9

Morphological Filters

In the discussion of the median filter in Chapter 5 (Sec. 5.4.2), we noticed that this type of filter can somehow alter 2D image structures. Figure 9.1 illustrates once more how corners are rounded off, holes of a certain size are filled, and small structures, such as single dots or thin lines, are removed. The median filter thus responds selectively to the local shape of image structures, a property that might be useful for other purposes if it can be applied not just randomly but in a controlled fashion. Altering the local structure in a predictable way is exactly what "morphological" filters can do, which we focus on in this chapter.

(a) (b) (c)

Fig. 9.1
Median filter applied to a binary image: original image (a) and results from a 3×3 pixel median filter (b) and a 5×5 pixel median filter (c).

In their original form, morphological filters are aimed at binary images, images with only two possible pixel values, 0 and 1 or *black* and *white*, respectively. Binary images are found in many places, in particular in digital printing, document transmission (FAX) and storage, or as selection masks in image and video editing. Binary images can be obtained from grayscale images by simple thresholding (see Sec. 4.1.4) using either a global or a locally varying threshold value. We denote binary pixels with values 1 and 0 as *foreground* and *background* pixels, respectively. In most of the following examples, the foreground pixels are shown in black and background pixels are shown in white, as is common in printing.

At the end of this chapter, we will see that morphological filters are applicable not only to binary images but also to grayscale and

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_9

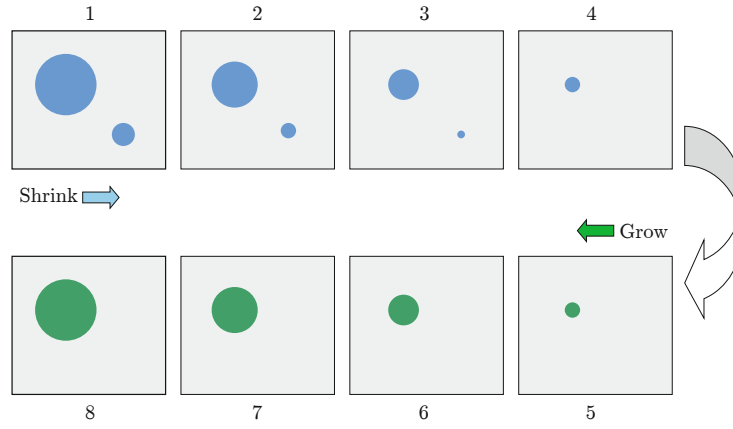


Fig. 9.2
Basic idea of size-dependent removal of image structures. Small structures may be eliminated by iterative shrinking and subsequent growing. Ideally, the "surviving" structures should be restored to their original shape.

even color images, though these operations differ significantly from their binary counterparts.

9.1 Shrink and Let Grow

Our starting point was the observation that a simple 3×3 pixel median filter can round off larger image structures and remove smaller structures, such as points and thin lines, in a binary image. This could be useful to eliminate structures that are below a certain size (e.g., to clean an image from noise or dirt). But how can we control the size and possibly the shape of the structures affected by such an operation?

Although its structural effects may be interesting, we disregard the median filter at this point and start with this task again from the beginning. Let's assume that we want to remove small structures from a binary image without significantly altering the remaining larger structures. The key idea for accomplishing this could be the following (Fig. 9.2):

1. First, all structures in the image are iteratively "shrunk" by peeling off a layer of a certain thickness around the boundaries.
2. Shrinking removes the smaller structures step by step, and only the larger structures remain.
3. The remaining structures are then grown back by the same amount.
4. Eventually the larger regions should have returned to approximately their original shapes, while the smaller regions have disappeared from the image.

All we need for this are two types of operations. "Shrinking" means to remove a layer of pixels from a foreground region around all its borders against the background (Fig. 9.3). The other way around, "growing", adds a layer of pixels around the border of a foreground region (Fig. 9.4).

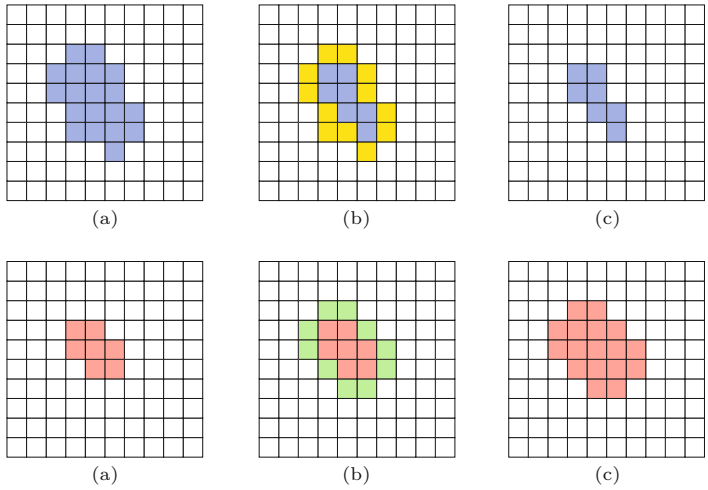


Fig. 9.3
"Shrinking" a foreground region by removing a layer of border pixels: original image (a), identified foreground pixels that are in direct contact with the background (b), and result after shrinking (c).

Fig. 9.4
"Growing" a foreground region by attaching a layer of pixels: original image (a), identified background pixels that are in direct contact with the region (b), and result after growing (c).

9.1.1 Neighborhood of Pixels

For both operations, we must define the meaning of two pixels being adjacent (i.e., being "neighbors"). Two definitions of "neighborhood" are commonly used for rectangular pixel grids (Fig. 9.5):

- **4-neighborhood** ($\mathcal{N}_4$): the four pixels adjacent to a given pixel in the horizontal and vertical directions;
- **8-neighborhood** ($\mathcal{N}_8$): the pixels contained in $\mathcal{N}_4$ plus the four adjacent pixels along the diagonals.

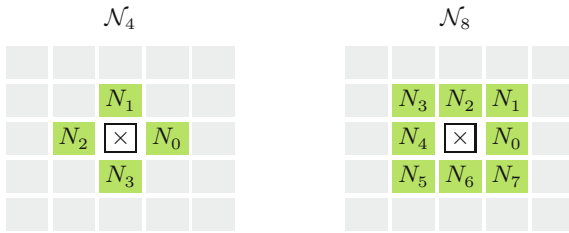


Fig. 9.5
Definitions of "neighborhood" on a rectangular pixel grid: *4-neighborhood* $\mathcal{N}_4 = \{N_1, \ldots, N_4\}$ and *8-neighborhood* $\mathcal{N}_8 = \mathcal{N}_4 \cup \{N_5, \ldots, N_8\}$.

9.2 Basic Morphological Operations

Shrinking and growing are indeed the two most basic morphological operations, which are referred to as "erosion" and "dilation", respectively. These morphological operations, however, are much more general than illustrated in the example in Sec. 9.1. They go well beyond removing or attaching single pixel layers and—in combination—can perform much more complex operations.

9.2.1 The Structuring Element

Similar to the coefficient matrix of a linear filter (see Sec. 5.2), the properties of a morphological filter are specified by elements in a matrix called a "structuring element". In binary morphology, the structuring element (just like the image itself) contains only the values 0

and 1,

$$H(i,j) \in \{0,1\},$$

and the *hot spot* marks the origin of the coordinate system of H (Fig. 9.6). Notice that the hot spot is not necessarily located at the center of the structuring element, nor must its value be 1.

Fig. 9.6 Binary structuring element (example). 1–elements are marked with •; 0–cells are empty. The hot spot (boxed) is not necessarily located at the center.

9.2.2 Point Sets

For the formal specification of morphological operations, it is sometimes helpful to describe binary images as *sets* of 2D coordinate points.[1]

For a binary image $I(u,v) \in \{0,1\}$, the corresponding point set $\mathcal{Q}_I$ consists of the coordinate pairs $\boldsymbol{p} = (u,v)$ of all foreground pixels,

$$Q_I = \{\boldsymbol{p} \mid I(\boldsymbol{p}) = 1\}. \tag{9.1}$$

Of course, as shown in Fig. 9.7, not only a binary image I but also a structuring element H can be described as a point set.

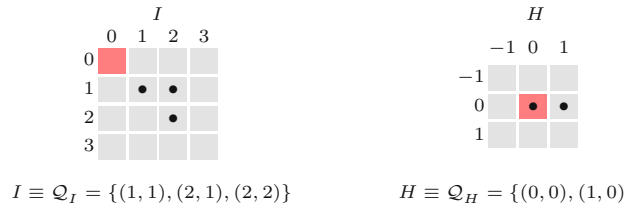


Fig. 9.7 A binary image I or a structuring element H can each be described as a set of coordinate pairs, $\mathcal{Q}_I$ and $\mathcal{Q}_H$, respectively. The dark shaded element in H marks the coordinate origin (hot spot).

With the description as point sets, fundamental operations on binary images can also be expressed as simple set operations. For example, *inverting* a binary image $I \to \bar{I}$ (i.e., exchanging foreground and background) is equivalent to building the *complementary* set

$$\mathcal{Q}_{\bar{I}} = \bar{\mathcal{Q}}_I = \{\boldsymbol{p} \in \mathbb{Z}^2 \mid \boldsymbol{p} \notin \mathcal{Q}_I\}. \tag{9.2}$$

Combining two binary images I_1 and I_2 by an OR operation between corresponding pixels, the resulting point set is the *union* of the individual point sets $\mathcal{Q}_{I_1}$ and $\mathcal{Q}_{I_2}$; that is,

$$Q_{I_1 \vee I_2} = Q_{I_1} \cup Q_{I_2}. \tag{9.3}$$

Since a point set $\mathcal{Q}_I$ is only an alternative representation of the binary image I (i.e., $I \equiv \mathcal{Q}_I$), we will use both image and set notations synonymously in the following. For example, we simply write $\bar{I}$ instead of $\bar{\mathcal{Q}}_I$ for an inverted image as in Eqn. (9.2) or $I_1 \cup I_2$ instead of $\mathcal{Q}_{I_1} \cup \mathcal{Q}_{I_2}$ in Eqn. (9.3). The meaning should always be clear in the given context.

[1] *Morphology* is a mathematical discipline dealing with the algebraic analysis of geometrical structures and shapes, with strong roots in set theory.

