10

Regions in Binary Images

In a binary image, pixels can take on exactly one of two values. These values are often thought of as representing the "foreground" and "background" in the image, even though these concepts often are not applicable to natural scenes. In this chapter we focus on connected regions in images and how to isolate and describe such structures.

Let us assume that our task is to devise a procedure for finding the number and type of objects contained in an image as shown in Fig. 10.1. As long as we continue to consider each pixel in isolation, we will not be able to determine how many objects there are overall in the image, where they are located, and which pixels belong to which objects. Therefore our first step is to find each object by grouping together all the pixels that belong to it. In the simplest case, an object is a group of touching foreground pixels, that is, a connected *binary region* or "component".

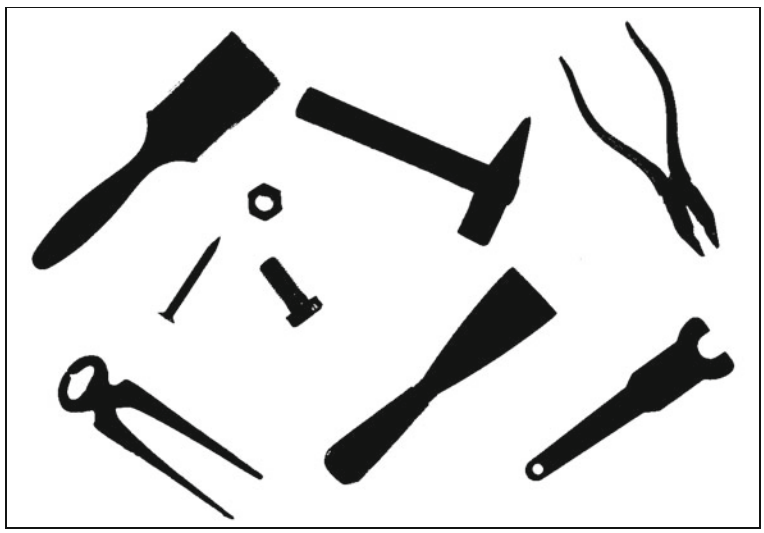


Fig. 10.1
Binary image with nine components. Each component corresponds to a connected region of (black) foreground pixels.

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_10

10.1 Finding Connected Image Regions

In the search for binary regions, the most important tasks are to find out which pixels belong to which regions, how many regions are in the image, and where these regions are located. These steps usually take place as part of a process called *region labeling* or *region coloring*. During this process, neighboring pixels are pieced together in a stepwise manner to build regions in which all pixels within that region are assigned a unique number ("label") for identification. In the following sections, we describe two variations on this idea. In the first method, region marking through *flood filling*, a region is filled in all directions starting from a single point or "seed" within the region. In the second method, *sequential region marking*, the image is traversed from top to bottom, marking regions as they are encountered. In Sec. 10.2.2, we describe a third method that combines two useful processes, region labeling and contour finding, in a single algorithm.

Independent of which of these methods we use, we must first settle on either the 4- or 8-connected definition of neighboring (see Ch. 9, Fig. 9.5) for determining when two pixels are "connected" to each other, since under each definition we can end up with different results. In the following region-marking algorithms, we use the following convention: the original binary image $I(u, v)$ contains the values 0 and 1 to mark the *background* and *foreground*, respectively; any other value is used for numbering (labeling) the regions, that is, the pixel values are

$$I(u,v) = \begin{cases} 0 & \text{background,} \\ 1 & \text{foreground,} \\ 2, 3, \ldots & \text{region label.} \end{cases} \tag{10.1}$$

10.1.1 Region Labeling by Flood Filling

The underlying algorithm for region marking by *flood filling* is simple: search for an unmarked foreground pixel and then fill (visit and mark) all the rest of the neighboring pixels in its region. This operation is called a "flood fill" because it is as if a flood of water erupts at the start pixel and flows out across a flat region. There are various methods for carrying out the fill operation that ultimately differ in how to select the coordinates of the next pixel to be visited during the fill. We present three different ways of performing the FloodFill() procedure: a recursive version, an iterative *depth-first* version, and an iterative *breadth-first* version (see Alg. 10.1):

A. Recursive Flood Filling: The recursive version (Alg. 10.1, line 8) does not make use of explicit data structures to keep track of the image coordinates but uses the local variables that are implicitly allocated by recursive procedure calls.[1] Within each region, a tree structure, rooted at the starting point, is defined by the neighborhood relation between pixels. The recursive step corresponds to a *depth-first traversal* [54] of this tree and results

[1] In Java, and similar imperative programming languages such as C and C++, local variables are automatically stored on the *call stack* at each procedure call and restored from the stack when the procedure returns.

Alg. 10.1 Region marking by *flood filling*. The binary input image I uses the value 0 for background pixels and 1 for foreground pixels. Unmarked foreground pixels are searched for, and then the region to which they belong is filled. Procedure FloodFill() is defined in three different versions: *recursive*, emphdepth-first and *breadth-first*.

```
1:  RegionLabeling(I)
      Input: I, an integer-valued image with initial values 0 = background, 1 = foreground. Returns nothing but modifies the image I.
2:    label ← 2                                  ▷ value of the next label to be assigned
3:    for all image coordinates u, v do
4:      if I(u, v) = 1 then                      ▷ a foreground pixel
5:        FloodFill(I, u, v, label)              ▷ any of the 3 versions below
6:        label ← label + 1.
7:    return
8:  FloodFill(I, u, v, label)                    ▷ Recursive Version
9:    if u, v is within the image boundaries and I(u, v) = 1 then
10:     I(u, v) ← label
11:     FloodFill(I, u+1, v, label)              ▷ recursive call to FloodFill()
12:     FloodFill(I, u, v+1, label)
13:     FloodFill(I, u, v−1, label)
14:     FloodFill(I, u−1, v, label)
15:   return
16: FloodFill(I, u, v, label)                    ▷ Depth-First Version
17:   S ← ( )                                    ▷ create an empty stack S
18:   S ← (u, v) ⌢ S                             ▷ push seed coordinate (u, v) onto S
19:   while S ≠ ( ) do                           ▷ while S is not empty
20:     (x, y) ← GetFirst(S)
21:     S ← Delete((x, y), S)                    ▷ pop first coordinate off the stack
22:     if x, y is within the image boundaries and I(x, y) = 1 then
23:       I(x, y) ← label
24:       S ← (x+1, y) ⌢ S                       ▷ push (x+1, y) onto S
25:       S ← (x, y+1) ⌢ S                       ▷ push (x, y+1) onto S
26:       S ← (x, y−1) ⌢ S                       ▷ push (x, y−1) onto S
27:       S ← (x−1, y) ⌢ S                       ▷ push (x−1, y) onto S
28:   return
29: FloodFill(I, u, v, label)                    ▷ Breadth-First Version
30:   Q ← ( )                                    ▷ create an empty queue Q
31:   Q ← Q ⌢ (u, v)                             ▷ append seed coordinate (u, v) to Q
32:   while Q ≠ ( ) do                           ▷ while Q is not empty
33:     (x, y) ← GetFirst(Q)
34:     Q ← Delete((x, y), Q)                    ▷ dequeue first coordinate
35:     if x, y is within the image boundaries and I(x, y) = 1 then
36:       I(x, y) ← label
37:       Q ← Q ⌢ (x+1, y)                       ▷ append (x+1, y) to Q
38:       Q ← Q ⌢ (x, y+1)                       ▷ append (x, y+1) to Q
39:       Q ← Q ⌢ (x, y−1)                       ▷ append (x, y−1) to Q
40:       Q ← Q ⌢ (x−1, y)                       ▷ append (x−1, y) to Q
41:   return
```

in very short and elegant program code. Unfortunately, since the maximum depth of the recursion—and thus the size of the required stack memory—is proportional to the size of the region, stack memory is quickly exhausted. Therefore this method is risky and really only practical for very small images.

B. Iterative Flood Filling (*depth-first*): Every recursive algorithm can also be reformulated as an iterative algorithm (Alg. 10.1, line 16) by implementing and managing its own *stacks*. In this case, the stack records the "open" (that is, the adjacent but not yet visited) elements. As in the recursive version (A), the corresponding tree of pixels is traversed in *depth-first* order. By making use of its own dedicated stack (which is created in the much larger *heap* memory), the depth of the tree is no longer limited to the size of the call stack.

C. Iterative Flood Filling (*breadth-first*): In this version, pixels are traversed in a way that resembles an expanding wave front propagating out from the starting point (Alg. 10.1, line 29). The data structure used to hold the as yet unvisited pixel coordinates is in this case a *queue* instead of a stack, but otherwise it is identical to version B.

Java implementation

The recursive version (A) of the algorithm is an exact blueprint of the Java implementation. However, a normal Java runtime environment does not support more than about 10,000 recursive calls of the FloodFill() procedure (Alg. 10.1, line 8) before the memory allocated for the call stack is exhausted. This is only sufficient for relatively small images with fewer than approximately 200×200 pixels.

Program 10.1 (line 1–17) gives the complete Java implementation for both variants of the iterative FloodFill() procedure. The stack (S) in the *depth-first* Version (B) and the queue (Q) in the *breadth-first* variant (C) are both implemented as instances of type `LinkedList`.[2] `<Point>` is specified as a type parameter for both generic container classes so they can only contain objects of type `Point`.[3]

Figure 10.2 illustrates the progress of the region marking in both variants within an example region, where the start point (i.e., seed point), which would normally lie on a contour edge, has been placed arbitrarily within the region in order to better illustrate the process. It is clearly visible that the *depth-first* method first explores *one* direction (in this case horizontally to the left) completely (that is, until it reaches the edge of the region) and only then examines the remaining directions. In contrast the *breadth-first* method markings proceed outward, layer by layer, equally in all directions.

Due to the way exploration takes place, the memory requirement of the *breadth-first* variant of the *flood-fill* version is generally much lower than that of the *depth-first* variant. For example, when flood filling the region in Fig. 10.2 (using the implementation given Prog. 10.1), the stack in the *depth-first* variant grows to a maximum of 28,822 elements, while the queue used by the *breadth-first* variant never exceeds a maximum of 438 nodes.

[2] The class `LinkedList` is part of Java's *collections framework*.

[3] Note that the depth-first and breadth-first implementations in Prog. 10.1 typically run slower than the recursive version described in Alg. 10.1, since they allocate (and immediately discard) large numbers of `Point` objects. A better solution is to use a queue or stack with elements of a primitive type (e.g., `int`) instead. See also Exercise 10.3.

Depth-first version (using a *stack*):

```
1 void floodFill(int u, int v, int label) {
2   Deque<Point> S = new LinkedList<Point>(); // stack S
3   S.push(new Point(u, v));
4   while (!S.isEmpty()) {
5     Point p = S.pop();
6     int x = p.x;
7     int y = p.y;
8     if ((x >= 0) && (x < width) && (y >= 0) && (y < height)
9         && ip.getPixel(x, y) == 1) {
10      ip.putPixel(x, y, label);
11      S.push(new Point(x + 1, y));
12      S.push(new Point(x, y + 1));
13      S.push(new Point(x, y - 1));
14      S.push(new Point(x - 1, y));
15    }
16  }
17 }
```

Breadth-first version (using a *queue*):

```
18 void floodFill(int u, int v, int label) {
19   Queue<Point> Q = new LinkedList<Point>(); // queue Q
20   Q.add(new Point(u, v));
21   while (!Q.isEmpty()) {
22     Point p = Q.remove(); // get the next point to process
23     int x = p.x;
24     int y = p.y;
25     if ((x >= 0) && (x < width) && (y >= 0) && (y < height)
26         && ip.getPixel(x, y) == 1) {
27       ip.putPixel(x, y, label);
28       Q.add(new Point(x + 1, y));
29       Q.add(new Point(x, y + 1));
30       Q.add(new Point(x, y - 1));
31       Q.add(new Point(x - 1, y));
32     }
33   }
34 }
```

Prog. 10.1
Java implementation of iterative flood filling (*depth-first* and *breadth-first* variants).

