than the total area of the pixel. Consider what happens when the edge moves across the pixel and the image is perfectly focused. In the light-sensitive area of the pixel, the gray value changes as expected when the edge moves across the pixel because the sensor integrates the incoming light. However, when the edge enters the light-insensitive area, the gray value no longer changes [54]. Consequently, the edge does not move in the image. In the real image, the focus is not perfect. Hence, the light is spread slightly over adjacent sensor elements. Therefore, the

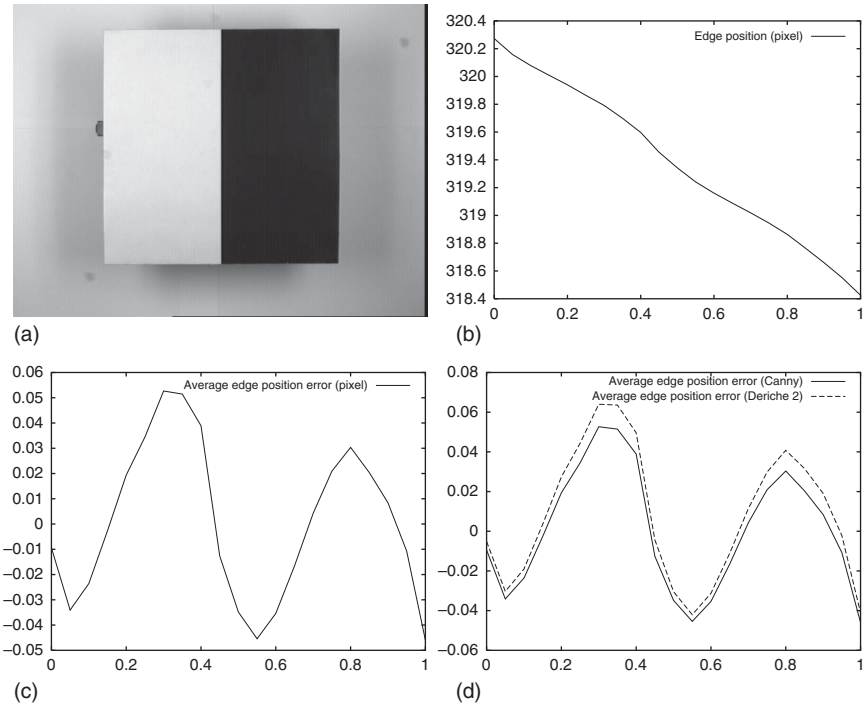


Figure 9.83 (a) Edge image used in the accuracy experiment. (b) Edge position extracted along a horizontal line in the image with the Canny filter. The edge position is given in pixels as a function of the true shift in millimeters. (c) Error of the edge positions obtained by fitting a line through the edge positions in (b) and subtracting the line from (b). (d) Comparison of the errors obtained with the Canny and second Deriche filters.

edges do not jump as they would in a perfectly focused image but shift continuously. Nevertheless, the poor fill factor causes errors in the edge positions. This can be seen very clearly from Figure 9.83c. Recall that the shift of 50 µm corresponds to 1/10 of a pixel. Consequently, the entire shift of 1mm corresponds to two pixels. This is why we see a sine wave with two periods in Figure 9.83c. Each period corresponds exactly to one pixel. That these effects are caused by the fill factor can also be seen if the lens is defocused. In this case, the light is spread over more sensor elements. This helps to create an artificially increased fill factor, which causes smaller errors.

From the above discussion, it would appear that the edge position can be extracted with an accuracy of 1/22 of a pixel. To check whether this is true, let us repeat the experiment with the second Deriche filter. Figure 9.83d shows the result of extracting the edges with $\alpha = 1$ and computing the errors with the line fitted through the Canny edge positions. The last part is done to make the errors comparable. We can see, surprisingly, that the Deriche edge positions are systematically shifted in one direction. Does this mean that the Deriche filter is less accurate than the Canny filter? Of course, it does not, since on ideal data both filters return the same result. It shows that another assumption must be

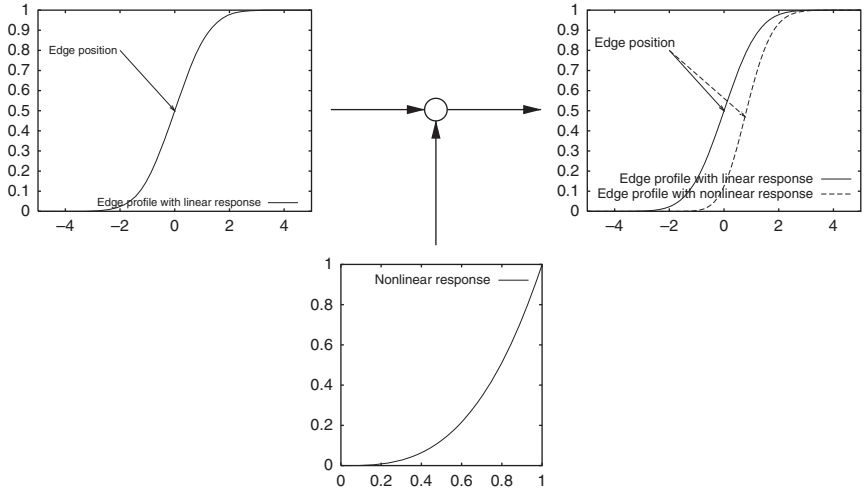


Figure 9.84 Result of applying a nonlinear gray value response curve to an ideal symmetric edge profile. The ideal edge profile is shown in the upper left graph and the nonlinear response in the bottom graph. The upper right graph shows the modified gray value profile along with the edge positions on the profiles. Note that the edge position is affected substantially by the nonlinear response.

violated. In this case, it is the assumption that the edge profile is symmetric with respect to the true edge position. This is the only reason why the two filters, which are symmetric themselves, can return different results.

There are many reasons why edge profiles may become asymmetric. One reason is that the gray value responses of the camera (and analog frame grabber, if used) are nonlinear. Figure 9.84 illustrates that an originally symmetric edge profile becomes asymmetric by a nonlinear gray value response function. It can be seen that the edge position accuracy is severely degraded by the nonlinear response. To correct the nonlinear response of the camera, it must be calibrated radiometrically with the methods described in Section 9.2.2.

Unfortunately, even if the camera has a linear response or is calibrated radiometrically, other factors may cause the edge profiles to become asymmetric. In particular, lens aberrations like coma, astigmatism, and chromatic aberrations may cause asymmetric profiles (see Section 4.5.3). Since lens aberrations cannot be corrected easily with image processing algorithms, they should be as small as possible.

While all the error sources discussed above influence the edge accuracy, we have so far neglected the largest source of errors. If the camera is not calibrated geometrically, extracting edges with subpixel accuracy is pointless because the lens distortions alone are sufficient to render any subpixel position meaningless. Let us, for example, assume that the lens has a distortion that is smaller than 1% in the entire field of view. At the corners of the image, this means that the edges are offset by four pixels for a 640×480 image. We can see that extracting edges with subpixel accuracy is an exercise in futility if the lens distortions are not corrected,

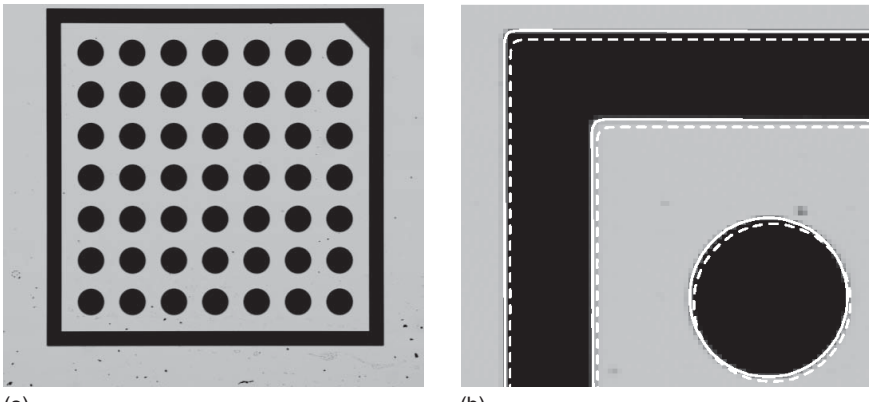


Figure 9.85 (a) Image of a calibration target. (b) Extracted subpixel-accurate edges (solid lines) and edges after the correction of the lens distortions (dashed lines). Note that the lens distortions cause an error of approximately three pixels.

even for this relatively small distortion. This is illustrated in Figure 9.85, where the result of correcting the lens distortions after calibrating the camera as described in Section 9.9 is shown. Note that, despite the fact that the application used a very high quality lens, the lens distortions cause an error of approximately three pixels.

Another detrimental influence on the accuracy of the extracted edges is caused by the perspective distortions in the image. They happen whenever we cannot mount the camera perpendicular to the objects we want to measure. Figure 9.86a shows the result of extracting the 1D edges along the ruler markings on a caliper. Because of the severe perspective distortions, the distances between the ruler markings vary greatly throughout the image. If the camera is calibrated, that is, its interior orientation and the exterior orientation of the plane in which the objects to measure lie have been determined with the approach described in Section 9.9, the measurements in the image can be converted into measurements in world coordinates in the plane determined by the calibration. This is done by intersecting the optical ray that corresponds to each edge point in the image with the plane in the world. Figure 9.86b displays the results of converting the measurements in Figure 9.86a into millimeters with this approach. Note that the measurements are extremely accurate even in the presence of severe perspective distortions.

From the above discussion, we can see that extracting edges with subpixel accuracy relies on careful selection of the hardware components. First of all, the gray value response of the camera (and analog frame grabber, if used) should be linear. To ensure this, the camera should be calibrated radiometrically. Furthermore, lenses with very small aberrations (such as coma and astigmatism) should be chosen. Furthermore, monochromatic light should be used to avoid the effects of chromatic aberrations. In addition, the fill factor of the camera should be as large as possible to avoid the effects of "blind spots." Finally, the camera should be calibrated geometrically to obtain meaningful results. All these requirements for

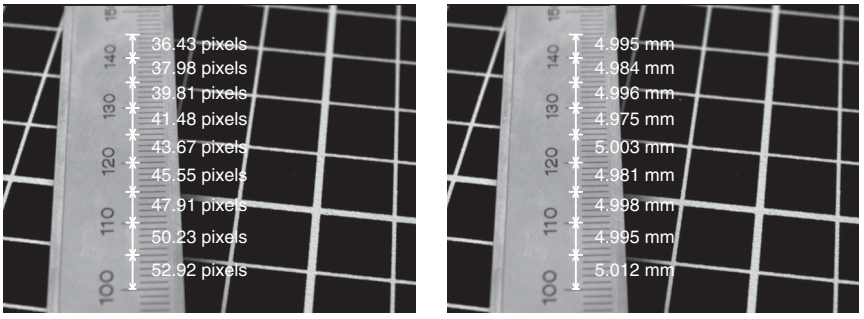


Figure 9.86 Result of extracting 1D edges along the ruler markings on a caliper. (a) Pixel distances between the markings. (b) Distances converted to world units using camera calibration.

the hardware components are, of course, also valid for other subpixel algorithms, for example, the subpixel-precise thresholding (see Section 9.4.3), the gray value moments (see Section 9.5.2), and the contour features (see Section 9.5.3).

9.8 Segmentation and Fitting of Geometric Primitives

In Sections 9.4 and 9.7, we have seen how to segment images by thresholding and edge extraction. In both cases, the boundary of objects either is returned explicitly or can be derived by some postprocessing (see Section 9.6.1). Therefore, for the purposes of this section, we can assume that the result of the segmentation is a contour with the points of the boundary, which may be subpixel-accurate. This approach often creates an enormous amount of data. For example, the subpixel-accurate edge of the hole in the workpiece in Figure 9.79d contains 172 contour points. However, we are typically not interested in such a large amount of information. For example, in the application in Figure 9.79d, we would probably be content with knowing the position and radius of the hole, which can be described with just three parameters. Therefore, in this section we will discuss methods to fit geometric primitives to contour data. We will only examine the most relevant geometric primitives: lines, circles, and ellipses. Furthermore, we will examine how contours can be segmented automatically into parts that correspond to the geometric primitives. This will enable us to substantially reduce the amount of data that needs to be processed, while also providing us with a symbolic description of the data. Furthermore, the fitting of the geometric primitives will enable us to reduce the influence of incorrectly or inaccurately extracted points (the so-called outliers). We will start by examining the fitting of the geometric primitives in Sections 9.8.1–9.8.3. In each case, we will assume that the contour or part of the contour we are examining corresponds to the primitive we are trying to fit, that is, we are assuming that the segmentation into the different primitives has already been performed. The segmentation itself will be discussed in Section 9.8.4.

9.8.1 Fitting Lines

If we want to fit lines, we first need to think about the representation of lines. In images, lines can occur in any orientation. Therefore, we have to use a representation that enables us to represent all lines. For example, the common representation $y = mx + b$ does not allow us to do this. One representation that can be used is the Hessian normal form of the line, given by

$$\alpha r + \beta c + \gamma = 0 \tag{9.113}$$

This is actually an over-parameterization, since the parameters (α, β, γ) are homogeneous [20, 21]. Therefore, they are defined only up to a scale factor. The scale factor in the Hessian normal form is fixed by requiring that $\alpha^2 + \beta^2 = 1$. This has the advantage that the distance of a point to the line can simply be obtained by substituting its coordinates into equation (9.113).

To fit a line through a set of points (r_i, c_i), $i = 1, \dots, n$, we can minimize the sum of the squared distances of the points to the line:

$$\varepsilon^2 = \sum_{i=1}^{n} (\alpha r_i + \beta c_i + \gamma)^2 \tag{9.114}$$

While this is correct in principle, it does not work in practice, because we can achieve a zero error if we select $\alpha = \beta = \gamma = 0$. This is caused by the over-parameterization of the line. Therefore, we must add the constraint $\alpha^2 + \beta^2 = 1$ as a Lagrange multiplier, and hence must minimize the following error:

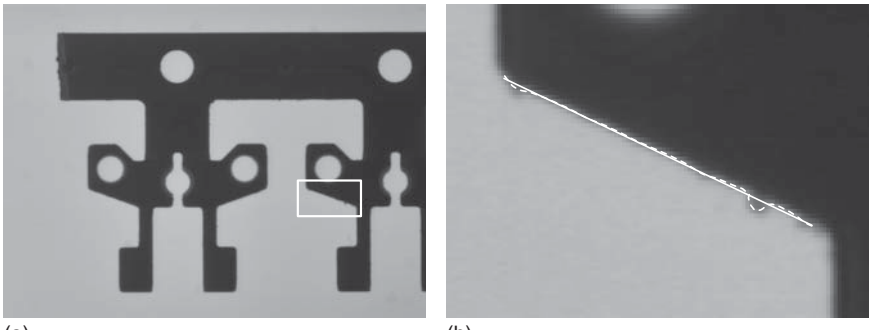
$$\varepsilon^2 = \sum_{i=1}^{n} (\alpha r_i + \beta c_i + \gamma)^2 - \lambda(\alpha^2 + \beta^2 - 1)n \tag{9.115}$$

The solution to this optimization problem is derived in [22]. It can be shown that (α, β) is the eigenvector corresponding to the smaller eigenvalue of the following matrix:

$$\begin{pmatrix} \mu_{2,0} & \mu_{1,1} \\ \mu_{1,1} & \mu_{0,2} \end{pmatrix} \tag{9.116}$$

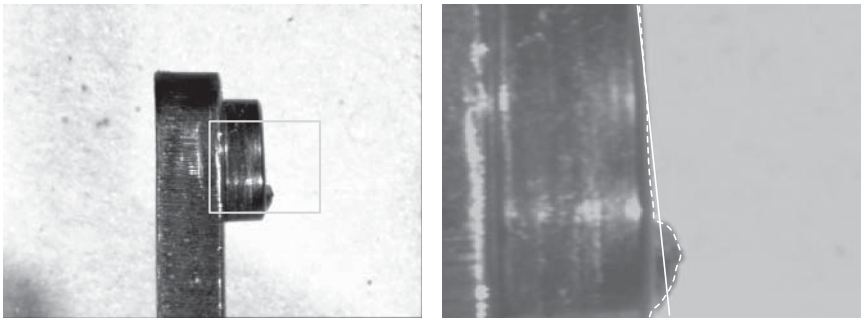
With this, γ is given by $\gamma = -(\alpha n_{1,0} + \beta n_{0,1})$. Here, $\mu_{2,0}$, $\mu_{1,1}$, and $\mu_{0,2}$ are the second-order central moments of the point set (r_i, c_i), while $n_{1,0}$ and $n_{0,1}$ are the normalized first-order moments (the center of gravity) of the point set. If we replace the area a of a region with the number n of points and sum over the points in the point set instead of the points in the region, the formulas to compute these moments are identical to the region moments of equations (9.55) and 9.56 in Section 9.5.1. It is interesting to note that the vector (α, β) thus obtained, which is the normal vector of the line, is the minor axis that would be obtained from the ellipse parameters of the point set. Consequently, the major axis of the ellipse is the direction of the line. This is a very interesting connection between the ellipse parameters and the line fitting, because the results were derived using different approaches and models.

Figure 9.87b illustrates the line-fitting procedure for an oblique edge of the workpiece shown in Figure 9.87a. Note that, by fitting the line, we were able



(a) (b)

Figure 9.87 (a) Image of a workpiece with the part shown in (b) indicated by the white rectangle. (b) Extracted edge within a region around the inclined edge of the workpiece (dashed line) and straight line fitted to the edge (solid line).



(a) (b)

Figure 9.88 (a) Image of a relay with the part shown in (b) indicated by the light gray rectangle. (b) Extracted edge within a region around the vertical edge of the relay (dashed line) and straight line fitted to the edge (solid line). To provide a better visibility of the edge and line, the contrast of the image has been reduced in (b).

to reduce the effects of the small protrusion on the workpiece. As mentioned previously, by inserting the coordinates of the edge points into the line equation (9.113), we can easily calculate the distances of the edge points to the line. Therefore, by thresholding the distances the protrusion can be detected easily.

As can be seen from the above example, the line fit is robust to small deviations from the assumed model (small outliers). However, Figure 9.88 shows that large outliers severely affect the quality of the fitted line. In this example, the line is fitted through the straight edge as well as the large arc caused by the relay contact. Since the line fit must minimize the sum of the squared distances of the contour points, the fitted line has a direction that deviates from that of the straight edge.

The least-squares line fit is not robust to large outliers since points that lie far from the line have a very large weight in the optimization because of the squared distances. To reduce the influence of distant points, we can introduce a weight w_i for each point. The weight should be $\ll 1$ for distant points. Let us assume for the

moment that we have a way to compute these weights. Then, the minimization becomes

$$\varepsilon^2 = \sum_{i=1}^{n} w_i(\alpha r_i + \beta c_i + \gamma)^2 - \lambda(\alpha^2 + \beta^2 - 1)n \tag{9.117}$$

The solution of this optimization problem is again given by the eigenvector corresponding to the smaller eigenvalue of a moment matrix like in equation (9.116) [55]. The only difference is that the moments are computed by taking the weights w_i into account. If we interpret the weights as gray values, the moments are identical to the gray value center of gravity and the second-order central gray value moments (see equations (9.63) and (9.64) in Section 9.5.2). As above, the fitted line corresponds to the major axis of the ellipse obtained from the weighted moments of the point set. Hence, there is an interesting connection to the gray value moments.

The only remaining problem is how to define the weights w_i. Since we want to give smaller weights to points with large distances, the weights must be based on the distances $\delta_i = |\alpha r_i + \beta c_i + \gamma|$ of the points to the line. Unfortunately, we do not know the distances without fitting the line, so this seems an impossible requirement. The solution is to fit the line in several iterations. In the first iteration, $w_i = 1$ is used, that is, a normal line fit is performed to calculate the distances δ_i. They are used to define weights for the following iterations by using a weight function $w(\delta)$. This method is called iteratively reweighted least-squares (IRLS) [56, 57]. In practice, often one of the following two weight functions is used. They both work very well. The first weight function was proposed by Huber [55, 58]. It is given by

$$w(\delta) = \begin{cases} 1, & |\delta| \le \tau \\ \tau/|\delta|, & |\delta| > \tau \end{cases} \tag{9.118}$$

The parameter τ is the clipping factor. It defines which points should be regarded as outliers. We will see how it is computed below. For now, note that all points with a distance $\le\tau$ receive a weight of 1. This means that, for small distances, the squared distance is used in the minimization. Points with a distance $>\tau$, on the other hand, receive a progressively smaller weight. In fact, the weight function is chosen such that points with large distances use the distance itself and not the squared distance in the optimization. Sometimes, these weights are not small enough to suppress outliers completely. In this case, the Tukey weight function can be used [55, 59]. It is given by

$$w(\delta) = \begin{cases} (1 - (\delta/\tau)^2)^2, & |\delta| \le \tau \\ 0, & |\delta| > \tau \end{cases} \tag{9.119}$$

Again, τ is the clipping factor. Note that this weight function completely disregards points that have a distance $>\tau$. For distances $\le\tau$, the weight changes smoothly from 1 to 0.

In the above two weight functions, the clipping factor specifies which points should be regarded as outliers. Since the clipping factor is a distance, it could simply be set manually. However, this would ignore the distribution of the noise and the outliers in the data, and consequently would have to be adapted for each

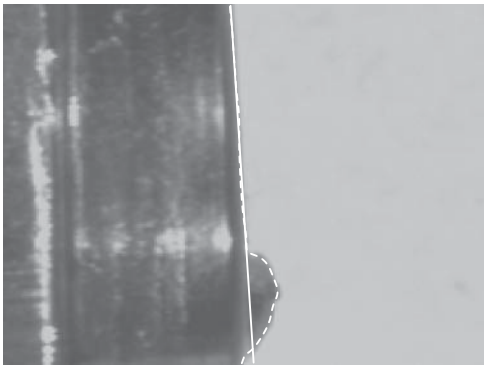


Figure 9.89 Straight line (solid line) fitted robustly to the vertical edge (dashed line). In this case, the Tukey weight function with a clipping factor of $\tau = 2\sigma_\delta$ with five iterations was used. Compared to Figure 9.88b, the line is now fitted to the straight-line part of the edge.

application. It is more convenient to derive the clipping factor from the data itself. This is typically done based on the standard deviation of the distances to the line. Since we expect outliers in the data, we cannot use the normal standard deviation, but must use a standard deviation that is robust to outliers. Typically, the following formula is used to compute the robust standard deviation:

$$\sigma_\delta = \frac{\text{median}|\delta_i|}{0.6745} \tag{9.120}$$

The constant in the denominator is chosen such that, for normally distributed distances, the standard deviation of the normal distribution is computed. The clipping factor is then set to a small multiple of σ_δ, for example, $\tau = 2\sigma_\delta$.

In addition to the Huber and Tukey weight functions, other weight functions can be defined. Several other possibilities are discussed in [20].

Figure 9.89 displays the result of fitting a line robustly to the edge of the relay using the Tukey weight function with a clipping factor of $\tau = 2\sigma_\delta$ with five iterations. If we compare this with the standard least-squares line fit in Figure 9.88b, we see that with the robust fit the line is now fitted to the straight-line part of the edge and the outliers caused by the relay contact have been suppressed.

It should also be noted that the above approach to outlier suppression by weighting down the influence of points with large distances can sometimes fail because the initial fit, which is a standard least-squares fit, can produce a solution that is dominated by the outliers. Consequently, the weight function will drop inliers. In this case, other robust methods must be used. The most important approach is the random sample consensus (RANSAC) algorithm, proposed by Fischler and Bolles [60]. Instead of dropping outliers successively, it constructs a solution (e.g., a line fit) from the minimum number of points (e.g., two for lines), which are selected randomly, and then checks how many points are consistent with the solution. The process of randomly selecting points, constructing the solution, and checking the number of consistent points is continued until a certain probability of having found the correct solution, for example, 99%, is achieved. At the end, the solution with the largest number of consistent points is selected.

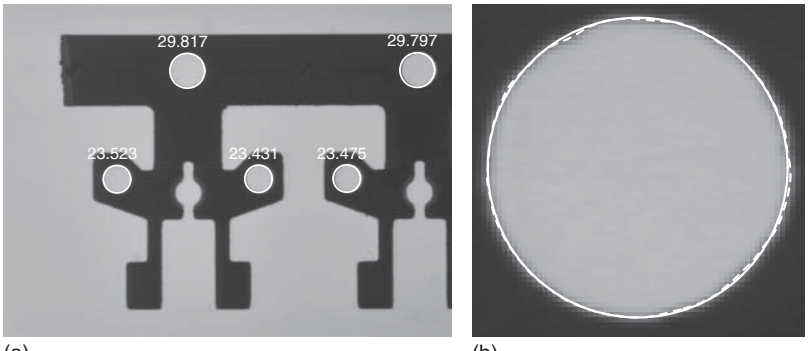


Figure 9.90 (a) Image of a workpiece with circles fitted to the edges of the holes in the workpiece. (b) Details of the upper right hole with the extracted edge (dashed line) and the fitted circle (solid line).

9.8.2 Fitting Circles

Fitting circles or circular arcs to a contour uses the same idea as fitting lines: that is, we want to minimize the sum of the squared distances of the contour points to the circle:

$$\varepsilon^2 = \sum_{i=1}^{n} \left(\sqrt{(r_i - \alpha)^2 + (c_i - \beta)^2} - \rho \right)^2 \tag{9.121}$$

Here, (α, β) is the center of the circle and ρ is its radius. Unlike the line fitting, this leads to a nonlinear optimization problem, which can only be solved iteratively using nonlinear optimization techniques. Details can be found in [22, 61, 62].

Figure 9.90a shows the result of fitting circles to the edges of the holes of a workpiece, along with the extracted radii in pixels. In Figure 9.90b, details of the upper right hole are shown. Note how well the circle fits the extracted edges.

Like the least-squares line fit, the least-squares circle fit is not robust to outliers. To make the circle fit robust, we can use the same approach that we used for the line fitting: we can introduce a weight that is used to reduce the influence of the outliers. Again, this requires that we perform a normal least-squares fit first and then use the distances that result from it to calculate the weights in later iterations. Since it is possible that large outliers prevent this algorithm from converging to the correct solution, a RANSAC approach might be necessary in extreme cases.

Figure 9.91 compares the standard circle fitting with the robust circle fitting using the BGA example of Figure 9.34. With the standard fitting (Figure 9.91b), the circle is affected by the error in the pad, which acts like an outlier. This is corrected with the robust fitting (Figure 9.91c).

To conclude this section, we should give some thought to what happens when a circle is fitted to a contour that only represents a part of a circle (a circular arc). In this case, the accuracy of the parameters becomes progressively worse as the

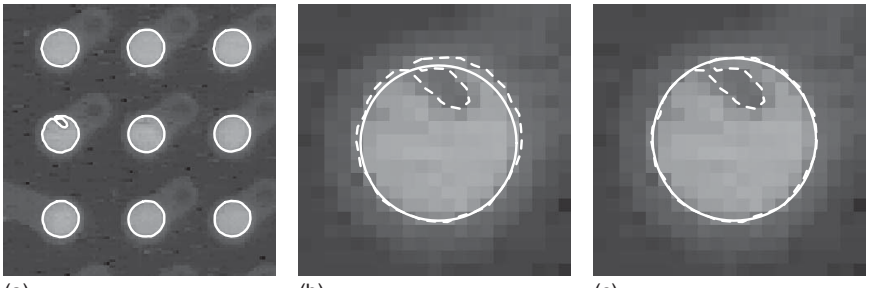


Figure 9.91 (a) Image of a BGA with pads extracted by subpixel-precise thresholding (see also Figure 9.34). (b) Circle fitted to the left pad in the center row of (a). The fitted circle is shown as a solid line, while the extracted contour is shown as a dashed line. The fitted circle is affected by the error in the pad, which acts like an outlier. (c) Result of robustly fitting a circle. The fitted circle corresponds to the true boundary of the pad.

angle of the circular arc becomes smaller. An excellent analysis of this effect is given in [61]. This effect is obvious from the geometry of the problem. Simply think about a contour that only represents a 5° arc. If the contour points are disturbed by noise, we have a very large range of radii and centers that lead to almost the same fitting error. On the other hand, if we fit to a complete circle, the geometry of the circle is much more constrained. This effect is caused by the geometry of the fitting problem and not by a particular fitting algorithm, that is, it will occur for all fitting algorithms.

9.8.3 Fitting Ellipses

To fit an ellipse to a contour, we would like to use the same principles as for lines and circles: that is, minimize the distance of the contour points to the ellipse. This requires us to determine the closest point to each contour point on the ellipse. While this can be determined easily for lines and circles, for ellipses it requires finding the roots of a fourth-degree polynomial. Since this is quite complicated and expensive, ellipses are often fitted by minimizing a different kind of distance. The principle is similar to the line fitting approach: we write down an implicit equation for ellipses (for lines, the implicit equation is given by equation (9.113)), and then substitute the point coordinates into the implicit equation to get a distance measure for the points to the ellipse. For the line-fitting problem, this procedure returns the true distance to the line. For the ellipse fitting, it only returns a value that has the same properties as a distance, but is not the true distance. Therefore, this distance is called the algebraic distance. Ellipses are described by the following implicit equation:

$$\alpha r^2 + \beta rc + \gamma c^2 + \delta r + \zeta c + \eta = 0 \tag{9.122}$$

Like for lines, the set of parameters is a homogeneous quantity, that is, only defined up to scale. Furthermore, equation (9.122) also describes hyperbolas and parabolas. Ellipses require $\beta^2 - 4\alpha\gamma < 0$. We can solve both problems by requiring $\beta^2 - 4\alpha\gamma = -1$. An elegant solution to fitting ellipses by minimizing the algebraic error with a linear method was proposed by Fitzgibbon.

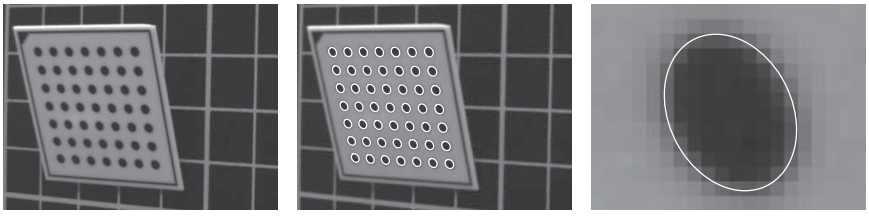


Figure 9.92 (a) Image of a calibration target. (b) Ellipses fitted to the extracted edges of the circular marks of the calibration target. (c) Detail of the center mark of the calibration target with the fitted ellipse.

The interested reader is referred to [63] for details. Unfortunately, minimizing the algebraic error can result in biased ellipse parameters. Therefore, if the ellipse parameters should be determined with maximum accuracy, the geometric error should be used. A nonlinear approach for fitting ellipses based on the geometric error is proposed in [62]. It is significantly more complicated than the linear approach in [63].

Like the least-squares line and circle fits, fitting ellipses via the algebraic or geometric distance is not robust to outliers. We can again introduce weights to create a robust fitting procedure. If the ellipses are fitted with the algebraic distance, this again results in a linear algorithm in each iteration of the robust fit [55]. In applications with a very large number of outliers or with very large outliers, a RANSAC approach might be necessary.

The ellipse fitting is very useful in camera calibration, where often circular marks are used on the calibration targets (see Section 9.9 and [54, 64, 65]). Since circles project to ellipses, fitting ellipses to the edges in the image is the natural first step in the calibration process. Figure 9.92a displays an image of a calibration target. The ellipses fitted to the extracted edges of the calibration marks are shown in Figure 9.92b. In Figure 9.92c, a detailed view of the center mark with the fitted ellipse is shown. Since the subpixel edge extraction is very accurate, there is hardly any visible difference between the edge and the ellipse, and consequently the edge is not shown in the figure.

To conclude this section, we should note that, if ellipses are fitted to contours that only represent a part of an ellipse, the same comments that were made for circular arcs at the end of the last section apply: the accuracy of the parameters will become worse as the angle that the arc subtends becomes smaller. The reason for this behavior lies in the geometry of the problem and not in the fitting algorithms we use.

9.8.4 Segmentation of Contours into Lines, Circles, and Ellipses

So far, we have assumed that the contours to which we are fitting the geometric primitives correspond to a single primitive of the correct type, for example, a line segment. Of course, a single contour may correspond to multiple primitives of different types. Therefore, in this section we will discuss how contours can be segmented into the different primitives.

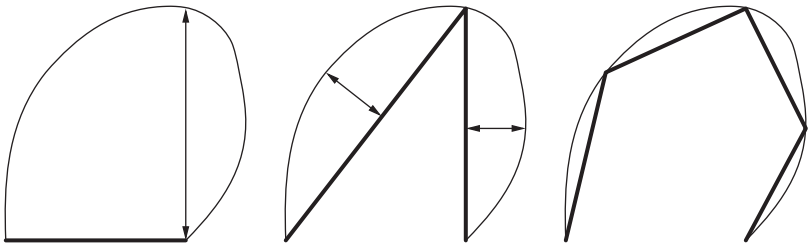


Figure 9.93 Example of the recursive subdivision that is performed in the Ramer algorithm. The contour is displayed as a thin line, while the approximating polygon is displayed as a thick line.

We will start by examining how a contour can be segmented into lines. To do so, we would like to find a polygon that approximates the contour sufficiently well. Let us call the contour points $p_i = (r_i, c_i)$, for $i = 1, \dots, n$. Approximating the contour by a polygon means that we want to find a subset p_{i_j}, for $j = 1, \dots, m$ with $m \leq n$, of the control points of the contour that describes the contour reasonably well. Once we have found the approximating polygon, each line segment $(p_{i_j}, p_{i_{j+1}})$ of the polygon is a part of the contour that can be approximated well with a line. Hence, we can fit lines to each line segment afterward to obtain a very accurate geometric representation of the line segments.

The question we need to answer is this: How do we define whether a polygon approximates the contour sufficiently well? A large number of different definitions have been proposed over the years. A very good evaluation of many polygonal approximation methods has been carried out by Rosin [66, 67]. In both cases, it was established that the algorithm proposed by Ramer [68], which curiously enough is one of the oldest algorithms, is the best overall method.

The Ramer algorithm performs a recursive subdivision of the contour until the resulting line segments have a maximum distance to the respective contour segments that is lower than a user-specified threshold $d_{\max}$. Figure 9.93 illustrates how the Ramer algorithm works. We start out by constructing a single line segment between the first and last contour points. If the contour is closed, we construct two segments: one from the first point to the point with index $n/2$, and the second one from $n/2$ to n. We then compute the distances of all the contour points to the line segment and find the point with the maximum distance to the line segment. If its distance is larger than the threshold we have specified, we subdivide the line segment into two segments at the point with the maximum distance. Then, this procedure is applied recursively to the new segments until no more subdivisions occur, that is, until all segments fulfill the maximum distance criterion.