Translating (shifting) a binary image I by some coordinate vector $\boldsymbol{d}$ creates a new image with the content

$$I_{\boldsymbol{d}}(\boldsymbol{p}+\boldsymbol{d}) = I(\boldsymbol{p}) \qquad \text{oder} \qquad I_{\boldsymbol{d}}(\boldsymbol{p}) = I(\boldsymbol{p}-\boldsymbol{d}), \tag{9.4}$$

which is equivalent to changing the coordinates of the original point set in the form

$$I_{\boldsymbol{d}} \equiv \big\{(\boldsymbol{p}+\boldsymbol{d}) \mid \boldsymbol{p} \in I\big\}. \tag{9.5}$$

In some cases, it is also necessary to *reflect* (mirror) a binary image or point set about its origin, which we denote as

$$I^* \equiv \{-\boldsymbol{p} \mid \boldsymbol{p} \in I\}. \tag{9.6}$$

9.2.3 Dilation

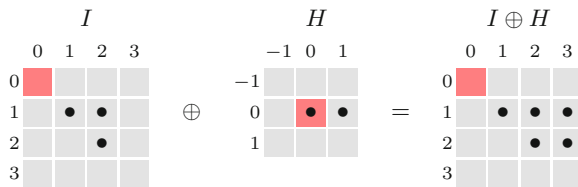
A *dilation* is the morphological operation that corresponds to our intuitive concept of "growing" as discussed already. As a set operation, it is defined as

$$I \oplus H \equiv \big\{(\boldsymbol{p}+\boldsymbol{q}) \mid \text{for all } \boldsymbol{p} \in I, \boldsymbol{q} \in H\big\}. \tag{9.7}$$

Thus the point set produced by a dilation is the (vector) sum of all possible pairs of coordinate points from the original sets I and H, as illustrated by a simple example in Fig. 9.8. Alternatively, one could view the dilation as the structuring element H being *replicated* at each foreground pixel of the image I or, conversely, the image I being replicated at each foreground element of H. Expressed in set notation,[2] this is

$$I \oplus H \equiv \bigcup_{\boldsymbol{p} \in I} H_{\boldsymbol{p}} = \bigcup_{\boldsymbol{q} \in H} I_{\boldsymbol{q}}, \tag{9.8}$$

with $H_{\boldsymbol{p}}, I_{\boldsymbol{q}}$ denoting the sets H, I shifted by $\boldsymbol{p}$ and $\boldsymbol{q}$, respectively (see Eqn. (9.5)).



$$I \equiv \{(1,1),(2,1),(2,2)\},\ H \equiv \{(\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{0})\}$$

$$\begin{aligned} I \oplus H \ \equiv\ \{\ &(1,1)+(\mathbf{0},\mathbf{0}),\ (1,1)+(\mathbf{1},\mathbf{0}),\\ &(2,1)+(\mathbf{0},\mathbf{0}),\ (2,1)+(\mathbf{1},\mathbf{0}),\\ &(2,2)+(\mathbf{0},\mathbf{0}),\ (2,2)+(\mathbf{1},\mathbf{0})\ \} \end{aligned}$$

Fig. 9.8
Binary dilation example. The binary image I is subject to dilation with the structuring element H. In the result $I \oplus H$ the structuring element H is replicated at every foreground pixel of the original image I.

[2] See also Sec. A.2 in the Appendix.

9.2.4 Erosion

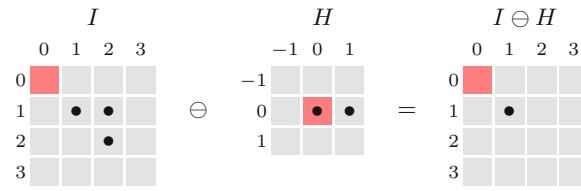
The quasi-inverse of dilation is the *erosion* operation, again defined in set notation as

$$I \ominus H \equiv \big\{\boldsymbol{p} \in \mathbb{Z}^2 \mid (\boldsymbol{p}+\boldsymbol{q}) \in I, \text{ for all } \boldsymbol{q} \in H\big\}. \tag{9.9}$$

This operation can be interpreted as follows. A position $\boldsymbol{p}$ is contained in the result $I \ominus H$ if (and only if) the structuring element H—when placed at this position $\boldsymbol{p}$—is *fully contained* in the foreground pixels of the original image; that is, if $H_{\boldsymbol{p}}$ is a subset of I. Equivalent to Eqn. (9.9), we could thus define binary erosion as

$$I \ominus H \equiv \{\boldsymbol{p} \in \mathbb{Z}^2 \mid H_{\boldsymbol{p}} \subseteq I\}. \tag{9.10}$$

Figure 9.9 shows a simple example for binary erosion.



$$I \equiv \{(1,1),(2,1),(2,2)\},\ H \equiv \{(\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{0})\}$$

$$I \ominus H \equiv \{\,(1,1)\,\} \text{ because}$$
$$(1,1)+(\mathbf{0},\mathbf{0}) = (1,1) \in I \quad \textbf{and} \quad (1,1)+(\mathbf{1},\mathbf{0}) = (2,1) \in I$$

Fig. 9.9 Binary erosion example. The binary image I is subject to erosion with H as the structuring element. H is only covered by I when placed at position $\boldsymbol{p} = (1,1)$, thus the resulting points set contains only the single coordinate $(1,1)$.

9.2.5 Formal Properties of Dilation and Erosion

The dilation operation is *commutative*,

$$I \oplus H = H \oplus I, \tag{9.11}$$

and therefore—just as in linear convolution—the image and the structuring element (filter) can be exchanged to get the same result. Dilation is also *associative*, that is,

$$(I_1 \oplus I_2) \oplus I_3 = I_1 \oplus (I_2 \oplus I_3), \tag{9.12}$$

and therefore the ordering of multiple dilations is not relevant. This also means—analogous to linear filters (cf. Eqn. (5.25))—that a dilation with a large structuring element of the form $H_{\text{big}} = H_1 \oplus H_2 \oplus \ldots \oplus H_K$ can be efficiently implemented as a sequence of multiple dilations with smaller structuring elements by

$$I \oplus H_{\text{big}} = (\ldots((I \oplus H_1) \oplus H_2) \oplus \ \ldots \ \oplus H_K) \tag{9.13}$$

There is also a *neutral element* (δ) for the dilation operation, similar to the Dirac function for the linear convolution (see Sec. 5.3.4),

$$I \oplus \delta = \delta \oplus I = I, \quad \text{with } \delta = \{(0,0)\}. \tag{9.14}$$

The *erosion* operation is, in contrast to dilation (but similar to arithmetic subtraction), *not* commutative, that is,

$$I \ominus H \neq H \ominus I, \tag{9.15}$$

in general. However, if erosion and dilation are combined, then—again in analogy with arithmetic subtraction and addition—the following chain rule holds:

$$(I_1 \ominus I_2) \ominus I_3 = I_1 \ominus (I_2 \oplus I_3). \tag{9.16}$$

Although dilation and erosion are not mutually inverse (in general, the effects of dilation cannot be undone by a subsequent erosion), there are still some strong formal relations between these two operations. For one, dilation and erosion are *dual* in the sense that a dilation of the *foreground* (I) can be accomplished by an erosion of the *background* ($\bar{I}$) and subsequent inversion of the result,

$$I \oplus H \;=\; \overline{(\bar{I} \ominus H^*)}, \tag{9.17}$$

where H^* denotes the *reflection* of H (Eqn. (9.6)). This works similarly the other way, too, namely

$$\bar{I} \ominus H = \overline{(I \oplus H^*)}, \tag{9.18}$$

effectively eroding the foreground by dilating the background with the mirrored structuring element, as illustrated by the example in Fig. 9.10 (see [88, pp. 521–524] for a formal proof).

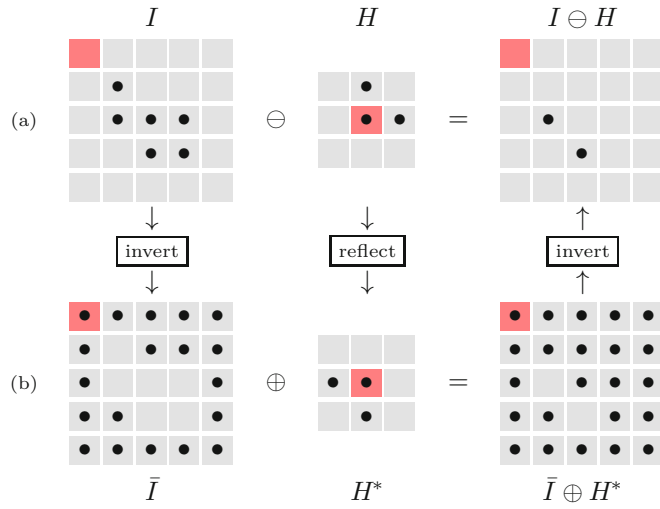


Fig. 9.10
Implementing erosion via dilation. The binary erosion of the foreground $I \ominus H$ (a) can be implemented by dilating the inverted (background) image $\bar{I}$ with the reflected structuring element H^* and subsequently inverting the result again (b).

Equation (9.18) is interesting because it shows that we only need to implement either dilation or erosion for computing both, considering that the foreground–background inversion is a very simple task. Algorithm 9.1 gives a simple algorithmic description of dilation and erosion based on the aforementioned relationships.

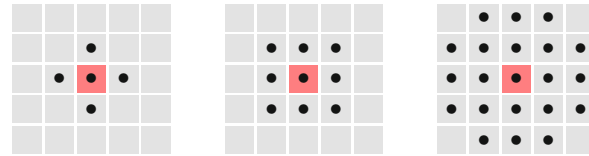
Alg. 9.1 Binary dilation and erosion. Procedure Dilate() implements the binary dilation as suggested by Eqn. (9.8). The original image I is displaced to each foreground coordinate of H and then copied into the resulting image I'. The hot spot of the structuring element H is assumed to be at coordinate $(0, 0)$. Procedure Erode() implements the binary erosion by dilating the inverted image $\bar{I}$ with the reflected structuring element H^*, as described by Eqn. (9.18).

Dilate(I, H)
Input: I, a binary image of size $M \times N$;
H, a binary structuring element.
Returns the dilated image $I' = I \oplus H$.

Create map $I' \colon M \times N \mapsto \{0, 1\}$ ▷ new binary image I'
for all $(\boldsymbol{p}) \in M \times N$ **do**
  $I'(\boldsymbol{p}) \leftarrow 0$ ▷ $I' \leftarrow \{\,\}$
for all $\boldsymbol{q} \in H$ **do**
  for all $\boldsymbol{p} \in I$ **do**
    $I'(\boldsymbol{p}+\boldsymbol{q}) \leftarrow 1$ ▷ $I' \leftarrow I' \cup \{(\boldsymbol{p}+\boldsymbol{q})\}$
return I' ▷ $I' = I \oplus H$

Erode(I, H)
Input: I, a binary image of size $M \times N$;
H, a binary structuring element.
Returns the eroded image $I' = I \ominus H$.

$\bar{I} \leftarrow \mathsf{Invert}(I)$ ▷ $\bar{I} \leftarrow \neg I$
$H^* \leftarrow \mathsf{Reflect}(H)$
$I' \leftarrow \mathsf{Invert}(\mathsf{Dilate}(\bar{I}, H^*))$ ▷ $I' = I \ominus H = \overline{(\bar{I} \oplus H^*)}$
return I'



(a) (b) (c)

Fig. 9.11 Typical binary structuring elements of various sizes. 4-neighborhood (a), 8-neighborhood (b), "small disk" (c).