10.1.2 Sequential Region Labeling

Sequential region marking is a classical, nonrecursive technique that is known in the literature as "region labeling". The algorithm consists of two steps: (1) preliminary labeling of the image regions and (2) resolving cases where more than one label occurs (i.e., has been assigned in the previous step) in the same connected region. Even though this algorithm is relatively complex, especially its second stage, its moderate memory requirements make it a good choice under limited memory conditions. However, this is not a major issue on modern computers and thus, in terms of overall efficiency, sequential labeling offers no clear advantage over the simpler methods described earlier. The sequential technique is nevertheless interesting (not only from a historic perspective) and inspiring. The complete process is summarized in Alg. 10.2, with the following main steps:

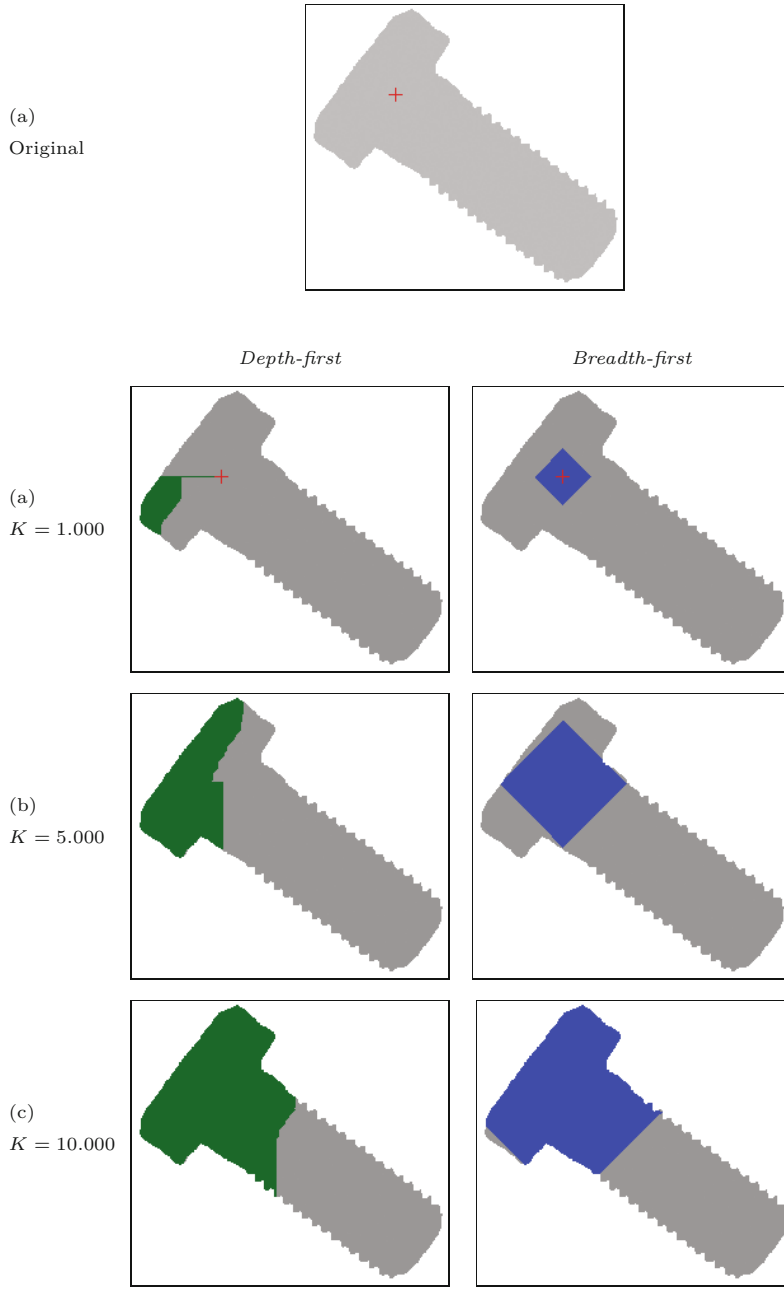


Fig. 10.2
Iterative *flood filling*—comparison between the *depth-first* and *breadth-first* approach. The starting point, marked + in the top two image (a), was arbitrarily chosen. Intermediate results of the *flood fill* process after 1000 (a), 5000 (b), and 10,000 (c) marked pixels are shown. The image size is 250 × 242 pixels.

Step 1: Initial labeling

In the first stage of region labeling, the image is traversed from top left to bottom right sequentially to assign a preliminary label to every foreground pixel. Depending on the definition of neighborhood (either 4- or 8-connected) used, the following neighbors in the direct vicinity of each pixel must be examined (× marks the current pixel at the position (u, v)):

Alg. 10.2 Sequential region labeling. The binary input image I uses the value $I(u,v) = 0$ for background pixels and $I(u,v) = 1$ for foreground (region) pixels. The resulting labels have the values $2, \ldots, label - 1$.

SequentialLabeling(I)
Input: I, an integer-valued image with initial values $0 = background$, $1 = foreground$. Returns nothing but modifies the image I.

Step 1 – Assign initial labels:

$(M, N) \leftarrow \mathsf{Size}(I)$
$label \leftarrow 2$ ▷ value of the next label to be assigned
$\mathsf{C} \leftarrow (\,)$ ▷ empty list of label collisions
for $v \leftarrow 0, \ldots, N-1$ **do**
  for $u \leftarrow 0, \ldots, M-1$ **do**
    if $I(u,v) = 1$ **then** ▷ $I(u,v)$ is a foreground pixel
      $\mathcal{N} \leftarrow \mathsf{GetNeighbors}(I, u, v)$ ▷ see Eqn. 10.2
      if $N_i = 0$ for all $N_i \in \mathcal{N}$ **then**
        $I(u,v) \leftarrow label$.
        $label \leftarrow label + 1$.
      else if exactly one $N_j \in \mathcal{N}$ has a value > 1 **then**
        set $I(u,v) \leftarrow N_j$
      else if more than one $N_k \in \mathcal{N}$ have values > 1 **then**
        $I(u,v) \leftarrow N_k$ ▷ select one $N_k > 1$ as the new label
        for all $N_l \in \mathcal{N}$, with $l \neq k$ and $N_l > 1$ **do**
          $\mathsf{C} \leftarrow \mathsf{C} \cup (N_k, N_l)$ ▷ register collision (N_k, N_l)

Remark: The image I now contains labels $0, 2, \ldots, label–1$.

Step 2 – Resolve label collisions:

Create a partitioning of the label set (sequence of 1-element sets):
$\mathsf{R} \leftarrow (\{2\}, \{3\}, \{4\}, \ldots, \{label–1\})$
for all collisions (A, B) in C **do**
  Find the sets $\mathsf{R}(a)$, $\mathsf{R}(b)$ holding labels A, B:
  $a \leftarrow$ index of the set $\mathsf{R}(a)$ that contains label A
  $b \leftarrow$ index of the set $\mathsf{R}(b)$ that contains label B
  if $a \neq b$ **then** ▷ A and B are contained in different sets
    $\mathsf{R}(a) \leftarrow \mathsf{R}(a) \cup \mathsf{R}(b)$ ▷ merge elements of $\mathsf{R}(b)$ into $\mathsf{R}(a)$
    $\mathsf{R}(b) \leftarrow \{\}$

Remark: All *equivalent* labels (i.e., all labels of pixels in the same connected component) are now contained in the same subset of R.

Step 3: Relabel the image:

for all $(u,v) \in M \times N$ **do**
  if $I(u,v) > 1$ **then** ▷ this is a labeled foreground pixel
    $j \leftarrow$ index of the set $\mathsf{R}(j)$ that contains label $I(u,v)$
    Choose a representative element k from the set $\mathsf{R}(j)$:
    $k \leftarrow \min(\mathsf{R}(j))$ ▷ e.g., pick the minimum value
    $I(u,v) \leftarrow k$ ▷ replace the image label
return

$$\mathcal{N}_4 = \begin{array}{ccc} & N_1 & \\ N_2 & \times & N_0 \\ & N_3 & \end{array} \quad \text{or} \quad \mathcal{N}_8 = \begin{array}{ccc} N_3 & N_2 & N_1 \\ N_4 & \times & N_0 \\ N_5 & N_6 & N_7 \end{array}. \tag{10.2}$$

When using the 4-connected neighborhood $\mathcal{N}_4$, only the two neighbors $N_1 = I(u-1, v)$ and $N_2 = I(u, v-1)$ need to be considered, but when using the 8-connected neighborhood $\mathcal{N}_8$, all four neighbors $N_1 \ldots N_4$ must be examined. In the following examples (Figs. 10.3–10.5), we use an 8-connected neighborhood and a very simple test image (Fig. 10.3(a)) to demonstrate the sequential region labeling process.

Propagating region labels

Again we assume that, in the image, the value $I(u, v) = 0$ represents background pixels and the value $I(u, v) = 1$ represents foreground pixels. We will also consider neighboring pixels that lie outside of the image matrix (e.g., on the array borders) to be part of the background. The neighborhood region $\mathcal{N}(u, v)$ is slid over the image horizontally and then vertically, starting from the top left corner. When the current image element $I(u, v)$ is a foreground pixel, it is either assigned a new region number or, in the case where one of its previously examined neighbors in $\mathcal{N}(u, v)$ was a foreground pixel, it takes on the region number of the neighbor. In this way, existing region numbers propagate in the image from the left to the right and from the top to the bottom, as shown in (Fig. 10.3(b–c)).

Label collisions

In the case where two or more neighbors have labels belonging to *different* regions, then a label collision has occurred; that is, pixels within a single connected region have different labels. For example, in a U-shaped region, the pixels in the left and right arms are at first assigned different labels since it is not immediately apparent that they are actually part of a single region. The two labels will propagate down independently from each other until they eventually collide in the lower part of the "U" (Fig. 10.3(d)).

When two labels a, b collide, then we know that they are actually "equivalent"; that is, they are contained in the same image region. These collisions are registered but otherwise not dealt with during the first step. Once all collisions have been registered, they are then resolved in the second step of the algorithm. The number of collisions depends on the content of the image. There can be only a few or very many collisions, and the exact number is only known at the end of the first step, once the whole image has been traversed. For this reason, collision management must make use of dynamic data structures such as lists or hash tables.

Upon the completion of the first steps, all the original foreground pixels have been provisionally marked, and all the collisions between labels within the same regions have been registered for subsequent processing. The example in Fig. 10.4 illustrates the state upon completion of step 1: all foreground pixels have been assigned preliminary labels (Fig. 10.4(a)), and the following collisions (depicted by circles) between the labels $(2, 4)$, $(2, 5)$, and $(2, 6)$ have been registered. The labels $\mathcal{L} = \{2, 3, 4, 5, 6, 7\}$ and collisions $\mathcal{C} = \{(2, 4), (2, 5), (2, 6)\}$ correspond to the nodes and edges of an undirected graph (Fig. 10.4(b)).

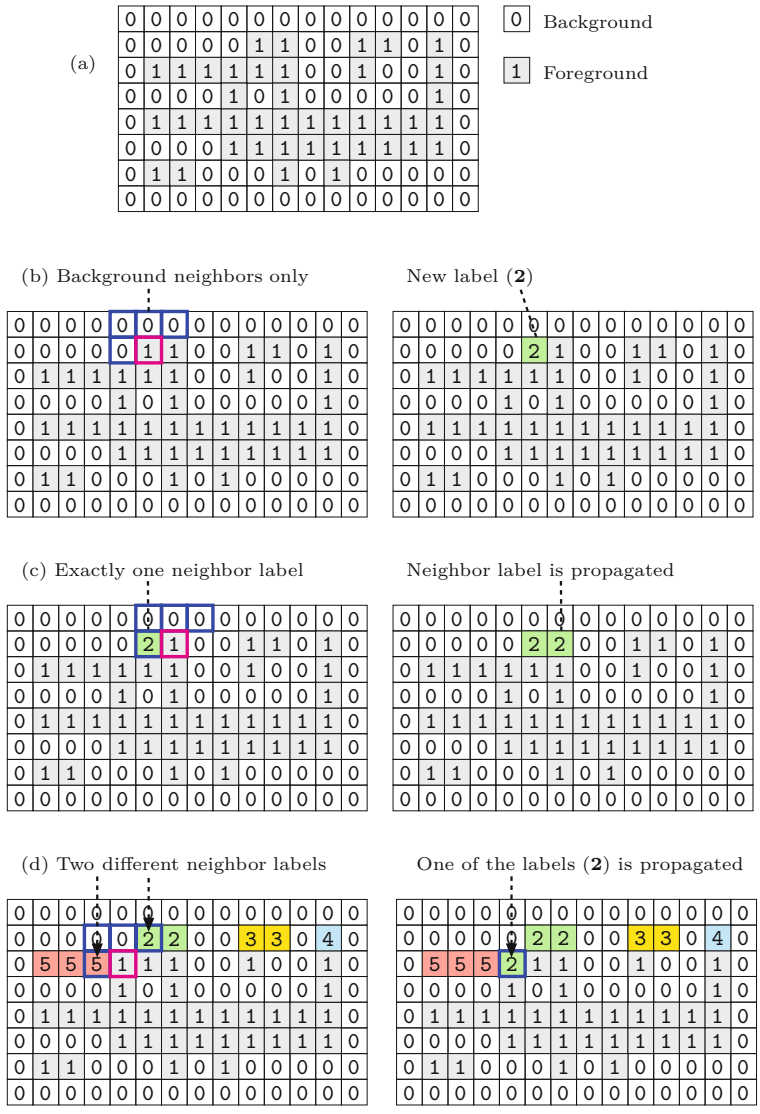


Fig. 10.3
Sequential region labeling—label propagation. Original image (a). The first foreground pixel (marked **1**) is found in (b): all neighbors are background pixels (marked **0**), and the pixel is assigned the first label (**2**). In the next step (c), there is exactly *one* neighbor pixel marked with the label **2**, so this value is propagated. In (d) there are *two* neighboring pixels, and they have differing labels (**2** and **5**); one of these values is propagated, and the collision (**2**, **5**) is registered.