Figure 9.94 illustrates the use of the polygonal approximation in a real application. In Figure 9.94a, a back-lit cutting tool is shown. In the application, the dimensions and angles of the cutting tool must be inspected. Since the tool consists of straight edges, the obvious approach is to extract edges with subpixel accuracy (Figure 9.94b) and to approximate them with a polygon using the Ramer algorithm. From Figure 9.94c we can see that the Ramer algorithm

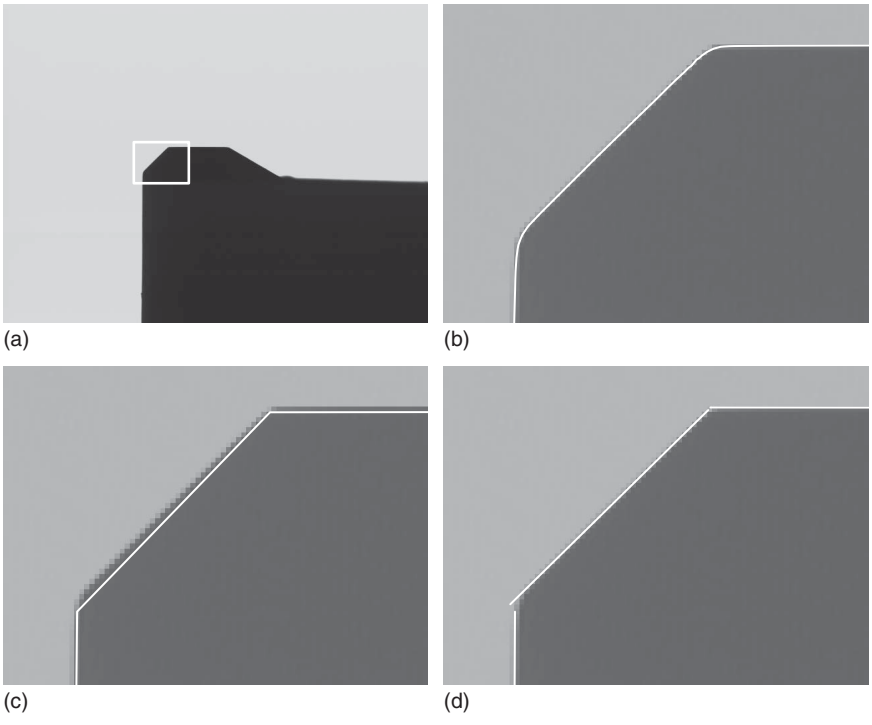


Figure 9.94 (a) Image of a back-lit cutting tool with the part that is shown in (b)–(d) overlaid as a white rectangle. To provide a better visibility of the results, the contrast of the image has been reduced in (b)–(d). (b) Subpixel-accurate edges extracted with the Lanser filter with $\alpha = 0.7$. (c) Polygons extracted with the Ramer algorithm with $d_{\max} = 2$. (d) Lines fitted robustly to the polygon segments using the Tukey weight function.

splits the edges correctly. We can also see the only slight drawback of the Ramer algorithm: it sometimes places the polygon control points into positions that are slightly offset from the true corners. In this application, this poses no problem, since to achieve maximum accuracy we must fit lines to the contour segments robustly anyway (Figure 9.94d). This enables us to obtain a concise and very accurate geometric description of the cutting tool. With the resulting geometric parameters, it can easily be checked whether the tool has the required dimensions.

While lines are often the only geometric primitive that occurs for the objects that should be inspected, in several cases the contour must be split into several types of primitives. For example, machined tools often consist of lines and circular arcs or lines and elliptic arcs. Therefore, we will now discuss how such a segmentation of the contours can be performed.

The approaches to segment contours into lines and circles can be classified into two broad categories. The first type of algorithm tries to identify breakpoints on the contour that correspond to semantically meaningful entities. For example, if two straight lines with different angles are next to each other, the tangent direction of the curve will contain a discontinuity. On the other hand, if

two circular arcs with different radii meet smoothly, there will be a discontinuity in the curvature of the contour. Therefore, the breakpoints typically are defined as discontinuities in the contour angle, which are equivalent to maxima of the curvature, and as discontinuities in the curvature itself. The first definition covers straight lines or circular arcs that meet at a sharp angle. The second definition covers smoothly joining circles or lines and circles [69, 70]. Since the curvature depends on the second derivative of the contour, it is an unstable feature that is very prone to even small errors in the contour coordinates. Therefore, to enable these algorithms to function properly, the contour must be smoothed substantially. This, in turn, can cause the breakpoints to shift from their desired positions. Furthermore, some breakpoints may be missed. Therefore, these approaches are often followed by an additional splitting and merging stage and a refinement of the breakpoint positions [70, 71].

While the above algorithms work well for splitting contours into lines and circles, they are quite difficult to extend to lines and ellipses because ellipses do not have constant curvature like circles. In fact, the two points on the ellipse on the major axis have locally maximal curvature and consequently would be classified as breakpoints by the above algorithms. Therefore, if we want to have a unified approach to segmenting contours into lines and circles or ellipses, the second type of algorithm is more appropriate. This type of algorithm is characterized by initially performing a segmentation of the contour into lines only. This produces an over-segmentation in the areas of the contour that correspond to circles and ellipses since here many line segments are required to approximate the contour. Therefore, in a second phase the line segments are examined as to whether they can be merged into circles or ellipses [55, 72]. For example, the algorithm in [55] initially performs a polygonal approximation with the Ramer algorithm. Then, it checks each pair of adjacent line segments to see whether it can be better approximated by an ellipse (or, alternatively, a circle). This is done by fitting an ellipse to the part of the contour that corresponds to the two line segments. If the fitting error of the ellipse is smaller than the maximum error of the two lines, the two line segments are marked as candidates for merging. After examining all pairs of line segments, the pair with the smallest fitting error is merged. In the following iterations, the algorithm also considers pairs of line and ellipse segments. The iterative merging is continued until there are no more segments that can be merged.

Figure 9.95 illustrates the segmentation into lines and circles. Like in the previous example, the application is the inspection of cutting tools. Figure 9.95a displays a cutting tool that consists of two linear parts and a circular part. The result of the initial segmentation into lines with the Ramer algorithm is shown in Figure 9.95b. Note that the circular arc is represented by four contour parts. The iterative merging stage of the algorithm successfully merges these four contour parts into a circular arc, as shown in Figure 9.95c. Finally, the angle between the linear parts of the tool is measured by fitting lines to the corresponding contour parts, while the radius of the circular arc is determined by fitting a circle to the circular arc part. Since the camera is calibrated in this application, the fitting is actually performed in world coordinates. Hence, the radius of the arc is calculated in millimeters.

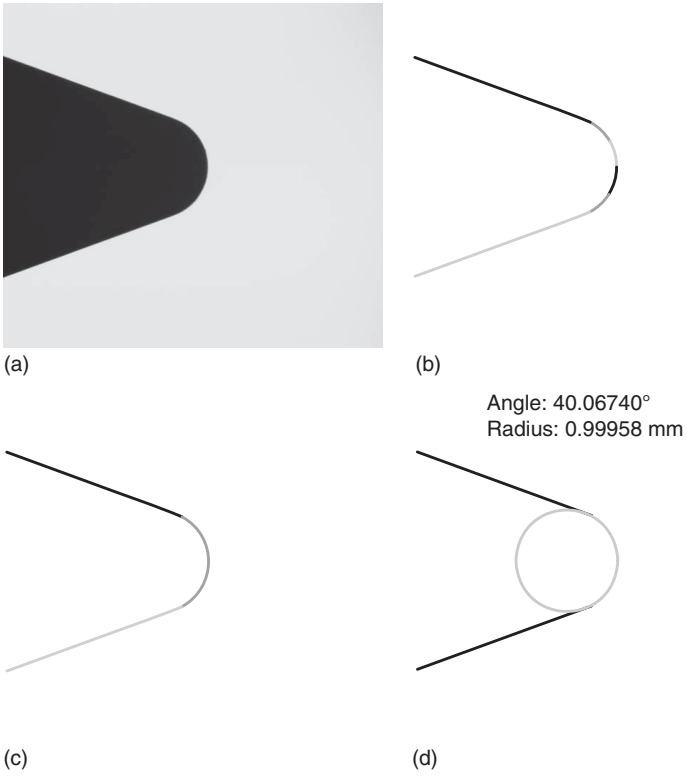


Figure 9.95 (a) Image of a back-lit cutting tool. (b) Contour parts corresponding to the initial segmentation into lines with the Ramer algorithm. The contour parts are displayed in three different gray values. (c) Result of the merging stage of the line and circle segmentation algorithm. In this case, two lines and one circular arc are returned. (d) Geometric measurements obtained by fitting lines to the linear parts of the contour and a circle to the circular part. Because the camera was calibrated, the radius is calculated in millimeters.

9.9 Camera Calibration

At the end of Section 9.7.4, we already discussed briefly that camera calibration is essential to obtain accurate measurements of objects. Chapter 5 discusses the approach that is used in close-range photogrammetry to calibrate camera setups with one or more pinhole cameras for multiview three-dimensional (3D) reconstruction applications. In this section, we will extend the discussion of the camera calibration to different types of camera and lens combinations that are used in machine vision applications. Furthermore, we will describe how metric results (e.g., object coordinates in meters or millimeters) can be obtained from single images.

To calibrate a camera, a model for the mapping of the 3D points of the world to the 2D image generated by the camera, lens, and frame grabber (if used) is necessary. Two different types of lenses are relevant for machine vision tasks: regular and telecentric lenses (see Chapter 4). Regular lenses perform a perspective

projection of the world into the image. We will call a camera with a regular lens a pinhole camera because of the connection between the projection performed by a regular lens and a pinhole camera that is described in Section 4.2.12.4. Telecentric lenses perform a parallel projection of the world into the image. We will call a camera with a telecentric lens a telecentric camera.

Furthermore, two types of sensors need to be considered: line sensors and area sensors. For area sensors, regular as well as telecentric lenses are in common use. For line sensors, only regular lenses are commonly used. Therefore, we will not introduce additional labels for the pinhole and telecentric cameras. Instead, it will be silently assumed that these two types of cameras use area sensors. For line sensors with regular lenses, we will simply use the term line scan camera.

For all three camera models, the mapping from three to two dimensions performed by the camera can be described by a certain number of parameters:

$$p = \pi(P_w, c_1, \ldots, c_n) \tag{9.123}$$

Here, p is the 2D image coordinate of the 3D point P_w produced by the projection π. Camera calibration is the process of determining the camera parameters $c_1, \ldots, c_n$.

9.9.1 Camera Models for Area Scan Cameras

Figure 9.96 displays the perspective projection performed by a pinhole camera. The world point P_w is projected through the projection center of the lens to the point p in the image plane. If there were no lens distortions present, p would lie on a straight line from P_w through the projection center, indicated by the dotted line. Lens distortions cause the point p to lie at a different position.

The image plane is located at a distance of f behind the projection center. As explained at the end of Section 4.2.12.4, f is the camera constant or principal

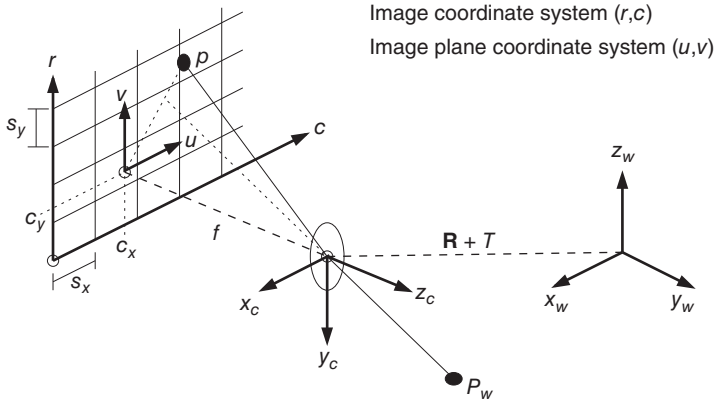


Figure 9.96 Camera model for a pinhole camera.

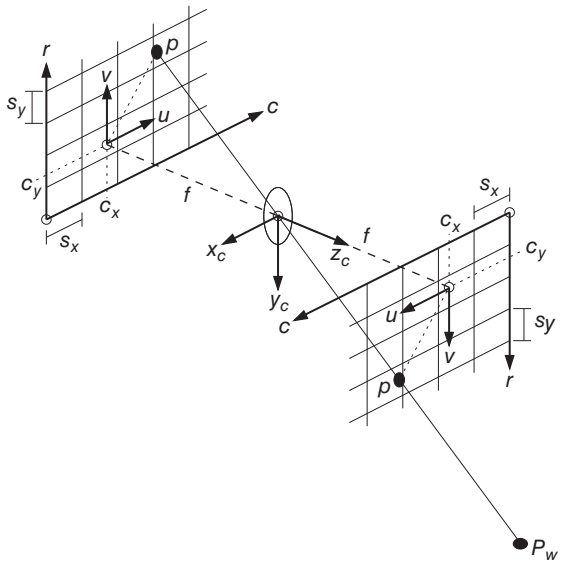


Figure 9.97 Image plane and virtual image plane.

distance, and not the focal length of the lens. Nevertheless, we will use the letter f to denote the principal distance for the purposes of this section.

Although the image plane in reality lies behind the projection center of the lens, it is easier to pretend that it lies at a distance of f in front of the projection center, as shown in Figure 9.97. This causes the image coordinate system to be aligned with the pixel coordinate system (row coordinates increase downward and column coordinates to the right) and simplifies many calculations.

We are now ready to describe the mapping of objects in 3D world coordinates to the 2D image plane and the corresponding camera parameters. First, we should note that the world points P_w are given in a world coordinate system (WCS). To make the projection into the image plane possible, they need to be transformed into the camera coordinate system (CCS). The CCS is defined such that its x- and y-axis are parallel to the column and row axes of the image, respectively, and the z-axis is perpendicular to the image plane and is oriented such that points in front of the camera have positive z coordinates. The transformation from the WCS to the CCS is a rigid transformation, that is, a rotation followed by a translation. Therefore, the point $P_w = (x_w, y_w, z_w)^\top$ in the WCS is given by the point $P_c = (x_c, y_c, z_c)^\top$ in the CCS, where

$$P_c = \mathrm{R} P_w + T \tag{9.124}$$

Here, $T = (t_x, t_y, t_z)^\top$ is a translation vector and $\mathrm{R} = \mathrm{R}(\alpha, \beta, \gamma)$ is a rotation matrix, which is determined by the three rotation angles γ (around the z-axis of the CCS), β (around the y-axis), and α (around the x-axis):

$$\mathrm{R}(\alpha, \beta, \gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{9.125}$$

The six parameters $(\alpha, \beta, \gamma, t_x, t_y, t_z)$ of R and T are called the exterior camera parameters, exterior orientation, or camera pose, because they determine the position of the camera with respect to the world.

The next step of the mapping is the projection of the 3D point P_c into the image plane coordinate system (IPCS). For the pinhole camera model, the projection is a perspective projection, which is given by

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{f}{z_c} \begin{pmatrix} x_c \\ y_c \end{pmatrix} \tag{9.126}$$

For the telecentric camera model, the projection is a parallel projection, which is given by

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \end{pmatrix} \tag{9.127}$$

As can be seen, there is no focal length f for telecentric cameras. Furthermore, the distance z_c of the object to the camera has no influence on the image coordinates.

After the projection to the image plane, lens distortions cause the coordinates $(u, v)^\top$ to be modified. Lens distortions are a transformation that can be modeled in the image plane alone, that is, 3D information is unnecessary. For many lenses, the distortion can be approximated sufficiently well by a radial distortion using the division model, which is given by [54, 55, 64]

$$\begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix} = \frac{2}{1 + \sqrt{1 - 4\kappa(u^2 + v^2)}} \begin{pmatrix} u \\ v \end{pmatrix} = \frac{2}{1 + \sqrt{1 - 4\kappa r^2}} \begin{pmatrix} u \\ v \end{pmatrix} \tag{9.128}$$

where $r^2 = u^2 + v^2$. The parameter κ models the magnitude of the radial distortions. If κ is negative, the distortion is barrel-shaped; while for positive κ it is pincushion-shaped. Figure 9.98 shows the effect of κ for an image of a calibration target, which we will use in the following to calibrate the camera.

The division model has the great advantage that the rectification of the distortion can be calculated analytically by

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{1 + \kappa(\tilde{u}^2 + \tilde{v}^2)} \begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix} = \frac{1}{1 + \kappa\tilde{r}^2} \begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix} \tag{9.129}$$

where $\tilde{r}^2 = \tilde{u}^2 + \tilde{v}^2$. This will be important when we compute world coordinates from image coordinates.

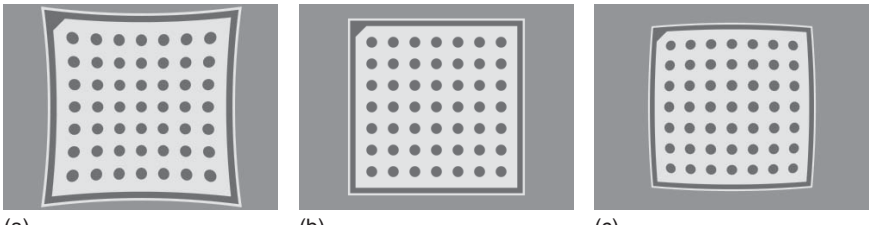


Figure 9.98 Effects of the distortion coefficient κ in the division model. (a) Pincushion distortion: $\kappa > 0$. (b) No distortion: $\kappa = 0$. (c) Barrel distortion: $\kappa < 0$.

If the division model is not sufficiently accurate for a particular lens, a polynomial distortion model that is able to model radial as well as decentering distortions can be used. Here, the rectification of the distortion is modeled by [65, 73, 74]

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \tilde{u}(1 + K_1\tilde{r}^2 + K_2\tilde{r}^4 + K_3\tilde{r}^6 + \cdots) \\ \quad +(2P_1\tilde{u}\tilde{v} + P_2(\tilde{r}^2 + 2\tilde{u}^2))(1 + P_3\tilde{r}^2 + \cdots) \\ \tilde{v}(1 + K_1\tilde{r}^2 + K_2\tilde{r}^4 + K_3\tilde{r}^6 + \cdots) \\ \quad +(P_1(\tilde{r}^2 + 2\tilde{v}^2) + 2P_2\tilde{u}\tilde{v})(1 + P_3\tilde{r}^2 + \cdots) \end{pmatrix} \tag{9.130}$$

The terms K_i describe a radial distortion, while the terms P_i describe a decentering distortion, which may occur if the optical axes of the individual lenses are not aligned perfectly with each other. In practice, the terms K_1, K_2, K_3, P_1, and P_2 are typically used, while higher order terms are neglected.

Note that (9.130) models the rectification of the distortion, that is, the analog of (9.129). In the polynomial model, the distortion (the analog of (9.128)) cannot be computed analytically. Instead, it must be computed numerically by a root-finding algorithm. This is no drawback since in applications we are typically interested in transforming image coordinates to measurements in the world (see Sections 9.9.4 and 9.10). Therefore, it is advantageous if the rectification can be computed analytically.

Figure 9.99 shows the effect of the parameters of the polynomial model on the distortion. In contrast to the division model, where $\kappa > 0$ leads to a pincushion distortion and $\kappa < 0$ to a barrel distortion, in the polynomial model $K_i > 0$ leads to a barrel distortion and $K_i < 0$ to a pincushion distortion. Furthermore, higher order terms lead to very strong distortions at the edges of the image, while they have a progressively smaller effect in the center of the image (if the distortions at the corners of the image are approximately the same). Decentering distortions cause an effect that is somewhat similar to a perspective distortion. However, they additionally bend the image in the horizontal or vertical direction.

There is a deeper connection between the radial distortion coefficients in the division and polynomial models, which can be seen by expanding the rectification

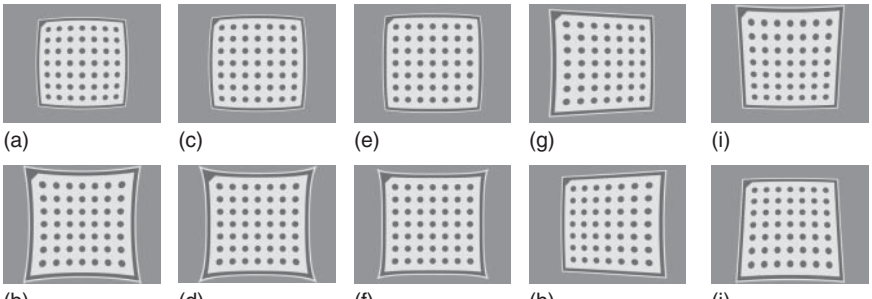


Figure 9.99 Effects of the distortion coefficients in the polynomial model. Coefficients that are not explicitly mentioned are 0. (a) $K_1 > 0$. (b) $K_1 < 0$. (c) $K_2 > 0$. (d) $K_2 < 0$. (e) $K_3 > 0$. (f) $K_3 < 0$. (g) $P_1 > 0$. (h) $P_1 < 0$. (i) $P_2 > 0$. (j) $P_2 < 0$.

factor $1/(1+\kappa\tilde{r}^2)$ in (9.129) into a geometric series:

$$\frac{1}{1+\kappa\tilde{r}^2} = \sum_{i=0}^{\infty} (-\kappa\tilde{r}^2)^i = 1 - \kappa\tilde{r}^2 + \kappa^2\tilde{r}^4 - \kappa^3\tilde{r}^6 + \cdots \tag{9.131}$$

Therefore, the division model corresponds to the polynomial model without decentering distortions and with infinitely many radial distortion terms K_i that all depend functionally on the single distortion coefficient κ: $K_i = (-\kappa)^i$.

Because of the complexity of the polynomial model, we will only use the division model in the discussion below. However, everything that will be discussed also holds if the division model is replaced by the polynomial model.

Finally, the point $(u, v)^\top$ is transformed from the IPCS into the image coordinate system (ICS):

$$\begin{pmatrix} r \\ c \end{pmatrix} = \begin{pmatrix} \frac{\tilde{v}}{s_y} + c_y \\ \frac{\tilde{u}}{s_x} + c_x \end{pmatrix} \tag{9.132}$$

Here, s_x and s_y are scaling factors. For pinhole cameras, they represent the horizontal and vertical pixel pitch on the sensor. For telecentric cameras, they represent the horizontal and vertical pixel pitch divided by the magnification factor of the lens (not taking into account the distortions of the lens). The point $(c_x, c_y)^\top$ is the principal point of the image. For pinhole cameras, this is the perpendicular projection of the projection center onto the image plane, that is, the point in the image from which a ray through the projection center is perpendicular to the image plane. It also defines the center of the distortions. For telecentric cameras, no projection center exists. Therefore, the principal point is solely defined by the distortions.

The six parameters $(f, \kappa, s_x, s_y, c_x, c_y)$ of the pinhole camera and the five parameters $(\kappa, s_x, s_y, c_x, c_y)$ of the telecentric camera are called the interior camera parameters or interior orientation, because they determine the projection from three dimensions to two dimensions performed by the camera.

9.9.2 Camera Model for Line Scan Cameras

Line scan cameras must move with respect to the object to acquire a useful image. The relative motion between the camera and the object is part of the interior orientation. By far the most frequent motion is a linear motion of the camera, that is, the camera moves with constant velocity along a straight line relative to the object, the orientation of the camera is constant with respect to the object, and the motion is equal for all images [75]. In this case, the motion can be described by the motion vector $V = (v_x, v_y, v_z)^\top$, as shown in Figure 9.100. The vector V is best described in units of meters per scan line in the camera coordinate system. As shown in Figure 9.100, the definition of V assumes a moving camera and a fixed object. If the camera is stationary and the object is moving, for example, on a conveyor belt, we can simply use $-V$ as the motion vector.

The camera model for a line scan camera is displayed in Figure 9.101. The origin of the CCS is the projection center. The z-axis is identical to the optical axis and

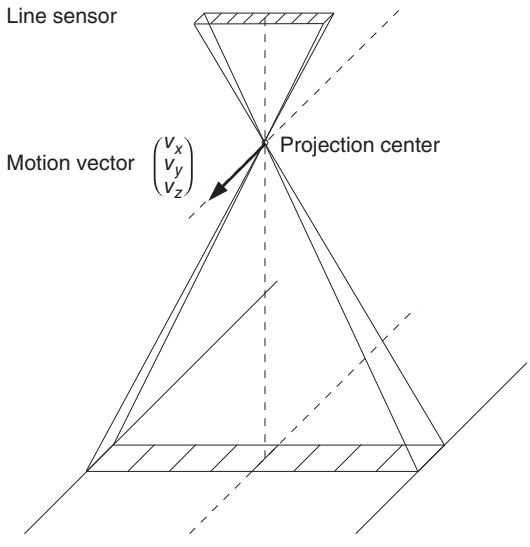


Figure 9.100 Principle of line scan image acquisition.

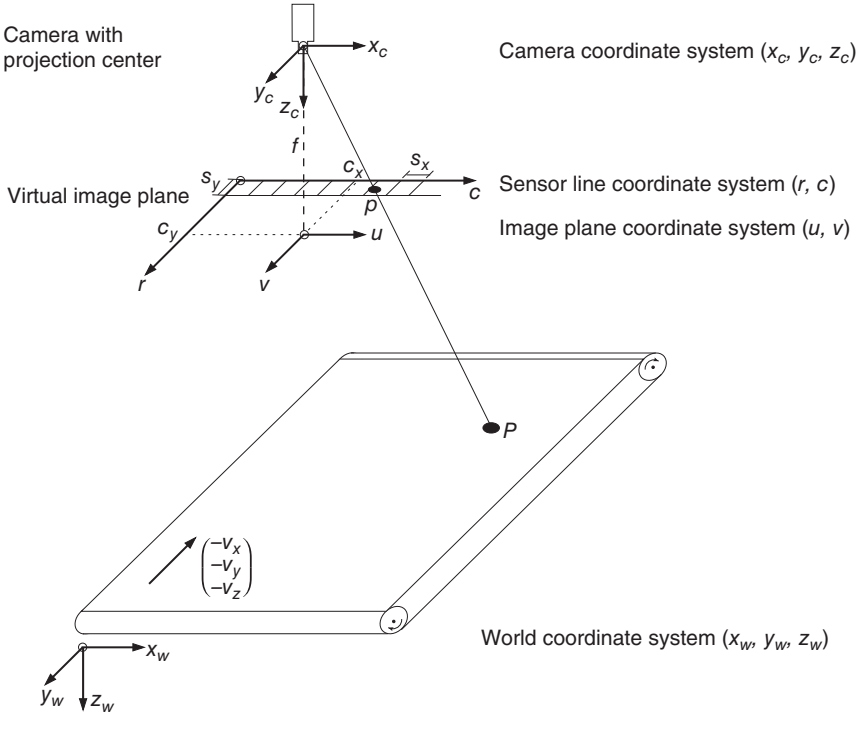


Figure 9.101 Camera model for a line scan camera.

is oriented such that points in front of the camera have positive z coordinates. The y-axis is perpendicular to the sensor line and to the z-axis. It is oriented such that the motion vector has a positive y component: that is, if a fixed object is assumed, the y-axis points in the direction in which the camera is moving. The x-axis is perpendicular to the y- and z-axis, so that the x, y, and z axes form a right-handed coordinate system.

Similar to area scan cameras, the projection of a point given in world coordinates into the image is modeled in two steps. First, the point is transformed from the WCS into the CCS. Then, it is projected into the image.

As the camera moves over the object during the image acquisition, the CCS moves with respect to the object, that is, each image is imaged from a different position. This means that each line has a different pose. To make things easier, we can use the fact that the motion of the camera is linear. Hence, it suffices to know the transformation from the WCS to the CCS for the first line of the image. The poses of the remaining lines can be computed from the motion vector, that is, the motion vector V is taken into account during the projection of P_c into the image. With this, the transformation from the WCS to the CCS is identical to equation (9.124). Like for area scan cameras, the six parameters $(\alpha, \beta, \gamma, t_x, t_y, t_z)$ are called the exterior camera parameters or exterior orientation, because they determine the position of the camera with respect to the world.

To obtain a model for the interior geometry of a line scan camera, we can regard a line sensor as one particular line of an area sensor. Therefore, like in area scan cameras, there is an IPCS that lies at a distance of f (the principal distance) behind the projection center. Again, the computations can be simplified if we pretend that the image plane lies in front of the projection center, as shown in Figure 9.101. We will defer the description of the projection performed by the line scan camera until later, since it is more complicated than for area scan cameras.

Let us assume that the point P_c has been projected to the point $(u, v)^\top$ in the IPCS. Like for area scan cameras, the point is now distorted by the radial distortion (9.128), which results in a distorted point $(\tilde{u}, \tilde{v})^\top$.

Finally, like for area scan cameras, $(\tilde{u}, \tilde{v})^\top$ is transformed into the ICS, resulting in the coordinates $(r, c)^\top$. Since we want to model the fact that the line sensor may not be mounted exactly behind the projection center, which often occurs in practice, we again have to introduce a principal point $(c_x, c_y)^\top$ that models how the line sensor is shifted with respect to the projection center, that is, it describes the relative position of the principal point with respect to the line sensor. Since $(\tilde{u}, \tilde{v})^\top$ is given in metric units, for example, meters, we need to introduce two scale factors s_x and s_y that determine how the IPCS units are converted to ICS units (i.e., pixels). Like for area scan cameras, s_x represents the horizontal pixel pitch on the sensor. As we will see in the following, s_y only serves as a scaling factor that enables us to specify the principal point in pixel coordinates. The values of s_x and s_y cannot be calibrated and must be set to the pixel size of the line sensor in the horizontal and vertical directions, respectively.

To determine the projection of the point $P_c = (x_c, y_c, z_c)^\top$ (specified in the CCS), we first consider the case where there are no radial distortions ($\kappa = 0$), the line sensor is mounted precisely behind the projection center ($c_y = 0$), and the motion is purely in the y direction of the CCS ($V = (0, v_y, 0)^\top$). In this case, the row

coordinate of the projected point p is proportional to the time it takes for the point P_c to appear directly under the sensor, that is, to appear in the xz-plane of the CCS. To determine this, we must solve $x_c - tv_y = 0$ for the "time" t (since V is specified in meters per scan line, the units of t are actually scan lines, i.e., pixels). Hence, $r = t = x_c/v_y$. Since $v_x = 0$, we also have $u = fx_c/z_c$ and $c = u/s_x + c_x$. Therefore, the projection is a perspective projection in the direction of the line sensor and a parallel projection perpendicular to the line sensor (i.e., in the special motion direction $(0, v_y, 0)^\top$).

For general motion vectors (i.e., $v_x \neq 0$ or $v_z \neq 0$), non-perfectly aligned line sensors ($c_y \neq 0$), and radial distortions ($\kappa \neq 0$), the equations become significantly more complicated. As above, we need to determine the "time" t when the point P_c appears in the "plane" spanned by the projection center and the line sensor. We put "plane" in quotes since the radial distortion will cause the back-projection of the line sensor to be a curved surface in space whenever $c_y \neq 0$ and $\kappa \neq 0$. To solve this problem, we construct the optical ray through the projection center and the projected point $p = (r, c)^\top$. Let us assume that we have transformed $(r, c)^\top$ into the distorted IPCS, where we have coordinates $(\tilde{u}, \tilde{v})^\top$. Here, $\tilde{v} = s_y c_y$ is the coordinate of the principal point in metric units. Then, we rectify $(\tilde{u}, \tilde{v})^\top$ by equation (9.129), that is, $(u, v)^\top = d(\tilde{u}, \tilde{v})^\top$, where $d = 1/(1 + \kappa(\tilde{u}^2 + \tilde{v}^2))$ is the rectification factor from equation (9.129). The optical ray is now given by the line equation $\lambda(u, v, f)^\top = \lambda(d\tilde{u}, d\tilde{v}, f)^\top$. The point P_c moves along the line given by $(x_c, y_c, z_c)^\top - t(v_x, v_y, v_z)^\top$ during the acquisition of the image. If p is the projection of P_c, both lines must intersect. Therefore, to determine the projection of P_c, we must solve the following nonlinear set of equations:

$$\begin{aligned} \lambda d\tilde{u} &= x_c - tv_x \\ \lambda d\tilde{v} &= y_c - tv_y \\ \lambda f &= z_c - tv_z \end{aligned} \tag{9.133}$$

for λ, $\tilde{u}$, and t, where d and $\tilde{v}$ are defined above. From $\tilde{u}$ and t, the pixel coordinates can be computed by

$$\begin{pmatrix} r \\ c \end{pmatrix} = \begin{pmatrix} t \\ \frac{\tilde{u}}{s_x} + c_x \end{pmatrix} \tag{9.134}$$

The nine parameters $(f, \kappa, s_x, s_y, c_x, c_y, v_x, v_y, v_z)$ of the line scan camera are called the interior orientation because they determine the projection from three dimensions to two dimensions performed by the camera.

Although the line scan camera geometry is conceptually simply a mixture of a perspective and a telecentric lens, precisely this mixture makes the camera geometry much more complex than the area scan geometries. Figure 9.102 displays some of the effects that can occur. In Figure 9.102a, the pixels are non-square because the motion is not tuned to the line frequency of the camera. In this example, either the line frequency would need to be increased or the motion speed would need to be decreased in order to obtain square pixels. Figure 9.102b shows the effect of a motion vector of the form $V = (v_x, v_y, 0)$ with $v_x = v_y/10$. We

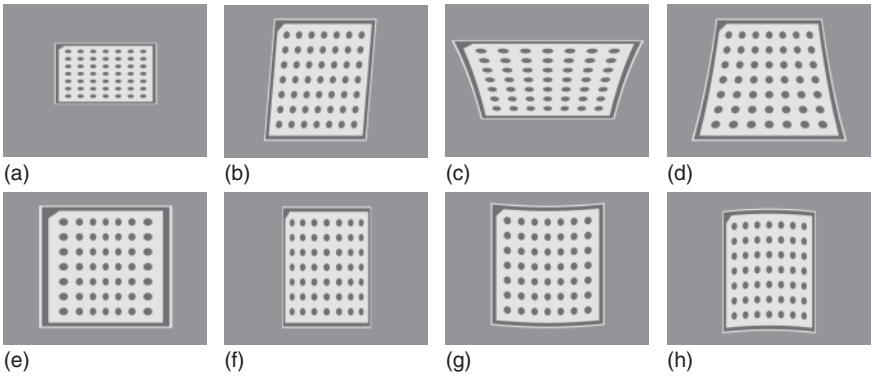


Figure 9.102 Some effects that occur for the line scan camera geometry. (a) Non-square pixels due to the not motion being tuned to the line frequency of the camera. (b) Skewed pixels due to the motion not being parallel to the *y*-axis of the CCS. (c) Straight lines can project to hyperbolic arcs, even if the line sensor is perpendicular to the motion. (d) This effect is more pronounced if the motion has a nonzero *z* component. Note that hyperbolic arcs occur even if the lens has no distortions ($\kappa = 0$). (e) Pincushion distortion ($\kappa > 0$) for $c_y = 0$. (f) Barrel distortion for $c_y = 0$. (g) Pincushion distortion ($\kappa > 0$) for $c_y > 0$. (h) Barrel distortion for $c_y > 0$.

obtain skew pixels. In this case, the camera would need to be better aligned with the motion vector. Figure 9.102c,d shows that straight lines can be projected to hyperbolic arcs [75], even if the lens has no distortions ($\kappa = 0$). This effect occurs even if $V = (0, v_y, 0)$, that is, if the line sensor is perfectly aligned perpendicular to the motion vector. The hyperbolic arcs are more pronounced if the motion vector has nonzero v_z. Figure 9.102e–h shows the effect of a lens with distortions ($\kappa \neq 0$) for the case where the sensor is located perfectly behind the projection center ($c_y = 0$) and for the case where the sensor is not perfectly aligned (here $c_y > 0$). For $c_y = 0$, the pincushion and barrel distortions only cause distortions within each row of the image. For $c_y \neq 0$, the rows are also bent.