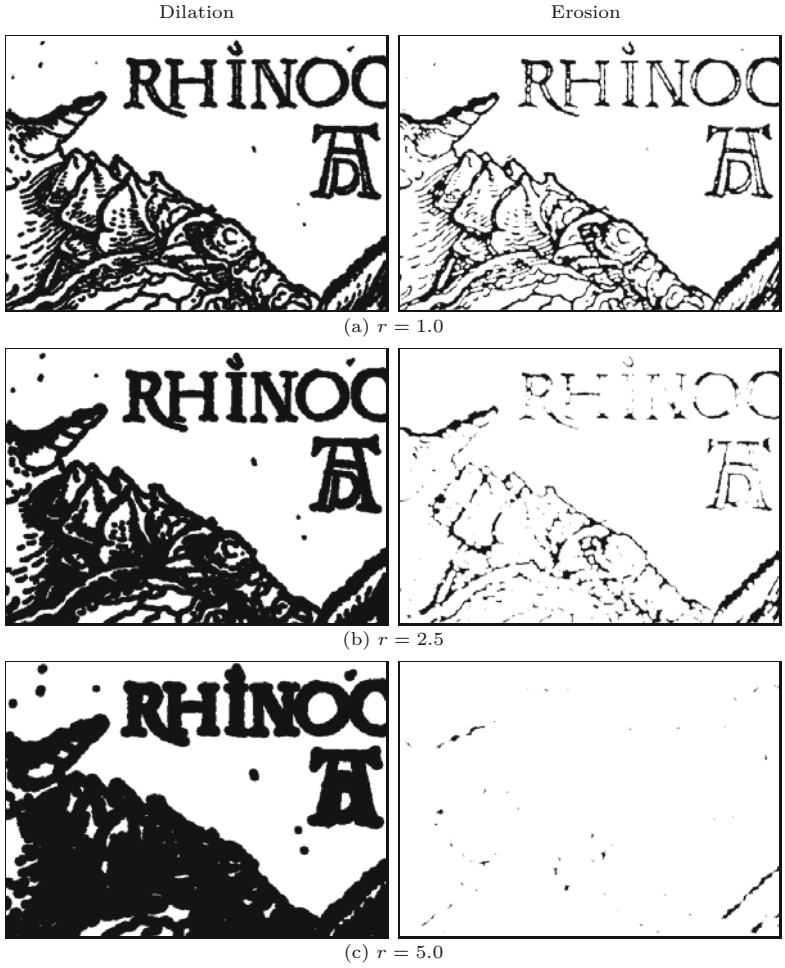
9.2.6 Designing Morphological Filters

A morphological filter is unambiguously specified by (a) the type of operation and (b) the contents of the structuring element. The appropriate size and shape of the structuring element depends upon the application, image resolution, etc. In practice, structuring elements of quasi-circular shape are frequently used, such as the examples shown in Fig. 9.11.

A dilation with a circular (disk-shaped) structuring element with radius r adds a layer of thickness r to any foreground structure in the image. Conversely, an erosion with that structuring element peels off layers of the same thickness. Figure 9.13 shows the results of dilation and erosion with disk-shaped structuring elements of different diameters applied to the original image in Fig. 9.12. Dilation and erosion results for various other structuring elements are shown in Fig. 9.14.

Disk-shaped structuring elements are commonly used to implement *isotropic* filters, morphological operations that have the same effect in every direction. Unlike linear filters (e.g., the 2D Gaussian filter in Sec. 5.3.3), it is generally not possible to compose an isotropic 2D structuring element H° from 1D structuring elements H_x and H_y since the dilation $H_x \oplus H_y$ always results in a rectangular (i.e., non-isotropic) structure. A remedy for approximating large disk-shaped filters is to alternately apply smaller disk-shaped operators of differ-

Fig. 9.12
Original binary image and the section used in the following examples (illustration by Albrecht Dürer, 1515).



(a) $r = 1.0$

(b) $r = 2.5$

(c) $r = 5.0$

Fig. 9.13
Results of binary dilation and erosion with disk-shaped structuring elements. The radius of the disk (r) is 1.0 (a), 2.5 (b), and 5.0 (c).

ent shapes, as illustrated in Fig. 9.15. The resulting filter is generally not fully isotropic but can be implemented efficiently as a sequence of small filters.

9.2.7 Application Example: Outline

A typical application of morphological operations is to extract the boundary pixels of the foreground structures. The process is very simple. First, we apply an erosion on the original image I to remove the boundary pixels of the foreground,

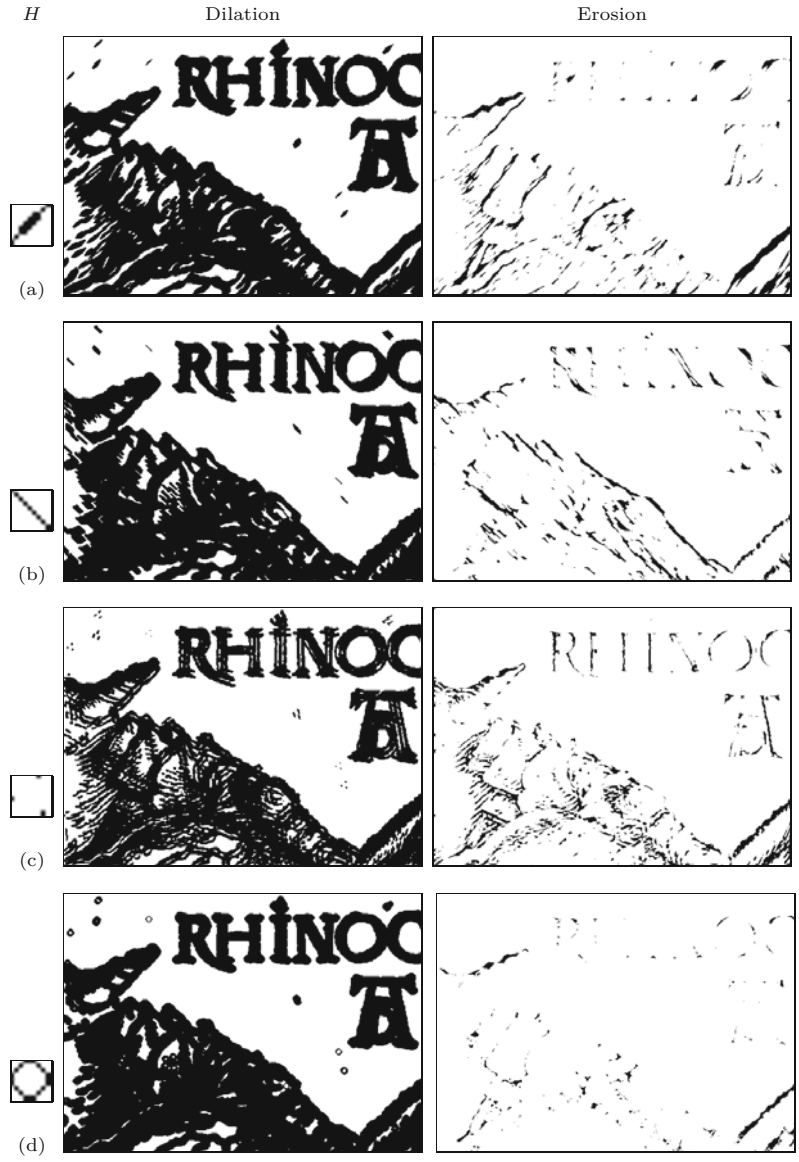


Fig. 9.14 Examples of binary dilation and erosion with various free-form structuring elements. The structuring elements H are shown in the left column (enlarged). Notice that the dilation expands every isolated foreground point to the shape of the structuring element, analogous to the *impulse response* of a linear filter. Under erosion, only those elements where the structuring element is fully contained in the original image survive.

$$I' = I \ominus H_n,$$

where H_n is a structuring element, for example, for a 4- or 8-neighborhood (Fig. 9.11) as the structuring element H_n. The actual boundary pixels B are those contained in the original image but *not* in the eroded image, that is, the *intersection* of the original image I and the inverted result $\bar{I}'$, or

$$B \leftarrow I \cap \overline{I'} = I \cap \overline{(I \ominus H_n)}\,. \tag{9.19}$$

Figure 9.17 shows an example for the extraction of region boundaries. Notice that using the 4-neighborhood as the structuring element H_n produces "8-connected" contours and vice versa [125, p. 504].

The process of boundary extraction is illustrated on a simple example in Fig. 9.16. As can be observed in this figure, the result B

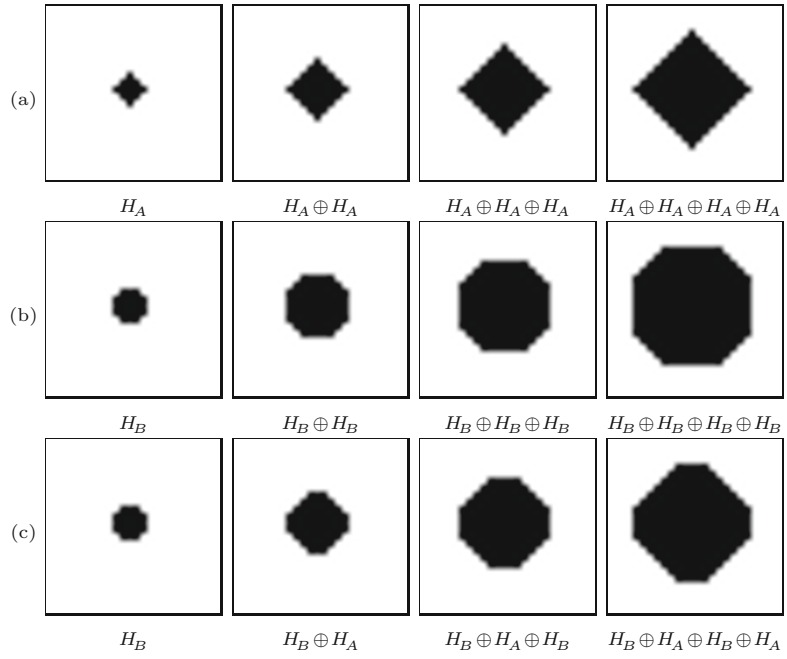


Fig. 9.15
Composition of large morphological filters by repeated application of smaller filters: repeated application of the structuring element H_A (a) and structuring element H_B (b); alternating application of H_B and H_A (c).

contains exactly those pixels that are *different* in the original image I and the eroded image $I' = I \ominus H_n$, which can also be obtained by an exclusive-OR (XOR) operation between pairs of pixels; that is, boundary extraction from a binary image can be implemented as

$$B(\boldsymbol{p}) \leftarrow I(\boldsymbol{p}) \text{ XOR } (I \ominus H_n)(\boldsymbol{p}), \quad \text{for all } \boldsymbol{p}. \tag{9.20}$$

Figure 9.17 shows a more complex example for isolating the boundary pixels in a real image.