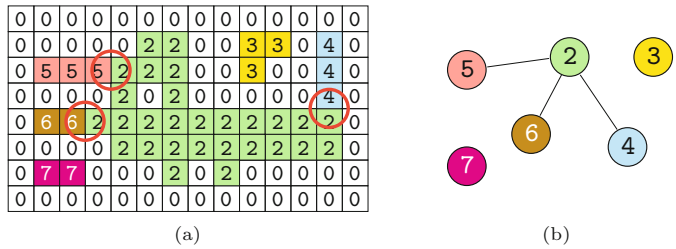


Fig. 10.4
Sequential region labeling—intermediate result after step 1. Label collisions indicated by circles (a); the nodes of the undirected graph (b) correspond to the labels, and its edges correspond to the collisions.

Step 2: Resolving label collisions

The task in the second step is to resolve the label collisions that arose in the first step in order to merge the corresponding "partial" regions. This process is nontrivial since it is possible for two regions with dif-

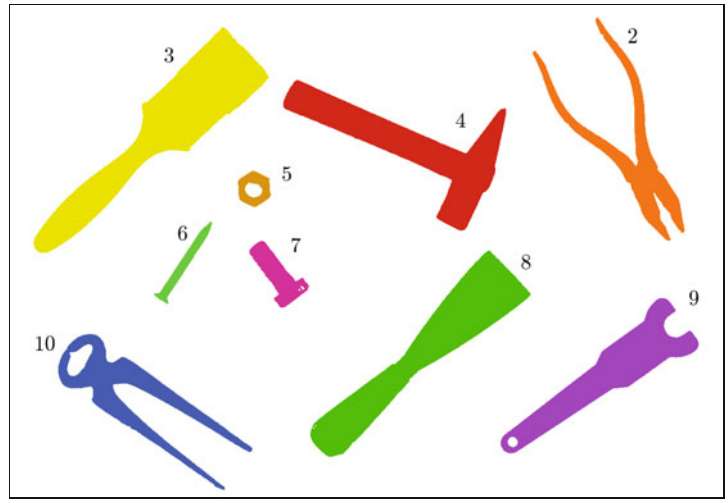
ferent labels to be connected transitively (e.g., $(a,b) \cap (b,c) \Rightarrow (a,c)$) through a third region or, more generally, through a series of regions. In fact, this problem is identical to the problem of finding the *connected components* of a graph [54], where the labels $\mathcal{L}$ determined in step 1 constitute the "nodes" of the graph and the registered collisions $\mathcal{C}$ make up its "edges" (Fig. 10.4(b)).

Once all the distinct labels within a single region have been collected, the labels of all the pixels in the region are updated so they carry the same label (e.g., choosing the smallest label number in the region), as depicted in Fig. 10.5. Figure 10.6 shows the complete segmentation with some region statistics that can be easily calculated from the labeling data.

Fig. 10.5
Sequential region labeling—final result after step 2. All equivalent labels have been replaced by the smallest label within that region.

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	2	0
0	2	2	2	2	2	2	0	0	3	0	0	2	0
0	0	0	0	2	0	2	0	0	0	0	0	2	0
0	2	2	2	2	2	2	2	2	2	2	2	2	0
0	0	0	0	2	2	2	2	2	2	2	2	2	0
0	7	7	0	0	0	2	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 10.6
Example of a complete region labeling. The pixels within each region have been colored according to the consecutive label values $2, 3, \ldots, 10$ they were assigned. The corresponding region statistics are shown in the table (total image size is 1212×836).



Label	Area (*pixels*)	Bounding Box (*left, top, right, bottom*)	Centroid (x_c, y_c)
2	14978	(887, 21, 1144, 399)	(1049.7, 242.8)
3	36156	(40, 37, 438, 419)	(261.9, 209.5)
4	25904	(464, 126, 841, 382)	(680.6, 240.6)
5	2024	(387, 281, 442, 341)	(414.2, 310.6)
6	2293	(244, 367, 342, 506)	(294.4, 439.0)
7	4394	(406, 400, 507, 512)	(454.1, 457.3)
8	29777	(510, 416, 883, 765)	(704.9, 583.9)
9	20724	(833, 497, 1168, 759)	(1016.0, 624.1)
10	16566	(82, 558, 411, 821)	(208.7, 661.6)

10.1.3 Region Labeling—Summary

In this section, we have described a selection of algorithms for finding and labeling connected regions in images. We discovered that the elegant idea of labeling individual regions using a simple recursive flood-filling method (Sec. 10.1.1) was not useful because of practical limitations on the depth of recursion and the high memory costs associated with it. We also saw that classical sequential region labeling (Sec. 10.1.2) is relatively complex and offers no real advantage over iterative implementations of the *depth-first* and *breadth-first* methods. In practice, the iterative breadth-first method is generally the best choice for large and complex images. In the following section we present a modern and efficient algorithm that performs region labeling and also delineates the regions' contours. Since contours are required in many applications, this combined approach is highly practical.

10.2 Region Contours

Once the regions in a binary image have been found, the next step is often to find the contours (that is, the outlines) of the regions. Like so many other tasks in image processing, at first glance this appears to be an easy one: simply follow along the edge of the region. We will see that, in actuality, describing this apparently simple process algorithmically requires careful thought, which has made contour finding one of the classic problems in image analysis.

10.2.1 External and Internal Contours

As we discussed in Chapter 9, Sec. 9.2.7, the pixels along the edge of a binary region (i.e., its border) can be identified using simple morphological operations and difference images. It must be stressed, however, that this process only *marks* the pixels along the contour, which is useful, for instance, for display purposes. In this section, we will go one step further and develop an algorithm for obtaining an *ordered sequence* of border pixel coordinates for describing a region's contour. Note that connected image regions contain exactly one *outer* contour, yet, due to holes, they can contain arbitrarily many *inner* contours. Within such holes, smaller regions may be found, which will again have their own outer contours, and in turn these regions may themselves contain further holes with even smaller regions, and so on in a recursive manner (Fig. 10.7). An additional complication arises when regions are connected by parts that taper down to the width of a single pixel. In such cases, the contour can run through the same pixel more than once and from different directions (Fig. 10.8). Therefore, when tracing a contour from a start point $\boldsymbol{x}_s$, returning to the start point is *not* a sufficient condition for terminating the contour-tracing process. Other factors, such as the current direction along which contour points are being traversed, must be taken into account.

One apparently simple way of determining a contour is to proceed in analogy to the two-stage process presented in Sec. 10.1; that is,

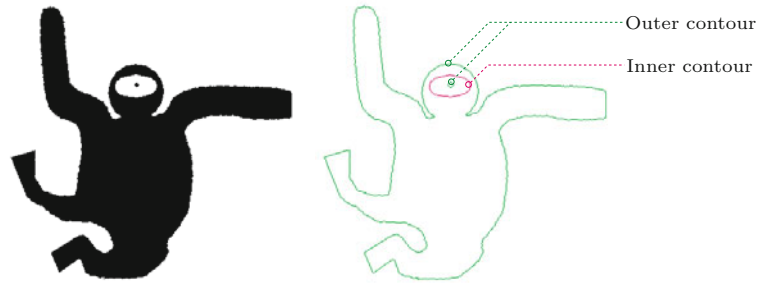


Fig. 10.7
Binary image with outer and inner contours. The outer contour lies along the outside of the foreground region (dark). The inner contour surrounds the space within the region, which may contain further regions (holes), and so on.

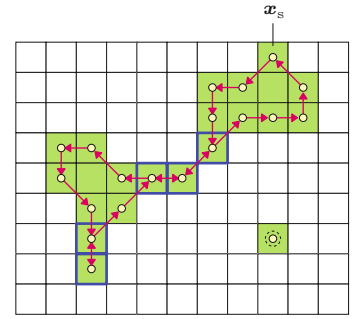


Fig. 10.8
The path along a contour as an ordered sequence of pixel coordinates with a given start point $\boldsymbol{x}_s$. Individual pixels may occur (be visited) more than once within the path, and a region consisting of a single isolated pixel will also have a contour (bottom right).

to *first* identify the connected regions in the image and *second*, for each region, proceed around it, starting from a pixel selected from its border. In the same way, an internal contour can be found by starting at a border pixel of a region's hole. A wide range of algorithms based on first finding the regions and then following along their contours have been published, including [202], [180, pp. 142–148], and [214, p. 296].

As a modern alternative, we present the following *combined* algorithm that, in contrast to the aforementioned classical methods, combines contour finding and region labeling in a single process.

10.2.2 Combining Region Labeling and Contour Finding

This method, based on [47], combines the concepts of sequential region labeling (Sec. 10.1) and traditional contour tracing into a single algorithm able to perform both tasks simultaneously during a single pass through the image. It identifies and labels regions and at the same time traces both their inner and outer contours. The algorithm does not require any complicated data structures and is relatively efficient when compared to other methods with similar capabilities. The key steps of this method are described here and illustrated in Fig. 10.9:

1. As in the sequential region labeling (Alg. 10.2), the binary image I is traversed from the top left to the bottom right. Such a traversal ensures that all pixels in the image are eventually examined and assigned an appropriate label.

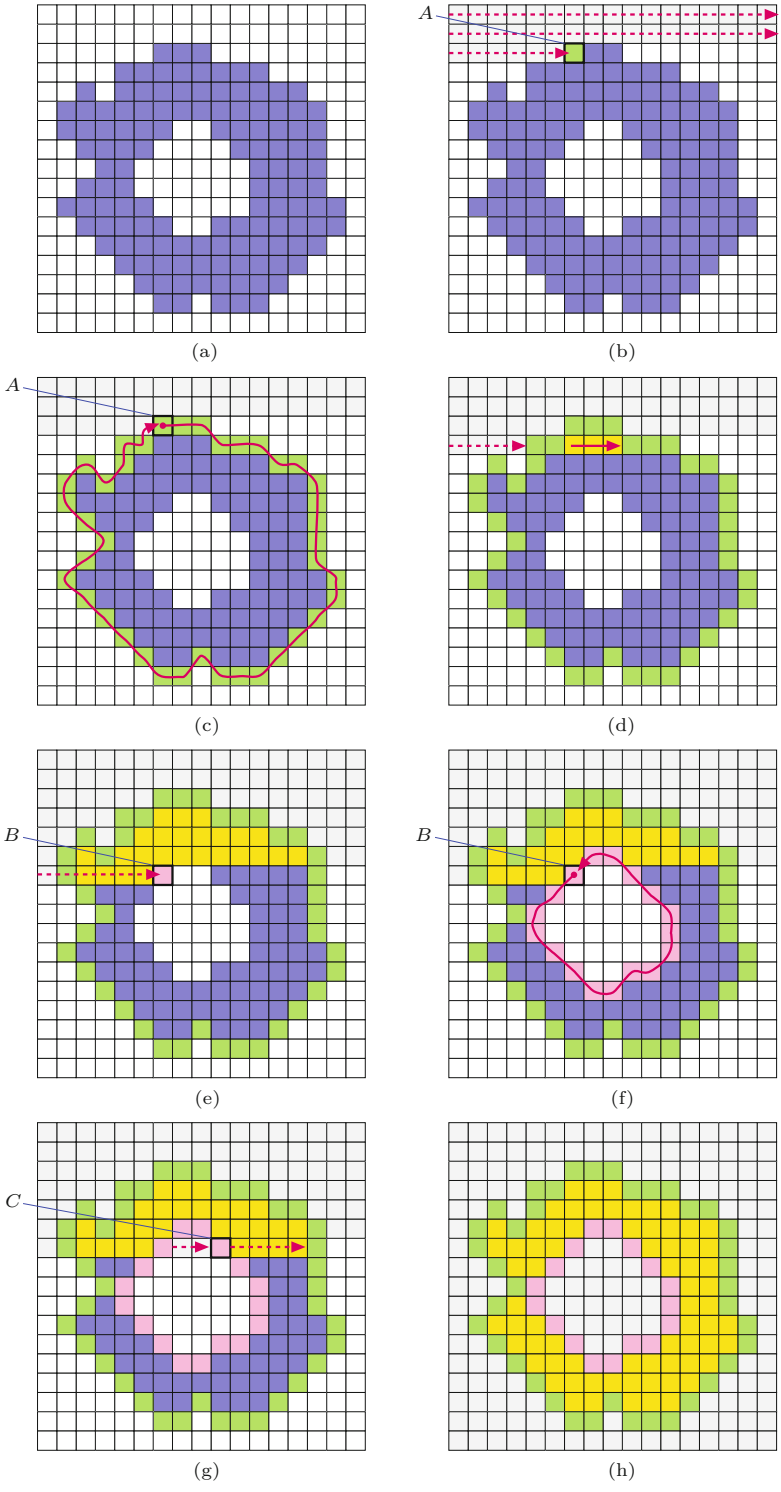


Fig. 10.9
Combined region labeling and contour following (after [47]). The image in (a) is traversed from the top left to the lower right, one row at a time. In (b), the first foreground pixel A on the outer edge of the region is found. Starting from point A, the pixels on the edge along the outer contour are visited and labeled until A is reached again (c). Labels picked up at the outer contour are propagated along the image line inside the region (d). In (e), B was found as the first point on the *inner contour*. Now the inner contour is traversed in clock-wise direction, marking the contour pixels until point B is reached again (f). The same tracing process is used as in step (c), with the inside of the region always lying to the right of the contour path. In (g) a previously marked point C on an inner contour is detected. Its label is again propagated along the image line inside the region. The final result is shown in (h).