9.9.3 Calibration Process

From the above discussion, we can see that camera calibration is the process of determining the interior and exterior camera parameters. To perform the calibration, it is necessary to know the location of a sufficiently large number of 3D points in world coordinates, and to be able to determine the correspondence between the world points and their projections in the image. To meet the first requirement, usually objects or marks that are easy to extract, for example, circles or linear grids, must be placed at known locations. If the location of a camera is to be known with respect to a given coordinate system, for example, with respect to the building plan of, say, a factory building, then each mark location must be measured very carefully within this coordinate system. Fortunately, it is often sufficient to know the position of a reference object with respect to the camera to be able to measure the object precisely, since the absolute position of the object in world coordinates is unimportant. Therefore, a movable calibration target that has been measured accurately can be used to calibrate the camera. This has the advantage that the calibration can be performed with the camera in place, for

example, already mounted in the machine. Furthermore, the position of the camera with respect to the objects can be recalibrated if required, for example, if the object type to be inspected changes.

The second requirement, that is, the necessity to determine the correspondence of the known world points and their projections in the image, is in general a hard problem. Therefore, calibration targets are usually constructed in such a way that this correspondence can be determined easily. For example, a planar calibration target with $m \times n$ circular marks within a rectangular border can be used. We have already seen examples of this kind of calibration target in Figures 9.85, 9.92, 9.98, and 9.102. Planar calibration targets have several advantages. First of all, they can be handled easily. Second, they can be manufactured very accurately. Finally, they can be used for back-light applications easily if a transparent medium is used as the carrier for the marks. A rectangular border around the calibration target allows the inner part of the calibration target to be found easily. A small orientation mark in one of the corners of the border enables the camera calibration algorithm to uniquely determine the orientation of the calibration target. Circular marks are used because their center point can be determined with high accuracy. Finally, the regular matrix layout of the rows and columns of the circles enables the camera calibration algorithm to determine the correspondence between the marks and their image points easily.

To extract the calibration target, we can make use of the fact that the border separates the inner part of the calibration target from the background. Consequently, the inner part can be found by a simple threshold operation (see Section 9.4.1). Since the correct threshold depends on the brightness of the calibration target in the image, different thresholds can be tried automatically until a region with $m \times n$ holes (corresponding to the calibration marks) has been found.

Once the region of the inner part of the calibration target has been found, the borders of the calibration marks can be extracted with subpixel-accurate edge extraction (see Section 9.7.3). Since the projections of the circular marks are ellipses, ellipses can then be fitted to the extracted edges with the algorithms described in Section 9.8.3 to obtain robustness against outliers in the edge points and to increase the accuracy. An example of the extraction of the calibration marks was already shown in Figure 9.92.

Finally, the correspondence between the calibration marks and their projections in the image can be determined easily based on the smallest enclosing quadrilateral of the extracted ellipses. Furthermore, the orientation of the marks can be determined uniquely based on the small triangular orientation mark. If it were not present, the orientation can only be determined modulo 90° for square calibration targets and modulo 180° for rectangular targets. It should be noted that the orientation of the calibration target is important only if the distances between the marks are not identical in both axes or in applications where the orientation must be determined uniquely across multiple images. One such example is stereo with cameras that are rotated significantly with respect to each other around their respective optical axes.

After the correspondence between the marks and their projections has been determined, the camera can be calibrated. Let us denote the 3D positions of the

centers of the marks by M_i. Since the calibration target is planar, we can place the calibration target in the plane $z = 0$. However, what we describe in the following is completely general and can be used for arbitrary calibration targets. Furthermore, let us denote the projections of the centers of the marks in the image by m_i. Here, we must take into account that the projection of the center of the circle is not the center of the ellipse [65, 76]. Finally, let us denote the camera parameters by a vector c. As described previously, c consists of the interior and exterior orientation parameters of the respective camera model. For example, $c = (f, \kappa, s_x, s_y, c_x, c_y, \alpha, \beta, \gamma, t_x, t_y, t_z)$ for pinhole cameras. Here, it should be noted that the exterior orientation is determined by attaching the WCS to the calibration target, for example, to the center mark, in such a way that the x- and y-axis of the WCS are aligned with the row and column directions of the marks on the calibration target and the z-axis points in the same direction as the optical axis if the calibration target is parallel to the image plane. Then, the camera parameters can be determined by minimizing the distance of the extracted mark centers m_i and their projections $\pi(M_i, c)$:

$$d(c) = \sum_{i=1}^{k} \| m_i - \pi(M_i, c) \|^2 \rightarrow \min \tag{9.135}$$

Here, $k = mn$ is the number of calibration marks. This is a difficult nonlinear optimization problem. Therefore, good starting values are required for the parameters. The interior orientation parameters can be determined from the specifications of the image sensor and the lens. The starting values for the exterior orientation are in general harder to obtain. For the planar calibration target described above, good starting values can be obtained based on the geometry and size of the projected circles [55, 64].

The optimization in equation (9.135) cannot determine all camera parameters because the physically motivated camera models we have chosen are over-parameterized. For example, for the pinhole camera, f, s_x, and s_y cannot be determined uniquely since they contain a common scale factor, as shown in Figure 9.103. For example, making the pixels twice as large and increasing the principal distance by a factor of 2 results in the same image. The solution to this problem is to keep s_y fixed in the optimization since the image is transmitted row-synchronously for all kinds of video signals. This fixes the common scale factor. On the other hand, s_x cannot be kept fixed in general since the video signal may not be sampled pixel-synchronously, at least for analog video signals. A similar effect happens for line scan cameras. Here, s_x and f cannot be determined uniquely. Consequently, s_x must be kept fixed in the calibration. Furthermore, as described in Section 9.9.2, s_y cannot be determined for line scan cameras and therefore also must be kept fixed.

Even if the above problem is solved, some degeneracies remain because we use a planar calibration target. For example, as shown in Figure 9.104, if the calibration target is parallel to the image plane, f and t_z cannot be determined uniquely since they contain a common scale factor. This problem also occurs if the calibration target is not parallel to the image plane. Here, f and a combination of parameters from the exterior orientation can be determined only up to a one-parameter

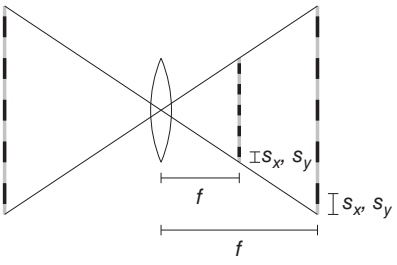


Figure 9.103 For pinhole cameras, f, s_x, and s_y cannot be determined uniquely.

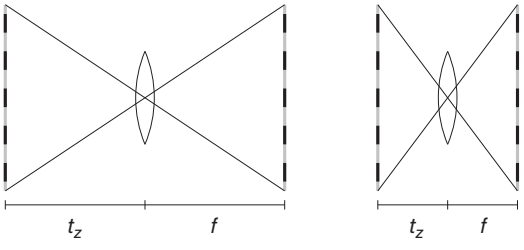


Figure 9.104 For pinhole cameras, f and t_z cannot be determined uniquely.

ambiguity. For example, if the calibration target is rotated around the x-axis, f, t_z, and α cannot be determined at the same time. For telecentric cameras, it is generally impossible to determine s_x, s_y, and the rotation angles from a single image. For example, a rotation of the calibration target around the x-axis can be compensated by a corresponding change in s_x. For line scan cameras, there are similar degeneracies as the ones described above plus degeneracies that include the motion vector.

To prevent the above degeneracies, the camera must be calibrated from multiple images in which the calibration target is positioned to avoid the degeneracies. For example, for pinhole cameras, the calibration targets must not be parallel to each other in all images; while for telecentric cameras, the calibration target must be rotated around all of its axes to avoid the above degeneracies. Suppose we use image l for the calibration. Then, we also have l sets of exterior orientation parameters $(\alpha_l, \beta_l, \gamma_l, t_{x,l}, t_{y,l}, t_{z,l})$ that must be determined. As above, we collect the interior orientation parameters and the l sets of exterior orientation parameters into the camera parameter vector c. Then, to calibrate the camera, we must solve the following optimization problem:

$$d(c) = \sum_{j=1}^{l} \sum_{i=1}^{k} \| m_{i,j} - \pi(M_i, c) \|^2 \rightarrow \min \tag{9.136}$$

Here, $m_{i,j}$ denotes the projection of the ith calibration mark in the jth image. If the calibration targets are placed and oriented suitably in the images, this will determine all the camera parameters uniquely. To ensure high accuracy of the camera parameters so determined, the calibration target should be placed at all four corners of the image. Since the distortion is largest in the corners, this will facilitate the determination of the distortion coefficient(s) with the highest possible accuracy.

To conclude this section, we mention that a flexible calibration algorithm will enable one to specify a subset of the parameters that should be determined.

For example, from the mechanical setup, some of the parameters of the exterior orientation may be known. One frequently encountered example is that the mechanical setup ensures with high accuracy that the calibration target is parallel to the image plane. In this case, we should set $\alpha = \beta = 0$, and the camera calibration should leave these parameters fixed. Another example is a camera with square pixels that transmits the video signal digitally. Here, $s_x = s_y$, and for pinhole cameras both parameters should be kept fixed. Finally, we will see in the next section that the calibration target determines the world plane in which measurements from a single image can be performed. In some applications, there is not enough space for the calibration target to be turned in three dimensions if the camera is mounted in its final position. Here, a two-step approach can be used. First, the interior orientation of the camera is determined with the camera not mounted in its final position. This ensures that the calibration target can be moved freely. (Note that this also determines the exterior orientation of the calibration target; however, this information is discarded.) Then, the camera is mounted in its final position. Here, it must be ensured that the focus and iris diaphragm settings of the lens are not changed, since this will change the interior orientation. In the final position, a single image of the calibration target is taken, and only the exterior orientation is optimized to determine the pose of the camera with respect to the measurement plane.

9.9.4 World Coordinates from Single Images

As mentioned previously and at the end of Section 9.7.4, if the camera is calibrated, it is possible in principle to obtain undistorted measurements in world coordinates. In general, this can be done only if two or more images of the same object are taken at the same time with cameras at different spatial positions. This is called stereo reconstruction. With this approach, discussed in Section 9.10, the reconstruction of 3D positions for corresponding points in the two images is possible because the two optical rays defined by the two optical centers of the cameras and the points in the image plane defined by the two image points can be intersected in 3D space to give the 3D position of that point. In some applications, however, it is impossible to use two cameras, for example, because there is not enough space to mount two cameras. Nevertheless, it is possible to obtain measurements in world coordinates for objects acquired through telecentric lenses and for objects that lie in a known plane, for example, on a conveyor belt, for pinhole and line scan cameras. Both of these problems can be solved by intersecting an optical ray (also called the line of sight) with a plane. With this, it is possible to measure objects that lie in a plane, even if the plane is tilted with respect to the optical axis.

Let us first look at the problem of determining world coordinates for telecentric cameras. In this case, the parallel projection in equation (9.127) discards any depth information completely. Therefore, we cannot hope to discover the distance of the object from the camera, that is, its z coordinate in the CCS. What we can recover, however, are the x and y coordinates of the object in the CCS (x_c and y_c in equation (9.127)), that is, the dimensions of the object in world units. Since the z coordinate of P_c cannot be recovered, in most cases it is unnecessary

to transform P_c into world coordinates by inverting equation (9.124). Instead, the point P_c is regarded as a point in world coordinates. To recover P_c, we can start by inverting equation (9.132) to transform the coordinates from the ICS into the IPCS:

$$\begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix} = \begin{pmatrix} s_x(c - c_x) \\ s_y(r - c_y) \end{pmatrix} \tag{9.137}$$

Then, we can rectify the lens distortions by applying equation (9.129) or (9.130) to obtain the rectified coordinates $(u, v)^\top$ in the image plane. Finally, the coordinates of P_c are given by

$$P_c = (x_c, y_c, z_c)^\top = (u, v, 0)^\top \tag{9.138}$$

Note that the above procedure is equivalent to intersecting the optical ray given by the point $(u, v, 0)^\top$ and the direction perpendicular to the image plane, that is, $(0, 0, 1)^\top$, with the plane $z = 0$.

The determination of world coordinates for pinhole cameras is slightly more complicated, but uses the same principle of intersecting an optical ray with a known plane. Let us look at this problem by using the application where this procedure is most useful. In many applications, the objects to be measured lie in a plane in front of the camera, for example, a conveyor belt. Let us assume for the moment that the location and orientation of this plane (its pose) are known. We will describe below how to obtain this pose. The plane can be described by its origin and a local coordinate system, that is, three orthogonal vectors, one of which is perpendicular to the plane. To transform the coordinates in the coordinate system of the plane (the WCS) into the CCS, a rigid transformation given by equation (9.124) must be used. Its six parameters $(\alpha, \beta, \gamma, t_x, t_y, t_z)$ describe the pose of the plane. If we want to measure objects in this plane, we need to determine the object coordinates in the WCS defined by the plane. Conceptually, we need to intersect the optical ray corresponding to an image point with the plane. To do so, we need to know two points that define the optical ray. Recalling Figures 9.96 and 9.97, obviously the first point is given by the projection center, which has the coordinates $(0, 0, 0)^\top$ in the CCS. To obtain the second point, we need to transform the point $(r, c)^\top$ from the ICS into the IPCS. This transformation is given by equations (9.137) and (9.129) or (9.130). To obtain the 3D point that corresponds to this point in the image plane, we need to take into account that the image plane lies at a distance of f in front of the optical center. Hence, the coordinates of the second point on the optical ray are given by $(u, v, f)^\top$. Therefore, we can describe the optical ray in the CCS by

$$L_c = (0, 0, 0)^\top + \lambda(u, v, f)^\top \tag{9.139}$$

To intersect this line with the plane, it is best to express the line L_c in the WCS of the plane, since in the WCS the plane is given by the equation $z = 0$. Therefore, we need to transform the two points $(0, 0, 0)^\top$ and $(u, v, f)^\top$ into the WCS. This can be done by inverting equation (9.124) to obtain

$$P_w = \mathtt{R}^{-1}(P_c - T) = \mathtt{R}^\top(P_c - T) \tag{9.140}$$

Here, $\mathtt{R}^{-1} = \mathtt{R}^\top$ is the inverse of the rotation matrix $\mathtt{R}$ in equation (9.124). Let us call the transformed optical center O_w, that is, $O_w = \mathtt{R}^\top((0, 0, 0)^\top - T) = -\mathtt{R}^\top T$,

and the transformed point in the image plane I_w, that is, $I_w = \mathtt{R}^\top((u, v, f)^\top - T)$. With this, the optical ray is given by

$$L_w = O_w + \lambda(I_w - O_w) = O_w + \lambda D_w \tag{9.141}$$

in the WCS. Here, D_w denotes the direction vector of the optical ray. With this, it is a simple matter to determine the intersection of the optical ray in equation (9.141) with the plane $z = 0$. The intersection point is given by

$$P_w = \begin{pmatrix} o_x - o_z\, d_x/d_z \\ o_y - o_z\, d_y/d_z \\ 0 \end{pmatrix} \tag{9.142}$$

where $O_w = (o_x, o_y, o_z)^\top$ and $D_w = (d_x, d_y, d_z)^\top$.

Up to now, we have assumed that the pose of the plane in which we want to measure objects is known. Fortunately, the camera calibration gives us this pose almost immediately since a planar calibration target is used. If the calibration target is placed on the plane, for example, the conveyor belt, in one of the images used for calibration, the exterior orientation of the calibration target in that image almost defines the pose of the plane we need in the above derivation. The pose would be the true pose of the plane if the calibration target were infinitely thin. To take the thickness of the calibration target into account, the WCS defined by the exterior orientation must be moved by the thickness of the calibration target in the positive z direction. This modifies the transformation from the WCS to the CCS in equation (9.124) as follows: the translation T simply becomes $\mathtt{R}D + T$, where $D = d(0, 0, 1)^\top$, and d is the thickness of the calibration target. This is the pose that must be used in equation (9.140) to transform the optical ray into the WCS. An example of computing edge positions in world coordinates for pinhole cameras is given at the end of Section 9.7.4 (see Figure 9.86).

For line scan cameras, the procedure to obtain world coordinates in a given plane is conceptually similar to the approaches described above. First, the optical ray is constructed from equations (9.134) and (9.133). Then, it is transformed into the WCS and intersected with the plane $z = 0$.

In addition to transforming image points, for example, 1D edge positions or subpixel-precise contours, into world coordinates, sometimes it is also useful to transform the image itself into world coordinates. This creates an image that would have resulted if the camera had looked perfectly perpendicularly without distortions onto the world plane. This image rectification is useful for applications that must work on the image data itself, for example, region processing, template matching, or OCR. It can be used whenever the camera cannot be mounted perpendicular to the measurement plane. To rectify the image, we conceptually cut out a rectangular region of the world plane $z = 0$ and sample it with a specified distance, for example, 200 µm. We then project each sample point into the image with the equations of the relevant camera model, and obtain the gray value through interpolation, for example, bilinear interpolation. Figure 9.105 shows an example of this process. In Figure 9.105a, the image of a caliper together with the calibration target that defines the world plane is shown. The unrectified and rectified images of the caliper are shown in Figure 9.105b,c, respectively.

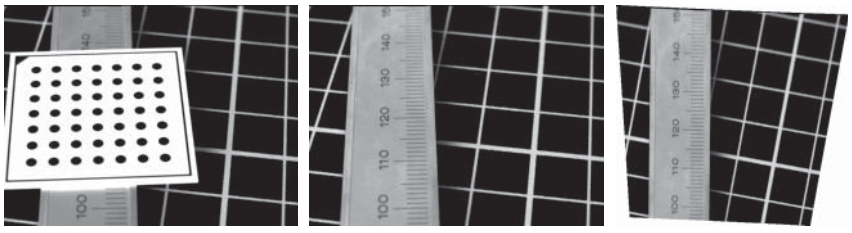


Figure 9.105 (a) Image of a caliper with a calibration target. (b) Unrectified image of the caliper. (c) Rectified image of the caliper.

Note that the rectification has removed the perspective and radial distortions from the image.

9.9.5 Accuracy of the Camera Parameters

We conclude the discussion of camera calibration by discussing two different aspects: the accuracy of the camera parameters, and the changes in the camera parameters that result from adjusting the focus and iris diaphragm settings on the lens.

As was already noted in Section 9.9.3, there are some cases where an inappropriate placement of the calibration target can result in degenerate configurations where one of the camera parameters or a combination of some of the parameters cannot be determined. These configurations must obviously be avoided if the camera parameters should be determined with high accuracy. Apart from this, the main influencing factor for the accuracy of the camera parameters is the number of images that are used to calibrate the camera. This is illustrated in Figure 9.106, where the standard deviations of the principal distance f, the radial distortion coefficient κ, and the principal point (c_x, c_y) are plotted as functions of the number of images that are used for calibration. To obtain this data, 20 images of a calibration target were taken. Then, every possible subset of l images ($l = 2, \ldots, 19$) from the 20 images was used to calibrate the camera. The standard deviations were calculated from the resulting camera parameters when l of the 20 images were used for the calibration. From Figure 9.106 it is obvious that the accuracy of the camera parameters increases significantly as the number of images l increases. This is not surprising when we consider that each image serves to constrain the parameters. If the images were independent measurements, we could expect the standard deviation to decrease proportionally to $l^{-0.5}$. In this particular example, f decreases roughly proportionally to $l^{-1.5}$, κ decreases roughly proportionally to $l^{-1.1}$, and c_x and c_y decrease roughly proportionally to $l^{-1.2}$, that is, much faster than $l^{-0.5}$.

From Figure 9.106, it can also be seen that a comparatively large number of calibration images is required to determine the camera parameters accurately. This happens because there are nonnegligible correlations between the camera parameters, which can only be resolved through multiple independent measurements. To obtain accurate camera parameters, it is important that the calibration target covers the entire field of view and that it tries to cover the range of

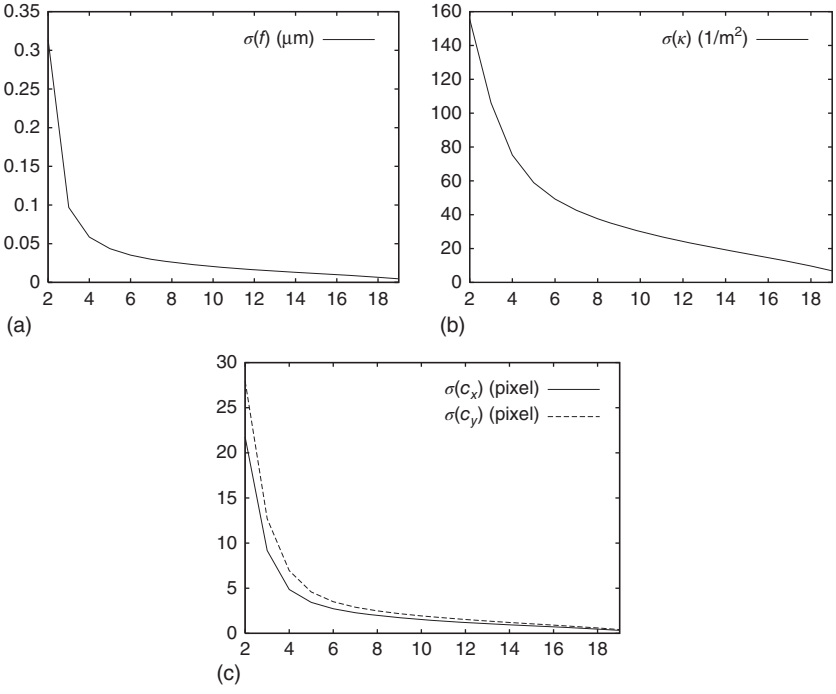


Figure 9.106 Standard deviations of (a) the principal distance f, (b) the radial distortion coefficient κ, and (c) the principal point (c_x, c_y) as functions of the number of images that are used for calibration.

exterior orientations as well as possible. In particular, κ can be determined more accurately if the calibration target is placed into each corner of the image. Furthermore, all parameters can be determined more accurately if the calibration target covers a large depth range. This can be achieved by turning the calibration target around its x- and y-axes, and by placing it at different depths relative to the camera.

We now turn to a discussion of whether changes in the lens settings change the camera parameters. Figure 9.107 displays the effect of changing the focus

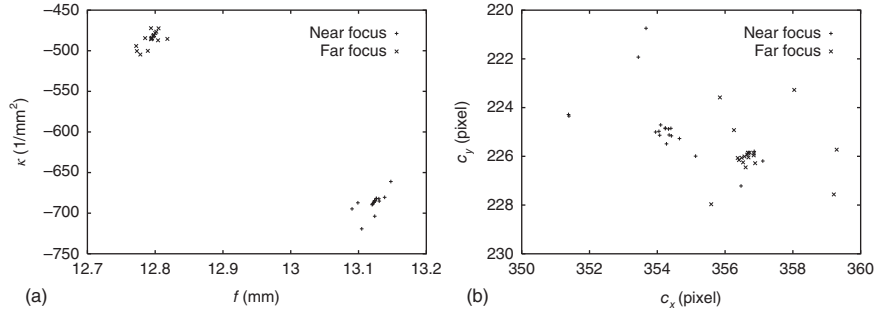


Figure 9.107 (a) Principal distances and radial distortion coefficients and (b) principal points for a lens with two different focus settings.

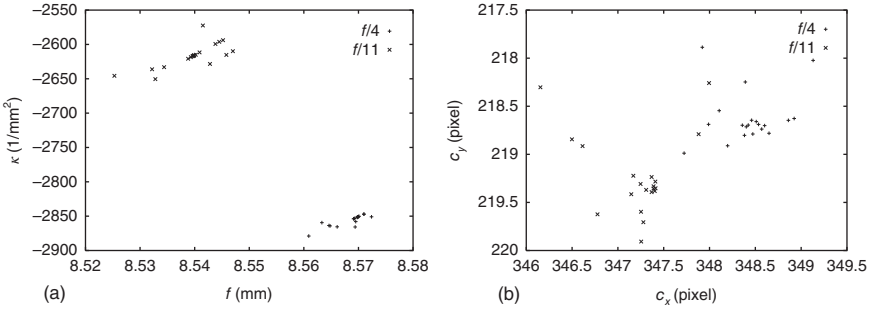


Figure 9.108 (a) Principal distances and radial distortion coefficients and (b) principal points for a lens with two different iris diaphragm settings (f-numbers: $f/4$ and $f/11$).

setting of the lens. Here, a 12.5 mm lens with a 1mm extension tube was used. The lens was set to the nearest and farthest focal settings. In the near focus setting, the camera was calibrated with a calibration target of size 1 cm × 1 cm, while for the far focus setting a calibration target of size 3 cm × 3 cm was used. This resulted in the same size of the calibration target in the focusing plane for both settings. Care was taken to use images of calibration targets with approximately the same range of depths and positions in the images. For each setting, 20 images of the calibration target were taken. To be able to evaluate statistically whether the camera parameters are different, all 20 subsets of 19 of the 20 images were used to calibrate the camera. As can be expected from the discussion in Section 4.2, changing the focus will change the principal distance. From Figure 9.107a, we can see that this clearly is the case. Furthermore, the radial distortion coefficient κ also changes significantly. From Figure 9.107b, it is also obvious that the principal point changes. The probability that the two means of the respective principal points are identical is less than 10^{-7}.

Finally, we examine what can happen when the iris diaphragm on the lens is changed. To test this, a similar setup as above was used. The camera was set to f-numbers $f/4$ and $f/11$. For each setting, 20 images of a 3 cm × 3 cm calibration target were taken. Care was taken to position the calibration targets in similar positions for the two settings. The lens is an 8.5mm lens with a 1mm extension tube. Again, all 20 subsets of 19 of the 20 images were used to calibrate the camera. Figure 9.108a displays the principal distance and radial distortion coefficient. Clearly, the two parameters change in a statistically significant way. Figure 9.108b also shows that the principal point changes significantly. All parameters have a probability of being equal that is less than 10^{-7}. The changes in the parameters mean that there is an overall difference in the point coordinates across the image diagonal of ≈ 1.5 pixels. Therefore, we can see that changing the f-number on the lens requires a recalibration, at least for some lenses.

9.10 Stereo Reconstruction

In Sections 9.9 and 9.7.4, we have seen that we can perform very accurate measurements from a single image by calibrating the camera and by determining its

exterior orientation with respect to a plane in the world. We could then convert the image measurements to world coordinates within the plane by intersecting optical rays with the plane. Note, however, that these measurements are still 2D measurements within the world plane. In fact, from a single image we cannot reconstruct the 3D geometry of the scene because we can only determine the optical ray for each point in the image. We do not know at which distance on the optical ray the point lies in the world. In the approach in Section 9.9.4, we had to assume a special geometry in the world to be able to determine the distance of a point along the optical ray. Note that this is not a true 3D reconstruction. To perform a 3D reconstruction, we must use at least two images of the same scene taken from different positions. Typically, this is done by simultaneously taking the images with two cameras. This process is called stereo reconstruction. In this section, we will examine the case of binocular stereo, that is, we will concentrate on the two-camera case. Throughout this section, we will assume that the cameras have been calibrated, that is, their interior orientations and relative orientation are known. While uncalibrated reconstruction is also possible [20, 21], the corresponding methods have not yet been used in industrial applications.

9.10.1 Stereo Geometry

Before we can discuss the stereo reconstruction, we must examine the geometry of two cameras, as shown in Figure 9.109. Since the cameras are assumed to be calibrated, we know their interior orientations, that is, their principal points, focal lengths (more precisely, their principal distances, that is, the distance between the image plane and the projection center), pixel size, and distortion coefficient(s). In Figure 9.109, the principal points are shown by the points C_1 and C_2 in the first and second image, respectively. Furthermore, the projection centers are shown by the points O_1 and O_2. The dashed line between the projection centers and principal points shows the principal distances. Note that, since the image planes physically lie behind the projection centers, the image is turned upside down. Consequently, the origin of the ICS lies in the lower right corner, with the row

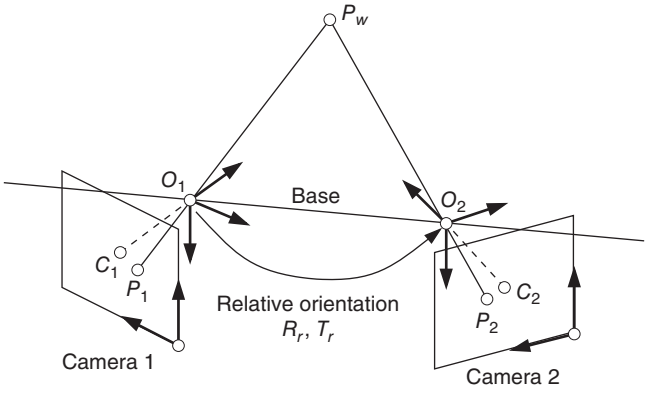


Figure 9.109 Stereo geometry of two cameras.

axis pointing upward and the column axis pointing leftward. The CCS axes are defined such that the x-axis points to the right, the y-axis points downwards, and the z-axis points forward from the image plane, that is, along the viewing direction. The position and orientation of the two cameras with respect to each other are given by the relative orientation, which is a rigid 3D transformation specified by the rotation matrix $\mathtt{R}_r$ and the translation vector T_r. It can be interpreted either as the transformation of the camera coordinate system of the first camera into the CCS of the second camera or as a transformation that transforms point coordinates in the camera coordinate system of the second camera into point coordinates of the CCS of the first camera: $P_{c1} = \mathtt{R}_r P_{c2} + T_r$. The translation vector T_r, which specifies the translation between the two projection centers, is also called the base. With this, we can see that a point P_w in the world is mapped to a point P_1 in the first image and to a point P_2 in the second image. If there are no lens distortions (which we will assume for the moment), the points P_w, O_1, O_2, P_1, and P_2 all lie in a single plane.

To calibrate the stereo system, we can extend the method of Section 9.9.3 as follows. Let M_i denote the positions of the calibration marks. We extract their projections in both images with the methods described in Section 9.9.3. Let us denote the projection of the centers of the marks in the first set of calibration images by $m_{i,j,1}$ and in the second set by $m_{i,j,2}$. Furthermore, let us denote the camera parameters by a vector c. The camera parameters c include the interior orientation of the first and second camera, the exterior orientation of the l calibration targets in the second image, and the relative orientation of the two cameras. From the above discussion of the relative orientation, it follows that these parameters determine the mappings $\pi_1(M_i, c)$ and $\pi_2(M_i, c)$ into the first and second images completely. Hence, to calibrate the stereo system, the following optimization problem must be solved:

$$d(c) = \sum_{j=1}^{l} \sum_{i=1}^{k} \| m_{i,j,1} - \pi_1(M_i, c)\|^2 + \| m_{i,j,2} - \pi_2(M_i, c)\|^2 \rightarrow \min \quad (9.143)$$

To illustrate the relative orientation and the stereo calibration, Figure 9.110 shows an image pair taken from a sequence of 15 image pairs that were used to calibrate a binocular stereo system. The calibration returns a translation vector of (0.1534 m, −0.0037 m, 0.0449 m) between the cameras, that is, the second camera is 15.34 cm to the right, 0.37 cm above, and 4.49 cm in front of the first camera, expressed in the camera coordinates of the first camera. Furthermore, the calibration returns a rotation angle of 40.1139° around the axis (−0.0035, 1.0000, 0.0008), that is, almost around the vertical y-axis of the CCS. Hence, the cameras are verging inward, like in Figure 9.109.

To reconstruct 3D points, we must find the corresponding points in the two images. "Corresponding" means that the two points P_1 and P_2 in the images belong to the same point P_w in the world. At first, it might seem that, given a point P_1 in the first image, we would have to search in the entire second image for the corresponding point P_2. Fortunately, this is not the case. In Figure 9.109, we already noted that the points P_w, O_1, O_2, P_1, and P_2 all lie in a single plane. The situation of trying to find a corresponding point for P_1 is shown in Figure 9.111.

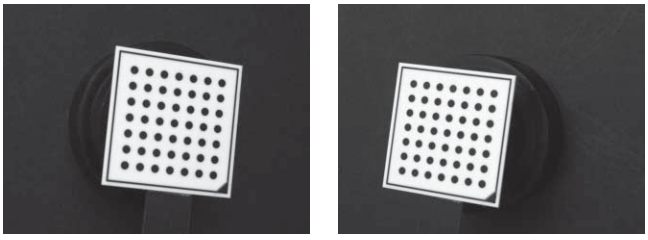


Figure 9.110 One image pair taken from a sequence of 15 image pairs that are used to calibrate a binocular stereo system. The calibration returns a translation vector (base) of (0.1534 m, − 0.0037 m, 0.0449 m) between the cameras and a rotation angle of 40.1139° around the axis (−0.0035, 1.0000, 0.0008), that is, almost around the *y*-axis of the camera coordinate system. Hence, the cameras are verging inward.

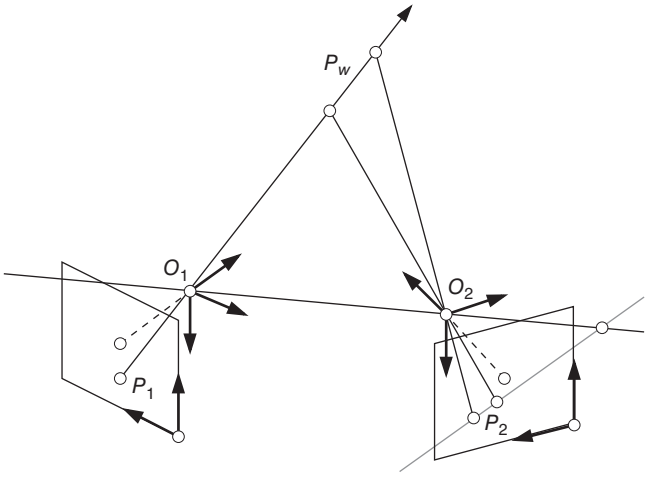


Figure 9.111 Epipolar geometry of two cameras. Given the point P_1 in the first image, the point P_2 in the second image can only lie on the epipolar line of P_1, which is the projection of the epipolar plane spanned by P_1, O_1, and O_2 onto the second image.

We can note that we know P_1, O_1, and O_2. We do not know at which distance the point P_w lies on the optical ray defined by P_1 and O_1. However, we know that P_w is coplanar with the plane spanned by P_1, O_1, and O_2 (the epipolar plane). Hence, we can see that the point P_2 can only lie on the projection of the epipolar plane onto the second image. Since O_2 lies on the epipolar plane, the projection of the epipolar plane is a line called the epipolar line.

It is obvious that the above construction is symmetric for both images, as shown in Figure 9.112. Hence, given a point P_2 in the second image, the corresponding point can only lie on the epipolar line in the first image. Furthermore, from Figure 9.112, we can see that different points typically define different epipolar lines. We can also see that all epipolar lines of one image intersect at a single point called the epipole. The epipoles are the projections of the opposite projective centers onto the respective image. Note that, since all epipolar planes contain

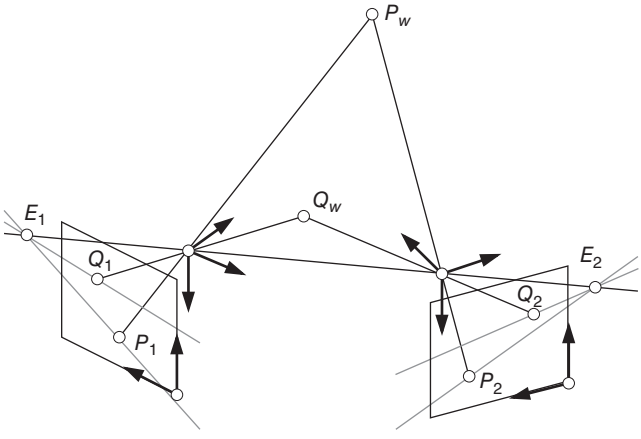


Figure 9.112 The epipolar geometry is symmetric between the two images. Furthermore, different points typically define different epipolar lines. All epipolar lines intersect at the epipoles E_1 and E_2, which are the projections of the opposite projective centers onto the respective image.