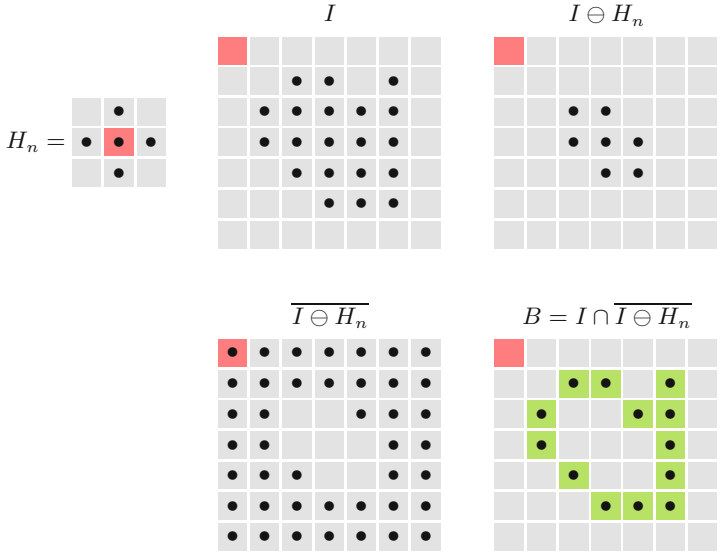
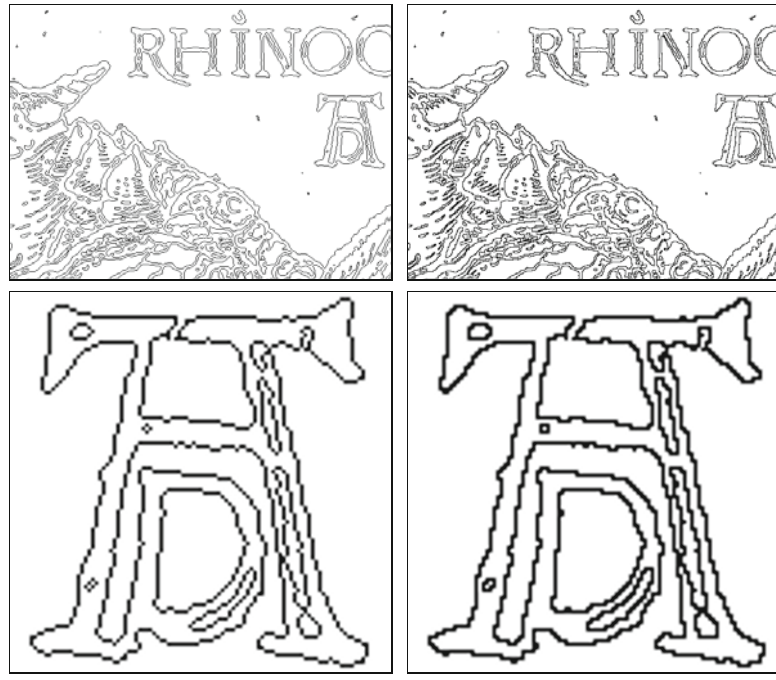


Fig. 9.16
Outline example using a 4-neighborhood structuring element H_n. The image I is first eroded ($I \ominus H_n$) and subsequently inverted ($\overline{I \ominus H_n}$). The boundary pixels are finally obtained as the intersection $I \cap \overline{I \ominus H_n}$.



(a) (b)

Fig. 9.17
Extraction of boundary pixels using morphological operations. The 4-neighborhood structuring element used in (a) produces 8-connected contours. Conversely, using the 8-neighborhood as the structuring element gives 4-connected contours (b).

9.3 Composite Morphological Operations

Due to their semiduality, dilation and erosion are often used together in composite operations, two of which are so important that they even carry their own names and symbols: "opening" and "closing". They are probably the most frequently used morphological operations in practice.

9.3.1 Opening

A binary opening $I \circ H$ denotes an erosion followed by a dilation with the *same* structuring element H,

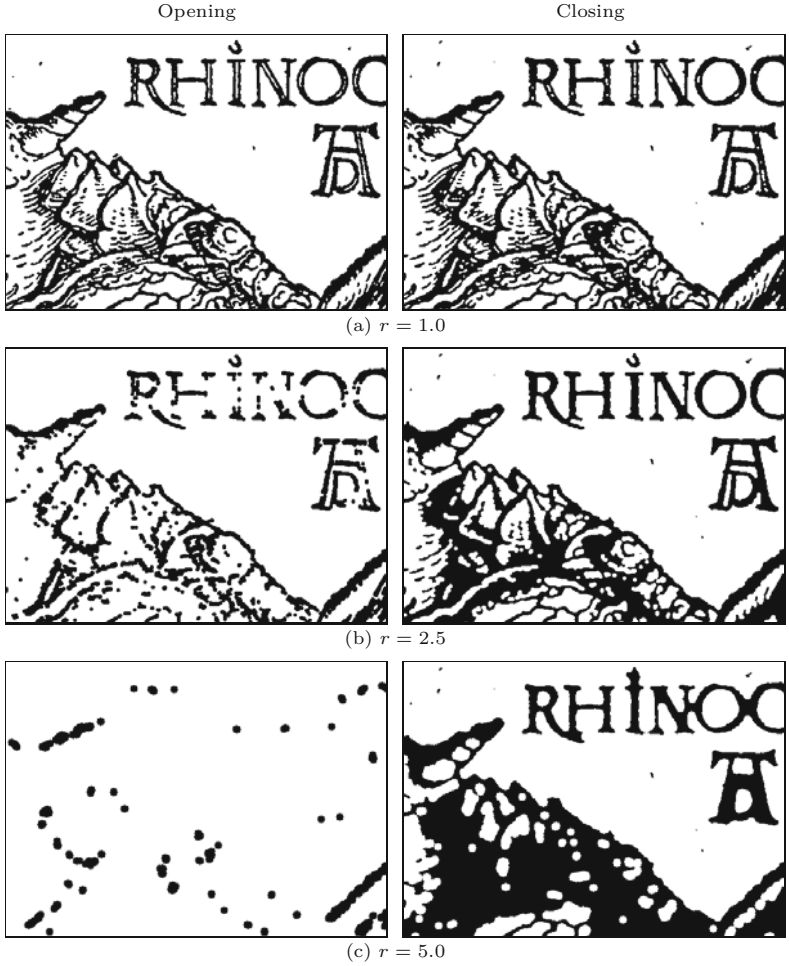
$$I \circ H = (I \ominus H) \oplus H. \tag{9.21}$$

The main effect of an opening is that all foreground structures that are smaller than the structuring element are eliminated in the first step (erosion). The remaining structures are smoothed by the subsequent dilation and grown back to approximately their original size, as demonstrated by the examples in Fig. 9.18. This process of shrinking and subsequent growing corresponds to the idea for eliminating small structures that we had initially sketched in Sec. 9.1.

9.3.2 Closing

When the sequence of erosion and dilation is reversed, the resulting operation is called a closing and denoted $I \bullet H$,

$$I \bullet H = (I \oplus H) \ominus H. \tag{9.22}$$



(a) $r = 1.0$

(b) $r = 2.5$

(c) $r = 5.0$

Fig. 9.18
Binary opening and closing with disk-shaped structuring elements. The radius r of the structuring element H is 1.0 (top), 2.5 (center), or 5.0 (bottom).

A *closing* removes (closes) holes and fissures in the foreground structures that are smaller than the structuring element H. Some examples with typical disk-shaped structuring elements are shown in Fig. 9.18.

9.3.3 Properties of Opening and Closing

Both operations, opening as well as closing, are *idempotent*, meaning that their results are "final" in the sense that any subsequent application of the same operation no longer changes the result, that is,

$$\begin{aligned} I \circ H &= (I \circ H) \circ H = ((I \circ H) \circ H) \circ H = \dots , \\ I \bullet H &= (I \bullet H) \bullet H = ((I \bullet H) \bullet H) \bullet H = \dots . \end{aligned} \tag{9.23}$$

Also, opening and closing are "duals" in the sense that opening the foreground is equivalent to closing the background and vice versa, that is,

$$I \circ H = \overline{\bar{I} \bullet H} \quad \text{and} \quad I \bullet H = \overline{\bar{I} \circ H}. \tag{9.24}$$

9.4 Thinning (Skeletonization)

Thinning is a common morphological technique which aims at shrinking binary structures down to a maximum thickness of one pixel without splitting them into multiple parts. This is accomplished by iterative "conditional" erosion. It is applied to a local neighborhood only if a sufficiently thick structure remains and the operation does not cause a separation to occur. This requires that, depending on the local image structure, a decision must be made at every image position whether another erosion step may be applied or not. The operation continues until no more changes appear in the resulting image. It follows that, compared to the ordinary ("homogeneous") morphological discussed earlier, thinning is computationally expensive in general. A frequent application of thinning is to calculate the "skeleton" of a binary region, for example, for structural matching of 2D shapes.

Thinning is also known by the terms *center line detection* and *medial axis transform*. Many different implementations of varied complexity and efficiency exist (see, e.g., [2, 7, 68, 108, 201]). In the following, we describe the classic algorithm by Zhang and Suen [265] and its implementation as a representative example.[3]

9.4.1 Thinning Algorithm by Zhang and Suen

The input to this algorithm is a binary image I, with foreground pixels carrying the value 1 and background pixels with value 0. The algorithm scans the image and at each position (u, v) examines a 3×3 neighborhood with the central element P and the surrounding values $\mathsf{N} = (N_0, N_1, \ldots, N_7)$, as illustrated in Fig. 9.5(b). The complete process is summarized in Alg. 9.2.

For classifying the contents of the local neighborhood N we first define the function

$$B(\mathsf{N}) = N_0 + N_1 + \cdots + N_7 = \sum_{i=0}^{7} N_i, \tag{9.25}$$

which simply counts surrounding foreground pixels. We also define the so-called "connectivity number" to express how many binary components are connected via the current center pixel at position (u, v). This quantity is equivalent to the number of $1 \to 0$ transitions in the sequence $(N_0, \ldots, N_7, N_0)$, or expressed in arithmetic terms,

$$C(\mathsf{N}) = \sum_{i=0}^{7} N_i \cdot [N_i - N_{(i+1) \bmod 8}]. \tag{9.26}$$

Figure 9.19 shows some selected examples for the neighborhood N and the associated values for the functions $B(\mathsf{N})$ and $C(\mathsf{N})$. Based on the above functions, we finally define two Boolean predicates R_1, R_2 on the neighborhood N,

[3] The built-in thinning operation in ImageJ is also based on this algorithm.

$$R_1(\mathsf{N}) := [\,2 \le B(\mathsf{N}) \le 6\,] \wedge [\,C(\mathsf{N}) = 1\,] \wedge [\,N_6 \cdot N_0 \cdot N_2 = 0\,] \wedge [\,N_4 \cdot N_6 \cdot N_0 = 0\,], \quad (9.27)$$

$$R_2(\mathsf{N}) := [\,2 \le B(\mathsf{N}) \le 6\,] \wedge [\,C(\mathsf{N}) = 1\,] \wedge [\,N_0 \cdot N_2 \cdot N_4 = 0\,] \wedge [\,N_2 \cdot N_4 \cdot N_6 = 0\,]. \quad (9.28)$$

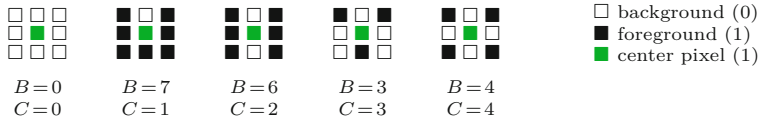


Fig. 9.19
Selected binary neighborhood patterns N and associated function values $B(\mathsf{N})$ and $C(\mathsf{N})$ (see Eqns. (9.25)–(9.26)).