2. At a given position in the image, the following cases may occur:

   Case A: The transition from a background pixel to a previously unmarked foreground pixel means that this pixel lies on the outer edge of a new region. A new *label* is assigned and the associated *outer* contour is traversed and marked by calling the method TraceContour (see Alg. 10.3 and Fig. 10.9(a)). Furthermore, all background pixels directly bordering the region are marked with the special label −1.

   Case B: The transition from a foreground pixel B to an unmarked background pixel means that this pixel lies on an *inner* contour (Fig. 10.9(b)). Starting from B, the inner contour is traversed and its pixels are marked with labels from the surrounding region (Fig. 10.9(c)). Also, all bordering background pixels are again assigned the special label value −1.

   Case C: When a foreground pixel does not lie on a contour, then the neighboring pixel to the left has already been labeled (Fig. 10.9(d)) and this label is propagated to the current pixel.

In Algs. 10.3–10.4, the entire procedure is presented again and explained precisely. Procedure RegionContourLabeling traverses the image line-by-line and calls procedure TraceContour whenever a new inner or outer contour must be traced. The labels of the image elements along the contour, as well as the neighboring foreground pixels, are stored in the "label map" L (a rectangular array of the same size as the image) by procedure FindNextContourPoint in Alg. 10.4.

10.2.3 Java Implementation

The Java implementation of the combined region labeling and contour tracing algorithm can be found online in class `RegionContourLabeling`[4] (for details see Sec. 10.9). It almost exactly follows Algs. 10.3–10.4, only the image I and the associated *label map* L are initially *padded* (i.e., enlarged) by a surrounding layer of background pixels. This simplifies the process of tracing the outer region contours, since no special treatment is needed at the image borders. Program 10.2 shows a minimal example of its usage within the `run()` method of an ImageJ plugin (class `Trace_Contours`).

Examples

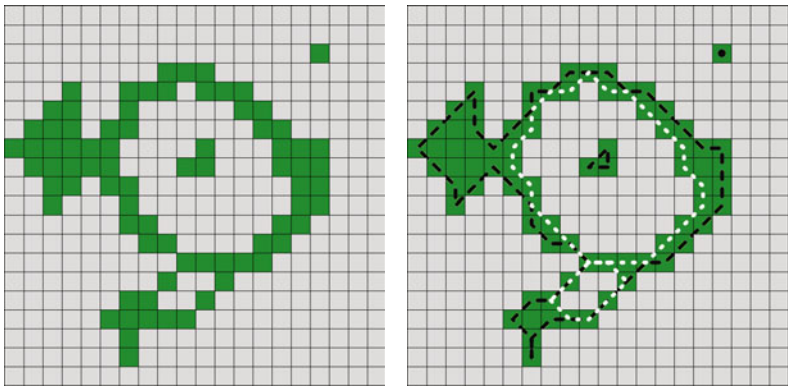
This combined algorithm for region marking and contour following is particularly well suited for processing large binary images since it is efficient and has only modest memory requirements. Figure 10.10 shows a synthetic test image that illustrates a number of special situations, such as isolated pixels and thin sections, which the algorithm must deal with correctly when following the contours. In the resulting plot, outer contours are shown as black polygon lines running trough the centers of the contour pixels, and inner contours are drawn white. Contours of single-pixel regions are marked by small circles filled with the corresponding color. Figure 10.11 shows the results for a larger section taken from a real image (Fig. 9.12).

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

```
1:  RegionContourLabeling(I)
        Input: I, a binary image with 0 = background, 1 = foreground.
        Returns sequences of outer and inner contours and a map of
        region labels.
2:      (M, N) ← Size(I)
3:      C_out ← ( )                          ▷ empty list of outer contours
4:      C_in ← ( )                           ▷ empty list of inner contours
5:      Create map L : M × N ↦ ℤ             ▷ create the label map L
6:      for all (u, v) do
7:          L(u, v) ← 0                      ▷ initialize L to zero
8:      r ← 0                                ▷ region counter
9:      for v ← 0, ..., N−1 do               ▷ scan the image top to bottom
10:         label ← 0
11:         for u ← 0, ..., M−1 do           ▷ scan the image left to right
12:             if I(u, v) > 0 then          ▷ I(u, v) is a foreground pixel
13:                 if (label ≠ 0) then      ▷ continue existing region
14:                     L(u, v) ← label
15:                 else
16:                     label ← L(u, v)
17:                     if (label = 0) then  ▷ hit a new outer contour
18:                         r ← r + 1
19:                         label ← r
20:                         x_s ← (u, v)
21:                         C ← TraceContour(x_s, 0, label, I, L)  ▷ outer c.
22:                         C_out ← C_out ∪ (C)   ▷ collect outer contour
23:                         L(u, v) ← label
24:             else                         ▷ I(u, v) is a background pixel
25:                 if (label ≠ 0) then
26:                     if (L(u, v) = 0) then  ▷ hit new inner contour
27:                         x_s ← (u−1, v)
28:                         C ← TraceContour(x_s, 1, label, I, L)  ▷ inner
                                cntr.
29:                         C_in ← C_in ∪ (C)   ▷ collect inner contour
30:                     label ← 0
31:     return (C_out, C_in, L)
```

continued in Alg. 10.4 ▷▷

Alg. 10.3
Combined contour tracing and region labeling (part 1). Given a binary image I, the application of RegionContourLabeling(I) returns a set of contours and an array containing region labels for all pixels in the image. When a new point on either an outer or inner contour is found, then an ordered list of the contour's points is constructed by calling procedure TraceContour (line 21 and line 28). TraceContour itself is described in Alg. 10.4.



(a) (b)

Fig. 10.10
Combined contour and region marking. Original image, with foreground pixels marked green (a); located contours with black lines for *outer* and white lines for *inner* contours (b). Contour polygons pass through the pixel centers. Outer contours of single-pixel regions (e.g., in the upper-right of (b)) are marked by a single dot.

Alg. 10.4 Combined contour finding and region labeling (part 2, continued from Alg. 10.3). Starting from $\boldsymbol{x}_s$, the procedure TraceContour traces along the contour in the direction $d_S = 0$ for outer contours or $d_S = 1$ for inner contours. During this process, all contour points as well as neighboring background points are marked in the label array L. Given a point $\boldsymbol{x}_c$, TraceContour uses FindNextContourPoint() to determine the next point along the contour (line 9). The function Delta() returns the next coordinate in the sequence, taking into account the search direction d.

```
1:  TraceContour(x_s, d_s, label, I, L)
        Input: x_s, start position; d_s, initial search direction; label, the
        label assigned to this contour; I, the binary input image; L, label
        map. Returns a new outer or inner contour (sequence of points)
        starting at x_s.
2:      (x, d) ← FindNextContourPoint(x_s, d_s, I, L)
3:      c ← (x)                              ▷ new contour with the single point x
4:      x_p ← x_s                            ▷ previous position x_p = (u_p, v_p)
5:      x_c ← x                              ▷ current position x_c = (u_c, v_c)
6:      done ← (x_s ≡ x)                     ▷ isolated pixel?
7:      while (¬done) do
8:          L(u_c, v_c) ← label
9:          (x_n, d) ← FindNextContourPoint(x_c, (d + 6) mod 8, I, L)
10:         x_p ← x_c
11:         x_c ← x_n
12:         done ← (x_p ≡ x_s ∧ x_c ≡ x)     ▷ back at starting position?
13:         if (¬done) then
14:             c ← c ∪ (x_n)                ▷ add point x_n to contour c
15:     return c                             ▷ return this contour

16: FindNextContourPoint(x, d, I, L)
        Input: x, initial position; d ∈ [0, 7], search direction, I, binary
        input image; L, the label map.
        Returns the next point on the contour and the modified search
        direction.
17:     for i ← 0, ..., 6 do                 ▷ search in 7 directions
18:         x_n ← x + Delta(d)
19:         if I(x_n) = 0 then               ▷ I(u_n, v_n) is a background pixel
20:             L(x_n) ← −1                  ▷ mark background as visited (−1)
21:             d ← (d + 1) mod 8
22:         else                             ▷ found a non-background pixel at x_n
23:             return (x_n, d)
24:     return (x, d)                        ▷ found no next node, return start position

25: Delta(d) := (Δx, Δy),  with
```

d	0	1	2	3	4	5	6	7
Δx	1	1	0	−1	−1	−1	0	1
Δy	0	1	1	1	0	−1	−1	−1

Prog. 10.2 Example of using the class ContourTracer. (plugin Trace_Contours). First (in line 9) a new instance of RegionContourLabeling is created for the input image I. The segmentation into regions and contours is done by the constructor. In lines 11–12 the outer and inner contours are retrieved as (possibly empty) lists of type Contour. Finally, the list of connected regions is obtained in line 14.

```
1  import imagingbook.pub.regions.BinaryRegion;
2  import imagingbook.pub.regions.Contour;
3  import imagingbook.pub.regions.RegionContourLabeling;
4  import java.util.List;
5  ...
6  public void run(ImageProcessor ip) {
7    // Make sure we have a proper byte image:
8    ByteProcessor I = ip.convertToByteProcessor();
9    // Create the region labeler / contour tracer:
10   RegionContourLabeling seg = new RegionContourLabeling(I);
11   // Get all outer/inner contours and connected regions:
12   List<Contour> outerContours = seg.getOuterContours();
13   List<Contour> innerContours = seg.getInnerContours();
14   List<BinaryRegion> regions = seg.getRegions();
15   ...
16 }
```

Fig. 10.11
Example of a complex contour (original image in Ch. 9, Fig. 9.12). Outer contours are marked in black and inner contours in white.

10.3 Representing Image Regions

10.3.1 Matrix Representation

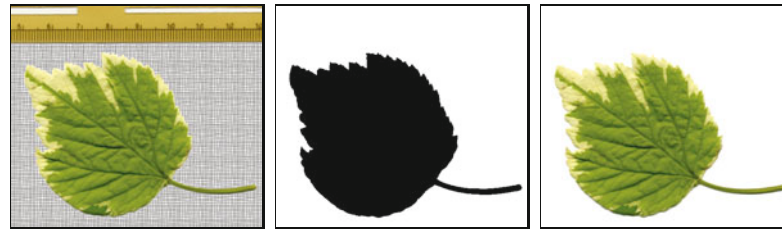
A natural representation for images is a matrix (i.e., a two-dimensional array) in which elements represent the intensity or the color at a corresponding position in the image. This representation lends itself, in most programming languages, to a simple and elegant mapping onto two-dimensional arrays, which makes possible a very natural way to work with raster images. One possible disadvantage with this representation is that it does not depend on the content of the image. In other words, it makes no difference whether the image contains only a pair of lines or is of a complex scene because the amount of memory required is constant and depends only on the dimensions of the image.

Regions in an image can be represented using a logical mask in which the area within the region is assigned the value *true* and the area without the value *false* (Fig. 10.12). Since these values can be represented by a single bit, such a matrix is often referred to as a "bitmap".[5]

10.3.2 Run Length Encoding

In *run length encoding* (RLE), sequences of adjacent foreground pixels can be represented compactly as "runs". A run, or contiguous

[5] Java does not provide a genuine 1-bit data type. Even variables of type `boolean` are represented internally (i.e., within the Java virtual machine) as 32-bit `int`s.



(a) (b) (c)

Fig. 10.12 Use of a binary mask to specify a region of an image: original image (a), logical (bit) mask (b), and masked image (c).

block, is a maximal length sequence of adjacent pixels of the same type within either a row or a column. Runs of arbitrary length can be encoded compactly using three integers,

$$Run_i = \langle row_i, column_i, length_i \rangle,$$

as illustrated in Fig. 10.13. When representing a sequence of runs within the same row, the number of the row is redundant and can be left out. Also, in some applications, it is more useful to record the coordinate of the end column instead of the length of the run.

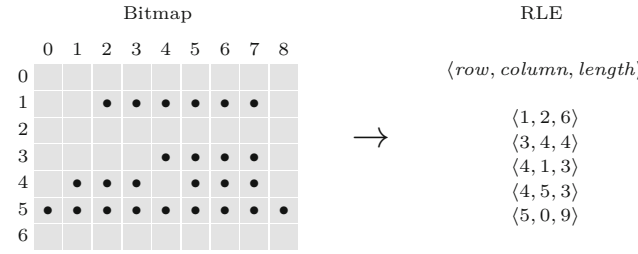


Fig. 10.13 Run length encoding in row direction. A run of pixels can be represented by its starting point $(1, 2)$ and its length (6).