O_1 and O_2, the epipoles lie on the line defined by the two projection centers (the base line).

Figure 9.113 shows an example of the epipolar lines. The stereo geometry is identical to Figure 9.110. The images show a PCB. In Figure 9.113a, four points are marked. They have been selected manually to lie at the tips of the triangles on the

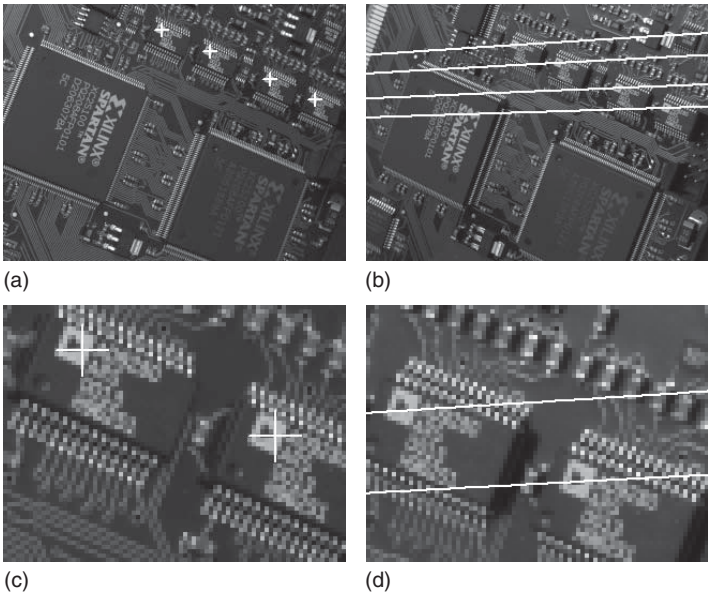


Figure 9.113 Stereo image pair of a PCB. (a) Four points marked in the first image. (b) Corresponding epipolar lines in the second image. (c) Detail of (a). (d) Detail of (b). The four points in (a) have been selected manually at the tips of the triangles on the four small ICs. Note that the epipolar lines pass through the tips of the triangles in the second image.

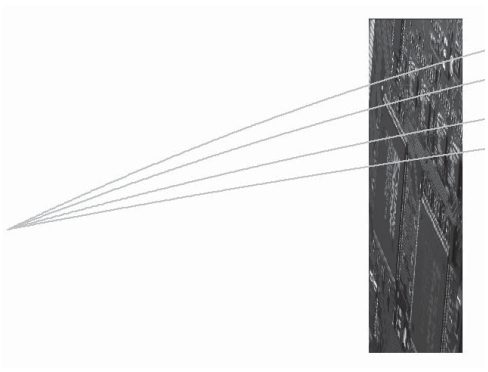


Figure 9.114 Because of lens distortions, the epipolar lines are generally not straight. The image shows the same image as Figure 9.113b. The zoom has been set so that the epipole is shown in addition to the image. The aspect ratio has been chosen so that the curvature of the epipolar lines is clearly visible.

four small ICs, as shown in the detailed view in Figure 9.113c. The corresponding epipolar lines in the second image are shown in Figure 9.113b,d. Note that the epipolar lines pass through the tips of the triangles in the second image.

As noted previously, we have so far assumed that the lenses have no distortions. In reality, this is very rarely true. In fact, by looking closely at Figure 9.113b, we can already perceive a curvature in the epipolar lines because the camera calibration has determined the radial distortion coefficient for us. If we set the displayed image part as in Figure 9.114, we can clearly see the curvature of the epipolar lines in real images. Furthermore, we can see the epipole of the image clearly.

From the above discussion, we can see that the epipolar lines are different for different points. Furthermore, because of lens distortions, they typically are not even straight. This means that, when we try to find corresponding points, we must compute a new, complicated epipolar line for each point that we are trying to match, typically for all points in the first image. The construction of the curved epipolar lines would be much too time consuming for real-time applications. Hence, we can ask ourselves whether the construction of the epipolar lines can be simplified for particular stereo geometries. This is indeed the case for the stereo geometry shown in Figure 9.115. Here, both image planes lie in the same plane and are vertically aligned. Furthermore, it is assumed that there

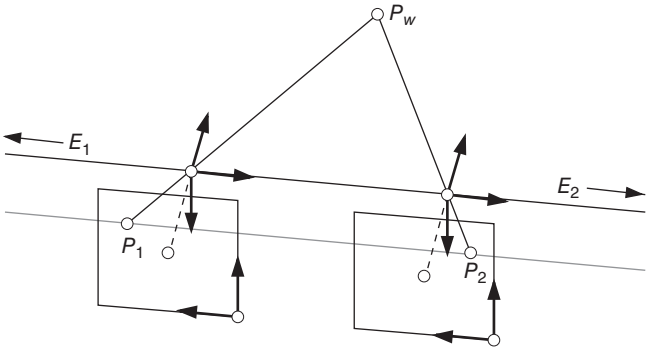


Figure 9.115 The epipolar standard geometry is obtained if both image planes lie in the same plane and are vertically aligned. Furthermore, it is assumed that there are no lens distortions. In this geometry, the epipolar line for a point is simply the line that has the same row coordinate as the point, that is, the epipolar lines are horizontal and vertically aligned.

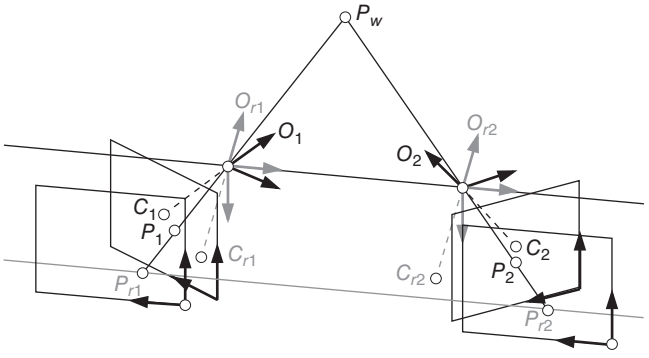


Figure 9.116 Transformation of a stereo configuration into the epipolar standard geometry.

are no lens distortions. Note that this implies that the two principal distances are identical, that the principal points have the same row coordinate, that the images are rotated such that the column axis is parallel to the base, and that the relative orientation contains only a translation in the x direction and no rotation. Since the image planes are parallel to each other, the epipoles lie infinitely far away on the base line. It is easy to see that this stereo geometry implies that the epipolar line for a point is simply the line that has the same row coordinate as the point, that is, the epipolar lines are horizontal and vertically aligned. Hence, they can be computed without any overhead at all. Since almost all stereo matching algorithms assume this particular geometry, we can call it the epipolar standard geometry.

While the epipolar standard geometry results in very simple epipolar lines, it is extremely difficult to align real cameras into this configuration. Furthermore, it is quite difficult and expensive to obtain distortion-free lenses. Fortunately, almost any stereo configuration can be transformed into the epipolar standard geometry, as indicated in Figure 9.116 [77]. The only exceptions are if an epipole happens to lie within one of the images. This typically does not occur in practical stereo configurations. The process of transforming the images to the epipolar standard geometry is called image rectification. To rectify the images, we need to construct two new image planes that lie in the same plane. To keep the 3D geometry identical, the projective centers must remain at the same positions in space, that is, $O_{r1} = O_1$ and $O_{r2} = O_2$. Note, however, that we need to rotate the CCSs such that their x-axes become identical to the base line. Furthermore, we need to construct two new principal points C_{r1} and C_{r2}. Their connecting vector must be parallel to the base. Furthermore, the vectors from the principal points to the projection centers must be perpendicular to the base. This leaves us two degrees of freedom. First of all, we must choose a common principal distance. Second, we can rotate the common plane in which the image planes lie around the base. These parameters can be chosen by requiring that the image distortion should be minimized [77]. The image dimensions are then typically chosen such that the original images are completely contained within the rectified images. Of course, we must also remove the lens distortions in the rectification.

To obtain the gray value for a pixel in the rectified image, we construct the optical ray for this pixel and intersect it with the original image plane. This is shown,

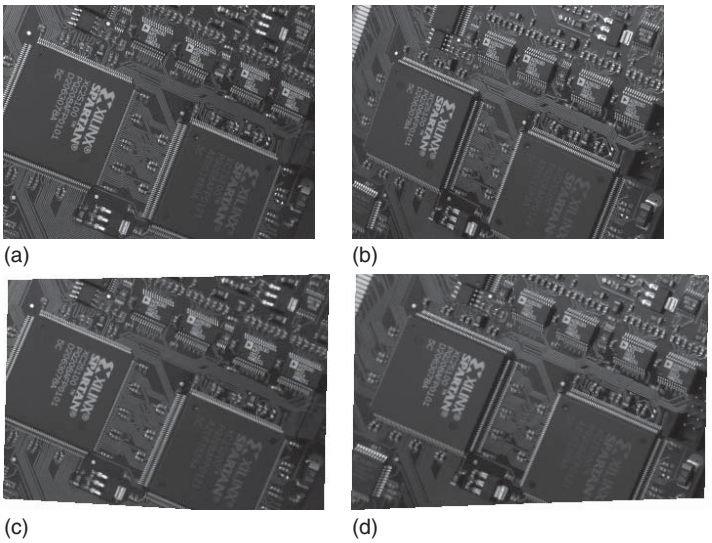


Figure 9.117 Example of the rectification of a stereo image pair. The images in (a) and (b) have the same relative orientation as those in Figure 9.110. The rectified images are shown in (c) and (d). Note the trapezoidal shape of the rectified images, which is caused by the verging cameras. Also note that the rectified images are slightly wider than the original images.

for example, for the points P_{r1} and P_1 in Figure 9.116. Since this typically results in subpixel coordinates, the gray values must be interpolated with the techniques described in Section 9.3.3.

While it may seem that image rectification is a very time-consuming process, the entire transformation can be computed once offline and stored in a table. Hence, images can be rectified very efficiently online.

Figure 9.117 shows an example of image rectification. The input image pair is shown in Figure 9.117a,b. The images have the same relative orientation as the images in Figure 9.110. The principal distances of the cameras are 13.05 mm and 13.16 mm, respectively. Both images have dimensions 320×240. Their principal points are (155.91, 126.72) and (163.67, 119.20), that is, they are very close to the image center. Finally, the images have a slight barrel-shaped distortion. The rectified images are shown in Figure 9.117c,d. Their relative orientation is given by the translation vector (0.1599 m, 0 m, 0 m). As expected, the translation is solely along the x-axis. Of course, the length of the translation vector is identical to that in Figure 9.110, since the position of the projective centers has not changed. The new principal distance of both images is 12.27mm. The new principal points are given by (−88.26, 121.36) and (567.38, 121.36). As can be expected from Figure 9.116, they lie well outside the rectified images. Also, as expected, the row coordinates of the principal points are identical. The rectified images have dimensions 336×242 and 367×242, respectively. Note that they exhibit a trapezoidal shape that is characteristic of the verging camera configuration. The barrel-shaped distortion has been removed from the images. Clearly, the epipolar lines are horizontal in both images.

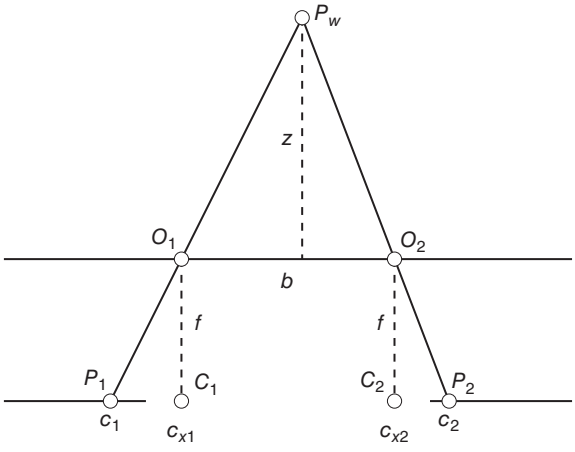


Figure 9.118 Reconstruction of the depth z of a point depends only on the disparity $d = c_2 - c_1$ of the points, that is, the difference of the column coordinates in the rectified images.

Apart from the fact that rectifying the images results in a particularly simple structure for the epipolar lines, it also results in a very simple reconstruction of the depth, as shown in Figure 9.118. In this figure, the stereo configuration is displayed as viewed along the direction of the row axis of the images, that is, the y-axis of the camera coordinate system. Hence, the image planes are shown as the lines at the bottom of the figure. The depth of a point is quite naturally defined as its z coordinate in the camera coordinate system. By examining the similar triangles $O_1O_2P_w$ and $P_1P_2P_w$, we can see that the depth of P_w depends only on the difference of the column coordinates of the points P_1 and P_2 as follows. From the similarity of the triangles, we have $z/b = (z+f)/(d_w + b)$. Hence, the depth is given by $z = bf/d_w$. Here, b is the length of the base, f is the principal distance, and d_w is the sum of the signed distances of the points P_1 and P_2 to the principal points C_1 and C_2. Since the coordinates of the principal points are given in pixels, but d_w is given in world units, for example, meters, we have to convert d_w to pixel coordinates by scaling it with the size of the pixels in the x direction: $d_p = d_w/s_x$. Now, we can easily see that $d_p = (c_{x1} - c_1) + (c_2 - c_{x2})$, where c_1 and c_2 denote the column coordinates of the points P_1 and P_2, while c_{x1} and c_{x2} denote the column coordinates of the principal points. Rearranging the terms, we find

$$d_p = (c_{x1} - c_{x2}) + (c_2 - c_1) \tag{9.144}$$

Since $c_{x1} - c_{x2}$ is constant for all points and known from the calibration and rectification, we can see that the depth z depends only on the difference of the column coordinates $d = c_2 - c_1$. This difference is called the disparity. Hence, we can see that, to reconstruct the depth of a point, we must determine its disparity.

9.10.2 Stereo Matching

As we have seen in the previous section, the main step in the stereo reconstruction is the determination of the disparity of each point in one of the images, typically the first image. Since one calculates, or at least attempts to calculate, a disparity for each point, these algorithms are called dense reconstruction algorithms. It should be noted that there is another class of algorithms that only try to

reconstruct the depth for selected features, for example, straight lines or points. Since these algorithms typically require expensive feature extraction, they are seldom used in industrial applications. Therefore, we will concentrate on dense reconstruction algorithms.

Reviews of dense stereo reconstruction algorithms published until 2002 are given in [78, 79]. Since then, many new stereo algorithms have been published. Evaluations of newly proposed algorithms are constantly updated on various stereo vision benchmark web sites (see, e.g., [80, 81]). While many of these algorithms offer stereo reconstructions of somewhat better quality than the algorithms we will discuss in the following, they also often are much too slow or have too demanding memory requirements for industrial applications.

Since the goal of dense reconstruction is to find the disparity for each point in the image, the determination of the disparity can be regarded as a template matching problem. Given a rectangular window of size $(2n+1) \times (2n+1)$ around the current point in the first image, we must find the most similar window along the epipolar line in the second image. Hence, we can use the techniques that will be described in greater detail in Section 9.11 to match a point. The gray value matching methods described in Section 9.11.1 are of particular interest because they do not require a costly model generation step, which would have to be performed for each point in the first image. Therefore, the gray value matching methods typically are the fastest methods for stereo reconstruction. The simplest similarity measures are the SAD (sum of absolute gray value differences) and SSD (sum of squared gray value differences) measures described later (see equations (9.149) and (9.150)). For the stereo matching problem, they are given by

$$\mathrm{SAD}(r,c,d) = \frac{1}{(2n+1)^2} \sum_{i=-n}^{n} \sum_{j=-n}^{n} |g_1(r+i,c+j) - g_2(r+i,c+j+d)| \tag{9.145}$$

and

$$\mathrm{SSD}(r,c,d) = \frac{1}{(2n+1)^2} \sum_{i=-n}^{n} \sum_{j=-n}^{n} (g_1(r+i,c+j) - g_2(r+i,c+j+d))^2 \tag{9.146}$$

As will be discussed in Section 9.11.1, these two similarity measures can be computed very quickly. Fast implementations for stereo matching using the SAD are given in [82, 83]. Unfortunately, these similarity measures have the disadvantage that they are not robust against illumination changes, which frequently happen in stereo reconstruction because of the different viewing angles along the optical rays. One way to deal with this problem is to perform a suitable preprocessing of the stereo images to remove illumination variations [84]. The preprocessing, however, is rarely invariant to arbitrary illumination changes. Consequently, in some applications it may be necessary to use the normalized cross-correlation (NCC) described later (see equation (9.151)) as the similarity measure, which has been shown to be robust to a very large range of illumination changes that can

occur in stereo reconstruction [84]. For the stereo matching problem, it is given by

$$\mathrm{NCC}(r,c,d) = \frac{1}{(2n+1)^2} \sum_{i=-n}^{n} \sum_{j=-n}^{n} \frac{g_1(r+i,c+j) - m_1(r+i,c+j)}{\sqrt{s_1^2(r+i,c+j)}} \cdot \frac{g_2(r+i,c+j+d) - m_2(r+i,c+j+d)}{\sqrt{s_2^2(r+i,c+j+d)}} \tag{9.147}$$

Here, m_i and s_i $(i = 1, 2)$ denote the mean and standard deviation of the window in the first and second images. They are calculated similar to their template matching counterparts in equations (9.152) and (9.153). The advantage of NCC is that it is invariant against linear illumination changes. However, it is more expensive to compute.

From the above discussion, it might appear that, to match a point, we would have to compute the similarity measure along the entire epipolar line in the second image. Fortunately, this is not the case. Since the disparity is inversely related to the depth of a point, and we typically know in which range of distances the objects we are interested in occur, we can restrict the disparity search space to a much smaller interval than the entire epipolar line. Hence, we have $d \in [d_{\min}, d_{\max}]$, where $d_{\min}$ and $d_{\max}$ can be computed from the minimum and maximum expected distance in the images. Therefore, the length of the disparity search space is given by $l = d_{\max} - d_{\min} + 1$.

After we have computed the similarity measure for the disparity search space for a point to be matched, we might be tempted to simply use the disparity with the minimum (SAD and SSD) or maximum (NCC) similarity measure as the match for the current point. However, this typically will lead to many false matches, since some windows may not have a good match in the second image. In particular, this happens if the current point is occluded because of perspective effects in the second image. Therefore, it is necessary to threshold the similarity measure, that is, to accept matches only if their similarity measure is below (SAD and SSD) or above (NCC) a threshold. Obviously, if we perform this thresholding, some points will not have a reconstruction, and consequently the reconstruction will not be completely dense.

With the above search strategy, the matching process has a complexity of $O(whln^2)$. This is much too expensive for real-time performance. Fortunately, it can be shown that with a clever implementation the above similarity measures can be computed recursively. With this, the complexity can be made independent of the window size n, and becomes $O(whl)$. With this, real-time performance becomes possible. The interested reader is referred to [85] for details.

Once we have computed the match with an accuracy of one disparity step from the extremum (minimum or maximum) of the similarity measure, the accuracy can be refined with an approach similar to the subpixel extraction of matches that will be described in Section 9.11.3. Since the search space is one dimensional in the stereo matching, a parabola can be fitted through the three points around the extremum, and the extremum of the parabola can be extracted analytically.

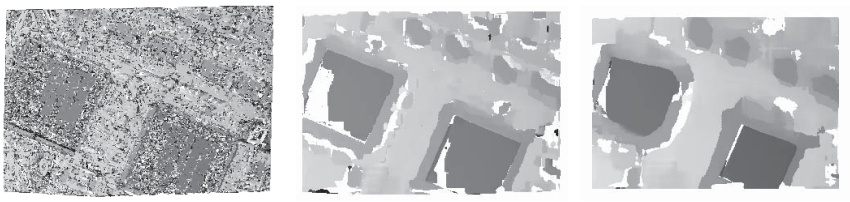


Figure 9.119 Distance reconstructed for the rectified image pair in Figure 9.117c,d with the NCC. (a) Window size 3 × 3. (b) Window size 17 × 17. (c) Window size 31 × 31. White areas correspond to the points that could not be matched because the similarity was too small.

Obviously, this will also result in a more accurate reconstruction of the depth of the points.

To perform stereo matching, we need to set one parameter: the size of the gray value windows n. It has a major influence on the result of the matching, as shown by the reconstructed depths in Figure 9.119. Here, window sizes of 3×3, 17×17, and 31×31 have been used with the NCC as the similarity measure. We can see that, if the window size is too small, many erroneous results will be found, despite the fact that a threshold of 0.4 has been used to select good matches. This happens because the matching requires a sufficiently distinctive texture within the window. If the window is too small, the texture is not distinctive enough, leading to erroneous matches. From Figure 9.119b, we see that the erroneous matches are mostly removed by the 17×17 window. However, because there is no texture in some parts of the image, especially in the lower left corners of the two large ICs, some parts of the image cannot be reconstructed. Note also that the areas of the leads around the large ICs are broader than in Figure 9.119a. This happens because the windows now straddle height discontinuities in a larger part of the image. Since the texture of the leads is more significant than the texture on the ICs, the matching finds the best matches at the depth of the leads. To fill the gaps in the reconstruction, we could try to increase the window size further, since this leads to more positions in which the windows have a significant texture. The result of setting the window size to 31×31 is shown in Figure 9.119c. Note that now most of the image can be reconstructed. Unfortunately, the lead area has broadened even more, which is undesirable.

From the above example, we can see that too small window sizes lead to many erroneous matches. In contrast, larger window sizes generally lead to fewer erroneous matches and a more complete reconstruction in areas with little texture. Furthermore, larger window sizes lead to a smoothing of the result, which may sometimes be desirable. However, larger window sizes lead to worse results at height discontinuities, which effectively limits the window sizes that can be used in practice.

Despite the fact that larger window sizes generally lead to fewer erroneous matches, they typically cannot be excluded completely based on the window size alone. Therefore, additional techniques are sometimes desirable to reduce the number of erroneous matches even further. An overview of methods to detect unreliable and erroneous matches is given in [86].

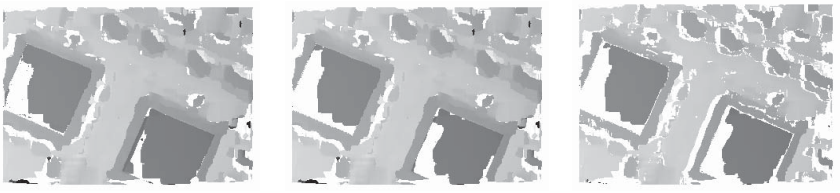


Figure 9.120 Increasing levels of robustness of the stereo matching. (a) Standard matching from the first to the second image with a window size of 17×17 using the NCC. (b) Result of requiring that the standard deviations of the windows is ≥ 5. (c) Result of performing the check where matching from the second to the first image results in the same disparity.

Erroneous matches occur mainly for two reasons: weak texture and occlusions. Erroneous matches caused by weak texture can sometimes be eliminated based on the matching score. However, in general it is best to exclude windows with weak texture *a priori* from the matching. Whether a window contains a weak texture can be decided on the basis of the output of a texture filter. Typically, the standard deviation of the gray values within the window is used as the texture filter. It has the advantage that it is computed in the NCC anyway, while it can be computed with just a few extra operations in the SAD and SSD. Therefore, to exclude windows with weak textures, we require that the standard deviation of the gray values within the window should be large.

The second reason why erroneous matches can occur are perspective occlusions, which, for example, occur at height discontinuities. To remove these errors, we can perform a consistency check that works as follows. First, we find the match from the first to the second image as usual. We then check whether matching the window around the match in the second image results in the same disparity, that is, finds the original point in the first image. If this is implemented naively, the runtime increases by a factor of 2. Fortunately, with a little extra bookkeeping the disparity consistency check can be performed with very few extra operations, since most of the required data have already been computed during the matching from the first to the second image.

Figure 9.120 shows the results of the different methods to increase robustness. For comparison, Figure 9.120a displays the result of the standard matching from the first to the second image with a window size of 17×17 using the NCC. The result of applying a texture threshold of 5 is shown in Figure 9.120b. It mainly removes untextured areas on the two large ICs. Figure 9.120c shows the result of applying the disparity consistency check. Note that it mainly removes matches in the areas where occlusions occur.

9.11 Template Matching

In the previous sections, we have discussed various techniques that can be combined to write algorithms to find objects in an image. While these techniques can in principle be used to find any kind of object, writing a robust recognition algorithm for a particular type of object can be quite cumbersome. Furthermore, if the

objects to be recognized change frequently, a new algorithm must be developed for each type of object. Therefore, a method to find any kind of object that can be configured simply by showing the system a prototype of the class of objects to be found would be extremely useful.

The above goal can be achieved by template matching. Here, we describe the object to be found by a template image. Conceptually, the template is found in the image by computing the similarity between the template and the image for all relevant poses of the template. If the similarity is high, an instance of the template has been found. Note that the term "similarity" is used here in a very general sense. We will see below that it can be defined in various ways, for example, based on the gray values of the template and the image, or based on the closeness of template edges to image edges.

Template matching can be used for several purposes. First of all, it can be used to perform completeness checks. Here, the goal is to detect the presence or absence of the object. Furthermore, template matching can be used for object discrimination, that is, to distinguish between different types of objects. In most cases, however, we already know which type of object is present in the image. In these cases, template matching is used to determine the pose of the object in the image. If the orientation of the objects can be fixed mechanically, the pose is described by a translation. In most applications, however, the orientation cannot be fixed completely, if at all. Therefore, often the orientation of the object, described by rotation, must also be determined. Hence, the complete pose of the object is described by a translation and a rotation. This type of transformation is called a rigid transformation. In some applications, additionally, the size of the objects in the image can change. This can happen if the distance of the objects to the camera cannot be kept fixed, or if the real size of the objects can change. Hence, a uniform scaling must be added to the pose in these applications. This type of pose (translation, rotation, and uniform scaling) is called a similarity transformation. If even the 3D orientation of the camera with respect to the objects can change and the objects to be recognized are planar, the pose is described by a projective transformation (see Section 9.3.2). Consequently, for the purposes of this chapter, we can regard the pose of the objects as a specialization of an affine or projective transformation.

In most applications, a single object is present in the search image. Therefore, the goal of template matching is to find this single instance. In some applications, more than one object is present in the image. If we know *a priori* how many objects are present, we want to find exactly this number of objects. If we do not have this knowledge, we typically must find all instances of the template in the image. In this mode, one of the goals is also to determine how many objects are present in the image.

9.11.1 Gray-Value-Based Template Matching

In this section, we will examine the simplest kind of template matching algorithms, which are based on the raw gray values in the template and the image. As mentioned previously, template matching is based on computing a similarity between the template and the image. Let us formalize this notion. For the

moment, we will assume that the object's pose is described by a translation. The template is specified by an image $t(r, c)$ and its corresponding ROI T. To perform the template matching, we can visualize moving the template over all positions in the image and computing a similarity measure $\mathbf{s}$ at each position. Hence, the similarity measure $\mathbf{s}$ is a function that takes the gray values in the template $t(r, c)$ and the gray values in the shifted ROI of the template at the current position in the image $f(r + u, c + v)$ and calculates a scalar value that measures the similarity based on the gray values within the respective ROI. With this approach, a similarity measure is returned for each point in the transformation space, which for translations can be regarded as an image. Hence, formally, we have

$$s(r, c) = \mathbf{s}\{t(u, v), f(r + u, c + v);\ (u, v) \in T\} \tag{9.148}$$

To make this abstract notation concrete, we will discuss several possible gray-value-based similarity measures [87].

The simplest similarity measures are to sum the absolute or squared gray value differences between the template and the image (SAD and SSD). They are given by

$$\mathrm{SAD}(r, c) = \frac{1}{n} \sum_{(u,v)\in T} |t(u, v) - f(r + u, c + v)| \tag{9.149}$$

and

$$\mathrm{SSD}(r, c) = \frac{1}{n} \sum_{(u,v)\in T} (t(u, v) - f(r + u, c + v))^2 \tag{9.150}$$

In both cases, n is the number of points in the template ROI, that is, $n = |T|$. Note that both similarity measures can be computed very efficiently with just two operations per pixel. These similarity measures have similar properties: if the template and the image are identical, they return a similarity measure of 0. If the image and template are not identical, a value greater than 0 is returned. As the dissimilarity increases, the value of the similarity measure increases. Hence, in this case the similarity measure should probably be better called a dissimilarity measure. To find instances of the template in the search image, we can threshold the similarity image $\mathrm{SAD}(r, c)$ with a certain upper threshold. This typically gives us a region that contains several adjacent pixels. To obtain a unique location for the template, we must select the local minima of the similarity image within each connected component of the thresholded region.

Figure 9.121 shows a typical application for template matching. Here, the goal is to locate the position of a fiducial mark on a PCB. The ROI used for the template is displayed in Figure 9.121a. The similarity computed with the sum of the SAD, given by equation (9.149), is shown in Figure 9.121b. For this example, the SAD was computed with the same image from which the template was generated. If the similarity is thresholded with a threshold of 20, only a region around the position of the fiducial mark is returned (Figure 9.121c). Within this region, the local minimum of the SAD must be computed (not shown) to obtain the position of the fiducial mark.

The SAD and SSD similarity measures work very well as long as the illumination can be kept constant. However, if the illumination can change, they both return

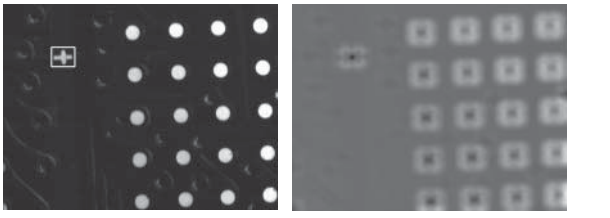


Figure 9.121 (a) Image of a PCB with a fiducial mark, which is used as the template (indicated by the white rectangle). (b) SAD computed with the template in (a) and the image in (a). (c) Result of thresholding (b) with a threshold of 20. Only a region around the fiducial is selected.

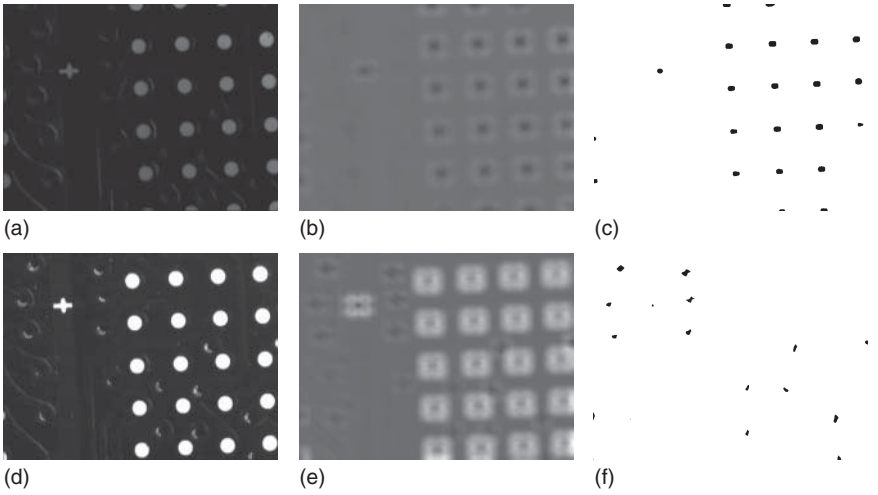


Figure 9.122 (a) Image of a PCB with a fiducial mark with a lower contrast. (b) SAD computed with the template in Figure 9.121a and the image in (a). (c) Result of thresholding (b) with a threshold of 35. (d) Image of a PCB with a fiducial mark with higher contrast. (e) SAD computed with the template of Figure 9.121a and the image in (d). (f) Result of thresholding (e) with a threshold of 35. In both cases, it is impossible to select a threshold that returns only the region of the fiducial mark.

larger values, even if the same object is contained in the image, because the gray values are no longer identical. This effect is illustrated in Figure 9.122. Here, a darker and brighter image of the fiducial mark are shown. They were obtained by adjusting the illumination intensity. The SAD computed with the template of Figure 9.121a is displayed in Figure 9.122b,e. The result of thresholding them with a threshold of 35 is shown in Figure 9.122c,f. The threshold was chosen such that the true fiducial mark is extracted in both cases. Note that, because of the contrast change, many extraneous instances of the template have been found.

As we can see from the above examples, the SAD and SSD similarity measures work well as long as the illumination can be kept constant. In applications where this cannot be ensured, a different kind of similarity measure is required. Ideally, this similarity measure should be invariant to all linear illumination changes

(see Section 9.2.1). A similarity measure that achieves this is the NCC, given by

$$\mathrm{NCC}(r,c) = \frac{1}{n} \sum_{(u,v)\in T} \frac{t(u,v) - m_t}{\sqrt{s_t^2}} \cdot \frac{f(r+u, c+v) - m_f(r,c)}{\sqrt{s_f^2(r,c)}} \tag{9.151}$$

Here, m_t is the mean gray value of the template and s_t^2 is the variance of the gray values, that is,

$$\begin{aligned} m_t &= \frac{1}{n} \sum_{(u,v)\in T} t(u,v) \\ s_t^2 &= \frac{1}{n} \sum_{(u,v)\in T} (t(u,v) - m_t)^2 \end{aligned} \tag{9.152}$$

Analogously, $m_f(r,c)$ and $s_f^2(r,c)$ are the mean value and variance in the image at a shifted position of the template ROI:

$$\begin{aligned} m_f(r,c) &= \frac{1}{n} \sum_{(u,v)\in T} f(r+u, c+v) \\ s_f^2(r,c) &= \frac{1}{n} \sum_{(u,v)\in T} (f(r+u, c+v) - m_f(r,c))^2 \end{aligned} \tag{9.153}$$

The NCC has a very intuitive interpretation. First of all, we should note that $-1 \leq \mathrm{NCC}(r,c) \leq 1$. Furthermore, if $\mathrm{NCC}(r,c) = \pm 1$, the image is a linearly scaled version of the template:

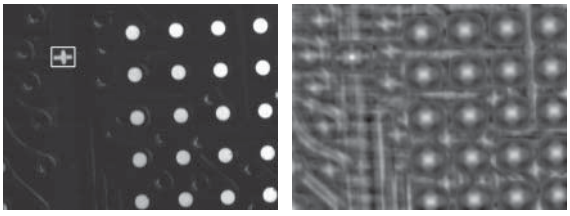
$$\mathrm{NCC}(r,c) = \pm 1 \quad \Leftrightarrow \quad f(r+u, c+v) = at(u,v) + b \tag{9.154}$$

For $\mathrm{NCC}(r,c) = 1$, we have $a > 0$, that is, the template and the image have the same polarity; while $\mathrm{NCC}(r,c) = -1$ implies that $a < 0$, that is, the polarity of the template and image are reversed. Note that this property of the NCC implies the desired invariance against linear illumination changes. The invariance is achieved by explicitly subtracting the mean gray values, which cancels additive changes, and by dividing by the standard deviation of the gray values, which cancels multiplicative changes.

While the template matches the image perfectly only if $\mathrm{NCC}(r,c) = \pm 1$, large absolute values of the NCC generally indicate that the template closely corresponds to the image part under examination, while values close to zero indicate that the template and image do not correspond well.

Figure 9.123 displays the results of computing the NCC for the template in Figure 9.121a (reproduced in Figure 9.123a). The NCC is shown in Figure 9.123b, while the result of thresholding the NCC with a threshold of 0.75 is shown in Figure 9.123c. This selects only a region around the fiducial mark. In this region, the local maximum of the NCC must be computed to derive the location of the fiducial mark (not shown). The results for the darker and brighter images in Figure 9.122 are not shown because they are virtually indistinguishable from the results in Figure 9.123b,c.