Depending on the outcome of $R_1(\mathsf{N})$ and $R_2(\mathsf{N})$, the foreground pixel at the center position of N is either deleted (i.e., eroded) or marked as non-removable (see Alg. 9.2, lines 16 and 27).

Figure 9.20 illustrates the effect of layer-by-layer thinning performed by procedure ThinOnce(). In every iteration, only one "layer" of foreground pixels is selectively deleted. An example of thinning applied to a larger binary image is shown in Fig. 9.21.

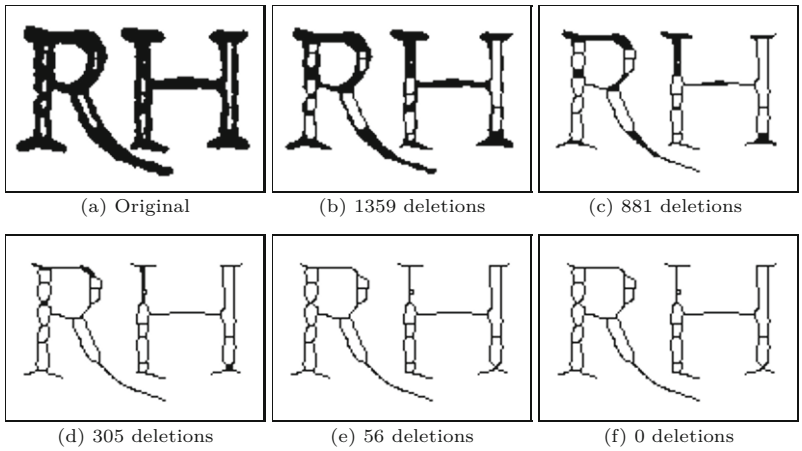


Fig. 9.20
Iterative application of the ThinOnce() procedure. The "deletions" indicated in (b–f) denote the number of pixels that were removed from the previous image. No deletions occurred in the final iteration (from (e) to (f)). Thus five iterations were required to thin this image.

9.4.2 Fast Thinning Algorithm

In a binary image, only $2^8 = 256$ different combinations of zeros and ones are possible inside any 8-neighborhood. Since the expressions in Eqns. (9.27)–(9.27) are relatively costly to evaluate it makes sense to pre-calculate and tabulate all 256 instances (see Fig. 9.22). This is the basis of the fast version of Zhang and Suen's algorithm, summarized in Alg. 9.3. It uses a decision table Q, which is constant and calculated only once by procedure MakeDeletionCodeTable() in Alg. 9.3 (lines 34–45). The table contains the binary codes

$$\mathsf{Q}(i) \in \{0,1,2,3\} = \{00_\mathrm{b}, 01_\mathrm{b}, 10_\mathrm{b}, 11_\mathrm{b}\}, \quad (9.29)$$

for $i = 0, \ldots, 255$, where the two bits correspond to the predicates R_1 and R_2, respectively. The associated test is found in procedure ThinOnceFast() in line 19. The two passes are in this case controlled by a separate loop variable ($p = 1, 2$). In the concrete implementation, the map Q is not calculated at the start but defined as a constant array (see Prog. 9.1 for the actual Java code).

Alg. 9.2 Iterative thinning algorithm by Zhang und Suen [265]. Procedure ThinOnce() performs a single thinning step on the supplied binary image I_b and returns the number of deleted foreground pixels. It is iteratively invoked by Thin() until no more pixels are deleted. The required pixel deletions are only registered in the binary map D and executed en-bloc at the end of every iteration. Lines 40–42 define the functions $R_1()$, $R_2()$, $B()$ and $C()$ used to characterize the local pixel neighborhoods. Note that the order of processing the image positions (u, v) in the **for all** loops in **Pass 1** and **Pass 2** is completely arbitrary. In particular, positions could be processed simultaneously, so the algorithm may be easily parallelized (and thereby accelerated).

```
1:  Thin(I_b, i_max)
        Input: I_b, binary image with background = 0, foreground > 0;
        i_max, max. number of iterations. Returns the number of iterations
        performed and modifies I_b.
2:      (M, N) ← Size(I_b)
3:      Create a binary map D: M × N ↦ {0, 1}
4:      i ← 0
5:      do
6:          n_d ← ThinOnce(I_b, D)
7:          i ← i + 1
8:      while (n_d > 0 ∧ i < i_max) ▷ do ... while more deletions required
9:      return i
```

```
10: ThinOnce(I_b, D)
        Pass 1:
11:     n_1 ← 0                                          ▷ deletion counter
12:     for all image positions (u, v) ∈ M × N do
13:         D(u, v) ← 0
14:         if I_b(u, v) > 0 then
15:             N ← GetNeighborhood(I_b, u, v)
16:             if R_1(N) then                           ▷ see Eq. 9.27
17:                 D(u, v) ← 1                          ▷ mark pixel (u, v) for deletion
18:                 n_1 ← n_1 + 1
19:     if n_1 > 0 then                                  ▷ at least 1 deletion required
20:         for all image positions (u, v) ∈ M × N do
21:             I_b(u, v) ← I_b(u, v) − D(u, v)          ▷ delete all marked pixels
        Pass 2:
22:     n_2 ← 0
23:     for all image positions (u, v) ∈ M × N do
24:         D(u, v) ← 0
25:         if I_b(u, v) > 0 then
26:             N ← GetNeighborhood(I_b, u, v)
27:             if R_2(N) then                           ▷ see Eq. 9.28
28:                 D(u, v) ← 1                          ▷ mark pixel (u, v) for deletion
29:                 n_2 ← n_2 + 1
30:     if n_2 > 0 then                                  ▷ at least 1 deletion required
31:         for all image positions (u, v) ∈ M × N do
32:             I_b(u, v) ← I_b(u, v) − D(u, v)          ▷ delete all marked pixels
33:     return n_1 + n_2
```

```
34: GetNeighborhood(I_b, u, v)
35:     N_0 ← I_b(u + 1, v),      N_1 ← I_b(u + 1, v − 1)
36:     N_2 ← I_b(u, v − 1),      N_3 ← I_b(u − 1, v − 1)
37:     N_4 ← I_b(u − 1, v),      N_5 ← I_b(u − 1, v + 1)
38:     N_6 ← I_b(u, v + 1),      N_7 ← I_b(u + 1, v + 1)
39:     return (N_0, N_1, ..., N_7)
```

40: $R_1(\mathsf{N}) := [2 \le B(\mathsf{N}) \le 6] \wedge [C(\mathsf{N}) = 1] \wedge [N_6 \cdot N_0 \cdot N_2 = 0] \wedge [N_4 \cdot N_6 \cdot N_0 = 0]$

41: $R_2(\mathsf{N}) := [2 \le B(\mathsf{N}) \le 6] \wedge [C(\mathsf{N}) = 1] \wedge [N_0 \cdot N_2 \cdot N_4 = 0] \wedge [N_2 \cdot N_4 \cdot N_6 = 0]$

42: $B(\mathsf{N}) := \sum_{i=0}^{7} N_i, \qquad C(\mathsf{N}) := \sum_{i=0}^{7} N_i \cdot [N_i - N_{(i+1) \bmod 8}]$

Alg. 9.3 Thinning algorithm by Zhang und Suen (accelerated version of Alg. 9.2). This algorithm employs a pre-calculated table of "deletion codes" (Q). Procedure GetNeighborhood() has been replaced by GetNeighborhoodIndex(), which does not return the neighboring pixel values themselves but the associated 8-bit index c with possible values in $0, \ldots, 255$ (see Fig. 9.22). For completeness, the calculation of table Q is included in procedure MakeDeletionCodeTable(), although this table is fixed and may be simply defined as a constant array (see Prog. 9.1).

1: **ThinFast**$(I_\mathrm{b}, i_\mathrm{max})$
Input: I_b, binary image with background = 0, foreground > 0; i_max, max. number of iterations. Returns the number of iterations performed and modifies I_b.
2: $(M, N) \leftarrow \mathsf{Size}(I_\mathrm{b})$
3: $\mathsf{Q} \leftarrow \mathsf{MakeDeletionCodeTable}()$
4: Create a binary map $\mathsf{D}\colon M \times N \mapsto \{0, 1\}$
5: $i \leftarrow 0$
6: **do**
7: $n_\mathrm{d} \leftarrow \mathsf{ThinOnce}(I_\mathrm{b}, \mathsf{D})$
8: **while** $(n_\mathrm{d} > 0 \wedge i < i_\mathrm{max})$ ▷ do ... while more deletions required
9: **return** i

10: **ThinOnceFast**$(I_\mathrm{b}, \mathsf{D})$ ▷ performs a single thinning iteration
11: $n_\mathrm{d} \leftarrow 0$ ▷ number of deletions in both passes
12: **for** $p \leftarrow 1, 2$ **do** ▷ pass counter (2 passes)
13: $n \leftarrow 0$ ▷ number of deletions in current pass
14: **for all** image positions (u, v) **do**
15: $\mathsf{D}(u, v) \leftarrow 0$
16: **if** $I_\mathrm{b}(u, v) = 1$ **then** ▷ $I_\mathrm{b}(u, v) = P$
17: $c \leftarrow \mathsf{GetNeighborhoodIndex}(I_\mathrm{b}, u, v)$
18: $q \leftarrow \mathsf{Q}(c)$ ▷ $q \in \{0, 1, 2, 3\} = \{00_\mathrm{b}, 01_\mathrm{b}, 10_\mathrm{b}, 11_\mathrm{b}\}$
19: **if** $(p$ and $q) \neq 0$ **then** ▷ bitwise 'and' operation
20: $\mathsf{D}(u, v) \leftarrow 1$ ▷ mark pixel (u, v) for deletion
21: $n \leftarrow n + 1$
22: **if** $n > 0$ **then** ▷ at least 1 deletion is required
23: $n_\mathrm{d} \leftarrow n_\mathrm{d} + n$
24: **for all** image positions (u, v) **do**
25: $I_\mathrm{b}(u, v) \leftarrow I_\mathrm{b}(u, v) - \mathsf{D}(u, v)$ ▷ delete all marked pixels
26: **return** n_d