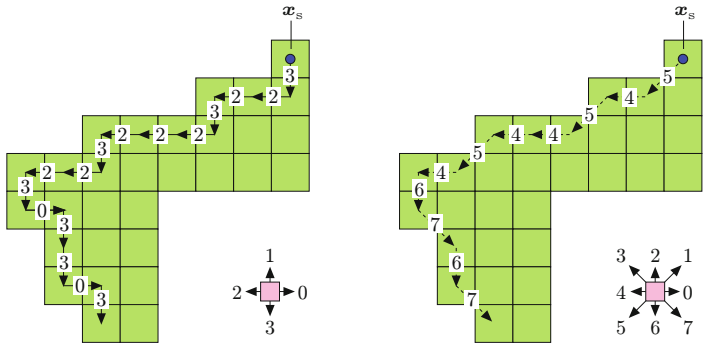
Since the RLE representation can be easily implemented and efficiently computed, it has long been used as a simple lossless compression method. It forms the foundation for fax transmission and can be found in a number of other important codecs, including TIFF, GIF, and JPEG. In addition, RLE provides precomputed information about the image that can be used directly when computing certain properties of the image (for example, statistical moments; see Sec. 10.5.2).

10.3.3 Chain Codes

Regions can be represented not only using their interiors but also by their contours. Chain codes, which are often referred to as Freeman codes [79], are a classical method of contour encoding. In this encoding, the contour beginning at a given start point $\boldsymbol{x}_\mathrm{s}$ is represented by the sequence of directional changes it describes on the discrete image grid (Fig. 10.14).

Absolute chain code

For a closed contour of a region $\mathcal{R}$, described by the sequence of points $\boldsymbol{c}_\mathcal{R} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$ with $\boldsymbol{x}_i = \langle u_i, v_i \rangle$, we create the elements of its chain code sequence $\boldsymbol{c}'_\mathcal{R} = (c'_0, c'_1, \ldots c'_{M-1})$ with



(a) 4-Chain Code
3223222322303303...111
length = 28

(b) 8-Chain Code
54544546767...222
length = $16 + 6\sqrt{2} \approx 24.5$

Fig. 10.14
Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point $\boldsymbol{x}_s$. Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) neighborhoods. The length of the resulting path, calculated as the sum of the individual segments, can be used to approximate the true length of the contour.

$$c'_i = \mathsf{Code}(u', v'), \tag{10.3}$$

where

$$(u', v') = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \le i < M-1, \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1, \end{cases} \tag{10.4}$$

and $\mathsf{Code}(u', v')$ being defined (assuming an 8-connected neighborhood) by the following table:

u'	1	1	0	−1	−1	−1	0	1
v'	0	1	1	1	0	−1	−1	−1
$\mathsf{Code}(u', v')$	0	1	2	3	4	5	6	7

Chain codes are compact since instead of storing the absolute coordinates for every point on the contour, only that of the starting point is recorded. The remaining points are encoded relative to the starting point by indicating in which of the eight possible directions the next point lies. Since only 3 bits are required to encode these eight directions the values can be stored using a smaller numeric type.

Differential chain code

Directly comparing two regions represented using chain codes is difficult since the description depends on the starting point selected $\boldsymbol{x}_s$, and for instance simply rotating the region by 90° results in a completely different chain code. When using a *differential* chain code, the situation improves slightly. Instead of encoding the difference in the *position* of the next contour point, the change in the *direction* along the discrete contour is encoded. A given *absolute* chain code $\boldsymbol{c}'_{\mathcal{R}} = (c'_0, c'_1, \ldots c'_{M-1})$ can be converted element by element to a *differential* chain code $\boldsymbol{c}''_{\mathcal{R}} = (c''_0, c''_1, \ldots c''_{M-1})$, with[6]

$$c''_i = \begin{cases} (c'_{i+1} - c'_i) \bmod 8 & \text{for } 0 \le i < M-1, \\ (c'_0 - c'_i) \bmod 8 & \text{for } i = M-1, \end{cases} \tag{10.5}$$

[6] For the implementation of the mod operator see Sec. F.1.2 in the Appendix.

again under the assumption of an 8-connected neighborhood. The element c''_i thus describes the change in direction (curvature) of the contour between two successive segments c'_i and c'_{i+1} of the original chain code $\boldsymbol{c}'_{\mathcal{R}}$. For the contour in Fig. 10.14(b), for example, the result is

$$\begin{aligned}\boldsymbol{c}'_{\mathcal{R}} &= (5,4,5,4,4,5,4,6,7,6,7,\ldots,2,2,2),\\ \boldsymbol{c}''_{\mathcal{R}} &= (7,1,7,0,1,7,2,1,7,1,1,\ldots,0,0,3).\end{aligned}$$

Given the start position $\boldsymbol{x}_\mathrm{s}$ and the (absolute) initial direction c_0, the original contour can be unambiguously reconstructed from the differential chain code.

Shape numbers

While the differential chain code remains the same when a region is rotated by 90°, the encoding is still dependent on the selected starting point. If we want to determine the similarity of two contours of the same length M using their differential chain codes $\boldsymbol{c}''_1$, $\boldsymbol{c}''_2$, we must first ensure that the same start point was used when computing the codes. A method that is often used [15,88] is to interpret the elements c''_i in the differential chain code as the digits of a number to the base b ($b = 8$ for an 8-connected contour or $b = 4$ for a 4-connected contour) and the numeric value

$$\mathsf{Val}(\boldsymbol{c}''_{\mathcal{R}}) = c''_0 \cdot b^0 + c''_1 \cdot b^1 + \ \ldots\ + c''_{M-1} \cdot b^{M-1} = \sum_{i=0}^{M-1} c''_i \cdot b^i. \quad (10.6)$$

Then the sequence $\boldsymbol{c}''_{\mathcal{R}}$ is shifted circularly until the numeric value of the corresponding number reaches a maximum. We use the expression $\boldsymbol{c}''_{\mathcal{R}} \rhd k$ to denote the sequence $\boldsymbol{c}''_{\mathcal{R}}$ being circularly shifted by k positions to the right.[7] For example, for $k = 2$ this is

$$\begin{aligned}\boldsymbol{c}''_{\mathcal{R}} &= (0,1,3,2,\ldots,5,3,7,4),\\ \boldsymbol{c}''_{\mathcal{R}} \rhd 2 &= (7,4,0,1,3,2,\ldots,5,3),\end{aligned}$$

and

$$k_{\max} = \operatorname*{argmax}_{0 \le k < M} \mathsf{Val}(\boldsymbol{c}''_{\mathcal{R}} \rhd k), \quad (10.7)$$

denotes the shift required to maximize the corresponding arithmetic value. The resulting code sequence or *shape number*,

$$\boldsymbol{s}_{\mathcal{R}} = \boldsymbol{c}''_{\mathcal{R}} \rhd k_{\max}, \quad (10.8)$$

is *normalized* with respect to the starting point and can thus be directly compared element by element with other normalized code sequences. Since the function $\mathsf{Val}()$ in Eqn. (10.6) produces values that are in general too large to be actually computed, in practice the relation

$$\mathsf{Val}(\boldsymbol{c}''_1) > \mathsf{Val}(\boldsymbol{c}''_2)$$

[7] That is, $(\boldsymbol{c}''_{\mathcal{R}} \rhd k)(\,i\,) = \boldsymbol{c}''_{\mathcal{R}}((i-k) \bmod M)$.

is determined by comparing the *lexicographic ordering* between the sequences $\boldsymbol{c}_1''$ and $\boldsymbol{c}_2''$ so that the arithmetic values need not be computed at all.

Unfortunately, comparisons based on chain codes are generally not very useful for determining the similarity between regions simply because rotations at arbitrary angles ($\neq 90°$) have too great of an impact (change) on a region's code. In addition, chain codes are not capable of handling changes in size (scaling) or other distortions. Section 10.4 presents a number of tools that are more appropriate in these types of cases.

Fourier shape descriptors

An elegant approach to describing contours are so-called Fourier shape descriptors, which interpret the two-dimensional contour $C = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{M-1})$ with $\boldsymbol{x}_i = (u_i, v_i)$ as a sequence of values in the complex plane, where

$$z_i = (u_i + \mathrm{i} \cdot v_i) \in \mathbb{C}. \tag{10.9}$$

From this sequence, one obtains (using a suitable method of interpolation in case of an 8-connected contour), a discrete, one-dimensional periodic function $f(s) \in \mathbb{C}$ with a constant sampling interval over s, the path length around the contour. The coefficients of the 1D *Fourier spectrum* (see Sec. 18.3) of this function $f(s)$ provide a shape description of the contour in frequency space, where the lower spectral coefficients deliver a gross description of the shape. The details of this classical method can be found, for example, in [88, 97, 126, 128, 222]. This technique is described in considerable detail in Chapter 26.

10.4 Properties of Binary Regions

Imagine that you have to describe the contents of a digital image to another person over the telephone. One possibility would be to call out the value of each pixel in some agreed upon order. A much simpler way of course would be to describe the image on the basis of its properties—for example, "a red rectangle on a blue background", or at an even higher level such as "a sunset at the beach with two dogs playing in the sand". While using such a description is simple and natural for us, it is not (yet) possible for a computer to generate these types of descriptions without human intervention. For computers, it is of course simpler to calculate the mathematical properties of an image or region and to use these as the basis for further classification. Using features to classify, be they images or other items, is a fundamental part of the field of pattern recognition, a research area with many applications in image processing and computer vision [64, 169, 228].

10.4.1 Shape Features

The comparison and classification of binary regions is widely used, for example, in optical character recognition (OCR) and for automating

processes ranging from blood cell counting to quality control inspection of manufactured products on assembly lines. The analysis of binary regions turns out to be one of the simpler tasks for which many efficient algorithms have been developed and used to implement reliable applications that are in use every day.

By a *feature* of a region, we mean a specific numerical or qualitative measure that is computable from the values and coordinates of the pixels that make up the region. As an example, one of the simplest features is its *size* or *area*; that is the number of pixels that make up a region. In order to describe a region in a compact form, different features are often combined into a *feature vector*. This vector is then used as a sort of "signature" for the region that can be used for classification or comparison with other regions. The best features are those that are simple to calculate and are not easily influenced (robust) by irrelevant changes, particularly translation, rotation, and scaling.

10.4.2 Geometric Features

A region $\mathcal{R}$ of a binary image can be interpreted as a two-dimensional distribution of foreground points $\boldsymbol{p}_i = (u_i, v_i)$ on the discrete plane $\mathbb{Z}^2$, that is, as a set

$$\mathcal{R} = \{\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{N-1}\} = \{(u_0, v_0), (u_1, v_1), \ldots, (u_{N-1}, v_{N-1})\}.$$

Most geometric properties are defined in such a way that a region is considered to be a set of pixels that, in contrast to the definition in Sec. 10.1, does not necessarily have to be connected.

Perimeter

The perimeter (or circumference) of a region $\mathcal{R}$ is defined as the length of its outer contour, where $\mathcal{R}$ must be connected. As illustrated in Fig. 10.14, the type of neighborhood relation must be taken into account for this calculation. When using a 4-neighborhood, the measured length of the contour (except when that length is 1) will be larger than its actual length.

In the case of 8-neighborhoods, a good approximation is reached by weighing the horizontal and vertical segments with 1 and diagonal segments with $\sqrt{2}$. Given an 8-connected chain code $\boldsymbol{c}'_{\mathcal{R}} = (c'_0, c'_1, \ldots c'_{M-1})$, the perimeter of the region is arrived at by

$$\mathsf{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \mathrm{length}(c'_i), \tag{10.10}$$

with

$$\mathrm{length}(c) = \begin{cases} 1 & \text{for } c = 0, 2, 4, 6, \\ \sqrt{2} & \text{for } c = 1, 3, 5, 7. \end{cases} \tag{10.11}$$

However, with this conventional method of calculation, the real perimeter $P(\mathcal{R})$ is systematically overestimated. As a simple remedy, an empirical correction factor of 0.95 works satisfactorily even for relatively small regions, that is,

$$P(\mathcal{R}) \approx 0.95 \cdot \mathsf{Perimeter}(\mathcal{R}). \tag{10.12}$$

Area

The area of a binary region $\mathcal{R}$ can be found by simply counting the image pixels that make up the region, that is,

$$A(\mathcal{R}) = N = |\mathcal{R}|. \tag{10.13}$$

The area of a connected region without holes can also be approximated from its closed contour, defined by M coordinate points $(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$, where $\boldsymbol{x}_i = (u_i, v_i)$, using the Gaussian area formula for polygons:

$$A(\mathcal{R}) \approx \frac{1}{2} \cdot \Big| \sum_{i=0}^{M-1} \big(u_i \cdot v_{(i+1) \bmod M} \; - \; u_{(i+1) \bmod M} \cdot v_i\big) \Big|. \tag{10.14}$$

When the contour is already encoded as a chain code $\boldsymbol{c}'_{\mathcal{R}} = (c'_0, c'_1, \ldots c'_{M-1})$, then the region's area can be computed (trivially) with Eqn. (10.14) by expanding C_{abs} into a sequence of contour points from an arbitrary starting point (e.g., $(0, 0)$). However, the area can also be calculated directly from the chain code representation without expanding the contour [263] (see also Exercise 10.12).

While simple region properties such as area and perimeter are not influenced (except for quantization errors) by translation and rotation of the region, they are definitely affected by changes in size; for example, when the object to which the region corresponds is imaged from different distances. However, as will be described, it is possible to specify combined features that are *invariant* to translation, rotation, *and* scaling as well.