In the above discussion, we have assumed that the similarity measures must be evaluated completely for every translation. This is, in fact, unnecessary, since the result of calculating the similarity measure will be thresholded with a threshold



(a) (b) (c)

Figure 9.123 (a) Image of a PCB with a fiducial mark, which is used as the template (indicated by the white rectangle). This is the same image as in Figure 9.121a. (b) NCC computed with the template in (a) and the image in (a). (c) Result of thresholding (b) with a threshold of 0.75. The results for the darker and brighter images in Figure 9.122 are not shown because they are virtually indistinguishable from the results in (b) and (c).

t_s later on. For example, thresholding the SAD in equation (9.149) means that we require

$$\mathrm{SAD}(r,c) = \frac{1}{n}\sum_{i=1}^{n} |t(u_i,v_i) - f(r+u_i, c+v_i)| \le t_s \tag{9.155}$$

Here, we have explicitly numbered the points $(u,v) \in T$ by (u_i,v_i). We can multiply both sides by n to obtain

$$\mathrm{SAD}'(r,c) = \sum_{i=1}^{n} |t(u_i,v_i) - f(r+u_i, c+v_i)| \le nt_s \tag{9.156}$$

Suppose we have already evaluated the first j terms in the sum in equation (9.156). Let us call this partial result $\mathrm{SAD}'_j(r,c)$. Then, we have

$$\mathrm{SAD}'(r,c) = \mathrm{SAD}'_j(r,c) + \underbrace{\sum_{i=j+1}^{n} |t(u_i,v_i) - f(r+u_i, c+v_i)|}_{\ge 0} \le nt_s \tag{9.157}$$

Hence, we can stop the evaluation as soon as $\mathrm{SAD}'_j(r,c) > nt_s$, because we are certain that we can no longer achieve the threshold. If we are looking for a maximum number of m instances of the template, we can even adapt the threshold t_s based on the instance with the mth best similarity found so far. For example, if we are looking for a single instance with $t_s = 20$ and we have already found a candidate with $\mathrm{SAD}(r,c) = 10$, we can set $t_s = 10$ for the remaining poses that need to be checked. Of course, we need to calculate the local minima of $\mathrm{SAD}(r,c)$ and use the corresponding similarity values to ensure that this approach works correctly if more than one instance should be found.

For the NCC, there is no simple criterion to stop the evaluation of the terms. Of course, we can use the fact that the mean m_t and standard deviation $\sqrt{s_t^2}$ of the template can be computed once offline because they are identical for every translation of the template. The only other optimization we can make, analogous to the SAD, is that we can adapt the threshold t_s based on the matches we have found so far [88].

The above stopping criteria enable us to stop the evaluation of the similarity measure as soon as we are certain that the threshold can no longer be reached. Hence, they prune unwanted parts of the space of allowed poses. It is interesting to note that improvements for the pruning of the search space are still actively being investigated. For example, in [88] further optimizations for the pruning of the search space when using the NCC are discussed. In [89] and [90], strategies for pruning the search space when using SAD or SSD are discussed. They rely on transforming the image into a representation in which a large portion of the SAD and SSD can be computed with very few evaluations so that the above stopping criteria can be reached as soon as possible.

9.11.2 Matching Using Image Pyramids

The evaluation of the similarity measures on the entire image is very time consuming, even if the stopping criteria discussed above are used. If they are not used, the runtime complexity is $O(whn)$, where w and h are the width and height of the image and n is the number of points in the template. The stopping criteria typically result in a constant factor for the speed-up, but do not change the complexity. Therefore, a method to further speed up the search is necessary to be able to find the template in real time.

To derive a faster search strategy, we note that the runtime complexity of the template matching depends on the number of translations, that is, poses, that need to be checked. This is the $O(wh)$ part of the complexity. Furthermore, it depends on the number of points in the template. This is the $O(n)$ part. Therefore, to gain a speed-up, we can try to reduce the number of poses that need to be checked as well as the number of template points. Since the templates typically are large, one way to do this would be to take into account only every ith point of the image and template in order to obtain an approximate pose of the template, which could later be refined by a search with a finer step size around the approximate pose. This strategy is identical to subsampling the image and template. Since subsampling can cause aliasing effects (see Sections 9.2.4 and 9.3.3), this is not a very good strategy because we might miss instances of the template because of the aliasing effects. We have seen in Section 9.3.3 that we must smooth the image to avoid the aliasing effects. Furthermore, typically it is better to scale the image down multiple times by a factor of 2 than only once by a factor of $i > 2$. Scaling down the image (and template) multiple times by a factor of 2 creates a data structure that is called an image pyramid. Figure 9.124 displays why the name was chosen: we can visualize the smaller versions of the image stacked on top of each other. Since their width and height are halved in each step, they form a pyramid.

When constructing the pyramid, speed is essential. Therefore, the smoothing is performed by applying a 2×2 mean filter, that is, by averaging the gray value of each 2×2 block of pixels [91]. The smoothing could also be performed by a Gaussian filter [92]. Note, however, that, in order to avoid the introduction of unwanted shifts into the image pyramid, the Gaussian filter must have an even mask size. Therefore, the smallest mask size would be a 4×4 filter. Hence, using the Gaussian filter would incur a severe speed penalty in the construction of the image pyramid. Furthermore, the 2×2 mean filter does not have the frequency

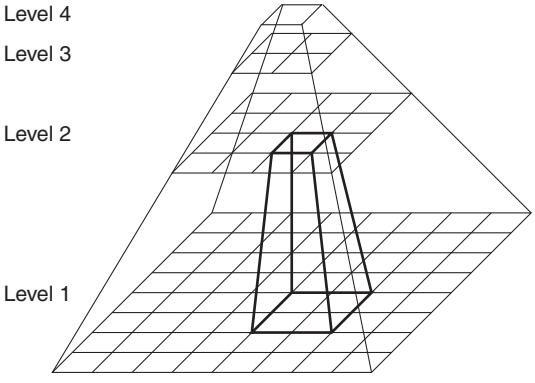


Figure 9.124 An image pyramid is constructed by successively halving the resolution of the image and combining 2×2 blocks of pixels in a higher resolution into a single pixel at the next lower resolution.

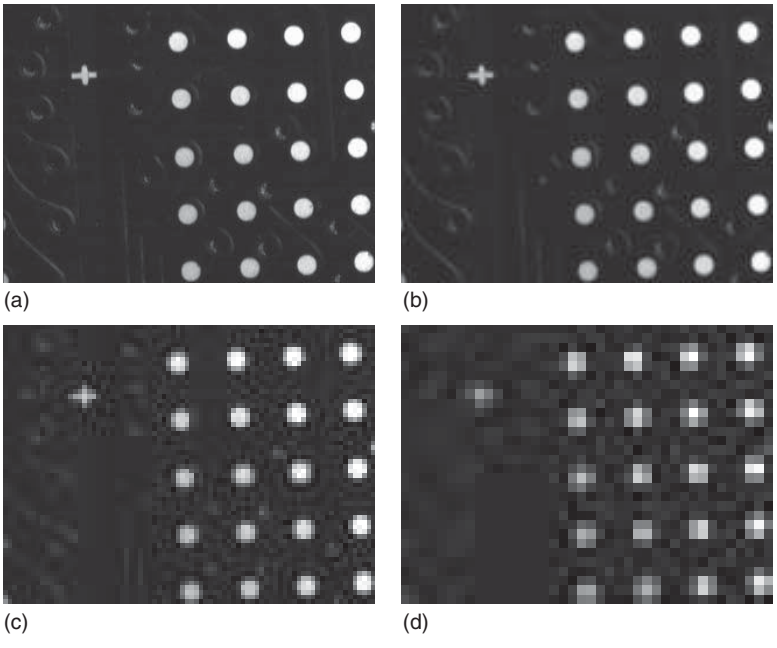


Figure 9.125 (a)–(d) Image pyramid levels 2–5 of the image in Figure 9.121a. Note that in level 5 the fiducial mark can no longer be discerned from the BGA pads.

response problems that the larger versions of the filter have (see Section 9.2.3). In fact, it drops off smoothly toward a zero response for the highest frequencies, like the Gaussian filter. Finally, it simulates the effects of a perfect camera with a fill factor of 100%. Therefore, the mean filter is the preferred filter for constructing image pyramids.

Figure 9.125 displays the image pyramid levels 2–5 of the image in Figure 9.121a. We can see that on levels 1–4 the fiducial mark can still be discerned from the BGA pads. This is no longer the case on level 5. Therefore, if we want to find an approximate location of the template, we can start the search on level 4.

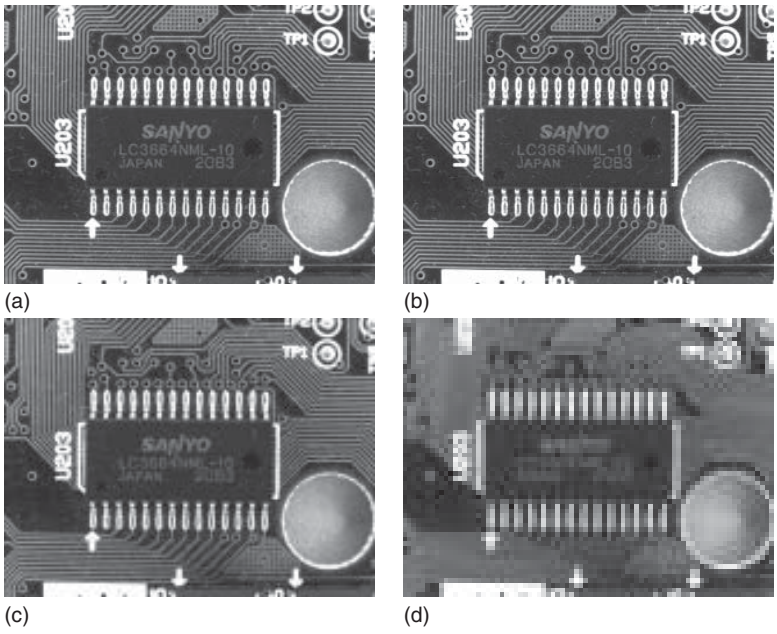


Figure 9.126 (a)–(d) Image pyramid levels 1–4 of an image of a PCB. Note that in level 4 all the tracks are merged into large components with identical gray values because of the smoothing that is performed when the pyramid is constructed.

The above example produces the expected result: the image is progressively smoothed and subsampled. The fiducial mark we are interested in can no longer be recognized as soon as the resolution becomes too low. Sometimes, however, creating an image pyramid can produce results that are unexpected at first glance. One example of this behavior is shown in Figure 9.126. Here, the image pyramid levels 1–4 of an image of a PCB are shown. We can see that on pyramid level 4 all the tracks are suddenly merged into large components with identical gray values. This happens because of the smoothing that is performed when the pyramid is constructed. Here, the neighboring thin lines start to interact with each other once the smoothing is large enough, that is, once we reach a pyramid level that is large enough. Hence, we can see that sometimes valuable information is destroyed by the construction of the image pyramid. If we were interested in matching, say, the corners of the tracks, we could only go as high as level 3 in the image pyramid.

Based on the image pyramids, we can define a hierarchical search strategy as follows. First, we calculate an image pyramid on the template and search image with an appropriate number of levels. How many levels can be used is mainly defined by the objects we are trying to find. On the highest pyramid level, the relevant structures of the object must still be discernible. Then, a complete matching is performed on the highest pyramid level. Here, of course, we take the appropriate stopping criterion into account. What gain does this give us? In each pyramid level, we reduce the number of image points and template points by a factor of 4. Hence, each pyramid level results in a speed-up of a factor of 16. Therefore, if we

perform the complete matching, for example, on level 4, we reduce the amount of computations by a factor of 4096.

All instances of the template that have been found on the highest pyramid level are then tracked to the lowest pyramid level. This is done by projecting the match to the next lower pyramid level, that is, by multiplying the coordinates of the found match by 2. Since there is an uncertainty in the location of the match, a search area is constructed around the match in the lower pyramid level, for example, a 5×5 rectangle. Then, the matching is performed within this small ROI, that is, the similarity measure is computed and thresholded, and the local extrema are extracted. This procedure is continued until the match is lost or tracked to the lowest level. Since the search spaces for the larger templates are very small, tracking the match down to the lowest level is very efficient.

While matching the template on the higher pyramid levels, we need to take the following effect into account: the gray values at the border of the object can change substantially on the highest pyramid level depending on where the object lies on the lowest pyramid level. This happens because a single pixel shift of the object translates to a subpixel shift on higher pyramid levels, which manifests itself as a change in the gray values on the higher pyramid levels. Therefore, on the higher pyramid levels we need to be more lenient with the matching threshold to ensure that all potential matches are being found. Hence, for the SAD and SSD similarity measures we need to use slightly higher thresholds, and for the NCC similarity measure we need to use slightly lower thresholds on the higher pyramid levels.

The hierarchical search is seen in Figure 9.127. The template is the fiducial mark shown in Figure 9.121a. The template is searched in the same image from which the template was created. As discussed previously, four pyramid levels are used in this case. The search starts on level 4. Here, the ROI is the entire image. The NCC and found matches on level 4 are displayed in Figure 9.127a. As we can see, 12 potential matches are initially found. They are tracked down to level 3 (Figure 9.127b). The ROIs created from the matches on level 4 are shown in white. For visualization purposes, the NCC is displayed for the entire image. In reality, it is, of course, only computed within the ROIs, that is, for a total of $12 \times 25 = 300$ translations. Note that at this level the true match turns out to be the only viable match. It is tracked down through levels 2 and 1 (Figure 9.127c,d). In both cases, only 25 translations need to be checked. Therefore, the match is found extremely efficiently. The zoomed part of the NCC in Figure 9.127b–d also shows that the pose of the match is progressively refined as the match is tracked down the pyramid.

9.11.3 Subpixel-Accurate Gray-Value-Based Matching

So far, we have located the pose of the template with pixel precision. This has been done by extracting the local minima (SAD, SSD) or maxima (NCC) of the similarity measure. To obtain the pose of the template with higher accuracy, the local minima or maxima can be extracted with subpixel precision. This can be done in a manner that is analogous to the method we have used in edge extraction (see Section 9.7.3): we simply fit a polynomial to the similarity measure in a 3×3

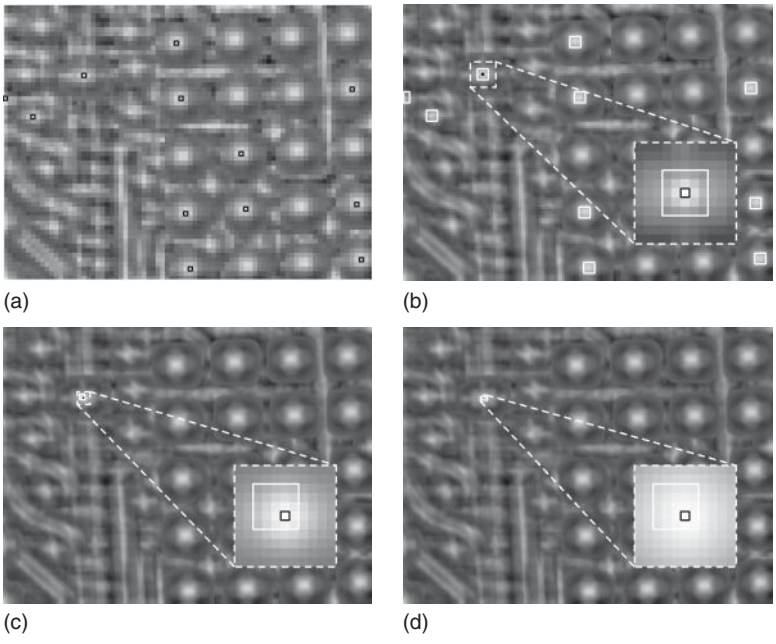


Figure 9.127 Hierarchical template matching using image pyramids. The template is the fiducial mark shown in Figure 9.121a. To provide a better visualization, the NCC is shown for the entire image on each pyramid level. In reality, however, it is only calculated within the appropriate ROI on each level, shown in white. The found matches are displayed in black. (a) On pyramid level 4, the matching is performed in the entire image. Here, 12 potential matches are found. (b) The matching is continued within the white ROIs on level 3. Only one viable match is found in the 12 ROIs. The similarity measure and ROI around the match are displayed zoomed in the lower right corner. (c,d) The match is tracked through pyramid levels 2 and 1.

neighborhood around the local minimum or maximum. Then, we extract the local minimum or maximum of the polynomial analytically. Another approach is to perform a least-squares matching of the gray values of the template and the image [93]. Since the least-squares matching of the gray values is not invariant to illumination changes, the illumination changes must be modeled explicitly, and their parameters must be determined in the least-squares fitting in order to achieve robustness to illumination changes [94].

9.11.4 Template Matching with Rotations and Scalings

Up to now, we have implicitly restricted the template matching to the case where the object must have the same orientation and scale in the template and the image, that is, the space of possible poses was assumed to be the space of translations. The similarity measures we have discussed previously can tolerate only small rotations and scalings of the object in the image. Therefore, if the object does not have the same orientation and size as the template, the object will not be found. If we want to be able to handle a larger class of transformations, for example, rigid or similarity transformations, we must modify the matching approach. For simplicity, we will only discuss rotations, but the method can be

extended to scalings and even more general classes of transformations in an analogous manner.

To find a rotated object, we can create the template in multiple orientations, that is, we discretize the search space of rotations in a manner that is analogous to the discretization of the translations that is imposed by the pixel grid [95]. Unlike for the translations, the discretization of the orientations of the template depends on the size of the template, since the similarity measures are less tolerant to small angle changes for large templates. For example, a typical value is to use an angle step size of 1° for templates with a radius of 100 pixels. Larger templates must use smaller angle steps, while smaller templates can use larger angle steps. To find the template, we simply match all rotations of the template with the image. Of course, this is done only on the highest pyramid level. To make the matching in the pyramid more efficient, we can also use the fact that the templates become smaller by a factor of 2 on each pyramid level. Consequently, the angle step size can be increased by a factor of 2 for each pyramid level. Hence, if an angle step size of 1° is used on the lowest pyramid level, a step size of 8° can be used on the fourth pyramid level.

While tracking potential matches through the pyramid, we also need to construct a small search space for the angles in the next lower pyramid level, analogous to the small search space that we already use for the translations. Once we have tracked the match to the lowest pyramid level, we typically want to refine the pose to an accuracy that is higher than the resolution of the search space we have used. In particular, if rotations are used, the pose should consist of a subpixel translation and an angle that is more accurate than the angle step size we have chosen. The techniques for subpixel-precise localization of the template described above can easily be extended for this purpose.

9.11.5 Robust Template Matching

The above template matching algorithms have served for many years as the methods of choice to find objects in machine vision applications. Over time, however, there has been an increasing demand to find objects in images even if they are occluded or disturbed in other ways so that parts of the object are missing. Furthermore, the objects should be found even if there are a large number of disturbances on the object itself. These disturbances are often referred to as clutter. Finally, objects should be found even if there are severe nonlinear illumination changes. The gray-value-based template matching algorithms we have discussed so far cannot handle these kinds of disturbances. Therefore, in the remainder of this section, we will discuss several approaches that have been designed to find objects in the presence of occlusion, clutter, and nonlinear illumination changes.

We have already discussed a feature that is robust to nonlinear illumination changes in Section 9.7: edges are not (or at least very little) affected by illumination changes. Therefore, they are frequently used in robust matching algorithms. The only problem when using edges is the selection of a suitable threshold to segment the edges. If the threshold is chosen too low, there will be many clutter edges in the image. If it is chosen too high, important edges of the object will be missing. This has the same effect as if parts of the object are occluded. Since the

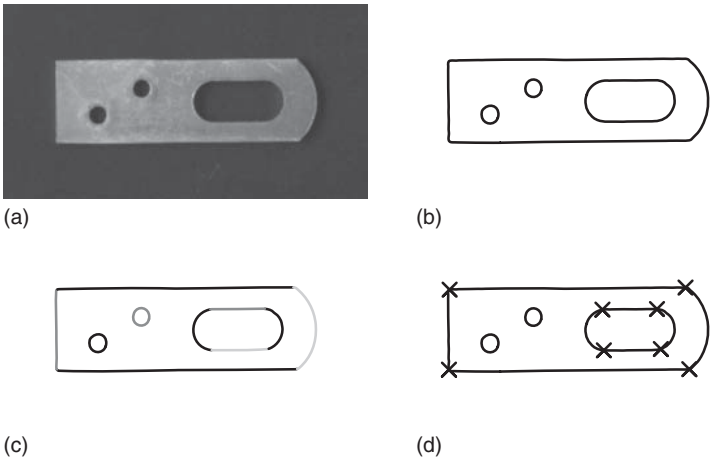


Figure 9.128 (a) Image of a model object. (b) Edges of (a). (c) Segmentation of (b) into lines and circles. (d) Salient points derived from the segmentation in (c).

threshold can never be chosen perfectly, this is another reason why the matching must be able to handle occlusions and clutter robustly.

To match objects using edges, several strategies exist. First of all, we can use the raw edge points, possibly augmented with some features per edge point, for the matching (see Figure 9.128b). Another strategy is to derive geometric primitives by segmenting the edges with the algorithms discussed in Section 9.8.4, and to match these to segmented geometric primitives in the image (see Figure 9.128c). Finally, based on a segmentation of the edges, we can derive salient points and match them to salient points in the image (see Figure 9.128d). It should be noted that the salient points can also be extracted directly from the image without extracting edges first [96, 97].

A large class of algorithms for edge matching is based on the distance of the edges in the template to the edges in the image. These algorithms typically use the raw edge points for the matching. One natural similarity measure based on this idea is to minimize the mean squared distance between the template edge points and the closest image edge points [98]. Hence, it appears that we must determine the closest image edge point for every template edge point, which would be extremely costly. Fortunately, since we are only interested in the distance to the closest edge point and not in which point is the closest point, this can be done in an efficient manner by calculating the distance transform of the complement of the segmented edges in the search image [98]. See Figure 9.129b,d for examples of the distance transform. A model is considered as being found if the mean distance of the template edge points to the image edge points is below a threshold. Of course, to obtain a unique location of the template, we must calculate the local minimum of this similarity measure. If we want to formalize this similarity measure, we can denote the edge points in the model by T and the distance transform of the complement of the segmented edge region in the search image by $d(r, c)$. Hence, the mean squared edge distance (SED) for the case of

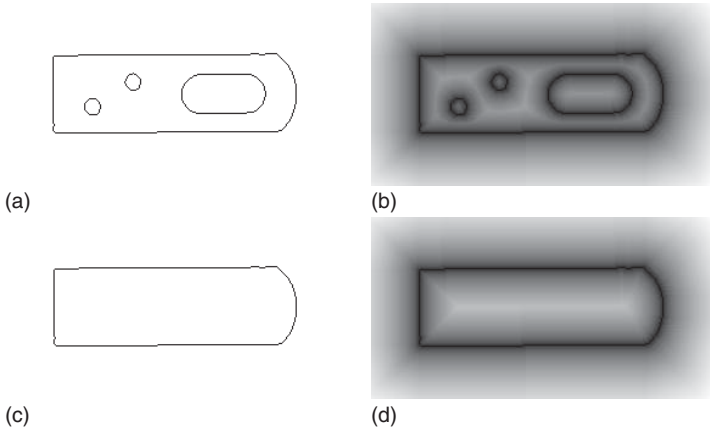


Figure 9.129 (a) Template edges. (b) Distance transform of the complement of (a). For better visualization, a square root look-up table (LUT) is used. (c) Search image with missing edges. (d) Distance transform of the complement of (c). If the template in (a) is matched to a search image in which the edges are complete and which possibly contains more edges than the template, the template will be found. If the template in (a) is matched to a search image in which template edges are missing, the template may not be found because a missing edge will have a large distance to the closest existing edge.

translations is given by

$$\mathrm{SED}(r,c) = \frac{1}{n} \sum_{(u,v)\in T} d(r+u, c+v)^2 \tag{9.158}$$

Note that this is very similar to the SSD similarity measure in equation (9.150) if we set $t(u,v) = 0$ there and use the distance transform image for $f(u,v)$. Consequently, the SED matching algorithm can be implemented very easily if we already have an implementation of the SSD matching algorithm. Of course, if we use the mean distance instead of the mean squared distance, we could use an existing implementation of the SAD matching, given by equation (9.149), for the edge matching.

We can now ask ourselves whether the SED fulfills the above criteria for robust matching. Since it is based on edges, it is robust to arbitrary illumination changes. Furthermore, since clutter, that is, extra edges in the search image, can only decrease the distance to the closest edge in the search image, it is robust to clutter. However, if edges are missing in the search image, the distance of the missing template edges to the closest image edges may become very large, and consequently the model may not be found. This is illustrated in Figure 9.129. Imagine what happens when the model in Figure 9.129a is searched in a search image in which some of the edges are missing (Figure 9.129c,d). Here, the missing edges will have a very large squared distance, which will increase the SED significantly. This will make it quite difficult to find the correct match.

Because of the above problems of the SED, edge matching algorithms using a different distance have been proposed. They are based on the Hausdorff distance of two point sets. Let us call the edge points in the template T and the edge points

in the image E. Then, the Hausdorff distance of the two point sets is given by

$$H(T,E) = \max(h(T,E), h(E,T)) \tag{9.159}$$

where

$$h(T,E) = \max_{t \in T} \min_{e \in E} \| t - e \| \tag{9.160}$$

and $h(E,T)$ is defined symmetrically. Hence, the Hausdorff distance consists of determining the maximum of two distances: the maximum distance of the template edges to the closest image edges, and the maximum distance of the image edges to the closest template edges [99]. It is immediately clear that, to achieve a low overall distance, every template edge point must be close to an image edge point, and vice versa. Therefore, the Hausdorff distance is robust to neither occlusion nor clutter. With a slight modification, however, we can achieve the desired robustness. The reason for the bad performance for occlusion and clutter is that in equation (9.160) the maximum distance of the template edges to the image edges is calculated. If we want to achieve robustness to occlusion, instead of computing the largest distance, we can compute a distance with a different rank, for example, the fth largest distance, where $f = 0$ denotes the largest distance. With this, the Hausdorff distance will be robust to 100 f/n % occlusion, where n is the number of edge points in the template. To make the Hausdorff distance robust to clutter, we can similarly modify $h(E,T)$ to use the rth largest distance. However, normally the model covers only a small part of the search image. Consequently, typically there are many more image edge points than template edge points, and hence r would have to be chosen very large to achieve the desired robustness against clutter. Therefore, $h(E,T)$ must be modified to be calculated only within a small ROI around the template. With this, the Hausdorff distance can be made robust to 100 r/m % clutter, where m is the number of edge points in the ROI around the template [99]. Like the SED, the Hausdorff distance can be computed based on distance transforms: one for the edge region in the image and one for each pose (excluding translations) of the template edge region. Therefore, we must compute either a very large number of distance transforms offline, which requires an enormous amount of memory, or the distance transforms of the model during the search, which requires a large amount of computation.

As we can see, one of the drawbacks of the Hausdorff distance is the enormous computational load that is required for the matching. In [99], several possibilities are discussed to reduce the computational load, including pruning regions of the search space that cannot contain the template. Furthermore, a hierarchical subdivision of the search space is proposed. This is similar to the effect that is achieved with image pyramids. However, the method in [99] only subdivides the search space, but does not scale the template or image. Therefore, it is still very slow. A Hausdorff distance matching method using image pyramids is proposed in [100].

The major drawback of the Hausdorff distance, however, is that, even with very moderate amounts of occlusion, many false instances of the template will be detected in the image [101]. To reduce the false detection rate, in [101] a modification of the Hausdorff distance is proposed that takes the orientation

of the edge pixels into account. Conceptually, the edge points are augmented with a third coordinate that represents the edge orientation. Then, the distance of these augmented 3D points and the corresponding augmented 3D image points is calculated as the modified Hausdorff distance. Unfortunately, this requires the calculation of a 3D distance transform, which makes the algorithm too expensive for machine vision applications. A further drawback of all approaches based on the Hausdorff distance is that it is quite difficult to obtain the pose with subpixel accuracy based on the interpolation of the similarity measure.

Another algorithm to find objects that is based on the edge pixels themselves is the generalized Hough transform proposed by Ballard [102]. The original Hough transform [103, 104] is a method that was designed to find straight lines in segmented edges. It was later extended to detect other shapes that can be described analytically, for example, circles or ellipses. The principle of the generalized Hough transform can be best explained by looking at a simple case. Let us try to find circles with a known radius in an edge image. Since circles are rotationally symmetric, we only need to consider translations in this case. If we want to find the circles as efficiently as possible, we can observe that, for circles that are brighter than the background, the gradient vector of the edge of the circle is perpendicular to the circle. This means that it points in the direction of the center of the circle. If the circle is darker than its background, the negative gradient vector points toward the center of the circle. Therefore, since we know the radius of the circle, we can theoretically determine the center of the circle from a single point on the circle. Unfortunately, we do not know which points lie on the circle (this is actually the task we would like to solve). However, we can detect the circle by observing that all points on the circle will have the property that, based on the gradient vector, we can construct the circle center. Therefore, we can accumulate evidence provided by all edge points in the image to determine the circle. This can be done as follows. Since we want to determine the circle center (i.e., the translation of the circle), we can set up an array that accumulates the evidence that a circle is present as a particular translation. We initialize this array with zeros. Then, we loop through all the edge points in the image and construct the potential circle center based on the edge position, the gradient direction, and the known circle radius. With this information, we increment the accumulator array at the potential circle center by one. After we have processed all the edge points, the accumulator array should contain a large amount of evidence, that is, a large number of votes, at the locations of the circle centers. We can then threshold the accumulator array and compute the local maxima to determine the circle centers in the image.

An example of this algorithm is shown in Figure 9.130. Suppose we want to locate the circle on top of the capacitor in Figure 9.130a and that we know that it has a radius of 39 pixels. The edges extracted with a Canny filter with $\sigma = 2$ and hysteresis thresholds of 80 and 20 are shown in Figure 9.130b. Furthermore, for every eighth edge point, the gradient vector is shown. Note that for the circle they all point toward the circle center. The accumulator array that is obtained with the algorithm described above is displayed in Figure 9.130c. Note that there is only one significant peak. In fact, most of the cells in the accumulator array have received so few votes that a square root LUT had to be used to visualize

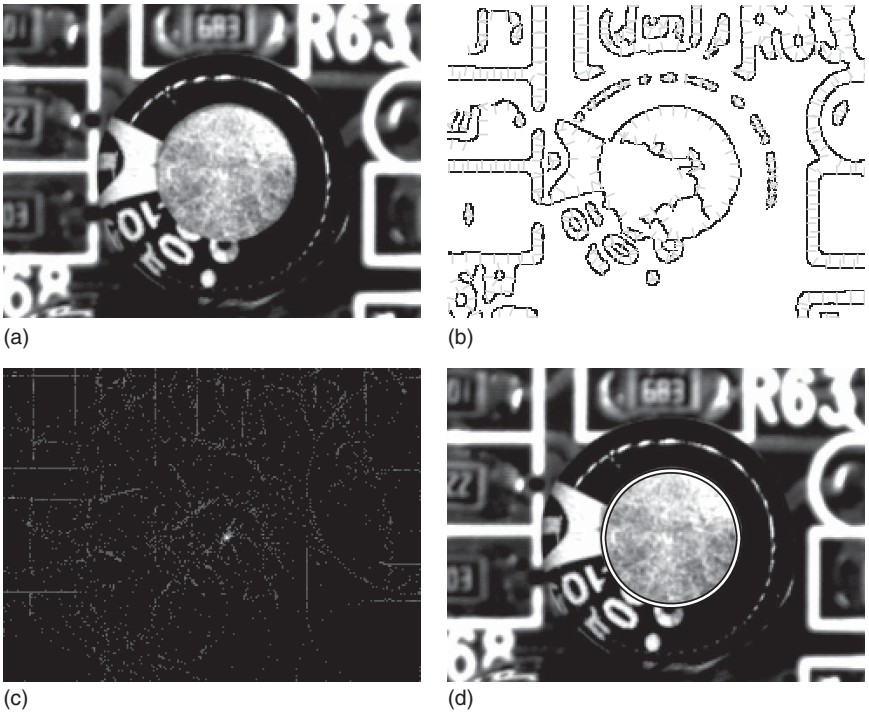


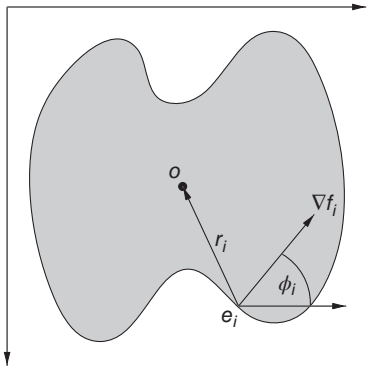
Figure 9.130 Using the Hough transform to detect a circle. (a) Image of a PCB showing a capacitor. (b) Detected edges. For every eighth edge point, the corresponding orientation is visualized by displaying the gradient vector. (c) Hough accumulator array obtained by performing the Hough transform using the edge points and orientations. A square root LUT is used to make the less populated regions of the accumulator space more visible. If a linear LUT were used, only the peak would be visible. (d) Circle detected by thresholding (c) and computing the local maxima.

whether there are any votes at all in the rest of the accumulator array. If the accumulator array is thresholded and the local maxima are calculated, the circle in Figure 9.130d is obtained.

From the above example, we can see that we can find circles in the image extremely efficiently. If we know the polarity of the circle, that is, whether it is brighter or darker than the background, we only need to perform a single increment of the accumulator array per edge point in the image. If we do not know the polarity of the edge, we need to perform two increments per edge point. Hence, the runtime is proportional to the number of edge points in the image and not to the size of the template, that is, the size of the circle. Ideally, we would like to find an algorithm that is equally efficient for arbitrary objects.

What can we learn from the above example? First of all, it is clear that, for arbitrary objects, the gradient direction does not necessarily point to a reference point of the object like it did for circles. Nevertheless, the gradient direction of the edge point provides a constraint where the reference point of the object can be, even for arbitrarily shaped objects. This is shown in Table 9.2. Suppose we have singled out the reference point o of the object. For the circle, the natural choice

Table 9.2 Principle of constructing the R-table in the generalized Hough transform (GHT). The R-table (on the right) is constructed based on the gradient angle ϕ_i of each edge point of the model object and the vector r_i from each edge point to the reference point o of the template.



j	ϕ_j	r_i
0	0	$\{r_i \mid \phi_i = 0\}$
1	$\Delta\phi$	$\{r_i \mid \phi_i = \Delta\phi\}$
2	$2\Delta\phi$	$\{r_i \mid \phi_i = 2\Delta\phi\}$
⋮	⋮	⋮

would be its center. For an arbitrary object, we can, for example, use the center of gravity of the edge points. Now consider an edge point e_i. We can see that the gradient vector ∇f_i and the vector r_i from e_i to o always enclose the same angle, no matter how the object is translated, rotated, and scaled. For simplicity, let us consider only translations for the moment. Then, if we find an edge point in the image with a certain gradient direction or gradient angle ϕ_i, we could calculate the possible location of the template with the vector r_i and increment the accumulator array accordingly. Note that for circles the gradient vector ∇f_i has the same direction as the vector r_i. For arbitrary shapes this no longer holds. From Table 9.2, we can also see that the edge direction does not necessarily uniquely constrain the reference point, since there may be multiple points on the edges of the template that have the same orientation. For circles, this is not the case. For example, in the lower left part of the object in Table 9.2, there is a second point that has the same gradient direction as the point labeled e_i, which has a different offset vector than the reference point. Therefore, in the search we have to increment all accumulator array elements that correspond to the edge points in the template with the same edge direction. Hence, during the search we must be able to quickly determine all the offset vectors that correspond to a given edge direction in the image. This can be achieved in a preprocessing step in the template generation that is performed offline. Basically, we construct a table, called the R-table, that is indexed by the gradient angle ϕ. Each table entry contains all the offset vectors r_i of the template edges that have the gradient angle ϕ. Since the table must be discrete to enable efficient indexing, the gradient angles are discretized with a certain step size $\Delta\phi$. The concept of the R-table is also shown in Table 9.2. With the R-table, it is very simple to find the offset vectors for incrementing the accumulator array in the search: we simply calculate the gradient angle in the image and use it as an index into the R-table. After the construction of the accumulator array, we threshold the array and calculate the local maxima to find the possible locations of the object. This approach can also be extended

easily to deal with rotated and scaled objects [102]. In real images, we also need to consider that there are uncertainties in the location of the edges in the image and in the edge orientations. We have already seen in equation (9.111) that the precision of the Canny edges depends on the signal-to-noise ratio. Using similar techniques, it can be shown that the precision of the edge angle ϕ for the Canny filter is given by $\sigma_\phi^2 = \sigma_n^2/(4\sigma^2 a^2)$. These values must be used in the online phase to determine a range of cells in the accumulator array that must be incremented to ensure that the cell corresponding to the true reference point is incremented.