27: **GetNeighborhoodIndex**(I_b, u, v)
28: $N_0 \leftarrow I_\mathrm{b}(u + 1, v)$, $N_1 \leftarrow I_\mathrm{b}(u + 1, v - 1)$
29: $N_2 \leftarrow I_\mathrm{b}(u, v - 1)$, $N_3 \leftarrow I_\mathrm{b}(u - 1, v - 1)$
30: $N_4 \leftarrow I_\mathrm{b}(u - 1, v)$, $N_5 \leftarrow I_\mathrm{b}(u - 1, v + 1)$
31: $N_6 \leftarrow I_\mathrm{b}(u, v + 1)$, $N_7 \leftarrow I_\mathrm{b}(u + 1, v + 1)$
32: $c \leftarrow N_0 + N_1 \cdot 2 + N_2 \cdot 4 + N_3 \cdot 8 + N_4 \cdot 16 + N_5 \cdot 32 + N_6 \cdot 64 + N_7 \cdot 128$
33: **return** c ▷ $c \in [0, 255]$

34: **MakeDeletionCodeTable**()
35: Create maps $\mathsf{Q}\colon [0, 255] \mapsto \{0, 1, 2, 3\}$, $\mathsf{N}\colon [0, 7] \mapsto \{0, 1\}$
36: **for** $i \leftarrow 0, \ldots, 255$ **do** ▷ list all possible neighborhoods
37: **for** $k \leftarrow 0, \ldots, 7$ **do** ▷ check neighbors $0, \ldots, 7$
38: $\mathsf{N}(k) \leftarrow \begin{cases} 1 & \text{if } (i \text{ and } 2^k) \neq 0 \\ 0 & \text{otherwise} \end{cases}$ ▷ test the k^{th} bit of i
39: $q \leftarrow 0$
40: **if** $R_1(\mathsf{N})$ **then** ▷ see Alg. 9.2, line 40
41: $q \leftarrow q + 1$ ▷ set bit 0 of q
42: **if** $R_2(\mathsf{N})$ **then** ▷ see Alg. 9.2, line 41
43: $q \leftarrow q + 2$ ▷ set bit 1 of q
44: $\mathsf{Q}(i) \leftarrow q$ ▷ $q \in \{0, 1, 2, 3\} = \{00_\mathrm{b}, 01_\mathrm{b}, 10_\mathrm{b}, 11_\mathrm{b}\}$
45: **return** Q

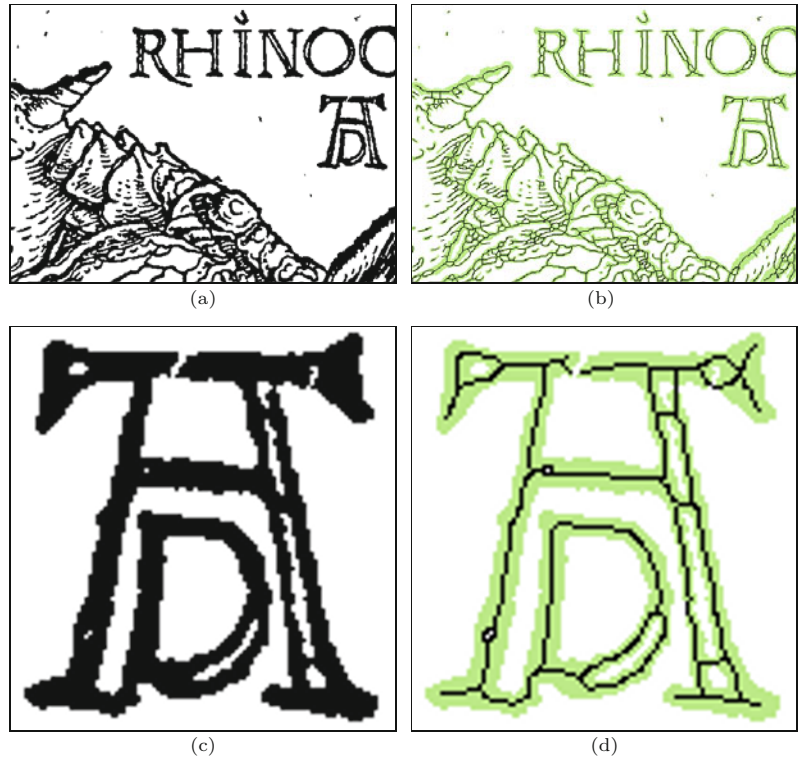


Fig. 9.21
Thinning a binary image (Alg. 9.2 or 9.3). Original image with enlarged detail (a, c) and results after thinning (b, d). The original foreground pixels are marked green, the resulting pixels are black.

Prog. 9.1
Java definition for the "deletion code" table `Q` (see Fig. 9.22).

```
1  static final byte[] Q = {
2    0, 0, 0, 3, 0, 0, 3, 3, 0, 0, 0, 0, 3, 0, 3, 3,
3    0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 3, 1,
4    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
5    3, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 3, 1,
6    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
7    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
8    3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
9    3, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 1, 0,
10   0, 3, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 3,
11   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
12   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
13   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
14   3, 3, 0, 3, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2,
15   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
16   3, 3, 0, 3, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0,
17   3, 2, 0, 2, 0, 0, 0, 0, 3, 2, 0, 0, 1, 0, 0, 0
18 };
```

9.4.3 Java Implementation

The complete Java source code for the morphological operations on binary images is available online as part of the `imagingbook`[4] library.

[4] Package `imagingbook.pub.morphology`.

Fig. 9.22
"Deletion codes" for the 256 possible binary 8-neighborhoods tabulated in map Q(c) of Alg. 9.3. □ = 0 and ■ = 1 denote background and foreground pixels, respectively. The 2-bit codes are color coded as indicated at the bottom.

`BinaryMorphologyFilter` class

This class implements several morphological operators for binary images of type `ByteProcessor`. It defines the sub-classes `Box` and `Disk` with different structuring elements. The class provides the following constructors:

`BinaryMorphologyFilter()`
Creates a morphological filter with a (default) structuring element of size 3×3 as depicted in Fig. 9.11(b).

`BinaryMorphologyFilter(int[][] H)`
Creates a morphological filter with a structuring element specified by the 2D array `H`, which may contain 0/1 values only (all values > 0 are treated as 1).

`BinaryMorphologyFilter.Box(int rad)`
Creates a morphological filter with a square structuring element of radius `rad` ≥ 1 and side length $2 \cdot$ `rad` $+ 1$ pixels.

`BinaryMorphologyFilter.Disk(double rad)`
Creates a morphological filter with a disk-shaped structuring element with radius `rad` ≥ 1 and diameter $2 \cdot \text{round}(\texttt{rad}) + 1$ pixels.

The key methods[5] of `BinaryMorphologyFilter` are:

`void applyTo(ByteProcessor I, OpType op)`
Destructively applies the morphological operator `op` to the image `I`. Possible arguments for `op` are `Dilate`, `Erode`, `Open`, `Close`, `Outline`, `Thin`.

`void dilate(ByteProcessor I)`
Performs (destructive) *dilation* on the binary image `I` with the initial structuring element of this filter.

`void erode(ByteProcessor I)`
Performs (destructive) *erosion* on the binary image `I`.

`void open(ByteProcessor I)`
Performs (destructive) *opening* on the binary image `I`.

`void close(ByteProcessor I)`
Performs (destructive) *closing* on the binary image `I`.

`void outline(ByteProcessor I)`
Performs a (destructive) *outline* operation on the binary image `I` using a 3×3 structuring element (see Sec. 9.2.7).

`void thin(ByteProcessor I)`
Performs a (destructive) *thinning* operation on the binary image `I` using a 3×3 structuring element (with at most $i_{\max} = 1500$ iterations, see Alg. 9.3).

`void thin(ByteProcessor I, int iMax)`
Performs a thinning operation with at most `iMax` iterations (see Alg. 9.3).

`int thinOnce(ByteProcessor I)`
Performs a single iteration of the thinning operation and returns the number of pixel deletions (see Alg. 9.3).

The methods listed here *always* treat image pixels with value 0 as background and values > 0 as foreground. Unlike ImageJ's built-in implementation of morphological operations (described in Sec. 9.4.4), the display lookup table (LUT, typically only used for display purposes) of the image is *not* taken into account at all.

[5] See the online documentation for additional methods.

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.ByteProcessor;
4  import ij.process.ImageProcessor;
5  import imagingbook.pub.morphology.BinaryMorphologyFilter;
6  import imagingbook.pub.morphology.BinaryMorphologyFilter.
       OpType;
7
8  public class Bin_Dilate_Disk_Demo implements PlugInFilter {
9    static double radius = 5.0;
10   static OpType op = OpType.Dilate; // Erode, Open, Close, ...
11
12   public int setup(String arg, ImagePlus imp) {
13     return DOES_8G;
14   }
15
16   public void run(ImageProcessor ip) {
17     BinaryMorphologyFilter bmf =
18             new BinaryMorphologyFilter.Disk(radius);
19     bmf.applyTo((ByteProcessor) ip, op);
20   }
21 }
```

Prog. 9.2
Example for using class `BinaryMorphologyFilter` (see Sec. 9.4.3) inside a ImageJ plugin. The actual filter operator is instantiated in line 18 and subsequently (in line 19) applied to the image `ip` of type `ByteProcessor`. Available operations (`OpType`) are `Dilate`, `Erode`, `Open`, `Close`, `Outline` and `Thin`. Note that the results depend strictly on the pixel values of the input image, with values 0 taken as background and values > 0 taken as foreground. The display lookup-table (LUT) is irrelevant.

The example in Prog. 9.2 shows the use of class `BinaryMorphologyFilter` in a complete ImageJ plugin that performs dilation with a disk-shaped structuring element of radius 5 (pixel units). Other examples can be found in the online code repository.

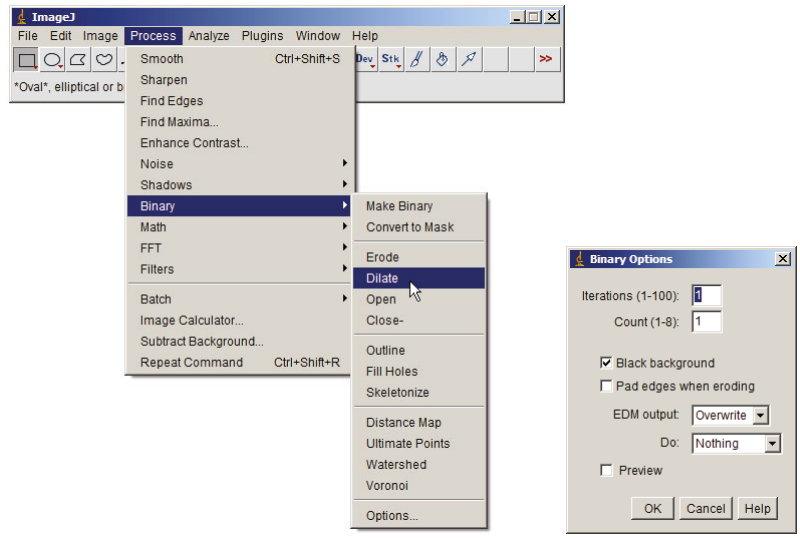
9.4.4 Built-in Morphological Operations in ImageJ

Apart from the implementation described in the previous section, the ImageJ API provides built-in methods for basic morphological operations, such as `dilate()` and `erode()`. These methods use a 3×3 structuring element (analogous to Fig. 9.11(b)) and are only defined for images of type `ByteProcessor` and `ColorProcessor`. In the case of RGB color images (`ColorProcessor`) the morphological operation is applied individually to the three color channels. All these and other morphological operations can be applied interactively through ImageJ's Process ▷ Binary menu (see Fig. 9.23(a)).

Note that ImageJ's `dilate()` and `erode()` methods use the current settings of display lookup table (LUT) to discriminate between background and foreground pixels. Thus the results of morphological operations depend not only on the stored pixel values but how they are being displayed (in addition to the settings in Process ▷ Binary ▷ Options..., see Fig. 9.23(b)).[6] It is therefore recommended to use the methods (defined for `ByteProcessor` only)