Compactness and roundness

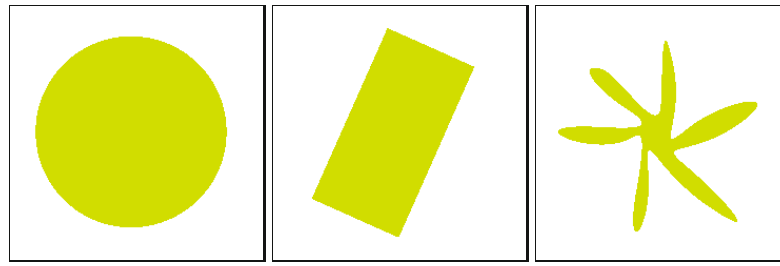
Compactness is understood as the relation between a region's area and its perimeter. We can use the fact that a region's perimeter P increases linearly with the enlargement factor while the area A increases quadratically to see that, for a particular shape, the ratio A/P^2 should be the same at any scale. This ratio can thus be used as a feature that is invariant under translation, rotation, and scaling. When applied to a circular region of any diameter, this ratio has a value of $\frac{1}{4\pi}$, so by normalizing it against a filled circle, we create a feature that is sensitive to the *roundness* or *circularity* of a region,

$$\mathsf{Circularity}(\mathcal{R}) = \; 4\pi \cdot \frac{A(\mathcal{R})}{P^2(\mathcal{R})}, \tag{10.15}$$

which results in a maximum value of 1 for a perfectly round region $\mathcal{R}$ and a value in the range $[0, 1]$ for all other shapes (Fig. 10.15). If an absolute value for a region's roundness is required, the corrected perimeter estimate (Eqn. (10.12)) should be employed. Figure 10.15 shows the circularity values of different regions as computed with the formulation in Eqn. (10.15).

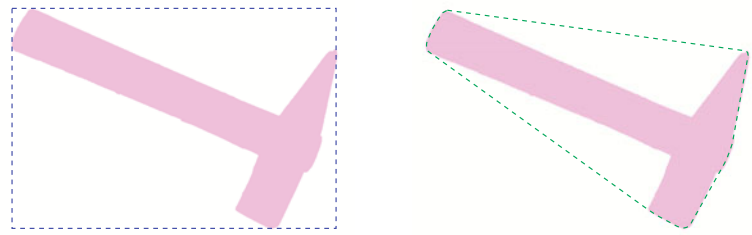
Bounding box

The bounding box of a region $\mathcal{R}$ is the minimal axis-parallel rectangle that encloses all points of $\mathcal{R}$,



(a) 0.904 (1.001) (b) 0.607 (0.672) (c) 0.078 (0.086)

Fig. 10.15 Circularity values for different shapes. Shown are the corresponding estimates for $\mathsf{Circularity}(\mathcal{R})$ as defined in Eqn. (10.15). Corrected values calculated with Eqn. (10.12) are shown in parentheses.



(a) (b)

Fig. 10.16 Example bounding box (a) and convex hull (b) of a binary image region.

$$\mathsf{BoundingBox}(\mathcal{R}) = \langle u_{\min}, u_{\max}, v_{\min}, v_{\max} \rangle, \tag{10.16}$$

where $u_{\min}, u_{\max}$ and $v_{\min}, v_{\max}$ are the minimal and maximal coordinate values of all points $(u_i, v_i) \in \mathcal{R}$ in the x and y directions, respectively (Fig. 10.16(a)).

Convex hull

The convex hull is the smallest convex polygon that contains all points of the region $\mathcal{R}$. A physical analogy is a board in which nails stick out in correspondence to each of the points in the region. If you were to place an elastic band around *all* the nails, then, when you release it, it will contract into a convex hull around the nails (see Figs. 10.16(b) and 10.21(c)). Given N contour points, the convex hull can be computed in time $\mathcal{O}(N \log V)$, where V is the number vertices in the polygon of the resulting convex hull [17].

The convex hull is useful, for example, for determining the convexity or the *density* of a region. The *convexity* is defined as the relationship between the length of the convex hull and the original perimeter of the region. *Density* is then defined as the ratio between the area of the region and the area of its convex hull. The *diameter*, on the other hand, is the maximal distance between any two nodes on the convex hull.

10.5 Statistical Shape Properties

When computing statistical shape properties, we consider a region $\mathcal{R}$ to be a collection of coordinate points distributed within a two-dimensional space. Since statistical properties can be computed for point distributions that do not form a connected region, they can

be applied before segmentation. An important concept in this context are the *central moments* of the region's point distribution, which measure characteristic properties with respect to its midpoint or *centroid*.

10.5.1 Centroid

The centroid or center of gravity of a connected region can be easily visualized. Imagine drawing the region on a piece of cardboard or tin and then cutting it out and attempting to balance it on the tip of your finger. The location on the region where you must place your finger in order for the region to balance is the *centroid* of the region.[8]

The centroid $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})^\mathsf{T}$ of a binary (not necessarily connected) region is the arithmetic mean of the pont coordinates $\boldsymbol{x}_i = (u_i, v_i)$, that is,

$$\bar{\boldsymbol{x}} = \frac{1}{|\mathcal{R}|} \cdot \sum_{\boldsymbol{x}_i \in \mathcal{R}} \boldsymbol{x}_i \tag{10.17}$$

or

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u_i,v_i)} u_i \qquad \text{and} \qquad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u_i,v_i)} v_i \,. \tag{10.18}$$

10.5.2 Moments

The formulation of the region's centroid in Eqn. (10.18) is only a special case of the more general statistical concept of a *moment*. Specifically, the expression

$$m_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot u^p \cdot v^q \tag{10.19}$$

describes the (ordinary) moment of order p, q for a discrete (image) function $I(u, v) \in \mathbb{R}$; for example, a grayscale image. All the following definitions are also generally applicable to regions in grayscale images. The moments of connected binary regions can also be calculated directly from the coordinates of the contour points [212, p. 148].

In the special case of a binary image $I(u, v) \in \{0, 1\}$, only the foreground pixels with $I(u, v) = 1$ in the region $\mathcal{R}$ need to be considered, and therefore Eqn. (10.19) can be simplified to

$$m_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} u^p \cdot v^q \,. \tag{10.20}$$

In this way, the *area* of a binary region can be expressed as its *zero*-order moment,

$$A(\mathcal{R}) = |\mathcal{R}| \;= \sum_{(u,v)} 1 \;= \sum_{(u,v)} u^0 \cdot v^0 = m_{00}(\mathcal{R}) \tag{10.21}$$

and similarly the *centroid* $\bar{\boldsymbol{x}}$ Eqn. (10.18) can be written as

[8] Assuming you did not imagine a region where the centroid lies outside of the region or within a hole in the region, which is of course possible.

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)} u^1 \cdot v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})},$$
$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)} u^0 \cdot v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})}. \tag{10.22}$$

These moments thus represent concrete physical properties of a region. Specifically, the area m_{00} is in practice an important basis for characterizing regions, and the centroid $(\bar{x}, \bar{y})$ permits the reliable and (within a fraction of a pixel) exact specification of a region's position.

10.5.3 Central Moments

To compute position-independent (translation-invariant) region features, the region's centroid, which can be determined precisely in any situation, can be used as a reference point. In other words, we can shift the origin of the coordinate system to the region's centroid $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ to obtain the *central* moments of order p, q:

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot (u - \bar{x})^p \cdot (v - \bar{y})^q. \tag{10.23}$$

For a binary image (with $I(u,v) = 1$ within the region $\mathcal{R}$), Eqn. (10.23) can be simplified to

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} (u - \bar{x})^p \cdot (v - \bar{y})^q. \tag{10.24}$$

10.5.4 Normalized Central Moments

Central moment values of course depend on the absolute size of the region since the value depends directly on the distance of all region points to its centroid. So, if a 2D shape is scaled uniformly by some factor $s \in \mathbb{R}$, its central moments multiply by the factor

$$s^{(p+q+2)}. \tag{10.25}$$

Thus size-invariant "normalized" moments are obtained by scaling with the reciprocal of the area $A = \mu_{00} = m_{00}$ raised to the required power in the form

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq} \cdot \Big(\frac{1}{\mu_{00}(\mathcal{R})}\Big)^{(p+q+2)/2}, \tag{10.26}$$

for $(p + q) \geq 2$ [126, p. 529].

10.5.5 Java Implementation

Program 10.3 gives a direct (brute force) Java implementation for computing the ordinary, central, and normalized central moments for binary images (`BACKGROUND = 0`). This implementation is only meant to clarify the computation, and naturally much more efficient implementations are possible (see, e.g., [131]).

```
// Ordinary moment:

double moment(ImageProcessor I, int p, int q) {
  double Mpq = 0.0;
  for (int v = 0; v < I.getHeight(); v++) {
    for (int u = 0; u < I.getWidth(); u++) {
      if (I.getPixel(u, v) > 0) {
        Mpq+= Math.pow(u, p) * Math.pow(v, q);
      }
    }
  }
  return Mpq;
}

// Central moments:

double centralMoment(ImageProcessor I, int p, int q) {
  double m00  = moment(I, 0, 0);  // region area
  double xCtr = moment(I, 1, 0) / m00;
  double yCtr = moment(I, 0, 1) / m00;
  double cMpq = 0.0;
  for (int v = 0; v < I.getHeight(); v++) {
    for (int u = 0; u < I.getWidth(); u++) {
      if (I.getPixel(u, v) > 0) {
        cMpq+= Math.pow(u-xCtr, p) * Math.pow(v-yCtr, q);
      }
    }
  }
  return cMpq;
}

// Normalized central moments:

double nCentralMoment(ImageProcessor I, int p, int q) {
  double m00 = moment(I, 0, 0);
  double norm = Math.pow(m00, 0.5 * (p + q + 2));
  return centralMoment(I, p, q) / norm;
}
```

Prog. 10.3
Example of directly computing moments in Java. The methods `moment()`, `centralMoment()`, and `nCentralMoment()` compute for a binary image the moments m_{pq}, μ_{pq}, and $\bar{\mu}_{pq}$ (Eqns. (10.20), (10.24), and (10.26)).

10.6 Moment-Based Geometric Properties

While normalized moments can be directly applied for classifying regions, further interesting and geometrically relevant features can be elegantly derived from statistical region moments.

10.6.1 Orientation

Orientation describes the direction of the major axis, that is, the axis that runs through the centroid and along the widest part of the region (Fig. 10.18(a)). Since rotating the region around the major axis requires less effort (smaller moment of inertia) than spinning it around any other axis, it is sometimes referred to as the major axis of rotation. As an example, when you hold a pencil between your hands and twist it around its major axis (that is, around the lead),

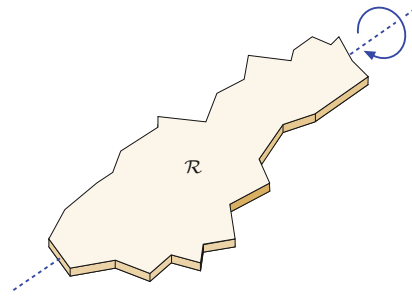


Fig. 10.17 Major axis of a region. Rotating an elongated region $\mathcal{R}$, interpreted as a physical body, around its major axis requires less effort (least moment of inertia) than rotating it around any other axis.

the pencil exhibits the least mass inertia (Fig. 10.17). As long as a region exhibits an orientation at all ($\mu_{20}(\mathcal{R}) \neq \mu_{02}(\mathcal{R})$), the direction $\theta_\mathcal{R}$ of the major axis can be found directly from the central moments μ_{pq} as

$$\tan(2\,\theta_\mathcal{R}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \tag{10.27}$$

and thus the corresponding angle is

$$\theta_\mathcal{R} = \frac{1}{2} \cdot \tan^{-1}\Big(\frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})}\Big) \tag{10.28}$$

$$= \frac{1}{2} \cdot \mathrm{ArcTan}\big(\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R}), 2 \cdot \mu_{11}(\mathcal{R})\big). \tag{10.29}$$

The resulting angle $\theta_\mathcal{R}$ is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.[9] Orientation measurements based on region moments are very accurate in general.

Calculating orientation vectors

When visualizing region properties, a frequent task is to plot the region's orientation as a line or arrow, usually anchored at the center of gravity $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})^\mathsf{T}$; for example, by a parametric line of the form

$$\boldsymbol{x} = \bar{\boldsymbol{x}} + \lambda \cdot \boldsymbol{x}_\mathrm{d} = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \lambda \cdot \begin{pmatrix} \cos(\theta_\mathcal{R}) \\ \sin(\theta_\mathcal{R}) \end{pmatrix}, \tag{10.30}$$

with the normalized orientation vector $\boldsymbol{x}_\mathrm{d}$ and the length variable $\lambda > 0$. To find the unit orientation vector $\boldsymbol{x}_d = (\cos\theta, \sin\theta)^\mathsf{T}$, we could first compute the inverse tangent to get 2θ (Eqn. (10.28)) and then compute the cosine and sine of θ. However, the vector $\boldsymbol{x}_d$ can also be obtained without using trigonometric functions as follows. Rewriting Eqn. (10.27) as

$$\tan(2\theta_\mathcal{R}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} = \frac{a}{b} = \frac{\sin(2\theta_\mathcal{R})}{\cos(2\theta_\mathcal{R})}, \tag{10.31}$$

we get (by Pythagora's theorem)

[9] See Sec. A.1 in the Appendix for the computation of angles with the ArcTan() (inverse tangent) function and Sec. F.1.6 for the corresponding Java method `Math.atan2()`.