The generalized Hough transform described previously is already quite efficient. On average, it increments a constant number of accumulator cells. Therefore, its runtime depends only on the number of edge points in the image. However, it is still not fast enough for machine vision applications because the accumulator space that must be searched to find the objects can quickly become very large, especially if rotations and scalings of the object are allowed. Furthermore, the accumulator uses an enormous amount of memory. Consider, for example, an object that should be found in a 640×480 image with an angle range of 360°, discretized in 1° steps. Let us suppose that two bytes are sufficient to store the accumulator array entries without overflow. Then, the accumulator array requires $640 \times 480 \times 360 \times 2 = 221\,184\,000$ bytes of memory, that is, 211 MB. This is unacceptably large for most applications. Furthermore, it means that initializing this array alone will require a significant amount of processing time. For this reason, a hierarchical generalized Hough transform is proposed in [105]. It uses image pyramids to speed up the search and to reduce the size of the accumulator array by using matches found on higher pyramid levels to constrain the search on lower pyramid levels. The interested reader is referred to [105] for details of the implementation. With this hierarchical generalized Hough transform, objects can be found in real time even under severe occlusions, clutter, and almost arbitrary illumination changes.

The algorithms we have discussed so far were based on matching edge points directly. Another class of algorithms is based on matching geometric primitives, for example, points, lines, and circles. These algorithms typically follow the hypothesize-and-test paradigm, that is, they hypothesize a match, typically from a small number of primitives, and then test whether the hypothetical match has enough evidence in the image.

The biggest challenge that this type of algorithm must solve is the exponential complexity of the correspondence problem. Let us, for the moment, suppose that we are only using one type of geometric primitive, for example, lines. Furthermore, let us suppose that all template primitives are visible in the image, so that potentially there is a subset of the primitives in the image that corresponds exactly to the primitives in the template. If the template consists of m primitives and there are n primitives in the image, there are $\binom{n}{m}$, that is, $O(n^m)$, potential correspondences between the template and image primitives. If the objects in the search image can be occluded, the number of potential matches is even larger, since we must allow that multiple primitives in the image can match a single primitive in the template, because a single primitive in the template may break up into several pieces, and that some primitives in the template are not present

in the search image. It is clear that, even for moderately large values of m and n, the cost of exhaustively checking all possible correspondences is prohibitive. Therefore, geometric constraints and strong heuristics must be used to perform the matching in an acceptable time.

One approach to perform the matching efficiently is called geometric hashing [106]. It was originally described for points as primitives, but can equally well be used with lines. Furthermore, the original description uses affine transformations as the set of allowable transformations. We will follow the original presentation and will note where modifications are necessary for other classes of transformations and lines as primitives. Geometric hashing is based on the observation that three points define an affine basis of the 2D plane. Thus, once we select three points e_{00}, e_{10}, and e_{01} in general position, that is, not collinear, we can represent every other point as a linear combination of these three points:

$$q = e_{00} + \alpha(e_{10} - e_{00}) + \beta(e_{01} - e_{00}) \tag{9.161}$$

The interesting property of this representation is that it is invariant to affine transformations, that is, (α, β) depend only on the three basis points (the basis triplet), but not on the affine transformation, that is, they are affine invariants. With this, the values (α, β) can be regarded as the affine coordinates of the point q. This property holds equally well for lines: three nonparallel lines that do not intersect in a single point can be used to define an affine basis. If we use a more restricted class of transformations, fewer points are sufficient to define a basis. For example, if we restrict the transformations to similarity transformations, two points are sufficient to define a basis. Note, however, that two lines are sufficient to determine only a rigid transformation.

The aim of geometric hashing is to reduce the amount of work that has to be performed to establish the correspondences between the template and image points. Therefore, it constructs a hash table that enables the algorithm to determine quickly the potential matches for the template. This hash table is constructed as follows. For every combination of three noncollinear points in the template, the affine coordinates (α, β) of the remaining $m - 3$ points of the template are calculated. The affine coordinates (α, β) serve as the index into the hash table. For every point, the index of the current basis triplet is stored in the hash table. If more than one template should be found, additionally the template index is stored; however, we will not consider this case further, so for our purposes only the index of the basis triplet is stored.

To find the template in the image, we randomly select three points in the image and construct the affine coordinates (α, β) of the remaining $n - 3$ points. We then use (α, β) as an index into the hash table. This returns us the index of the basis triplet. With this, we obtain a vote for the presence of a particular basis triplet in the image. If the randomly selected points do not correspond to a basis triplet of the template, the votes of all the points will not agree. However, if they correspond to a basis triplet of the template, many of the votes will agree and will indicate the index of the basis triplet. Therefore, if enough votes agree, we have a strong indication for the presence of the model. The presence of the model is then verified as described in the following. Since there is a certain probability that we have selected an inappropriate basis triplet in the image, the algorithm iterates

until it has reached a certain probability of having found the correct match. Here, we can make use of the fact that we only need to find one correct basis triplet to find the model. Therefore, if k of the m template points are present in the image, the probability of having selected at least one correct basis triplet in t trials is approximately

$$p = 1 - \left(1 - \left(\frac{k}{n}\right)^3\right)^t \tag{9.162}$$

If similarity transforms are used, only two points are necessary to determine the affine basis. Therefore, the inner exponent will change from 3 to 2 in this case. For example, if the ratio of visible template points to image points k/n is 0.2 and we want to find the template with a probability of 99% (i.e., $p = 0.99$), 574 trials are sufficient if affine transformations are used. For similarity transformations, 113 trials would suffice. Hence, geometric hashing can be quite efficient in finding the correct correspondences, depending on how many extra features are present in the image.

After a potential match has been obtained with the algorithm described above, it must be verified in the image. In [106], this is done by establishing point correspondences for the remaining template points based on the affine transformation given by the selected basis triplet. Based on these correspondences, an improved affine transformation is computed by a least-squares minimization over all corresponding points. This, in turn, is used to map all the edge points of the template, that is, not only the characteristic points that were used for the geometric hashing, to the pose of the template in the image. The transformed edges are compared to the image edges. If there is sufficient overlap between the template and image edges, the match is accepted and the corresponding points and edges are removed from the segmentation. Should more than one instance of the template be found, the entire process is repeated.

The algorithm described so far works well as long as the geometric primitives can be extracted with sufficient accuracy. If there are errors in the point coordinates, an erroneous affine transformation will result from the basis triplet. Therefore, all the affine coordinates (α, β) will contain errors, and hence the hashing in the online phase will access the wrong entry in the hash table. This is probably the largest drawback of the geometric hashing algorithm in practice. To circumvent this problem, the template points must be stored in multiple adjacent entries of the hash table. Which hash table entries must be used can in theory be derived through error propagation [106]. However, in practice, the accuracy of the geometric primitives is seldom known. Therefore, estimates have to be used, which must be well on the safe side for the algorithm not to miss any matches in the online phase. This, in turn, makes the algorithm slightly less efficient because more votes will have to be evaluated during the search.

The final class of algorithms we will discuss tries to match geometric primitives themselves to the image. Most of these algorithms use only line segments as the primitives [107–109]. One of the few exceptions to this rule is the approach in [110], which uses line segments and circular arcs. Furthermore, in 3D object recognition, sometimes line segments and elliptic arcs are used [111]. As we already discussed, exhaustively enumerating all potential correspondences

between the template and image primitives is prohibitively slow. Therefore, it is interesting to look at examples of different strategies that are employed to make the correspondence search tractable.

The approach in [107] segments the contours of the model object and the search image into line segments. Depending on the lighting conditions, the contours are obtained by thresholding or by edge detection. The 10 longest line segments in the template are singled out as privileged. Furthermore, the line segments in the model are ordered by adjacency as they trace the boundary of the model object. To generate a hypothesis, a privileged template line segment is matched to a line segment in the image. Since the approach is designed to handle similarity transforms, the angle, which is invariant under these transforms, to the preceding line segment in the image is compared with the angle to the preceding line segment in the template. If they are not close enough, the potential match is rejected. Furthermore, the length ratio of these two segments, which also is invariant, is used to check the validity of the hypothesis. The algorithm generates a certain number of hypotheses in this manner. These hypotheses are then verified by trying to match additional segments. The quality of the hypotheses, including the additionally matched segments, is then evaluated based on the ratio of the lengths of the matched segments to the length of the segments in the template. The matching is stopped once a high-quality match has been found or if enough hypotheses have been evaluated. Hence, we can see that the complexity is kept manageable by using privileged segments in conjunction with their neighboring segments.

In [109], a similar method is proposed. In contrast to [107], corners (combinations of two adjacent line segments of the boundary of the template that enclose a significant angle) are matched first. To generate a matching hypothesis, two corners must be matched to the image. Geometric constraints between the corners are used to reject false matches. The algorithm then attempts to extend the hypotheses with other segments in the image. The hypotheses are evaluated based on a dissimilarity criterion. If the dissimilarity is below a threshold, the match is accepted. Hence, the complexity of this approach is reduced by matching features that have distinctive geometric characteristics first.

The approach in [108] also generates matching hypotheses and tries to verify them in the image. Here, a tree of possible correspondences is generated and evaluated in a depth-first search. This search tree is called the interpretation tree. A node in the interpretation tree encodes a correspondence between a model line segment and an image line segment. Hence, the interpretation tree would exhaustively enumerate all correspondences, which would be prohibitively expensive. Therefore, the interpretation tree must be pruned as much as possible. To do this, the algorithm uses geometric constraints between the template line segments and the image line segments. Specifically, the distances and angles between pairs of line segments in the image and in the template must be consistent. This angle is checked by using normal vectors of the line segments that take the polarity of the edges into account. This consistency check prunes a large number of branches of the interpretation tree. However, since a large number of possible matchings still remain, a heuristic is used to explore the most promising hypotheses first. This is useful because the search is terminated once an acceptable match has been found. This early search termination is criticized in [112], and various strategies

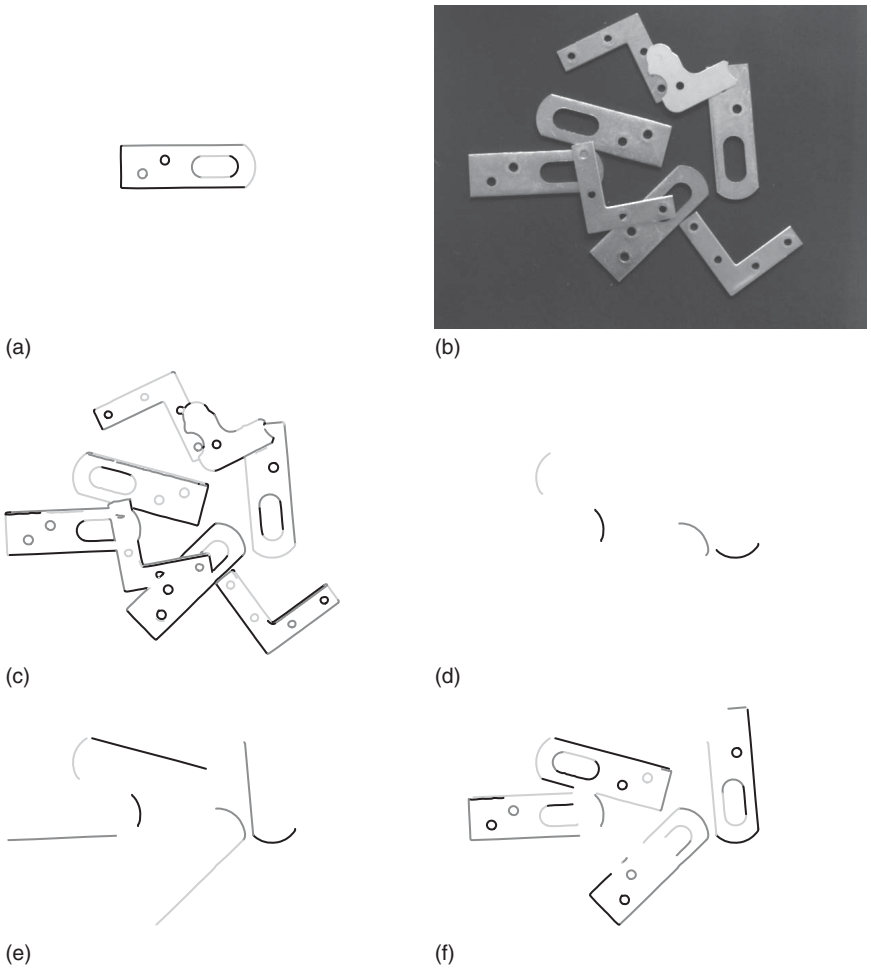


Figure 9.131 Example of matching an object in the image using geometric primitives. (a) The template consists of five line segments and five circular arcs. The model has been generated from the image in Figure 9.128a. (b) The search image contains four partially occluded instances of the template along with four clutter objects. (c) Edges extracted in (b) with a Canny filter with $\sigma = 1$ and split into line segments and circular arcs. (d) The matching in this case first tries to match the largest circular arc of the model and finds four hypotheses. (e) The hypotheses are extended with the lower of the long line segments in (a). These two primitives are sufficient to estimate a rigid transform that aligns the template with the features in the image. (f) The remaining primitives of the template are matched to the image. The resulting matched primitives are displayed.

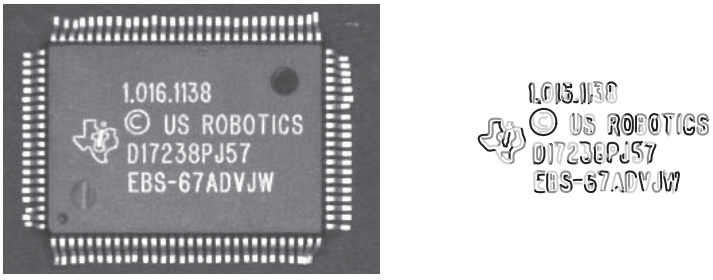
to speed up the search for all instances of the template in the image are discussed. The interested reader is referred to [112] for details.

To make the principles of the geometric matching algorithms clearer, let us examine a prototypical matching procedure on an example. The template to be found is shown in Figure 9.131a. It consists of five line segments and five circular

arcs. They were segmented automatically from the image in Figure 9.128a using a subpixel-precise Canny filter with $\sigma = 1$ and and by splitting the edge contours into line segments and circular arcs using the method described in Section 9.8.4. The template consists of the geometric parameters of these primitives as well as the segmented contours themselves. The image in which the template should be found is shown in Figure 9.131b. It contains four partially occluded instances of the model along with four clutter objects. The matching starts by extracting edges in the search image and by segmenting them into line segments and circular arcs (Figure 9.131c). Like for the template, the geometric parameters of the image primitives are calculated. The matching now determines possible matches for all of the primitives in the template. Of these, the largest circular arc is examined first because of a heuristic that rates moderately long circular arcs as more distinctive than even long line segments. The resulting matching hypotheses are shown in Figure 9.131d. Of course, the line segments could also have been examined first. Because in this case only rigid transformations are allowed, the matching of the circular arcs uses the radii of the circles as a matching constraint.

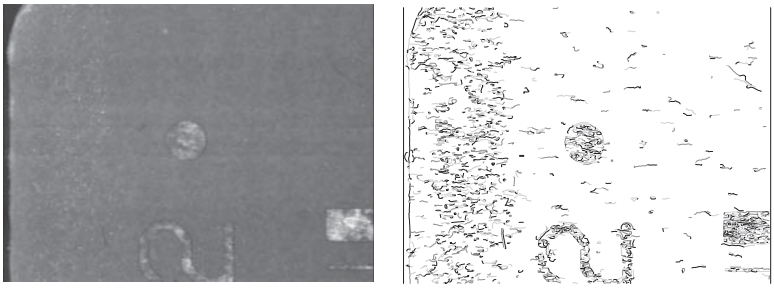
Since the matching should be robust to occlusions, the opening angle of the circular arcs is not used as a constraint. Because of this, the matched circles are not sufficient to determine a rigid transformation between the template and the image. Therefore, the algorithm tries to match an adjacent line segment (the long lower line segment in Figure 9.131a) to the image primitives while using the angle of intersection between the circle and the line as a geometric constraint. The resulting matches are shown in Figure 9.131e. With these hypotheses, it is possible to compute a rigid transformation that transforms the template to the image. Based on this, the remaining primitives can be matched to the image based on the distances of the image primitives and the transformed template primitives. The resulting matches are shown in Figure 9.131f. Note that, because of specular reflections, sometimes multiple parallel line segments are matched to a single line segment in the template. This could be fixed by taking the polarity of the edges into account. To obtain the rigid transformation between the template and the matches in the image as accurately as possible, a least-squares optimization of the distances between the edges in the template and the edges in the image can be used. An alternative is the minimal tolerance error zone optimization described in [110]. Note that the matching has already found the four correct instances of the template. For the algorithm, the search is not finished, however, since there might be more instances of the template in the image, especially instances for which the large circular arc is occluded more than in the leftmost instance in the image. Hence, the search is continued with other primitives as the first primitives to try. In this case, however, the search does not discover new viable matches.

After having discussed some of the approaches for robustly finding templates in an image, the question as to which of these algorithms should be used in practice naturally arises. We will say more on this topic in the following. From the above discussion, however, we can see that the effectiveness of a particular approach greatly depends on the shape of the template itself. Generally, the geometric matching algorithms have an advantage if the template and image contain only a few salient geometric primitives, like in the example in Figure 9.131. Here, the combinatorics of the geometric matching algorithms can work to their



(a) (b)

Figure 9.132 (a) Image of a template object that is not suitable for the geometric matching algorithms. Although the segmentation of the template into line segments and circular arcs in (b) only contains approximately 3 times as many edge points as the template in Figure 9.131, it contains 35 times as many geometric primitives, that is, 350.



(a) (b)

Figure 9.133 (a) A search image that is difficult for the geometric matching algorithms. Here, because of the poor contrast of the circular fiducial mark, the segmentation threshold must be chosen very low so that the relevant edges of the fiducial mark are selected. Because of this, the segmentation in (b) contains a very large number of primitives that must be examined in the search.

advantage. On the other hand, they work to their disadvantage if the template or search image contains a large number of geometric primitives.

A difficult model image is shown in Figure 9.132. The template contains fine structures that result in 350 geometric primitives, which are not particularly salient. Consequently, the search would have to examine an extremely large number of hypotheses that could be dismissed only after examining a large number of additional primitives. Note that the model contains 35 times as many primitives as the model in Figure 9.131, but only approximately 3 times as many edge points. Consequently, it could be easily found with pixel-based approaches like the generalized Hough transform.

A difficult search image is shown in Figure 9.133. Here, the goal is to find the circular fiducial mark. Since the contrast of the fiducial mark is very low, a small segmentation threshold must be used in the edge detection to find the relevant edges of the circle. This causes a very large number of edges and broken fragments that must be examined. Again, pixel-based algorithms will have little trouble with this image.

From the above examples, we can see that the pixel-based algorithms have the advantage that they can represent arbitrarily shaped templates without problems. Geometric matching algorithms, on the other hand, are restricted to relatively simple shapes that can be represented with a very small number of primitives. Therefore, in the remainder of this section, we will discuss a pixel-based robust template matching algorithm called shape-based matching [113–117] that works very well in practice [118, 119].

One of the drawbacks of all the algorithms that we have discussed so far is that they segment the edge image. This makes the object recognition algorithm invariant only against a narrow range of illumination changes. If the image contrast is lowered, progressively fewer edge points will be segmented, which has the same effect as progressively larger occlusion. Consequently, the object may not be found for low-contrast images. To overcome this problem, a similarity measure that is robust against occlusion, clutter, and nonlinear illumination changes must be used. This similarity measure can then be used in the pyramid-based recognition strategy described in Sections 9.11.2 and 9.11.4.

To define the similarity measure, we first define the model of an object as a set of points $p_i = (r_i, c_i)^\top$ and associated direction vectors $d_i = (t_i, u_i)^\top$, with $i = 1, \ldots, n$. The direction vectors can be generated by a number of different image processing operations. However, typically edge extraction (see Section 9.7.3) is used. The model is generated from an image of the object, where an arbitrary ROI specifies the part of the image in which the object is located. It is advantageous to specify the coordinates p_i relative to the center of gravity of the ROI of the model or to the center of gravity of the points of the model.

The image in which the model should be found can be transformed into a representation in which a direction vector $e_{r,c} = (v_{r,c}, w_{r,c})^\top$ is obtained for each image point (r, c). In the matching process, a transformed model must be compared with the image at a particular location. In the most general case considered here, the transformation is an arbitrary affine transformation (see Section 9.3.1). It is useful to separate the translation part of the affine transformation from the linear part. Therefore, a linearly transformed model is given by the points $p'_i = \mathtt{A} p_i$ and the accordingly transformed direction vectors $d'_i = (\mathtt{A}^{-1})^\top d_i$, where

$$\mathtt{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{9.163}$$

As discussed previously, the similarity measure by which the transformed model is compared with the image must be robust to occlusions, clutter, and illumination changes. One such measure is to sum the (unnormalized) dot product of the direction vectors of the transformed model and the image over all points of the model to compute a matching score at a particular point $q = (r, c)^\top$ of the image. That is, the similarity measure of the transformed model at the point q, which corresponds to the translation part of the affine transformation, is computed as follows:

$$s = \frac{1}{n} \sum_{i=1}^{n} d_i'^\top e_{q+p'} = \frac{1}{n} \sum_{i=1}^{n} t'_i \, v_{r+r'_i, c+c'_i} + u'_i \, w_{r+r'_i, c+c'_i} \tag{9.164}$$

If the model is generated by edge filtering and the image is preprocessed in the same manner, this similarity measure fulfills the requirements of robustness to occlusion and clutter. If parts of the object are missing in the image, there will be no edges at the corresponding positions of the model in the image, that is, the direction vectors will have a small length and hence contribute little to the sum. Likewise, if there are clutter edges in the image, there will either be no point in the model at the clutter position or it will have a small length, which means it will contribute little to the sum.

The similarity measure in equation (9.164) is not truly invariant against illumination changes however, because the length of the direction vectors depends on the brightness of the image if edge detection is used to extract the direction vectors. However, if a user specifies a threshold on the similarity measure to determine whether the model is present in the image, a similarity measure with a well-defined range of values is desirable. The following similarity measure achieves this goal:

$$s = \frac{1}{n}\sum_{i=1}^{n}\frac{\boldsymbol{d}_i'^{\top}\boldsymbol{e}_{q+p'}}{\|\boldsymbol{d}_i'\|\,\|\boldsymbol{e}_{q+p'}\|} = \frac{1}{n}\sum_{i=1}^{n}\frac{t_i'\,v_{r+r_i',c+c_i'} + u_i'\,w_{r+r_i',c+c_i'}}{\sqrt{t_i'^2 + u_i'^2}\,\sqrt{v_{r+r_i',c+c_i'}^2 + w_{r+r_i',c+c_i'}^2}} \tag{9.165}$$

Because of the normalization of the direction vectors, this similarity measure is additionally invariant to arbitrary illumination changes, since all vectors are scaled to a length of 1. What makes this measure robust against occlusion and clutter is the fact that, if a feature is missing, either in the model or in the image, noise will lead to random direction vectors, which, on average, will contribute nothing to the sum.

The similarity measure in equation (9.165) will return a high score if all the direction vectors of the model and the image align, that is, point in the same direction. If edges are used to generate the model and image vectors, this means that the model and image must have the same contrast direction for each edge. Sometimes it is desirable to be able to detect the object even if its contrast is reversed. This is achieved by

$$s = \left|\frac{1}{n}\sum_{i=1}^{n}\frac{\boldsymbol{d}_i'^{\top}\boldsymbol{e}_{q+p'}}{\|\boldsymbol{d}_i'\|\,\|\boldsymbol{e}_{q+p'}\|}\right| \tag{9.166}$$

In rare circumstances, it might be necessary to ignore even local contrast changes. In this case, the similarity measure can be modified as follows:

$$s = \frac{1}{n}\sum_{i=1}^{n}\frac{|\boldsymbol{d}_i'^{\top}\boldsymbol{e}_{q+p'}|}{\|\boldsymbol{d}_i'\|\,\|\boldsymbol{e}_{q+p'}\|} \tag{9.167}$$

The normalized similarity measures in equations (9.165)–(9.167) have the property that they return a number smaller than 1 as the score of a potential match. In all cases, a score of 1 indicates a perfect match between the model and the image. Furthermore, the score roughly corresponds to the portion of the model that is visible in the image. For example, if the object is 50% occluded, the score (on average) cannot exceed 0.5. This is a highly desirable property, because it gives

the user the means to select an intuitive threshold for when an object should be considered as recognized.

A desirable feature of the above similarity measures in equations (9.165)–(9.167) is that they do not need to be evaluated completely when object recognition is based on a user-defined threshold $s_{\min}$ for the similarity measure that a potential match must achieve. Let s_j denote the partial sum of the dot products up to the jth element of the model. For the match metric that uses the sum of the normalized dot products, this is

$$s_j = \frac{1}{n} \sum_{i=1}^{j} \frac{d_i'^{\top} e_{q+p'}}{\| d_i' \| \| e_{q+p'} \|} \tag{9.168}$$

Obviously, all the remaining terms of the sum are ≤ 1. Therefore, the partial score can never achieve the required score $s_{\min}$ if $s_j < s_{\min} - 1 + j/n$, and hence the evaluation of the sum can be discontinued after the jth element whenever this condition is fulfilled. This criterion speeds up the recognition process considerably.

As mentioned previously, to recognize the model, an image pyramid is constructed for the image in which the model should be found (see Section 9.11.2). For each level of the pyramid, the same filtering operation that was used to generate the model, for example, edge filtering, is applied to the image. This returns a direction vector for each image point. Note that the image is not segmented, that is, thresholding or other operations are not performed. This results in true robustness to illumination changes.

As discussed in Sections 9.11.2 and 9.11.4, to identify potential matches, an exhaustive search is performed for the top level of the pyramid, that is, all possible poses of the model are used on the top level of the image pyramid to compute the similarity measure via equations (9.165), (9.166), or (9.167). A potential match must have a score larger than $s_{\min}$, and the corresponding score must be a local maximum with respect to neighboring scores. The threshold $s_{\min}$ is used to speed up the search by terminating the evaluation of the similarity measure as early as possible. Therefore, this seemingly brute-force strategy actually becomes extremely efficient.

After the potential matches have been identified, they are tracked through the resolution hierarchy until they are found at the lowest level of the image pyramid. Once the object has been recognized on the lowest level of the image pyramid, its pose is extracted with a resolution better than the discretization of the search space with the approach described in Section 9.11.3.

While the pose obtained by the extrapolation algorithm is accurate enough for most applications, in some applications an even higher accuracy is desirable. This can be achieved through a least-squares adjustment of the pose parameters. To achieve a better accuracy than the extrapolation, it is necessary to extract the model points as well as the feature points in the image with subpixel accuracy. Then, the algorithm finds the closest image point for each model point, and then minimizes the sum of the squared distances of the image points to a line defined by their corresponding model point and the corresponding tangent to the model point, that is, the directions of the model points are taken to be correct and are

assumed to describe the direction of the object's border. If an edge detector is used, the direction vectors of the model are perpendicular to the object boundary, and hence the equation of a line through a model point tangent to the object boundary is given by

$$t_i(r - r_i) + u_i(c - c_i) = 0 \tag{9.169}$$

Let $q_i = (r'_i, c'_i)^\top$ denote the matched image points corresponding to the model points p_i. Then, the following function is minimized to refine the pose a:

$$d(a) = \sum_{i=1}^{n} (t_i(r'_i(a) - r_i) + u_i(c'_i(a) - c_i))^2 \rightarrow \min \tag{9.170}$$

The potential corresponding image points in the search image are obtained without thresholding by a non-maximum suppression and are extrapolated to subpixel accuracy. By this, a segmentation of the search image is avoided, which is important to preserve the invariance against arbitrary illumination changes. For each model point, the corresponding image point in the search image is chosen as the potential image point with the smallest Euclidian distance using the pose obtained by the extrapolation to transform the model to the search image. Since the point correspondences may change by the refined pose, an even higher accuracy can be gained by iterating the correspondence search and pose refinement. Typically, after three iterations the accuracy of the pose no longer improves.

Figure 9.134 shows six examples in which the shape-based matching algorithm finds the print on the IC shown in Figure 9.132. Note that the object is found despite severe occlusions and clutter.

Extensive tests with shape-based matching have been carried out in [118, 119]. The results show that shape-based matching provides extremely high recognition rates in the presence of severe occlusion and clutter as well as in the presence of nonlinear illumination changes. Furthermore, accuracies better than 1/30 of a pixel and better than 1/50 degree can be achieved.

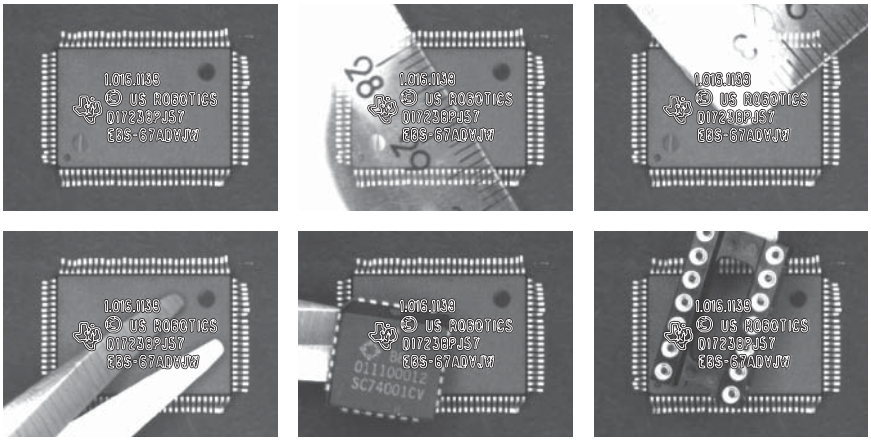


Figure 9.134 Six examples in which the shape-based matching algorithm finds an object (the print on the IC shown in Figure 9.132) despite severe occlusions and clutter.

The basic principle of the shape-based matching algorithm can be extended in various ways to handle larger classes of deformations and objects. For example, a method to recognize objects that consist of multiple parts that can move with respect to each other by rigid 2D transformations is described in [120–122]. An extension that is able to recognize planar objects under perspective transformations is proposed in [123, 124], while recognizing the 3D pose of planar objects is described in [124, 125]. A further extension that is able to handle smooth local deformations is presented in [124, 126]. Finally, the basic principle of the shape-based matching is also the foundation for a sophisticated algorithm that is able to recognize the 3D pose of objects from single images [127–130].

From the above discussion, we can see that the basic algorithms for implementing a robust template matching already are fairly complex. In reality, however, the complexity additionally resides in the time and effort that needs to be spent in making the algorithms very robust and fast. Additional complexity comes from the fact that, on one hand, templates with arbitrary ROIs should be possible to exclude undesired parts from the template, while, on the other, for speed reasons, it should also be possible to specify arbitrarily shaped ROIs for the search space in the search images. Consequently, these algorithms cannot be implemented easily. Therefore, wise machine vision users rely on standard software packages to provide this functionality rather than attempting to implement it themselves.

9.12 Optical Character Recognition

In quite a few applications, we face the challenge of having to read characters on the object we are inspecting. For example, traceability requirements often lead to the fact that the objects to be inspected are labeled with a serial number and that we have to read this serial number (see, e.g., Figures 9.22–9.24). In other applications, reading a serial number might be necessary to control the production flow.

OCR is the process of reading characters in images. It consists of two tasks: segmentation of the individual characters, and the classification of the segmented characters, that is, the assignment of a symbolic label to the segmented regions. We will examine these two tasks in this section.

9.12.1 Character Segmentation

The classification of the characters requires that we have segmented the text into individual characters, that is, each character must correspond to exactly one region.

To segment the characters, we can use all the methods that we have discussed in Section 9.4: thresholding with fixed and automatically selected thresholds, dynamic thresholding, and the extraction of connected components.

Furthermore, we might have to use the morphological operations of Section 9.6 to connect separate parts of the same character, for example, the dot of the character "i" to its main part (see Figure 9.43) or parts of the same character that are disconnected, for example, because of bad print quality. For characters

on difficult surfaces, for example, punched characters on a metal surface, gray value morphology may be necessary to segment the characters (see Figure 9.60).

Additionally, in some applications it may be necessary to perform a geometric transformation of the image to transform the characters into a standard position, typically such that the text is horizontal. This process is called image rectification. For example, the text may have to be rotated (see Figure 9.18), perspectively rectified (see Figure 9.20), or rectified with a polar transformation (see Figure 9.21).

Even though we have many segmentation strategies at our disposal, in some applications it may be difficult to segment the individual characters because the characters actually touch each other, either in reality or in the resolution at which we are looking at them in the image. Therefore, special methods to segment touching characters are sometimes required.

The simplest such strategy is to define a separate ROI for each character we are expecting in the image. This strategy sometimes can be used in industrial applications because the fonts typically have a fixed pitch (width) and we know *a priori* how many characters are present in the image, for example, if we are trying to read serial numbers with a fixed length. The main problem with this approach is that the character ROIs must enclose the individual characters we are trying to separate. This is difficult if the position of the text can vary in the image. If this is the case, we first need to determine the pose of the text in the image based on another strategy, for example, template matching to find a distinct feature in the vicinity of the text we are trying to read, and to use the pose of the text either to rectify the text to a standard position or to move the character ROIs to the appropriate position.

While defining separate ROIs for each character works well in some applications, it is not very flexible. A better method can be derived by realizing that the characters typically touch only with a small number of pixels. An example of this is shown in Figure 9.135a,b. To separate these characters, we can simply count the number of pixels per column in the segmented region. This is shown in Figure 9.135c. Since the touching part is only a narrow bridge between the characters, the number of pixels in the region of the touching part only has a very small number of pixels per column. In fact, we can simply segment the characters by splitting them vertically at the position of the minimum in Figure 9.135c. The result is shown in Figure 9.135d. Note that in Figure 9.135c the optimal splitting point is the global minimum of the number of pixels per column. However, in general, this may not be the case. For example, if the strokes between the vertical bars of the letter "m" were slightly thinner, the letter "m" might be split erroneously. Therefore, to make this algorithm more robust, it is typically necessary to define a search space for the splitting of the characters based on the expected width of the characters. For example, in this application the characters are approximately 20 pixels wide. Therefore, we could restrict the search space for the optimal splitting point to a range of ± 4 pixels (20% of the expected width) around the expected width of the characters. This simple splitting method works very well in practice. Further approaches for segmenting characters are discussed in [131].