```
dilate(int count, int background),
erode(int count, int background)
```

[6] These dependencies may be quite confusing because the same program will produce different results under different user setups.



(a) (b)

Fig. 9.23 Morphological operations in ImageJ's built-in standard menu Process ▷ Binary (a) and optional settings with Process ▷ Binary ▷ Options... (b). The choice "Black background" specifies if background pixels are bright or dark, which is taken into account by ImageJ's morphological operations.

instead, since they provide explicit control of the background pixel value and are thus independent from other settings. ImageJ's `ByteProcessor` class defines additional methods for morphological operations on binary images, such as `outline()` and `skeletonize()`. The method `outline()` implements the extraction of region boundaries using an 8-neighborhood structuring element, as described in Sec. 9.2.7. The method `skeletonize()`, on the other hand, implements a thinning process similar to Alg. 9.3.

9.5 Grayscale Morphology

Morphological operations are not confined to binary images but are also for intensity (grayscale) images. In fact, the definition of grayscale morphology is a *generalization* of binary morphology, with the binary OR and AND operators replaced by the arithmetic MAX and MIN operators, respectively. As a consequence, procedures designed for grayscale morphology can also perform binary morphology (but not the other way around).[7] In the case of color images, the grayscale operations are usually applied individually to each color channel.

9.5.1 Structuring Elements

Unlike in the binary scheme, the structuring elements for grayscale morphology are not defined as point sets but as real-valued 2D functions, that is,

$$H(i,j) \in \mathbb{R}, \qquad \text{for } (i,j) \in \mathbb{Z}^2. \tag{9.30}$$

The values in H may be negative or zero. Notice, however, that, in contrast to linear convolution (Sec. 5.3.1), zero elements in grayscale

[7] ImageJ provides a single implementation of morphological operations that handles both binary and grayscale images (see Sec. 9.4.4).

morphology generally *do* contribute to the result.[8] The design of structuring elements for grayscale morphology must therefore distinguish explicitly between cells containing the value 0 and empty ("don't care") cells, for example,

$$\begin{array}{|c|c|c|}\hline 0&1&0\\\hline 1&2&1\\\hline 0&1&0\\\hline\end{array} \neq \begin{array}{|c|c|c|}\hline &1&\\\hline 1&2&1\\\hline &1&\\\hline\end{array}\ . \tag{9.31}$$

9.5.2 Dilation and Erosion

The result of grayscale *dilation* $I \oplus H$ is defined as the *maximum* of the values in H added to the values of the current subimage of I, that is,

$$(I \oplus H)(u,v) = \max_{(i,j)\in H} \big(I(u+i, v+j) + H(i,j)\big). \tag{9.32}$$

Similarly, the result of grayscale *erosion* is the *minimum* of the differences,

$$(I \ominus H)(u,v) = \min_{(i,j)\in H} \big(I(u+i, v+j) - H(i,j)\big). \tag{9.33}$$

Figures 9.24 and 9.25 demonstrate the basic process of grayscale dilation and erosion, respectively, on a simple example.

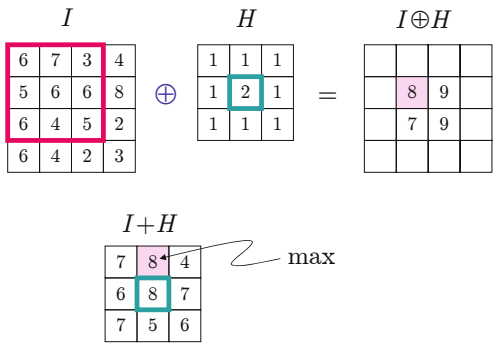


Fig. 9.24
Grayscale dilation $I \oplus H$. The 3×3 pixel structuring element H is placed on the image I in the upper left position. Each value of H is added to the corresponding element of I; the intermediate result $(I + H)$ for this particular position is shown below. Its maximum value $8 = 7 + 1$ is inserted into the result $(I \oplus H)$ at the current position of the filter origin. The results for three other filter positions are also shown.

In general, either operation may produce *negative* results that must be considered if the range of pixel values is restricted, for example, by clamping the results (see Ch. 4, Sec. 4.1.2). Some examples of grayscale dilation and erosion on natural images using disk-shaped structuring elements of various sizes are shown in Fig. 9.26. Figure 9.28 demonstrates the same operations with some freely designed structuring elements.

9.5.3 Grayscale Opening and Closing

Opening and closing on grayscale images are defined, identical to the binary case (Eqns. (9.21) and (9.22)), as operations composed