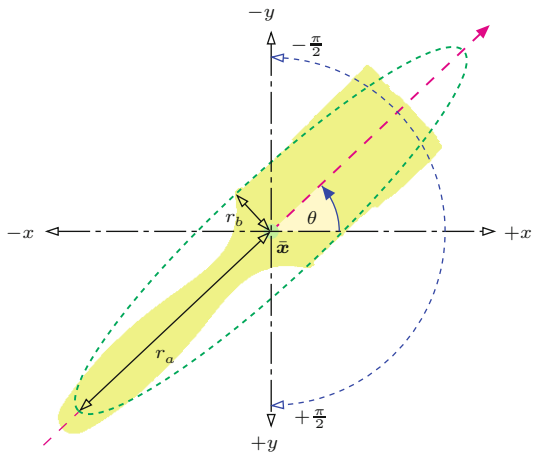


Fig. 10.18
Region orientation and eccentricity. The major axis of the region extends through its center of gravity $\bar{\boldsymbol{x}}$ at the orientation θ. Note that angles are in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ and increment in the *clockwise* direction because the y axis of the image coordinate system points downward (in this example, $\theta \approx -0.759 \approx -43.5°$). The eccentricity of the region is defined as the ratio between the lengths of the major axis (r_a) and the minor axis (r_b) of the "equivalent" ellipse.

$$\sin(2\theta_{\mathcal{R}}) = \frac{a}{\sqrt{a^2+b^2}} \qquad \text{and} \qquad \cos(2\theta_{\mathcal{R}}) = \frac{b}{\sqrt{a^2+b^2}},$$

where $A = 2\mu_{11}(\mathcal{R})$ and $B = \mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})$. Using the relations $\cos^2\alpha = \frac{1}{2}[1 + \cos(2\alpha)]$ and $\sin^2\alpha = \frac{1}{2}[1 - \cos(2\alpha)]$, we can compute the normalized orientation vector $\boldsymbol{x}_{\mathrm{d}} = (x_{\mathrm{d}}, y_{\mathrm{d}})^{\mathsf{T}}$ as

$$x_{\mathrm{d}} = \cos(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } a = b = 0, \\ \left[\frac{1}{2} \cdot \left(1 + \frac{b}{\sqrt{a^2+b^2}}\right)\right]^{\frac{1}{2}} & \text{otherwise,} \end{cases} \tag{10.32}$$

$$y_{\mathrm{d}} = \sin(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } a = b = 0, \\ \left[\frac{1}{2} \cdot \left(1 - \frac{b}{\sqrt{a^2+b^2}}\right)\right]^{\frac{1}{2}} & \text{for } a \geq 0, \\ -\left[\frac{1}{2} \cdot \left(1 - \frac{b}{\sqrt{a^2+b^2}}\right)\right]^{\frac{1}{2}} & \text{for } a < 0, \end{cases} \tag{10.33}$$

straight from the central region moments $\mu_{11}(\mathcal{R})$, $\mu_{20}(\mathcal{R})$, and $\mu_{02}(\mathcal{R})$, as defined in Eqn. (10.31). The horizontal component (x_d) in Eqn. (10.32) is always positive, while the case switch in Eqn. (10.33) corrects the sign of the vertical component (y_d) to map to the same angular range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ as Eqn. (10.28). The resulting vector $\boldsymbol{x}_d$ is normalized (i.e., $\|(x_d, y_d)\| = 1$) and could be scaled arbitrarily for display purposes by a suitable length λ, for example, using the region's eccentricity value described in Sec. 10.6.2 (see also Fig. 10.19).

10.6.2 Eccentricity

Similar to the region orientation, moments can also be used to determine the "elongatedness" or *eccentricity* of a region. A naive approach for computing the eccentricity could be to rotate the region until we can fit a bounding box (or enclosing ellipse) with a maximum aspect ratio. Of course this process would be computationally intensive simply because of the many rotations required. If we know the orientation of the region (Eqn. (10.28)), then we may fit a bounding box that is parallel to the region's major axis. In general, the proportions of the region's bounding box is not a good eccentricity measure

anyway because it does not consider the distribution of pixels inside the box.

Based on region moments, highly accurate and stable measures can be obtained without any iterative search or optimization. Also, moment-based methods do not require knowledge of the boundary length (as required for computing the circularity feature in Sec. 10.4.2), and they can also handle nonconnected regions or point clouds. Several different formulations of region eccentricity can be found in the literature [15, 126, 128] (see also Exercise 10.17). We adopt the following definition because of its simple geometrical interpretation:

$$\mathsf{Ecc}(\mathcal{R}) = \frac{a_1}{a_2} = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}, \tag{10.34}$$

where $a_1 = 2\lambda_1$, $a_2 = 2\lambda_2$ are proportional to the eigenvalues λ_1, λ_2 (with $\lambda_1 \geq \lambda_2$) of the symmetric 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{pmatrix}, \tag{10.35}$$

with the region's central moments μ_{11}, μ_{20}, μ_{02} (see Eqn. (10.23)).[10] The values of Ecc are in the range $[1, \infty)$, where $\mathsf{Ecc} = 1$ corresponds to a circular disk and elongated regions have values > 1.

The value returned by $\mathsf{Ecc}(\mathcal{R})$ is invariant to the region's orientation and size, that is, this quantity has the important property of being rotation and scale invariant. However, the values a_1, a_2 contain relevant information about the spatial structure of the region. Geometrically, the eigenvalues λ_1, λ_2 (and thus a_1, a_2) directly relate to the proportions of the "equivalent" ellipse, positioned at the region's center of gravity $(\bar{x}, \bar{y})$ and oriented at $\theta = \theta_{\mathcal{R}}$ Eqn. (10.28). The lengths of the major and minor axes, r_a and r_b, are

$$r_a = 2 \cdot \left(\frac{\lambda_1}{|\mathcal{R}|}\right)^{\frac{1}{2}} = \left(\frac{2\,a_1}{|\mathcal{R}|}\right)^{\frac{1}{2}}, \tag{10.36}$$

$$r_b = 2 \cdot \left(\frac{\lambda_2}{|\mathcal{R}|}\right)^{\frac{1}{2}} = \left(\frac{2\,a_2}{|\mathcal{R}|}\right)^{\frac{1}{2}}, \tag{10.37}$$

respectively, with a_1, a_2 as defined in Eqn. (10.34) and $|\mathcal{R}|$ being the number of pixels in the region. Given the axes' lengths r_a, r_b and the centroid $(\bar{x}, \bar{y})$, the parametric equation of this ellipse is

$$\boldsymbol{x}(t) = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} r_a \cdot \cos(t) \\ r_b \cdot \sin(t) \end{pmatrix} \tag{10.38}$$

$$= \begin{pmatrix} \bar{x} + \cos(\theta) \cdot r_a \cdot \cos(t) - \sin(\theta) \cdot r_b \cdot \sin(t) \\ \bar{y} + \sin(\theta) \cdot r_a \cdot \cos(t) + \cos(\theta) \cdot r_b \cdot \sin(t) \end{pmatrix}, \tag{10.39}$$

for $0 \leq t < 2\pi$. If entirely *filled*, the region described by this ellipse would have the same central moments as the original region $\mathcal{R}$. Figure 10.19 shows a set of regions with overlaid orientation and eccentricity results.

[10] $\mathbf{A}$ is actually the *covariance matrix* for the distribution of pixel positions inside the region (see Sec. D.2 in the Appendix).

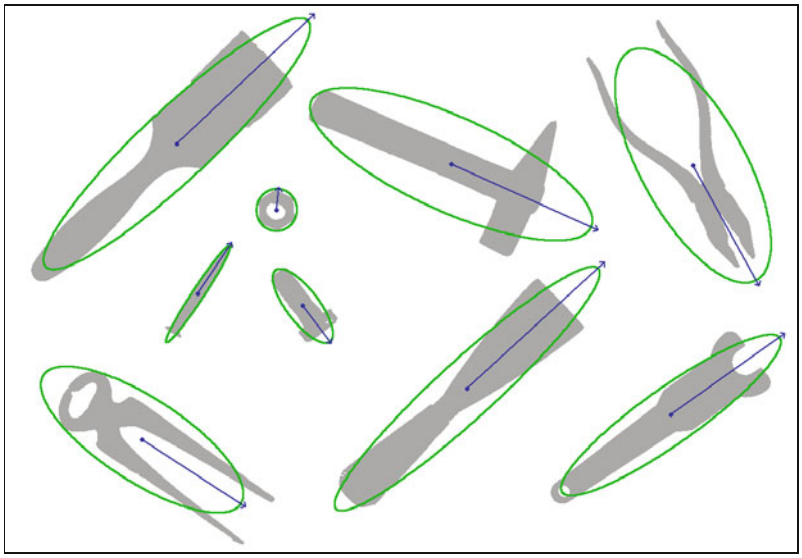


Fig. 10.19
Orientation and eccentricity examples. The orientation θ (Eqn. (10.28)) is displayed for each connected region as a vector with the length proportional to the region's eccentricity value $\mathsf{Ecc}(\mathcal{R})$ (Eqn. (10.34)). Also shown are the ellipses (Eqns. (10.36) and (10.37)) corresponding to the orientation and eccentricity parameters.

10.6.3 Bounding Box Aligned to the Major Axis

While the ordinary, x/y axis-aligned bounding box (see Sec. 10.4.2) is of little practical use (because it is sensitive to rotation), it may be interesting to see how to find a region's bounding box that is aligned with its major axis, as defined in Sec. 10.6.1. Given a region's orientation angle $\theta_{\mathcal{R}}$,

$$\boldsymbol{e}_a = \begin{pmatrix} x_a \\ y_a \end{pmatrix} = \begin{pmatrix} \cos(\theta_{\mathcal{R}}) \\ \sin(\theta_{\mathcal{R}}) \end{pmatrix} \tag{10.40}$$

is the unit vector parallel to its major axis; thus

$$\boldsymbol{e}_b = \boldsymbol{e}_a^{\perp} = \begin{pmatrix} y_a \\ -x_a \end{pmatrix} \tag{10.41}$$

is the unit vector orthogonal to $\boldsymbol{e}_a$.[11] The bounding box can now be determined as follows (see Fig. 10.20):

1. Project each region point[12] $\boldsymbol{u}_i = (u_i, v_i)$ onto the vector $\boldsymbol{e}_a$ (parallel to the region's major axis) by calculating the dot product[13]

$$a_i = \boldsymbol{u}_i \cdot \boldsymbol{e}_a \tag{10.42}$$

   and keeping the minimum and maximum values

$$a_{\min} = \min_{\boldsymbol{u}_i \in \mathcal{R}} a_i, \qquad a_{\max} = \max_{\boldsymbol{u}_i \in \mathcal{R}} a_i. \tag{10.43}$$

2. Analogously, project each region point $\boldsymbol{u}_i$ onto the *orthogonal axis* (specified by the vector $\boldsymbol{e}_b$) by

[11] $\boldsymbol{x}^{\perp} = \mathrm{perp}(\boldsymbol{x}) = \left(\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right) \cdot \boldsymbol{x}$.
[12] Of course, if the region's contour is available, it is sufficient to iterate over the contour points only.
[13] See Sec. B.3.1, Eqn. (B.19) in the Appendix.

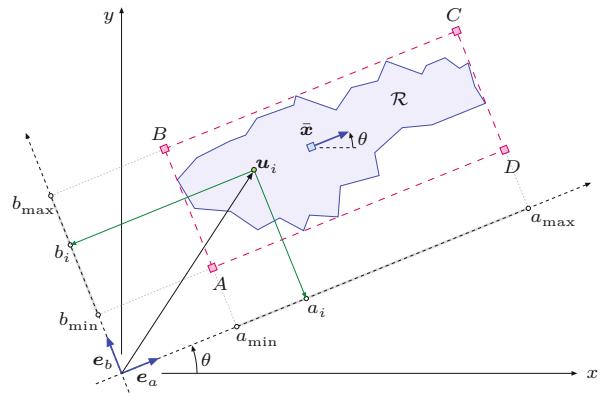


Fig. 10.20 Calculation of a region's major axis-aligned bounding box. The unit vector $\boldsymbol{e}_a$ is parallel to the region's major axis (oriented at angle θ); $\boldsymbol{e}_b$ is perpendicular to $\boldsymbol{e}_a$. The projection of a region point $\boldsymbol{u}_i$ onto the lines defined by $\boldsymbol{e}_a$ and $\boldsymbol{e}_b$ yields the lengths a_i and b_i, respectively (measured from the coordinate origin). The resulting quantities $a_{\min}$, $a_{\max}$, $b_{\min}$, $b_{\max}$ define the corner points (A, B, C, D) of the axis-aligned bounding box. Note that the position of the region's centroid ($\bar{\boldsymbol{x}}$) is not required in this calculation.

$$b_i = \boldsymbol{u}_i \cdot \boldsymbol{e}_b \tag{10.44}$$

and keeping the minimum and maximum values, that is,

$$b_{\min} = \min_{\boldsymbol{u}_i \in \mathcal{R}} b_i, \qquad b_{\max} = \max_{\boldsymbol{u}_i \in \mathcal{R}} b_i. \tag{10.45}$$

Note that steps 1 and 2 can be performed in a single iteration over all region points.