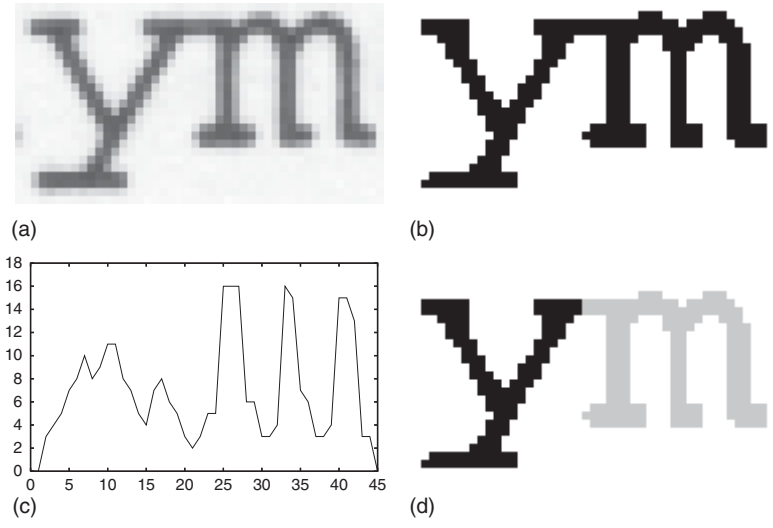


Figure 9.135 (a) An image of two touching characters. (b) Segmented region. Note that the characters are not separated. (c) Plot of the number of pixels in each column of (b). (d) The characters have been split at the minimum of (c) at position 21.

9.12.2 Feature Extraction

As mentioned previously, the reading of the characters corresponds to the classification of the regions, that is, the assignment of a class ω_i to a region. For the purposes of OCR, the classes ω_i can be thought of as the interpretation of the character, that is, the string that represents the character. For example, if an application must read serial numbers, the classes $\{\omega_1, \ldots, \omega_{10}\}$ are simply the strings $\{0, \ldots, 9\}$. If numbers and uppercase letters must be read, the classes are $\{\omega_1, \ldots, \omega_{36}\} = \{0, \ldots, 9, \mathrm{A}, \ldots, \mathrm{Z}\}$. Hence, classification can be thought of as a function f that maps to the set of classes $\Omega = \{\omega_i, i = 1, \ldots, m\}$. What is the input of this function? First of all, to make the above mapping well defined and easy to handle, we require that the number n of input values to the function f is constant. The input values to f are called features. Typically they are real numbers. With this, the function f that performs the classification can be regarded as a mapping $f : \mathbb{R}^n \mapsto \Omega$.

For OCR, the features that are used for the classification are features that we extract from the segmented characters. Any of the region features described in Section 9.5.1 and the gray value features described in Section 9.5.2 can be used as features. The main requirement is that the features enable us to discern the different character classes. Figure 9.136 illustrates this point. The input image is shown in Figure 9.136a. It contains examples of lowercase letters. Suppose that we want to classify the letters based on the region features anisometry and compactness. Figure 9.136b,c shows that the letters "c" and "o" as well as "i" and "j" can be distinguished easily based on these two features. In fact, they can be distinguished solely based on their compactness. As Figure 9.136d,e shows, however, these two features are not sufficient to distinguish between the classes "p" and "q" as well as "h" and "k."

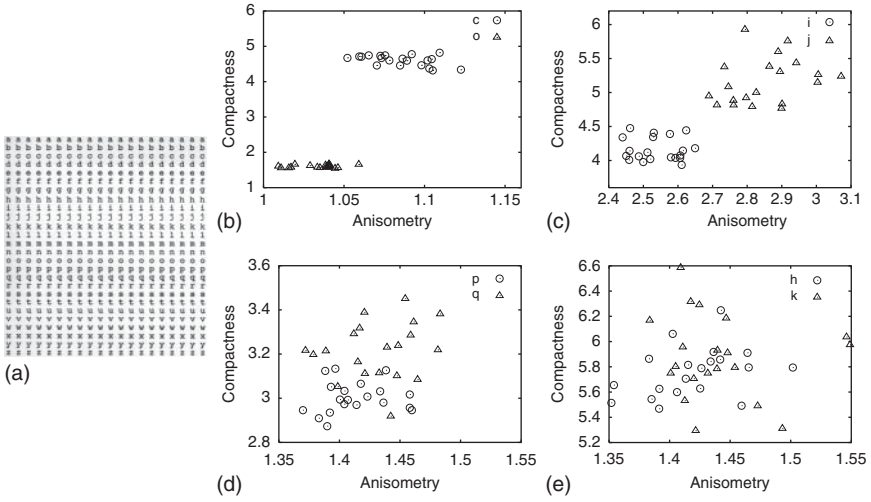


Figure 9.136 (a) Image with lowercase letters. (b)–(e) The features anisometry and compactness plotted for the letters "c" and "o" (b), "i" and "j" (c), "p" and "q" (d), and "h" and "k" (e). Note that the letters in (b) and (c) can be easily distinguished based on the selected features, while the letters in (d) and (e) cannot be distinguished.

From the above example, we can see that the features we use for the classification must be sufficiently powerful to enable us to classify all relevant classes correctly. The region and gray value features described in Sections 9.5.1 and 9.5.2, unfortunately, are often not powerful enough to achieve this. A set of features that is sufficiently powerful to distinguish all classes of characters is the gray values of the image themselves. Using the gray values directly, however, is not possible because the classifier requires a constant number of input features. To achieve this, we can use the smallest enclosing rectangle around the segmented character, enlarge it slightly to include a suitable amount of background of the character in the features (e.g., by one pixel in each direction), and then zoom the gray values within this rectangle to a standard size, for example, 8×10 pixels. While transforming the image, we must take care to use the interpolation and smoothing techniques discussed in Section 9.3.3. Note, however, that by zooming the image to a standard size based on the surrounding rectangle of the segmented character, we lose the ability to distinguish characters like "–" (minus sign) and "I" (upper case I in fonts without serifs). The distinction can easily be done based on a single additional feature: the ratio of the width and height of the smallest surrounding rectangle of the segmented character.

Unfortunately, the gray value features defined above are not invariant to illumination changes in the image. This makes the classification very difficult. To achieve invariance to illumination changes, two options exist. The first option is to perform a robust normalization of the gray values of the character, as described in Section 9.2.1, before the character is zoomed to the standard size. The second option is to convert the segmented character into a binary image before the character is zoomed to the standard size. Since the gray values generally contain more information, the first strategy is preferable in most cases. The second strategy can

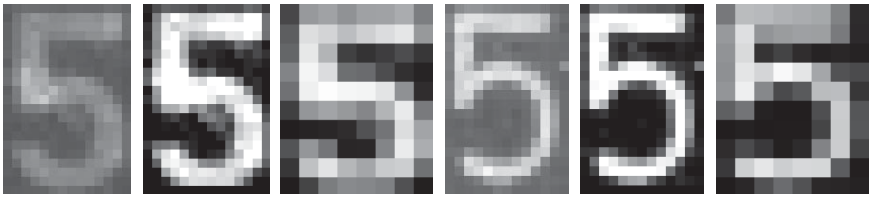


Figure 9.137 Gray value feature extraction for OCR. (a) Image of the letter "5" taken from the second row of characters in the image in Figure 9.23a. (b) Robust contrast normalization of (a). (c) Result of zooming (b) to a size of 8×10 pixels. (d) Image of the letter "5" taken from the second row of characters in the image in Figure 9.23b. (e) Robust contrast normalization of (d). (f) Result of zooming (e) to a size of 8×10 pixels.

be used whenever there is significant texture in the background of the segmented characters, which would make the classification more difficult.

Figure 9.137 displays two examples of the gray value feature extraction for OCR. Figure 9.137a,d displays two instances of the letter "5," taken from images with different contrast (Figure 9.23a,b). Note that the characters have different sizes (14×21 and 13×20 pixels, respectively). The result of the robust contrast normalization is shown in Figure 9.137b,e. Note that both characters now have full contrast. Finally, the result of zooming the characters to a size of 8×10 pixels is shown in Figure 9.137c,f. Note that this feature extraction automatically makes the OCR scale-invariant because of the zooming to a standard size.

To conclude the discussion about feature extraction, some words about the standard size are necessary. The discussion above has used the size 8×10. A large set of tests has shown that this size is a very good size to use for most industrial applications. If there are only a small number of classes to distinguish, for example, only numbers, it may be possible to use slightly smaller sizes. For some applications involving a larger number of classes, for example, numbers and uppercase and lowercase characters, a slightly larger size may be necessary (e.g., 10×12). On the other hand, using much larger sizes typically does not lead to better classification results because the features become progressively less robust against small segmentation errors if a large standard size is chosen. This happens because larger standard sizes imply that a segmentation error will lead to progressively larger position inaccuracies in the zoomed character as the standard size becomes larger. Therefore, it is best not to use a standard size that is much larger than the above recommendations. One exception to this rule is the recognition of an extremely large set of classes, for example, ideographic characters like the Japanese Kanji characters. Here, much larger standard sizes are necessary to distinguish the large number of different characters.

9.12.3 Classification

As we saw in the previous section, classification can be regarded as a mapping from the feature space to the set of possible classes, that is, $f : \mathbb{R}^n \mapsto \Omega$. We will now take a closer look at how the mapping can be constructed.

First of all, we can note that the feature vector x that serves as the input to the mapping can be regarded as a random variable because of the variations that the characters exhibit. In the application, we are observing this random feature vector for each character we are trying to classify. It can be shown that, to minimize the probability of erroneously classifying the feature vector, we should maximize the probability that the class ω_i occurs under the condition that we observe the feature vector x, that is, we should maximize $P(\omega_i|x)$ over all classes ω_i, for $i = 1, \ldots, m$ [132, 133]. The probability $P(\omega_i|x)$ is also called the *a posteriori* probability because of the above property that it describes the probability of class ω_i given that we have observed the feature vector x. This decision rule is called the Bayes decision rule. It yields the best classifier if all errors have the same weight, which is a reasonable assumption for OCR.

We now face the problem of how to determine the *a posteriori* probability. Using the Bayes theorem, $P(\omega_i|x)$ can be computed as follows:

$$P(\omega_i|x) = \frac{P(x|\omega_i)P(\omega_i)}{P(x)} \tag{9.171}$$

Hence, we can compute the *a posteriori* probability based on the *a priori* probability $P(x|\omega_i)$ that the feature vector x occurs given that the class of the feature vector is ω_i, the probability $P(\omega_i)$ that the class ω_i occurs, and the probability $P(x)$ that the feature vector x occurs. To simplify the calculations, we can note that the Bayes decision rule only needs to maximize $P(\omega_i|x)$ and that $P(x)$ is a constant if x is given. Therefore, the Bayes decision rule can be written as

$$x \in \omega_i \Leftrightarrow P(x|\omega_i)P(\omega_i) > P(x|\omega_j)P(\omega_j), \quad j = 1, \ldots, m, j \neq i \tag{9.172}$$

What do we gain by this transformation? As we will see in the following, the probabilities $P(x|\omega_i)$ and $P(\omega_i)$ can, in principle, be determined from training samples. This enables us to evaluate $P(\omega_i|x)$, and hence to classify the feature vector x. Before we examine this point in detail, however, let us assume that the probabilities in equation (9.172) are known. For example, let us assume that the feature space is one dimensional ($n = 1$) and that there are two classes ($m = 2$). Furthermore, let us assume that $P(\omega_1) = 0.3$, $P(\omega_2) = 0.7$, and that the features of the two classes have a normal distribution $N(\mu, \sigma)$ such that $P(x|\omega_1) \sim N(-3, 1.5)$ and $P(x|\omega_2) \sim N(3, 2)$. The corresponding likelihoods $P(x|\omega_i)P(\omega_i)$ are shown in Figure 9.138. Note that features to the left of $x \approx -0.7122$ are classified as belonging to ω_1, while features to the right are classified as belonging to ω_2. Hence, there is a dividing point $x \approx -0.7122$ that separates the classes from each other.

As a further example, consider a 2D feature space with three classes that have normal distributions with different means and covariances, as shown in Figure 9.139a. Again, there are three regions in the 2D feature space in which the respective class has the highest probability, as shown in Figure 9.139b. Note that now there are 1D curves that separate the regions in the feature space from each other.

As the above examples suggest, the Bayes decision rule partitions the feature space into mutually disjoint regions. This is obvious from the definition in equation (9.172): each region corresponds to the part of the feature space in which the class ω_i has the highest *a posteriori* probability. As also suggested by

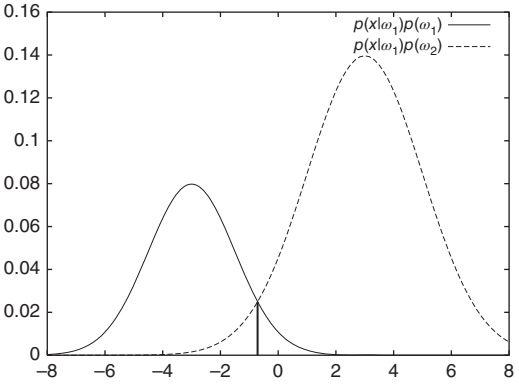


Figure 9.138 Example of a two-class classification problem in a 1D feature space in which $P(\omega_1) = 0.3$, $P(\omega_2) = 0.7$, $P(x|\omega_1) \sim N(-3,1.5)$, and $P(x|\omega_2) \sim N(3,2)$. Note that features to the left of $x \approx -0.7122$ are classified as belonging to ω_1, while features to the right are classified as belonging to ω_2.

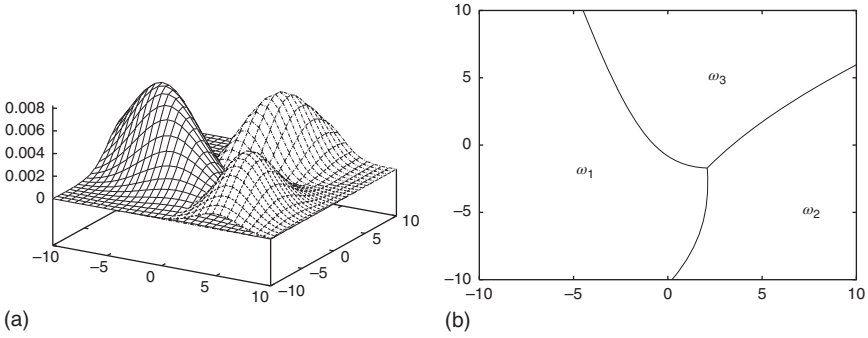


Figure 9.139 Example of a three-class classification problem in a 2D feature space in which the three classes have normal distributions with different means and covariances. (a) The *a posteriori* probabilities of the occurrence of the three classes. (b) Regions in the 2D feature space in which the respective class has the highest probability.

the above examples, the regions are separated by $(n-1)$-dimensional hypersurfaces (points for $n = 1$ and curves for $n = 2$, as in Figures 9.138 and 9.139). The hypersurfaces that separate the regions from each other are given by the points at which two classes are equally probable, that is, by $P(\omega_i|x) = P(\omega_j|x)$, for $i \neq j$.

Accordingly, we can identify two different types of classifiers. The first type of classifier tries to estimate the *a posteriori* probabilities, typically via the Bayes theorem, from the *a priori* probabilities of the different classes. In contrast, the second type of classifier tries to construct the separating hypersurfaces between the classes. In the following, we will examine representatives for both types of classifier.

All classifiers require a method with which the probabilities or separating hypersurfaces are determined. To do this, a training set is required. The training set is a set of sample feature vectors x_k with corresponding class labels ω_k. For OCR, the training set is a set of character samples from which the corresponding feature vectors can be calculated, along with the interpretation of the respective character. The training set should be representative of the data that can be expected in the application. In particular, for OCR the characters in the training set should contain the variations that will occur later, for example, different

character sets, stroke widths, noise, and so on. Since it is often difficult to obtain a training set with all variations, the image processing system must provide means to extend the training set over time with samples that are collected in the field, and should optionally also provide means to artificially add variations to the training set. Furthermore, to evaluate the classifier, in particular how well it has generalized the decision rule from the training samples, it is indispensable to have a test set that is independent of the training set. This test set is essential to determine the error rate that the classifier is likely to have in the application. Without the independent test set, no meaningful statement about the quality of the classifier can be made.

The classifiers that are based on estimating the probabilities, more precisely the probability densities, are called Bayes classifiers because they try to implement the Bayes decision rule via the probability densities. The first problem they have to solve is how to obtain the probabilities $P(\omega_i)$ of the occurrence of the class ω_i. There are two basic strategies for this purpose. The first strategy is to estimate $P(\omega_i)$ from the training set. Note that, for this, the training set must be representative not only in terms of the variations of the feature vectors but also in terms of the frequencies of the classes. Since this second requirement is often difficult to ensure, an alternative strategy for the estimation of $P(\omega_i)$ is to assume that each class is equally likely to occur, and hence to use $P(\omega_i) = 1/m$. Note that, in this case, the Bayes decision rule reduces to the classification according to the *a priori* probabilities since $P(\omega_i|x) \sim P(x|\omega_i)$ should now be maximized.

The remaining problem is how to estimate $P(x|\omega_i)$. In principle, this could be done by determining the histogram of the feature vectors of the training set in the feature space. To do so, we could subdivide each dimension of the feature space into b bins. Hence, the feature space would be divided into b^n bins in total. Each bin would count the number of occurrences of the feature vectors in the training set that lie within this bin. If the training set and b are large enough, the histogram would be a good approximation to the probability density $P(x|\omega_i)$. Unfortunately, this approach cannot be used in practice because of the so-called curse of dimensionality: the number of bins in the histogram is b^n, that is, its size grows exponentially with the dimension of the feature space. For example, if we use the 81 features described in the previous section and subdivide each dimension into a modest number of bins, for example, $b = 10$, the histogram would have 10^{81} bins, which is much too large to fit into any computer memory.

To obtain a classifier that can be used in practice, we can note that in the histogram approach the size of the bin is kept constant, while the number of samples in the bin varies. To get a different estimate for the probability of a feature vector, we can keep the number k of samples of class ω_i constant while varying the volume $v(x, \omega_i)$ of the region in space around the feature vector x that contains the k samples. Then, if there are t feature vectors in the training set, the probability of occurrence of the class ω_i is approximately given by

$$P(x|\omega_i) \approx \frac{k}{tv(x, \omega_i)} \tag{9.173}$$

Since the volume $v(x, \omega_i)$ depends on the k nearest neighbors of class ω_i, this type of density estimation is called the k nearest-neighbor density estimation.

In practice, this approach is often modified as follows. Instead of determining the k nearest neighbors of a particular class and computing the volume $v(x, \omega_i)$, the k nearest neighbors in the training set of any class are determined. The feature vector x is then assigned to the class that has the largest number of samples among the k nearest neighbors. This classifier is called the k nearest-neighbor classifier (kNN classifier). For $k = 1$, we obtain the nearest-neighbor classifier (NN classifier). It can be shown that the NN classifier has an error probability that is at most twice as large as the error probability of the optimal Bayes classifier that uses the correct probability densities [132], that is, $P_B \leq P_{\mathrm{NN}} \leq 2P_B$. Furthermore, if P_B is small, we have $P_{\mathrm{NN}} \approx 2P_B$ and $P_{\mathrm{3NN}} \approx P_B + 3P_B^2$. Hence, the 3NN classifier is almost as good as the optimal Bayes classifier. Nevertheless, kNN classifiers are difficult to use in practice because they require that the entire training set is stored with the classifier (which can easily contain several hundred thousands of samples). Furthermore, the search for the k nearest neighbors is time consuming even if optimized data structures are used to find exact [134] or approximate nearest neighbors [135, 136].

As we have seen from the above discussion, direct estimation of the probability density function is not practicable, either because of the curse of dimensionality for the histograms or because of efficiency considerations for the kNN classifier. To obtain an algorithm that can be used in practice, we can assume that $P(x|\omega_i)$ follows a certain distribution, for example, an n-dimensional normal distribution:

$$P(x|\omega_i) = \frac{1}{(2\pi)^{n/2}|\Sigma_i|^{1/2}} \exp\ (-\frac{1}{2}(x - \mu_i)^\top \Sigma_i^{-1}(x - \mu_i)) \tag{9.174}$$

With this, estimating the probability density function reduces to the estimation of the parameters of the probability density function. For the normal distribution, the parameters are the mean vector μ_i and the covariance matrix Σ_i of each class. Since the covariance matrix is symmetric, the normal distribution has $(n^2 + 3n)/2$ parameters in total. They can, for example, be estimated via the standard maximum likelihood estimators

$$\mu_i = \frac{1}{n_i}\sum_{j=1}^{n_i} x_{i,j} \quad \text{and} \quad \Sigma_i = \frac{1}{n_i - 1}\sum_{j=1}^{n_i}(x_{i,j} - \mu_i)(x_{i,j} - \mu_i)^\top \tag{9.175}$$

Here, n_i is the number of samples for class ω_i, while $x_{i,j}$ denotes the samples for class ω_i.

While the Bayes classifier based on the normal distribution can be quite powerful, often the assumption that the classes have a normal distribution does not hold in practice. In OCR applications, this happens frequently if characters in different fonts are to be recognized with the same classifier. One striking example of this is the shapes of the letters "a" and "g" in different fonts. For these letters, two basic shapes exist: a versus a and g versus g. It is clear that a single normal distribution is insufficient to capture these variations. In these cases, each font will typically lead to a different distribution. Hence, each class consists of a mixture of l_i different densities $P(x|\omega_i, k)$, each of which occurs with probability $P_{i,k}$:

$$P(x|\omega_i) = \sum_{k=1}^{l_i} P(x|\omega_i, k)P_{i,k} \tag{9.176}$$

Typically, the mixture densities $P(x|\omega_i, k)$ are assumed to be normally distributed. In this case, equation (9.176) is called a Gaussian mixture model. If we knew to which mixture density each sample belongs, we could easily estimate the parameters of the normal distribution with the above maximum likelihood estimators. Unfortunately, in real applications we typically do not have this knowledge, that is, we do not know k in equation (9.176). Hence, determining the parameters of the mixture model requires the estimation of not only the parameters of the mixture densities but also the mixture density labels k. This is a much harder problem, which can be solved by the expectation maximization algorithm (EM algorithm). The interested reader is referred to [132] for details. Another problem in the mixture model approach is that we need to specify how many mixture densities there are in the mixture model, that is, we need to specify l_i in equation (9.176). This is quite cumbersome to do manually. To solve this problem, algorithms that compute l_i automatically have been proposed. The interested reader is referred to [137, 138] for details.

Let us now turn our attention to classifiers that construct the separating hypersurfaces between the classes. Of all possible surfaces, the simplest ones are planes. Therefore, it is instructive to consider this special case first. Planes in the n-dimensional feature space are given by

$$w^\top x + b = 0 \tag{9.177}$$

Here, x is an n-dimensional vector that describes a point, while w is an n-dimensional vector that describes the normal vector to the plane. Note that this equation is linear. Because of this, classifiers based on separating hyperplanes are called linear classifiers.

Let us first consider the problem of classifying two classes with the plane. We can assign a feature vector to the first class ω_1 if x lies on one side of the plane, while we can assign it to the second class ω_2 if it lies on the other side of the plane. Mathematically, the test on which side of the plane a point lies is performed by looking at the sign of $w^\top x + b$. Without loss of generality, we can assign x to ω_1 if $w^\top x + b > 0$, while we assign x to ω_2 if $w^\top x + b < 0$.

For classification problems with more than two classes, we construct m separating planes (w_i, b_i) and use the following classification rule [132]:

$$x \in \omega_i \Leftrightarrow w_i^\top x + b_i > w_j^\top x + b_j, \quad j = 1, \ldots, m, j \neq i \tag{9.178}$$

Note that, in this case, the separating planes do not have the same meaning as in the two-class case, where the plane actually separates the data. The interpretation of equation (9.178) is that the plane is chosen such that the feature vectors of the correct class have the largest positive distance of all feature vectors from the plane.

Linear classifiers can also be regarded as neural networks, as shown in Figure 9.140 for the two-class and n-class cases. The neural network has processing units (neurons) that are visualized by circles. They first compute the linear combination of the feature vector x and the weights w: $w^\top x + b$. Then, a nonlinear activation function f is applied. For the two-class case, the activation function simply is $\text{sgn}(w^\top x + b)$, that is, the side of the hyperplane on which the feature vector lies. Hence, the output is mapped to its essence: -1 or $+1$.

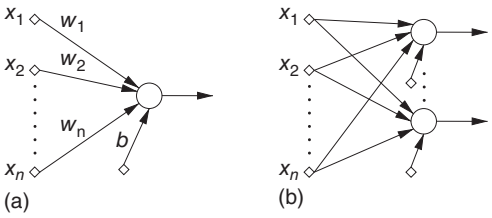


Figure 9.140 Architecture of a linear classifier expressed as a neural network (single-layer perceptron). (a) A two-class neural network. (b) An n-class neural network. In both cases, the neural network has a single layer of processing units that are visualized by circles. They first compute the linear combination of the feature vector and the weights. After this, a nonlinear activation function is computed, which maps the output to −1 or +1 (two-class neural network) or 0 or 1 (n-class neural network).

Note that this type of activation function essentially thresholds the input value. For the n-class case, the activation function f is typically chosen such that input values $\ll 0$ are mapped to 0, while input values ≥ 0 are mapped to 1. The goal in this approach is that a single processing unit returns the value 1, while all other units return the value 0. The index of the unit that returns 1 indicates the class of the feature vector. Note that the plane in equation (9.178) needs to be modified for this activation function to work since the plane is chosen such that the feature vectors have the largest distance from the plane. Therefore, $w_j^\top x + b_j$ is not necessarily <0 for all values that do not belong to the class. Nevertheless, the two definitions are equivalent. Note that, because the neural network has one layer of processing units, this type of neural network is also called a single-layer perceptron.

While linear classifiers are simple and easy to understand, they have very limited classification capabilities. By construction, the classes must be linearly separable, that is, separable by a hyperplane, for the classifier to produce the correct output. Unfortunately, this is rarely the case in practice. In fact, linear classifiers are unable to represent a simple function like the XOR function, as illustrated in Figure 9.141, because there is no line that can separate the two classes. Furthermore, for n-class linear classifiers, there is often no separating hyperplane for each class against all the other classes, although each pair of classes can be separated by a hyperplane. This happens, for example, if the samples of one class lie completely within the convex hull of all the other classes.

To get a classifier that is able to construct more general separating hypersurfaces, one approach is simply to add more layers to the neural network, as shown in Figure 9.142. Each layer first computes the linear combination of the feature

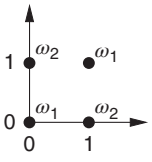


Figure 9.141 A linear classifier is not able to represent the XOR function because the two classes, corresponding to the two outputs of the XOR function, cannot be separated by a single line.

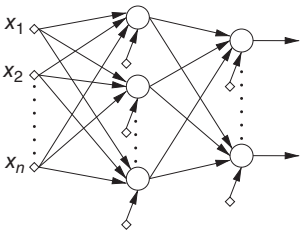


Figure 9.142 Architecture of a multilayer perceptron. The neural network has multiple layers of processing units that are visualized by circles. They compute the linear combination of the results of the previous layer and the network weights, and then pass the results through a nonlinear activation function.

vector or the results from the previous layer:

$$a_j^{(l)} = \sum_{i=1}^{n_l} w_{ji}^{(l)} x_i^{(l-1)} + b_j^{(l)} \tag{9.179}$$

Here, $x_i^{(0)}$ is simply the feature vector, while $x_i^{(l)}$, with $l \geq 1$, is the result vector of layer l. The coefficients $w_{ji}^{(l)}$ and $b_j^{(l)}$ are the weights of layer l. Then, the results are passed through a nonlinear activation function

$$x_j^{(l)} = f(a_j^{(l)}) \tag{9.180}$$

Let us assume for the moment that the activation function in each processing unit is the threshold function that is also used in the single-layer perceptron, that is, the function that maps input values <0 to 0 while mapping input values ≥ 0 to 1. Then, it can be seen that the first layer of processing units maps the feature space to the corners of the hypercube $\{0, 1\}^p$, where p is the number of processing units in the first layer. Hence, the feature space is subdivided by hyperplanes into half-spaces [132]. The second layer of processing units separates the points on the hypercube by hyperplanes. This corresponds to intersections of half-spaces, that is, convex polyhedra. Hence, the second layer is capable of constructing the boundaries of convex polyhedra as the separating hypersurfaces [132]. This is still not general enough however, since the separating hypersurfaces might need to be more complex than this. If a third layer is added, the network can compute unions of the convex polyhedra [132]. Hence, three layers are sufficient to approximate any separating hypersurface arbitrarily closely if the threshold function is used as the activation function.

In practice, the above threshold function is rarely used because it has a discontinuity at $x = 0$, which is very detrimental for the determination of the network weights by numerical optimization. Instead, often a sigmoid activation function is used, for example, the logistic function (see Figure 9.143a)

$$f(x) = \frac{1}{1 + e^x} \tag{9.181}$$

Similar to the hard threshold function, it maps its input to a value between 0 and 1. However, it is continuous and differentiable, which is a requirement for most numerical optimization algorithms. Another choice for the activation functions is to use the hyperbolic tangent function (see Figure 9.143b)

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{9.182}$$

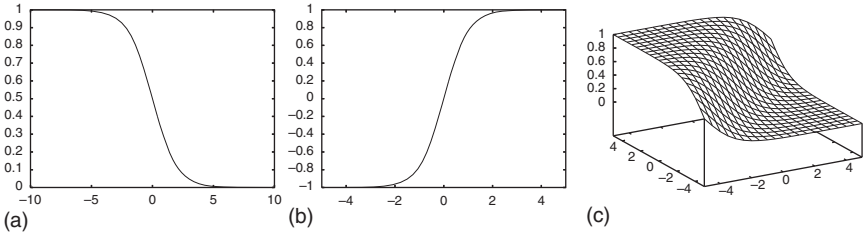


Figure 9.143 (a) Logistic activation function (9.181). (b) Hyperbolic tangent activation function (9.182). (c) Softmax activation function (9.183) for two classes.

in all layers except the output layer, in which the softmax activation function is used [139] (see Figure 9.143c)

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}} \tag{9.183}$$

The hyperbolic tangent function behaves similar to the logistic function. The major difference is that it maps its input to values between -1 and $+1$. There is experimental evidence that the hyperbolic tangent function leads to a faster training of the network [139]. In the output layer, the softmax function maps the input to the range [0,1], as desired. Furthermore, it ensures that the output values sum to 1, and hence have the same properties as a probability density [139]. With any of these choices of the activation function, it can be shown that two layers are sufficient to approximate any separating hypersurface and, in fact, any output function with values in [0,1], arbitrarily closely [139]. The only requirement for this is that there is a sufficient number of processing units in the first layer (the "hidden layer").

After having discussed the architecture of the multilayer perceptron, we can now examine how the network is trained. Training the network means that the weights $w_{ji}^{(l)}$ and $b_j^{(l)}$, with $l = 1, 2$, of the network must be determined. Let us denote the number of input features by n_i, the number of hidden units (first layer units) by n_h, and the number of output units (second layer units) by n_o. Note that n_i is the dimensionality of the feature vector, while n_o is the number of classes in the classifier. Hence, the only free parameter is the number n_h of units in the hidden layer. Note that there are $(n_i + 1)n_h + (n_h + 1)n_o$ weights in total. For example, if $n_i = 81$, $n_h = 40$, and $n_o = 10$, there are 3690 weights that must be determined. It is clear that this is a very complex problem and that we can hope to determine the weights uniquely only if the number of training samples is of the same order of magnitude as the number of weights.

As described previously, the training of the network is performed based on a training set, which consists of sample feature vectors x_k with corresponding class labels ω_k, for $k = 1, \ldots, l$. The sample feature vectors can be used as they are. The class labels, however, must be transformed into a representation that can be used in an optimization procedure. As described previously, ideally we would like to have the multilayer perceptron return a 1 in the output unit that corresponds to the class of the sample. Hence, a suitable representation of the classes is a target vector $y_k \in \{0, 1\}^{n_o}$, chosen such that there is a 1 at the index that corresponds

to the class of the sample and a 0 in all other positions. With this, we can train the network by minimizing, for example, the squared error of the outputs of the network on all the training samples [139]. In the notation of equation (9.180), we would like to minimize

$$\varepsilon = \sum_{k=1}^{l} \sum_{j=1}^{n_o} (x_j^{(2)} - y_{k,j})^2 \tag{9.184}$$

Here, $y_{k,j}$ is the jth element of the target vector y_k. Note that $x_j^{(2)}$ implicitly depends on all the weights $w_{ji}^{(l)}$ and $b_j^{(l)}$ of the network. Hence, minimization of equation (9.184) determines the optimum weights. To minimize equation (9.184), for a long time the back-propagation algorithm was used, which successively inputs each training sample into the network, determines the output error, and derives a correction term for the weights from the error. It can be shown that this procedure corresponds to the steepest descent minimization algorithm, which is well known to converge extremely slowly [18]. Currently, the minimization of equation (9.184) is typically being performed by sophisticated numerical minimization algorithms, such as the conjugate gradient algorithm [18, 139] or the scaled conjugate gradient algorithm [139].

Another approach to obtain a classifier that is able to construct arbitrary separating hypersurfaces is to transform the feature vector into a space of higher dimension, in which the features are linearly separable, and to use a linear classifier in the higher dimensional space. Classifiers of this type have been known for a long time as generalized linear classifiers [132]. One instance of this approach is the polynomial classifier, which transforms the feature vector by a polynomial of degree $\leq d$. For example, for $d = 2$ the transformation is

$$\Phi(x_1, \ldots, x_n) = (x_1, \ldots, x_n, x_1^2, \ldots, x_1 x_n, x_2^2 \cdots, x_2 x_n, \ldots, x_n^2) \tag{9.185}$$

The problem with this approach is again the curse of dimensionality: the dimension of the feature space grows exponentially with the degree d of the polynomial. In fact, there are $\binom{d+n-1}{d}$ monomials of degree $= d$ alone. Hence, the dimension of the transformed feature space is

$$n' = \sum_{i=1}^{d} \binom{i+n-1}{i} = \binom{d+n}{d} - 1 \tag{9.186}$$

For example, if $n = 81$ and $d = 5$, the dimension is 34 826 301. Even for $d = 2$, the dimension already is 3402. Hence, transforming the features into the larger feature space seems to be infeasible, at least from an efficiency point of view. Fortunately, however, there is an elegant way to perform the classification with generalized linear classifiers that avoids the curse of dimensionality. This is achieved by support vector machine (SVM) classifiers [140, 141].