[8] While a zero coefficient in a linear convolution matrix simply means that the corresponding image pixel is ignored.

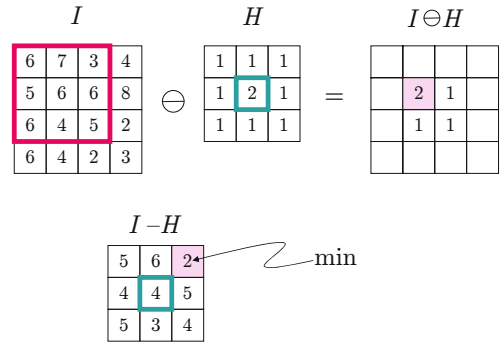


Fig. 9.25
Grayscale erosion $I \ominus H$. The 3×3 pixel structuring element H is placed on the image I in the upper left position. Each value of H is subtracted from the corresponding element of I; the intermediate result $(I - H)$ for this particular position is shown below. Its minimum value $3-1 = 2$ is inserted into the result $(I \ominus H)$ at the current position of the filter origin. The results for three other filter positions are also shown.

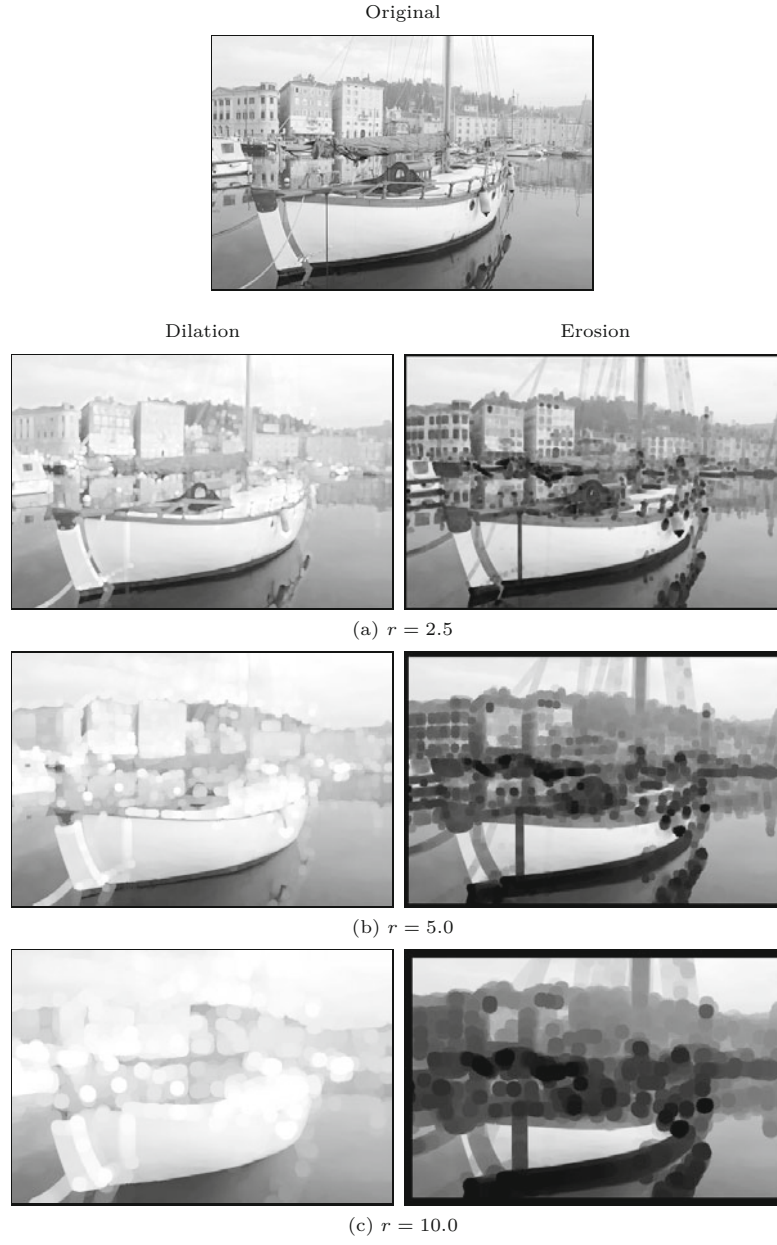


Fig. 9.26
Grayscale dilation and erosion with disk-shaped structuring elements. The radius r of the structuring element is 2.5 (a), 5.0 (b), and 10.0 (c).

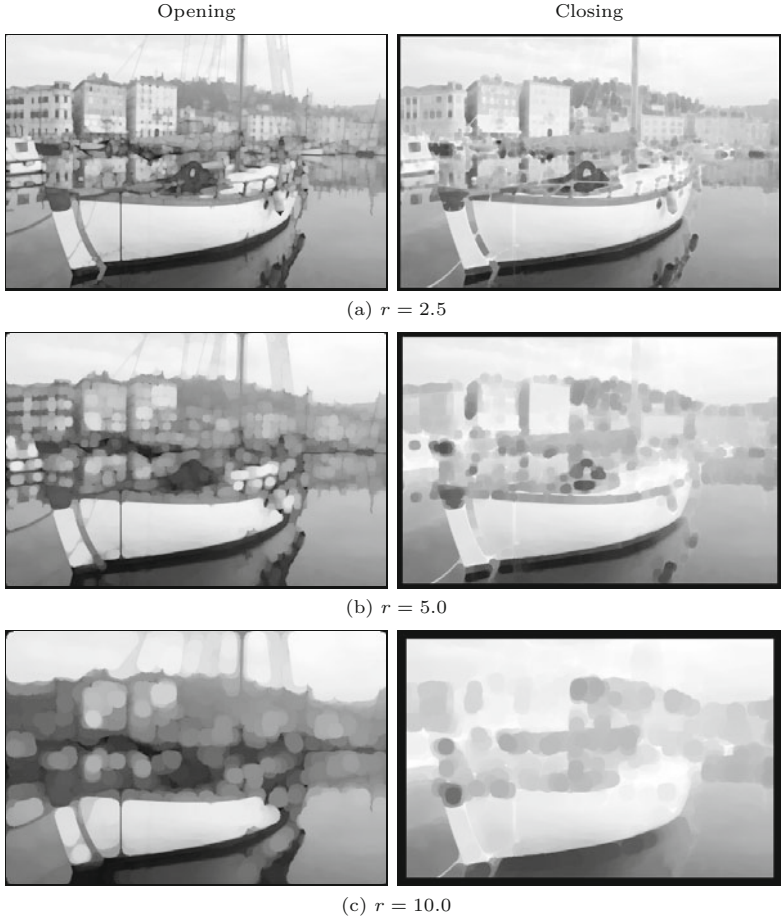


Fig. 9.27
Grayscale opening and closing with disk-shaped structuring elements. The radius r of the structuring element is 2.5 (a), 5.0 (b), and 10.0 (c).

of dilation and erosion with the same structuring element. Some examples are shown in Fig. 9.27 for disk-shaped structuring elements and in Fig. 9.29 for various nonstandard structuring elements. Notice that interesting effects can be obtained, particularly from structuring elements resembling the shape of brush or other stroke patterns.

As mentioned in Sec. 9.4.4, the morphological operations available in ImageJ can be applied to binary images as well as grayscale images. In addition, several additional plugins and complete morphological packages are available online,[9] including the morphology operators by Gabriel Landini and the Grayscale Morphology package by Dimiter Prodanov, which allows structuring elements to be interactively specified (a modified version was used for some examples in this chapter).

9.6 Exercises

Exercise 9.1. Manually calculate the results of dilation and erosion for the following image I and the structuring elements H_1 and H_2:

[9] See http://rsb.info.nih.gov/ij/plugins/.

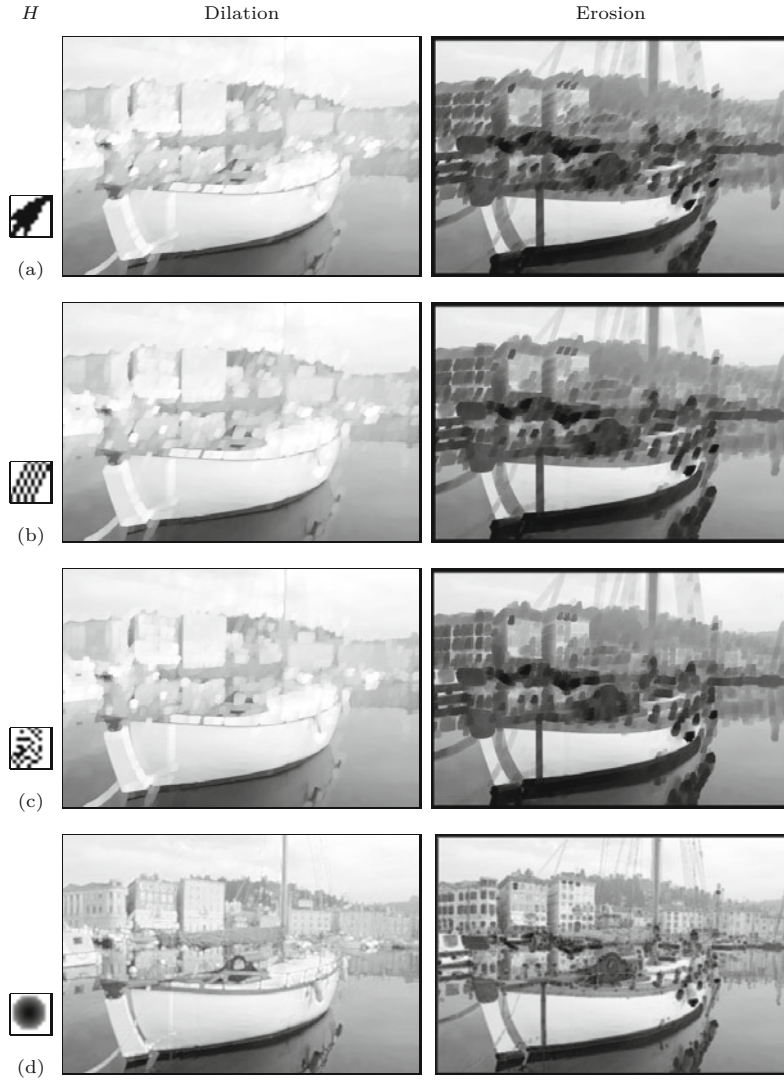
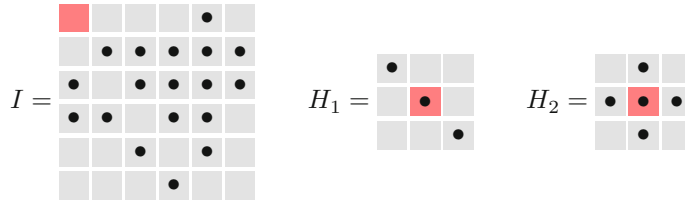


Fig. 9.28 Grayscale dilation and erosion with various free-form structuring elements.



Exercise 9.2. Assume that a binary image I contains unwanted foreground spots with a maximum diameter of 5 pixels that should be removed without damaging the remaining structures. Design a suitable morphological procedure, and evaluate its performance on appropriate test images.

Exercise 9.3. Investigate if the results of the thinning operation described in Alg. 9.2 (and implemented by the `thin()` method of class `BinaryMorphologyFilter`) are invariant against rotating the image

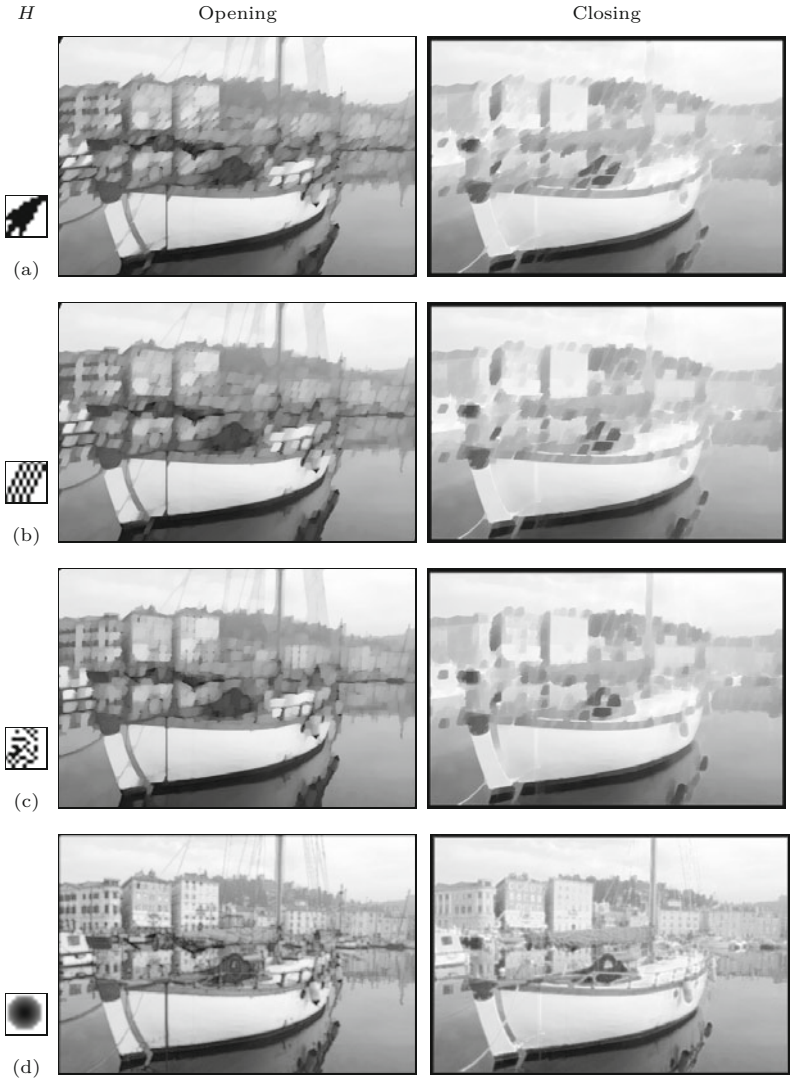
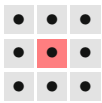
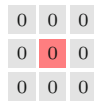


Fig. 9.29
Grayscale opening and closing with various free-form structuring elements.

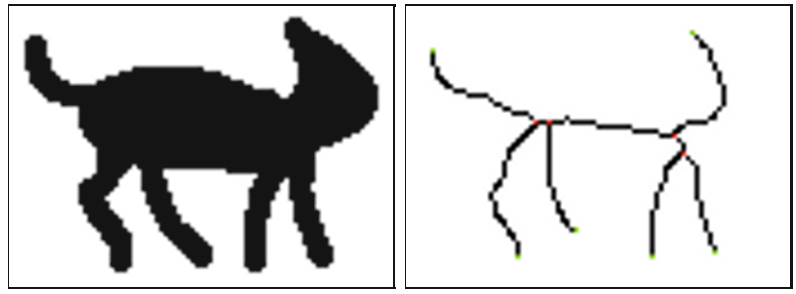
by 90° and horizontal or vertical mirroring. Use appropriate test images to see if the results are identical.

Exercise 9.4. Show that, in the special case of the structuring elements with the contents

 for *binary* and  for *grayscale* images,

dilation is equivalent to a 3×3 pixel maximum filter and erosion is equivalent to a 3×3 pixel minimum filter (see Ch. 5, Sec. 5.4.1).

Exercise 9.5. Thinning can be applied to extract the "skeleton" of a binary region, which in turn can be used to characterize the shape of the region. A common approach is to partition the skeleton into a graph, consisting of nodes and connecting segments, as a



(a) (b)

Fig. 9.30
Segmentation of a region skeleton. Original binary image (a) and the skeleton obtained by thinning (b). Terminal nodes are marked green, connecting (inner) nodes are marked red.

shape representation (see Fig. 9.30 for an example). Use ImageJ's `skeletonize()` method or the `thin()` methode of class `BinaryMorphologyFilter` (see Sec. 9.4.3) to generate the skeleton, then locate and mark the connecting and terminal nodes of this structure. Define precisely the properties of each type of node and use this definition in your implementation. Test your implementation on different examples. How would you generally judge the robustness of this approach as a 2D shape representation?