Before we can take a look at how SVMs avoid the curse of dimensionality, we have to take a closer look at how the optimal separating hyperplane can be constructed. Let us consider the two-class case. As described in equation (9.177) for linear classifiers, the separating hyperplane is given by $w^\top x + b = 0$. As noted previously, the classification is performed based on the sign of $w^\top x + b$. Hence, the

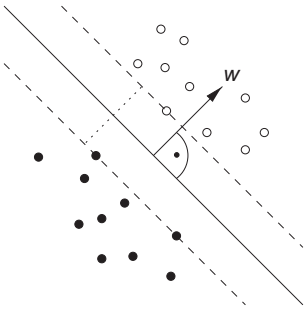


Figure 9.144 Optimal separating hyperplane between two classes. The samples of the two classes are represented by the filled and unfilled circles. The hyperplane is shown by the solid line. The margin is shown by the dotted line between the two dashed lines that show the hyperplanes in which samples are on the margin, that is, attain the minimum distance between the classes. The samples on the margin define the separating hyperplane. Since they "support" the margin hyperplanes, they are called support vectors.

classification function is

$$f(x) = \text{sgn}(w^\top x + b) \tag{9.187}$$

Let the training samples be denoted by x_i and their corresponding class labels by $y_i = \pm 1$. Then, a feature is classified correctly if $y_i(w^\top x_i + b) > 0$. However, this restriction is not sufficient to determine the hyperplane uniquely. This can be achieved by requiring that the margin between the two classes be as large as possible. The margin is defined as the closest distance of any training sample to the separating hyperplane.

Let us look at a small example of the optimal separating hyperplane, shown in Figure 9.144. Note that, if we want to maximize the margin (shown as a dotted line), there will be samples from both classes that attain the minimum distance to the separating hyperplane defined by the margin. These samples "support" the two hyperplanes that have the margin as the distance to the separating hyperplane (shown as the dashed lines). Hence, the samples are called the support vectors.

In fact, the optimal separating hyperplane is defined entirely by the support vectors, that is, a subset of the training samples: $w = \sum_{i=1}^{l} \alpha_i y_i x_i$, for $a_i \geq 0$, where $a_i > 0$ if and only if the training sample is a support vector [140]. With this, the classification function can be written as

$$f(x) = \text{sgn}(w^\top x + b) = \text{sgn}\left(\sum_{i=1}^{l} \alpha_i y_i x_i^\top x + b\right) \tag{9.188}$$

Hence, to determine the optimal hyperplane, the coefficients α_i of the support vectors must be determined. This can be achieved by solving the following quadratic programming problem [140]: maximize

$$\sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i^\top x_j \tag{9.189}$$

subject to

$$\alpha_i \geq 0, \quad i = 1, \ldots, l, \quad \text{and} \quad \sum_{i=1}^{l} \alpha_i y_i = 0 \tag{9.190}$$

Note that in both the classification function (9.188) and the optimization function (9.189), the feature vectors x, x_i, and x_j only are present in the dot product.

We now turn our attention back to the case in which the feature vector x is first transformed into a higher dimensional space by a function $\Phi(x)$, for example, by the polynomial function in equation (9.185). Then, the only change in the above discussion is that we substitute the feature vectors x, x_i, and x_j by their transformations $\Phi(x)$, $\Phi(x_i)$, and $\Phi(x_j)$. Hence, the dot products are simply computed in the higher dimensional space. The dot products become functions of two input feature vectors: $\Phi(x)^\top \Phi(x')$. These dot products of transformed feature vectors are called kernels in the SVM literature and are denoted by $k(x, x') = \Phi(x)^\top \Phi(x')$. Hence, the decision function becomes a function of the kernel $k(x, x')$:

$$f(x) = \operatorname{sgn}\left(\sum_{i=1}^{l} \alpha_i y_i k(x_i, x) + b\right) \tag{9.191}$$

The same happens with the optimization function equation (9.189).

So far, it seems that we do not gain anything from the kernel because we still have to transform the data into a feature space of a prohibitively large dimension. The ingenious trick of the SVM classification is that, for a large class of kernels, the kernel can be evaluated efficiently without explicitly transforming the features into the higher dimensional space, thus making the evaluation of the classification function (9.191) feasible. For example, if we transform the features by a polynomial of degree d, it can be easily shown that

$$k(x, x') = (x^\top x')^d \tag{9.192}$$

Hence, the kernel can be evaluated solely based on the input features without going to the higher dimensional space. This kernel is called a homogeneous polynomial kernel. As another example, the transformation by a polynomial of degree $\leq d$ can simply be evaluated as

$$k(x, x') = (x^\top x' + 1)^d \tag{9.193}$$

This kernel is called an inhomogeneous polynomial kernel. Further examples of possible kernels include the Gaussian radial basis function kernel

$$k(x, x') = \exp\left(-\frac{\| x - x' \|^2}{2\sigma^2}\right) \tag{9.194}$$

and the sigmoid kernel

$$k(x, x') = \tanh(\kappa x^\top x' + \vartheta) \tag{9.195}$$

Note that this is the same function that is also used in the hidden layer of the multilayer perceptron. With any of the above four kernels, SVMs can approximate any separating hypersurface arbitrarily closely.

Note that the above training algorithm that determines the support vectors still assumes that the classes can be separated by a hyperplane in the higher dimensional transformed feature space. This may not always be achievable. Fortunately, the training algorithm can be extended to handle overlapping classes. The reader is referred to [140] for details.

By its nature, the SVM classification can handle only two-class problems. To extend the SVM to multiclass problems, two basic approaches are possible.

The first strategy is to perform a pairwise classification of the feature vector against all pairs of classes and to use the class that obtains the most votes, that is, is selected most often as the result of the pairwise classification. Note that this implies that $m(m-1)/2$ classifications have to be performed if there are m classes. The second strategy is to perform m classifications of one class against the union of the rest of the classes. From an efficiency point of view, the second strategy may be preferable since it depends linearly on the number of classes. Note, however, that in the second strategy, typically there will be a larger number of support vectors than in the pairwise classification. Since the runtime depends linearly on the number of support vectors, this number must grow less than quadratically for the second strategy to be faster.

After having discussed the different types of classifiers, the natural question to ask is which of the classifiers should be used in practice. First of all, it must be noted that the quality of the classification results of all the classifiers depends to a large degree on the size and quality of the training set. Therefore, to construct a good classifier, the training set should be as large and as representative as possible.

If the main criterion for comparing classifiers is the classification accuracy, that is, the error rate on an independent test set, classifiers that construct the separating hypersurfaces, that is, the multilayer perceptron or the SVM, should be preferred. Of these two, the SVM is portrayed to have a slight advantage [140]. This advantage, however, is achieved by building certain invariances into the classifier that do not generalize to other classification tasks apart from OCR and a particular set of gray value features. The invariances built into the classifier in [140] are translations of the character by one pixel, rotations, and variations of the stroke width of the character. With the features in Section 9.12.2, the translation invariance is automatically achieved. The remaining invariances could also be achieved by extending the training set by systematically modified training samples. Therefore, neither the multilayer perceptron nor the SVM has a definite advantage in terms of classification accuracy.

Another criterion is the training speed. Here, SVMs have some advantage over the multilayer perceptron because the training times are often shorter. Therefore, if the speed in the training phase is important, SVMs currently should be preferred. Unfortunately, the picture is reversed in the online phase when unknown features must be classified. As noted previously, the classification time for the SVMs depends linearly on the number of support vectors, which usually is a substantial fraction of the training samples (typically, between 10% and 40%). Hence, if the training set consists of several hundred thousands of samples, tens of thousands of support vectors must be evaluated with the kernel in the online phase. In contrast, the classification time of the multilayer perceptron depends only on the topology of the net, that is, the number of processing units per layer. Therefore, if the speed in the online classification phase is important, the multilayer perceptron currently should be preferred.

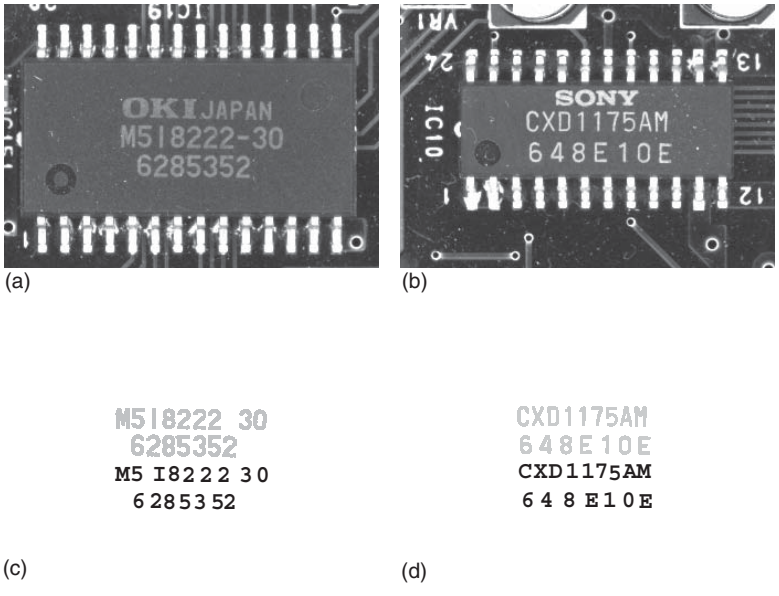
A final criterion is whether the classifier provides a simple means to decide whether the feature vector to be classified should be rejected because it does not belong to any of the trained classes. For example, if only digits have been trained, but the segmented character is the letter "M," it is important in some applications to be able to reject the character as being no digit. Another example is an erroneous segmentation, that is, a segmentation of an image part that corresponds to no character at all. Here, classifiers that construct the separating

Figure 9.145 (a) Samples in a 2D feature space for three classes, which are visualized by three gray levels. (b) Likelihood for class 1 determined in a square region of the feature space with a multilayer perceptron with five hidden units. (c) Likelihood for class 2. (d) Likelihood for class 3. Note that all classes have a very high likelihood for feature vectors that are far away from the training samples of each class. For example, class 3 has a very high likelihood in the lower corners of the displayed portion of the feature space. This high likelihood continues to infinity because of the architecture of the multilayer perceptron.

hypersurfaces provide no means to tell whether the feature vector is close to a class in some sense, since the only criterion is on which side of the separating hypersurface the feature lies. For the SVMs, this behavior is obvious from the architecture of the classifier. For the multilayer perceptron, this behavior is not immediately obvious since, as we have noted, the softmax activation function (9.183) has the same properties as a probability density. Hence, one could assume that the output reflects the likelihood of each class, and can thus be used to threshold unlikely classes. In practice, however, the likelihood is very close to 1 or 0 everywhere, except in areas in which the classes overlap in the training samples, as shown by the example in Figure 9.145. Note that all classes have a very high likelihood for feature vectors that are far away from the training samples of each class.

As can be seen from the above discussion, the behaviors of the multilayer perceptron and the SVMs are identical with respect to samples that lie far away from the training samples: they provide no means to evaluate the closeness of a sample to the training set. In applications where it is important that feature vectors can be rejected as not belonging to any class, there are two options. First, we can train an explicit rejection class. The problem with this approach is how to obtain or construct the samples for the rejection class. Second, we can use a classifier that provides a measure of closeness to the training samples, such as the Gaussian mixture model classifier. However, in this case we have to be prepared to accept slightly higher error rates for feature vectors that are close to the training samples.

We conclude this section with an example that uses the ICs that we have already used in Section 9.4. In this application, the goal is to read the characters in the last two lines of the print in the ICs. Figure 9.146a,b shows images of two sample ICs. The segmentation is performed with a threshold that is selected automatically based on the gray value histogram (see Figure 9.23). After this, the last two lines of characters are selected based on the smallest surrounding rectangle of the characters. For this, the character with the largest row coordinate is determined and characters lying within an interval above this character are selected. Furthermore, irrelevant characters like the "–" are suppressed based on the height of the characters. The characters are classified with a multilayer perceptron that has been trained with several tens of thousands of samples of characters on electronic components, which do not include the characters on the



(a) (b) (c) (d)

Figure 9.146 (a,b) Images of prints on ICs. (c,d) Result of the segmentation of the characters (light gray) and the OCR (black). The images are segmented with a threshold that is selected automatically based on the gray value histogram (see Figure 9.23). Furthermore, only the last two lines of characters are selected. Additionally, irrelevant characters like the "–" are suppressed based on the height of the characters.

ICs in Figure 9.146. The result of the segmentation and classification is shown in Figure 9.146c,d. Note that all characters have been read correctly.

References

1 Hagen, N. and Kudenov, M.W. (2013) Review of snapshot spectral imaging technologies. *Opt. Eng.*, **52** (9), 090 901–1–090 901–23.
2 Lapray, P.J., Wang, X., Thomas, J.B., and Gouton, P. (2014) Multispectral filter arrays: recent advances and practical implementation. *Sensors*, **14** (11), 21 626–21 659.
3 ISO 14524:2009. (2009) Photography – Electronic Still-Picture Cameras – Methods for Measuring Opto-Electronic Conversion Functions (OECFs), ISO, Geneva.
4 Mann, S. and Mann, R. (2001) Quantigraphic imaging: estimating the camera response and exposures from differently exposed images. Computer Vision and Pattern Recognition, vol. **I**, pp. 842–849.
5 Mitsunaga, T. and Nayar, S.K. (1999) Radiometric self calibration. Computer Vision and Pattern Recognition, vol. **I**, pp. 374–380.
6 Papoulis, A. (1991) *Probability, Random Variables, and Stochastic Processes*, 3rd edn, McGraw-Hill, New York.
7 Deriche, R. (1990) Fast algorithms for low-level vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, **12** (1), 78–87.

8 Lindeberg, T. (1994) *Scale-Space Theory in Computer Vision*, Kluwer Academic, Dordrecht.
9 Witkin, A.P. (1983) Scale-space filtering. 8th International Joint Conference on Artificial Intelligence, vol. **2**, pp. 1019–1022.
10 Babaud, J., Witkin, A.P., Baudin, M., and Duda, R.O. (1986) Uniqueness of the Gaussian kernel for scale-space filtering. *IEEE Trans. Pattern Anal. Mach. Intell.*, **8** (1), 26–33.
11 Florack, L.M.J., ter Haar Romeny, B.M., Koenderink, J.J., and Viergever, M.A. (1992) Scale and the differential structure of images. *Image Vision Comput.*, **10** (6), 376–388.
12 Deriche, R. (1993) Recursively Implementing the Gaussian and its Derivatives. Rapport de Recherche 1893, INRIA, Sophia Antipolis.
13 Young, I.T. and Lucas van Vliet, J. (1995) Recursive implementation of the Gaussian filter. *Signal Process.*, **44**, 139–151.
14 Huang, T.S., Yang, G.J., and Tang, G.Y. (1979) A fast two-dimensional median filtering algorithm. *IEEE Trans. Acoust. Speech Signal Process.*, **27** (1), 13–18.
15 Van Droogenbroeck, M. and Talbot, H. (1996) Fast computation of morphological operations with arbitrary structuring elements. *Pattern Recognit. Lett.*, **17** (14), 1451–1460.
16 Perreault, S. and Hébert, P. (2007) Median filtering in constant time. *IEEE Trans. Image Process.*, **16** (9), 2389–2394.
17 Brigham, E.O. (1988) *The Fast Fourier Transform and its Applications*, Prentice-Hall, Upper Saddle River, NJ.
18 Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. (1992) *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edn, Cambridge University Press, Cambridge.
19 Frigo, M. and Johnson, S.G. (2005) The design and implementation of FFTW3. *Proc. IEEE*, **93** (2), 216–231.
20 Hartley, R. and Zisserman, A. (2003) *Multiple View Geometry in Computer Vision*, 2nd edn, Cambridge University Press, Cambridge.
21 Faugeras, O. and Luong, Q.T. (2001) *The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications*, MIT Press, Cambridge, MA.
22 Haralick, R.M. and Shapiro, L.G. (1992) *Computer and Robot Vision*, vol. **I**, Addison-Wesley, Reading, MA.
23 Jain, R., Kasturi, R., and Schunck, B.G. (1995) *Machine Vision*, McGraw-Hill, New York.
24 Otsu, N. (1979) A threshold selection method from gray-level histograms. *IEEE Trans. Syst. Man Cybern.*, **9** (1), 62–66.
25 Sedgewick, R. (1990) *Algorithms in C*, Addison-Wesley, Reading, MA.
26 Hu, M.K. (1962) Visual pattern recognition by moment invariants. *IRE Trans. Inf. Theory*, **8**, 179–187.
27 Flusser, J. and Suk, T. (1993) Pattern recognition by affine moment invariants. *Pattern Recognit.*, **26** (1), 167–174.

28 Mamistvalov, A.G. (1998) *n*-dimensional moment invariants and conceptual mathematical theory of recognition *n*-dimensional solids. *IEEE Trans. Pattern Anal. Mach. Intell.*, **20** (8), 819–831.

29 Toussaint, G. (1983) Solving geometric problems with the rotating calipers. Proceedings of IEEE MELECON '83, IEEE Press, Los Alamitos, CA, pp. A10.02/1–4.

30 Welzl, E. (1991) Smallest enclosing disks (balls and ellipsoids), in *New Results and Trends in Computer Science*, Lecture Notes in Computer Science, vol. **555** (ed. H. Maurer), Springer-Verlag, Berlin, pp. 359–370.

31 de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (2000) *Computational Geometry: Algorithms and Applications*, 2nd edn, Springer-Verlag, Berlin.

32 O'Rourke, J. (1998) *Computational Geometry in C*, 2nd edn, Cambridge University Press, Cambridge.

33 Mendel, J.M. (1995) Fuzzy logic systems for engineering: a tutorial. *Proc. IEEE*, **83** (3), 345–377.

34 Steger, C. (1996) On the Calculation of Arbitrary Moments of Polygons. Technical Report FGBV–96–05, Forschungsgruppe Bildverstehen (FG BV), Informatik IX, Technische Universität München.

35 Soille, P. (2003) *Morphological Image Analysis*, 2nd edn, Springer-Verlag, Berlin.

36 Bloomberg, D.S. (2002) Implementation efficiency of binary morphology. International Symposium on Mathematical Morphology VI, pp. 209–218.

37 Lam, L., Lee, S.W., and Suen, C.Y. (1992) Thinning methodologies – a comprehensive survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, **14** (9), 869–885.

38 Eckhardt, U. and Maderlechner, G. (1993) Invariant thinning. *Int. J. Pattern Recognit. Artif. Intell.*, **7** (5), 1115–1144.

39 Borgefors, G. (1984) Distance transformation in arbitrary dimensions. *Comput. Vision Graph. Image Process.*, **27**, 321–345.

40 Danielsson, P.E. (1980) Euclidean distance mapping. *Comput. Graph. Image Process.*, **14**, 227–248.

41 Moganti, M., Ercal, F., Dagli, C.H., and Tsunekawa, S. (1996) Automatic PCB inspection algorithms: a survey. *Comput. Vision Image Understanding*, **63** (2), 287–313.

42 Gil, J. and Kimmel, R. (2002) Efficient dilation, erosion, opening, and closing algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, **24** (12), 1606–1617.

43 Canny, J. (1986) A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, **8** (6), 679–698.

44 Deriche, R. (1987) Using Canny's criteria to derive a recursively implemented optimal edge detector. *Int. J. Comput. Vision*, **1**, 167–187.

45 Steger, C. (1998) Unbiased extraction of curvilinear structures from 2D and 3D images. PhD thesis. Fakultät für Informatik, Technische Universität München. Herbert Utz Verlag, München.

46 Ando, S. (2000) Consistent gradient operators. *IEEE Trans. Pattern Anal. Mach. Intell.*, **22** (3), 252–265.

47 Lanser, S. and Eckstein, W. (1992) A modification of Deriche's approach to edge detection. 11th International Conference on Pattern Recognition, vol. **III**, pp. 633–637.

48 Steger, C. (2000) Subpixel-precise extraction of lines and edges. International Archives of Photogrammetry and Remote Sensing, vol. **XXXIII**, Part B3, pp. 141–156.

49 Haralick, R.M., Watson, L.T., and Laffey, T.J. (1983) The topographic primal sketch. *Int. J. Rob. Res.*, **2** (1), 50–72.

50 Haralick, R.M. and Shapiro, L.G. (1993) *Computer and Robot Vision*, vol. **II**, Addison-Wesley, Reading, MA.

51 JCGM 200:2012. (2012) International vocabulary of metrology – Basic and general concepts and associated terms (VIM), 3rd edition, JCGM.

52 Steger, C. (1998) Analytical and empirical performance evaluation of subpixel line and edge detection, in *Empirical Evaluation Methods in Computer Vision* (eds K.J. Bowyer and P.J. Phillips), IEEE Computer Society Press, Los Alamitos, CA, pp. 188–210.

53 Berzins, V. (1984) Accuracy of Laplacian edge detectors. *Comput. Vision Graph. Image Process.*, **27**, 195–210.

54 Lenz, R. and Fritsch, D. (1990) Accuracy of videometry with CCD sensors. *ISPRS J. Photogramm. Remote Sens.*, **45** (2), 90–110.

55 Lanser, S. (1997) Modellbasierte Lokalisation gestützt auf monokulare Videobilder. PhD thesis. Forschungs- und Lehreinheit Informatik IX, Technische Universität München. Shaker Verlag, Aachen.

56 Holland, P.W. and Welsch, R.E. (1977) Robust regression using iteratively reweighted least-squares. *Commun. Stat. Theory Methods*, **6** (9), 813–827.

57 Stewart, C.V. (1999) Robust parameter estimation in computer vision. *SIAM Rev.*, **41** (3), 513–537.

58 Huber, P.J. (1981) *Robust Statistics*, John Wiley & Sons, Inc., New York.

59 Mosteller, F. and Tukey, J.W. (1977) *Data Analysis and Regression*, Addison-Wesley, Reading, MA.

60 Fischler, M.A. and Bolles, R.C. (1981) Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, **24** (6), 381–395.

61 Joseph, S.H. (1994) Unbiased least squares fitting of circular arcs. *CVGIP: Graph. Models Image Process.*, **56** (5), 424–432.

62 Ahn, S.J., Rauh, W., and Warnecke, H.J. (2001) Least-squares orthogonal distances fitting of circle, sphere, ellipse, hyperbola, and parabola. *Pattern Recognit.*, **34** (12), 2283–2303.

63 Fitzgibbon, A., Pilu, M., and Fisher, R.B. (1999) Direct least square fitting of ellipses. *IEEE Trans. Pattern Anal. Mach. Intell.*, **21** (5), 476–480.

64 Lanser, S., Zierl, C., and Beutlhauser, R. (1995) Multibildkalibrierung einer CCD-Kamera, in *Mustererkennung, Informatik aktuell* (eds G. Sagerer, S. Posch, and F. Kummert), Springer-Verlag, Berlin, pp. 481–491.

65 Heikkilä, J. (2000) Geometric camera calibration using circular control points. *IEEE Trans. Pattern Anal. Mach. Intell.*, **22** (10), 1066–1077.

66 Rosin, P.L. (1997) Techniques for assessing polygonal approximations of curves. *IEEE Trans. Pattern Anal. Mach. Intell.*, **19** (6), 659–666.

67 Rosin, P.L. (2003) Assessing the behaviour of polygonal approximation algorithms. *Pattern Recognit.*, **36** (2), 508–518.

68 Ramer, U. (1972) An iterative procedure for the polygonal approximation of plane curves. *Comput. Graph. Image Process.*, **1**, 244–256.

69 Wuescher, D.M. and Boyer, K.L. (1991) Robust contour decomposition using a constant curvature criterion. *IEEE Trans. Pattern Anal. Mach. Intell.*, **13** (1), 41–51.

70 Sheu, H.T. and Hu, W.C. (1999) Multiprimitive segmentation of planar curves – a two-level breakpoint classification and tuning approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, **21** (8), 791–797.

71 Chen, J.M., Ventura, J.A., and Wu, C.H. (1996) Segmentation of planar curves into circular arcs and line segments. *Image Vision Comput.*, **14** (1), 71–83.

72 Rosin, P.L. and West, G.A.W. (1995) Nonparametric segmentation of curves into various representations. *IEEE Trans. Pattern Anal. Mach. Intell.*, **17** (12), 1140–1153.

73 Brown, D.C. (1971) Close-range camera calibration. *Photogramm. Eng.*, **37** (8), 855–866.

74 Gruen, A. and Huang, T.S. (eds) (2001) *Calibration and Orientation of Cameras in Computer Vision*, Springer-Verlag, Berlin.

75 Gupta, R. and Hartley, R.I. (1997) Linear pushbroom cameras. *IEEE Trans. Pattern Anal. Mach. Intell.*, **19** (9), 963–975.

76 Ahn, S.J., Warnecke, H.J., and Kotowski, R. (1999) Systematic geometric image measurement errors of circular object targets: mathematical formulation and correction. *Photogramm. Rec.*, **16** (93), 485–502.

77 Faugeras, O. (1993) *Three-Dimensional Computer Vision: A Geometric Viewpoint*, MIT Press, Cambridge, MA.

78 Scharstein, D. and Szeliski, R. (2002) A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vision*, **47** (1–3), 7–42.

79 Brown, M.Z., Burschka, D., and Hager, G.D. (2003) Advances in computational stereo. *IEEE Trans. Pattern Anal. Mach. Intell.*, **25** (8), 993–1008.

80 Scharstein, D. and Szeliski, R. Middlebury stereo vision page, http://vision.middlebury.edu/stereo/ (accessed 16 February 2015).

81 Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. The KITTI vision benchmark suite, http://www.cvlibs.net/datasets/kitti/ (accessed 16 February 2015).

82 Hirschmüller, H., Innocent, P.R., and Garibaldi, J. (2002) Real-time correlation-based stereo vision with reduced border errors. *Int. J. Comput. Vision*, **47** (1–3), 229–246.

83 Mühlmann, K., Maier, D., Hesser, J., and Männer, R. (2002) Calculating dense disparity maps from color stereo images, an efficient implementation. *Int. J. Comput. Vision*, **47** (1–3), 79–88.

84 Hirschmüller, H. and Scharstein, D. (2009) Evaluation of stereo matching costs on images with radiometric differences. *IEEE Trans. Pattern Anal. Mach. Intell.*, **31** (9), 1582–1599.

85 Faugeras, O., Hotz, B., Mathieu, H., Viéville, T., Zhang, Z., Fua, P., Théron, E., Moll, L., Berry, G., Vuillemin, J., Bertin, P., and Proy, C. (1993) Real Time

Correlation-based Stereo: Algorithm, Implementations and Applications. Rapport de Recherche 2013, INRIA, Sophia Antipolis.
86 Hu, X. and Mordohai, P. (2012) A quantitative evaluation of confidence measures for stereo vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, **34** (11), 2121–2133.
87 Brown, L.G. (1992) A survey of image registration techniques. *ACM Comput. Surv.*, **24** (4), 325–376.
88 Di Stefano, L., Mattoccia, S., and Mola, M. (2003) An efficient algorithm for exhaustive template matching based on normalized cross correlation. 12th International Conference on Image Analysis and Processing, pp. 322–327.
89 Gharavi-Alkhansari, M. (2001) A fast globally optimal algorithm for template matching using low-resolution pruning. *IEEE Trans. Image Process.*, **10** (4), 526–533.
90 Hel-Or, Y. and Hel-Or, H. (2003) Real time pattern matching using projection kernels. 9th International Conference on Computer Vision, vol. **2**, pp. 1486–1493.
91 Tanimoto, S.L. (1981) Template matching in pyramids. *Comput. Graph. Image Process.*, **16**, 356–369.
92 Glazer, F., Reynolds, G., and Anandan, P. (1983) Scene matching by hierarchical correlation. Computer Vision and Pattern Recognition, pp. 432–441.
93 Tian, Q. and Huhns, M.N. (1986) Algorithms for subpixel registration. *Comput. Vision Graph. Image Process.*, **35**, 220–233.
94 Lai, S.H. and Fang, M. (1999) Accurate and fast pattern localization algorithm for automated visual inspection. *Real-Time Imaging*, **5** (1), 3–14.
95 Anisimov, V.A. and Gorsky, N.D. (1993) Fast hierarchical matching of an arbitrarily oriented template. *Pattern Recognit. Lett.*, **14** (2), 95–101.
96 Förstner, W. (1994) A framework for low level feature extraction, in *3rd European Conference on Computer Vision*, Lecture Notes in Computer Science, vol. **801** (ed. J.O. Eklundh), Springer-Verlag, Berlin, pp. 383–394.
97 Schmid, C., Mohr, R., and Bauckhage, C. (2000) Evaluation of interest point detectors. *Int. J. Comput. Vision*, **37** (2), 151–172.
98 Borgefors, G. (1988) Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, **10** (6), 849–865.
99 Rucklidge, W.J. (1997) Efficiently locating objects using the Hausdorff distance. *Int. J. Comput. Vision*, **24** (3), 251–270.
100 Kwon, O.K., Sim, D.G., and Park, R.H. (2001) Robust Hausdorff distance matching algorithms using pyramidal structures. *Pattern Recognit.*, **34**, 2005–2013.
101 Olson, C.F. and Huttenlocher, D.P. (1997) Automatic target recognition by matching oriented edge pixels. *IEEE Trans. Image Process.*, **6** (1), 103–113.
102 Ballard, D.H. (1981) Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognit.*, **13** (2), 111–122.
103 Hough, P.V.C. (1962) Method and means for recognizing complex patterns. US Patent 3 069 654.
104 Duda, R.O. and Hart, P.E. (1972) Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM*, **15** (1), 11–15.

105 Ulrich, M., Steger, C., and Baumgartner, A. (2003) Real-time object recognition using a modified generalized Hough transform. *Pattern Recognit.*, **36** (11), 2557–2570.

106 Lamdan, Y., Schwartz, J.T., and Wolfson, H.J. (1990) Affine invariant model-based object recognition. *IEEE Trans. Rob. Autom.*, **6** (5), 578–589.

107 Ayache, N. and Faugeras, O.D. (1986) HYPER: a new approach for the recognition and positioning of two-dimensional objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, **8** (1), 44–54.

108 Grimson, W.E.L. and Lozano-Pérez, T. (1987) Localizing overlapping parts by searching the interpretation tree. *IEEE Trans. Pattern Anal. Mach. Intell.*, **9** (4), 469–482.

109 Koch, M.W. and Kashyap, R.L. (1987) Using polygons to recognize and locate partially occluded objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, **9** (4), 483–494.

110 Ventura, J.A. and Wan, W. (1997) Accurate matching of two-dimensional shapes using the minimal tolerance error zone. *Image Vision Comput.*, **15**, 889–899.

111 Costa, M.S. and Shapiro, L.G. (2000) 3D object recognition and pose with relational indexing. *Comput. Vision Image Understanding*, **79** (3), 364–407.

112 Joseph, S.H. (1999) Analysing and reducing the cost of exhaustive correspondence search. *Image Vision Comput.*, **17**, 815–830.

113 Steger, C. (2001) Similarity measures for occlusion, clutter, and illumination invariant object recognition, in *Pattern Recognition*, Lecture Notes in Computer Science, vol. **2191** (eds B. Radig and S. Florczyk), Springer-Verlag, Berlin, pp. 148–154.

114 Steger, C. (2002) Occlusion, clutter, and illumination invariant object recognition. International Archives of Photogrammetry and Remote Sensing, vol. **XXXIV**, Part 3A, pp. 345–350.

115 Steger, C. (2005) System and method for object recognition. European Patent 1 193 642.

116 Steger, C. (2006) System and method for object recognition. Japanese Patent 3 776 340.

117 Steger, C. (2006) System and method for object recognition. US Patent 7 062 093.

118 Ulrich, M. and Steger, C. (2001) Empirical performance evaluation of object recognition methods, in *Empirical Evaluation Methods in Computer Vision* (eds H.I. Christensen and P.J. Phillips), IEEE Computer Society Press, Los Alamitos, CA, pp. 62–76.

119 Ulrich, M. and Steger, C. (2002) Performance comparison of 2D object recognition techniques. International Archives of Photogrammetry and Remote Sensing, vol. **XXXIV**, Part 3A, pp. 368–374.

120 Ulrich, M., Baumgartner, A., and Steger, C. (2002) Automatic hierarchical object decomposition for object recognition. International Archives of Photogrammetry and Remote Sensing, vol. **XXXIV**, Part 5, pp. 99–104.

121 Ulrich, M. (2003) *Hierarchical Real-Time Recognition of Compound Objects in Images, Reihe C*, vol. **569**, Deutsche Geodätische Kommission bei der Bayerischen Akademie der Wissenschaften, München.

122 Ulrich, M. and Steger, C. (2007) Hierarchical component based object recognition. US Patent 7 239 929.

123 Hofhauser, A., Steger, C., and Navab, N. (2008) Edge-based template matching and tracking for perspectively distorted planar objects, in *Advances in Visual Computing*, Lecture Notes in Computer Science, vol. **5358** (eds G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, F. Porikli, J. Peters, J. Klosowski, L. Arns, Y.K. Chun, T.M. Rhyne, and L. Monroe), Springer-Verlag, Berlin, pp. 35–44.

124 Hofhauser, A. and Steger, C. (2012) System and method for deformable object recognition. US Patent 8 260 059.

125 Hofhauser, A., Steger, C., and Navab, N. (2009) Perspective planar shape matching, in *Image Processing: Machine Vision Applications II*, Proceedings of SPIE 7251 (eds K.S. Niel and D. Fofi), SPIE Press, Bellingham.

126 Hofhauser, A., Steger, C., and Navab, N. (2009) Edge-based template matching with a harmonic deformation model, in *Computer Vision and Computer Graphics: Theory and Applications International Conference VISIGRAPP 2008, Revised Selected Papers*, Communications in Computer and Information Science, vol. **24** (eds A.K. Ranchordas, H.J. Araujo, J.M. Pereira, and J. Braz), Springer-Verlag, Berlin, pp. 176–187.

127 Wiedemann, C., Ulrich, M., and Steger, C. (2008) Recognition and tracking of 3D objects, in *Pattern Recognition*, Lecture Notes in Computer Science, vol. **5096** (ed. G. Rigoll), Springer-Verlag, Berlin, pp. 132–141.

128 Ulrich, M., Wiedemann, C., and Steger, C. (2009) CAD-based recognition of 3D objects in monocular images. International Conference on Robotics and Automation, pp. 1191–1198.

129 Ulrich, M., Wiedemann, C., and Steger, C. (2012) Combining scale-space and similarity-based aspect graphs for fast 3D object recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, **34** (10), 1902–1914.

130 Wiedemann, C., Ulrich, M., and Steger, C. (2013) System and method for 3D object recognition. US Patent 8 379 014.

131 Casey, R.G. and Lecolinet, E. (1996) A survey of methods and strategies in character segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, **18** (7), 690–706.

132 Theodoridis, S. and Koutroumbas, K. (1999) *Pattern Recognition*, Academic Press, San Diego, CA.

133 Webb, A. (1999) *Statistical Pattern Recognition*, Arnold Publishers, London.

134 Friedman, J.H., Bentley, J.L., and Finkel, R.A. (1977) An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, **3** (3), 209–226.

135 Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., and Wu, A.Y. (1998) An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, **45** (6), 891–923.

136 Muja, M. and Lowe, D.G. (2014) Scalable nearest neighbor algorithms for high dimensional data. *IEEE Trans. Pattern Anal. Mach. Intell.*, **36** (11), 2227–2240.

137 Figueiredo, M.A.T. and Jain, A.K. (2002) Unsupervised learning of finite mixture models. *IEEE Trans. Pattern Anal. Mach. Intell.*, **24** (3), 381–396.

138 Wang, H.X., Luo, B., Zhang, Q.B., and Wei, S. (2004) Estimation for the number of components in a mixture model using stepwise split-and-merge EM algorithm. *Pattern Recognit. Lett.*, **25** (16), 1799–1809.

139 Bishop, C.M. (1995) *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford.

140 Schölkopf, B. and Smola, A.J. (2002) *Learning with Kernels – Support Vector Machines, Regularization, Optimization, and Beyond*, MIT Press, Cambridge, MA.

141 Christianini, N. and Shawe-Taylor, J. (2000) *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*, Cambridge University Press, Cambridge.