3. Finally, from the resulting quantities $a_{\min}$, $a_{\max}$, $b_{\min}$, $b_{\max}$, calculate the four corner points A, B, C, D of the bounding box as

$$\begin{aligned} A &= a_{\min}\cdot\boldsymbol{e}_a + b_{\min}\cdot\boldsymbol{e}_b, & B &= a_{\min}\cdot\boldsymbol{e}_a + b_{\max}\cdot\boldsymbol{e}_b, \\ C &= a_{\max}\cdot\boldsymbol{e}_a + b_{\max}\cdot\boldsymbol{e}_b, & D &= a_{\max}\cdot\boldsymbol{e}_a + b_{\min}\cdot\boldsymbol{e}_b. \end{aligned} \tag{10.46}$$

The complete calculation is summarized in Alg. 10.20; a typical example is shown in Fig. 10.21(d).

Alg. 10.5 Calculation of the major axis-aligned bounding box for a binary region $\mathcal{R}$. If the region's contour is available, it is sufficient to use the contour points only.

MajorAxisAlignedBoundingBox($\mathcal{R}$)
Input: $\mathcal{R} = \{\boldsymbol{u}_i\}$, a binary region containing points $\boldsymbol{u}_i \in \mathbb{R}^2$.
Returns the four corner points of the region's bounding box.

$\theta \leftarrow 0.5 \cdot \mathrm{ArcTan}(\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R}), 2 \cdot \mu_{11}(\mathcal{R}))$ ▷ see Eq. 10.28
$\boldsymbol{e}_a \leftarrow (\cos(\theta), \sin(\theta))^\mathsf{T}$ ▷ unit vector parall. to region's major axis
$\boldsymbol{e}_b \leftarrow (\sin(\theta), -\cos(\theta))^\mathsf{T}$ ▷ unit vector perpendic. to major axis
$a_{\min} \leftarrow \infty, \quad a_{\max} \leftarrow -\infty$
$b_{\min} \leftarrow \infty, \quad b_{\max} \leftarrow -\infty$
for all $\boldsymbol{u} \in \mathcal{R}$ **do**
  $a \leftarrow \boldsymbol{u} \cdot \boldsymbol{e}_a$ ▷ project $\boldsymbol{u}$ onto $\boldsymbol{e}_a$ (Eq. 10.42)
  $a_{\min} \leftarrow \min(a_{\min}, a)$
  $a_{\max} \leftarrow \max(a_{\max}, a)$
  $b \leftarrow \boldsymbol{u} \cdot \boldsymbol{e}_b$ ▷ project $\boldsymbol{u}$ onto $\boldsymbol{e}_b$ (Eq. 10.44)
  $b_{\min} \leftarrow \min(b_{\min}, b)$
  $b_{\max} \leftarrow \max(b_{\max}, b)$
$A \leftarrow a_{\min} \cdot \boldsymbol{e}_a + b_{\min} \cdot \boldsymbol{e}_b$
$B \leftarrow a_{\min} \cdot \boldsymbol{e}_a + b_{\max} \cdot \boldsymbol{e}_b$
$C \leftarrow a_{\max} \cdot \boldsymbol{e}_a + b_{\max} \cdot \boldsymbol{e}_b$
$D \leftarrow a_{\max} \cdot \boldsymbol{e}_a + b_{\min} \cdot \boldsymbol{e}_b$
return (A, B, C, D) ▷ corners of the bounding box

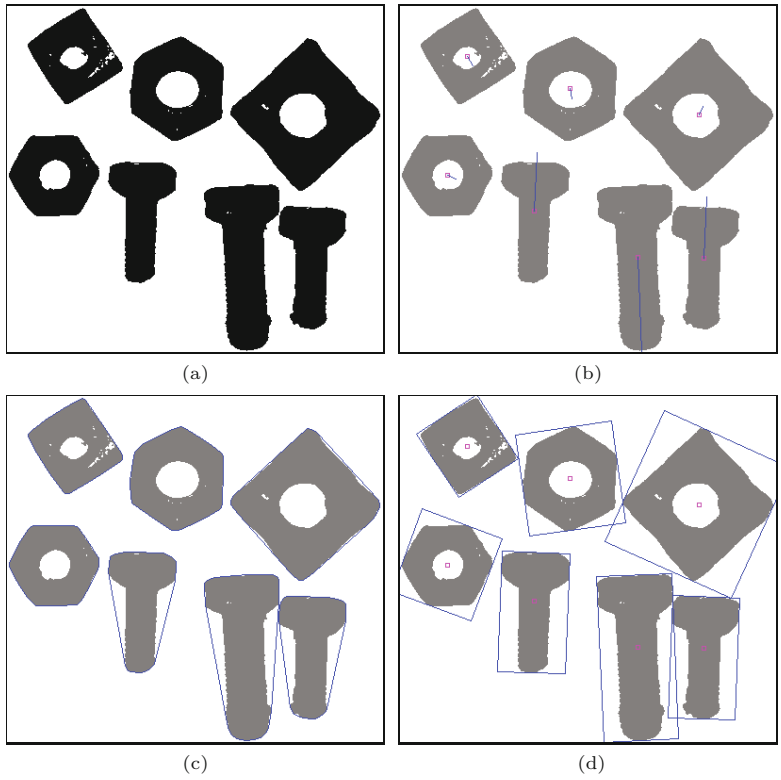


Fig. 10.21
Geometric region properties. Original binary image (a), centroid and orientation vector (length determined by the region's eccentricity) of the major axis (b), convex hull (c), and major axis-aligned bounding box (d).

10.6.4 Invariant Region Moments

Normalized central moments are not affected by the translation or uniform scaling of a region (i.e., the values are invariant), but in general rotating the image will change these values.

Hu's invariant moments

A classical solution to this problem is a clever combination of simpler features known as "Hu's Moments" [112]:[14]

$$
\begin{aligned}
\phi_1 &= \bar{\mu}_{20} + \bar{\mu}_{02}, \qquad (10.47)\\
\phi_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\,\bar{\mu}_{11}^2,\\
\phi_3 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12})^2 + (3\,\bar{\mu}_{21} - \bar{\mu}_{03})^2,\\
\phi_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2,\\
\phi_5 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2] +\\
&\quad (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot [3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2],\\
\phi_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2] +\\
&\quad 4\,\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}),\\
\phi_7 &= (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3\,(\bar{\mu}_{21} + \bar{\mu}_{03})^2] +\\
&\quad (3\,\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot [3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2].
\end{aligned}
$$

[14] In order to improve the legibility of Eqn. (10.47) the argument for the region ($\mathcal{R}$) has been dropped; as an example, with the region argument, the first line would read $H_1(\mathcal{R}) = \bar{\mu}_{20}(\mathcal{R}) + \bar{\mu}_{02}(\mathcal{R})$, and so on.

In practice, the logarithm of these quantities (that is, $\log(\phi_k)$) is used since the raw values may have a very large range. These features are also known as *moment invariants* since they are invariant under translation, rotation, and scaling. While defined here for binary images, they are also applicable to parts of grayscale images; examples can be found in [88, p. 517].

Flusser's invariant moments

It was shown in [72, 73] that Hu's moments, as listed in Eqn. (10.47), are partially redundant and incomplete. Based on so-called *complex moments* $c_{pq} \in \mathbb{C}$, Flusser designed an improved set of 11 rotation and scale-invariant features $\psi_1, \ldots, \psi_{11}$ (see Eqn. (10.51)) for characterizing 2D shapes. For grayscale images (with $I(u,v) \in \mathbb{R}$), the complex moments of order p, q are defined as

$$c_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot (x + \mathrm{i}\cdot y)^p \cdot (x - \mathrm{i}\cdot y)^q, \tag{10.48}$$

with centered positions $x = u - \bar{x}$ and $y = v - \bar{y}$, and $(\bar{x}, \bar{y})$ being the *centroid* of $\mathcal{R}$ (i denotes the imaginary unit). In the case of binary images (with $I(u,v) \in [0,1]$) Eqn. (10.48) simplifies to

$$c_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} (x + \mathrm{i}\cdot y)^p \cdot (x - \mathrm{i}\cdot y)^q. \tag{10.49}$$

Analogous to Eqn. (10.26), the complex moments can be *scale-normalized* to

$$\hat{c}_{p,q}(\mathcal{R}) = \frac{1}{A^{(p+q+2)/2}} \cdot c_{p,q}, \tag{10.50}$$

with A being the area of $\mathcal{R}$ [74, p. 29]. Finally, the derived rotation and scale invariant region moments of 2nd to 4th order are[15]

$$\begin{aligned}
\psi_1 &= \mathrm{Re}(\hat{c}_{1,1}), & \psi_2 &= \mathrm{Re}(\hat{c}_{2,1} \cdot \hat{c}_{1,2}), & \psi_3 &= \mathrm{Re}(\hat{c}_{2,0} \cdot \hat{c}_{1,2}^2),\\
\psi_4 &= \mathrm{Im}(\hat{c}_{2,0} \cdot \hat{c}_{1,2}^2), & \psi_5 &= \mathrm{Re}(\hat{c}_{3,0} \cdot \hat{c}_{1,2}^3), & \psi_6 &= \mathrm{Im}(\hat{c}_{3,0} \cdot \hat{c}_{1,2}^3),\\
\psi_7 &= \mathrm{Re}(\hat{c}_{2,2}), & \psi_8 &= \mathrm{Re}(\hat{c}_{3,1} \cdot \hat{c}_{1,2}^2), & \psi_9 &= \mathrm{Im}(\hat{c}_{3,1} \cdot \hat{c}_{1,2}^2),\\
\psi_{10} &= \mathrm{Re}(\hat{c}_{4,0} \cdot \hat{c}_{1,2}^4), & \psi_{11} &= \mathrm{Im}(\hat{c}_{4,0} \cdot \hat{c}_{1,2}^4). & &
\end{aligned} \tag{10.51}$$

Table 10.1 lists the normalized Flusser moments for five binary shapes taken from the Kimia dataset [134].

Shape matching with region moments

One obvious use of invariant region moments is shape matching and classification. Given two binary shapes A and B, with associated moment ("feature") vectors

$$\boldsymbol{f}_A = (\psi_1(A), \ldots, \psi_{11}(A)) \quad \text{and} \quad \boldsymbol{f}_B = (\psi_1(B), \ldots, \psi_{11}(B)),$$

respectively, one approach could be to simply measure the difference between shapes by the Euclidean distance of these vectors in the form

[15] In Eqn. (10.51), the use of Re( ) for the quantities ψ_1, ψ_2, ψ_7 (which are real-valued *per se*) is redundant.

ψ_1	0.3730017575	0.2545476083	0.2154034257	0.2124041195	0.3600613700
ψ_2	0.0012699373	0.0004247053	0.0002068089	0.0001089652	0.0017187073
ψ_3	0.0004041515	0.0000644829	0.0000274491	0.0000014248	-0.0003853999
ψ_4	0.0000097827	-0.0000076547	0.0000071688	-0.0000022103	-0.0001944121
ψ_5	0.0000012672	0.0000002327	0.0000000637	0.0000000083	-0.0000078073
ψ_6	0.0000001090	-0.0000000483	0.0000000041	0.0000000153	-0.0000061997
ψ_7	0.2687922057	0.1289708408	0.0814034374	0.0712567626	0.2340886626
ψ_8	0.0003192443	0.0000414818	0.0000134036	0.0000003020	-0.0002878997
ψ_9	0.0000053208	-0.0000032541	0.0000030880	-0.0000008365	-0.0001628669
ψ_{10}	0.0000103461	0.0000000091	0.0000000019	-0.0000000003	0.0000001922
ψ_{11}	0.0000000120	-0.0000000020	0.0000000008	-0.0000000000	0.0000003015

Table 10.1
Binary shapes and associated normalized Flusser moments $\psi_1, \ldots, \psi_{11}$. Notice the magnitude of the moments varies by a large factor.

	0.000	0.183	0.245	0.255	0.037
	0.183	0.000	0.062	0.071	0.149
	0.245	0.062	0.000	0.011	0.210
	0.255	0.071	0.011	0.000	0.220
	0.037	0.149	0.210	0.220	0.000

Table 10.2
Inter-class (Euclidean) distances $\mathrm{d}_{\mathrm{E}}(A, B)$ between normalized shape feature vectors for the five reference shapes (see Eqn. (10.52)). Off-diagonal values should be consistently large to allow good shape discrimination.

$$\mathrm{d}_{\mathrm{E}}(A, B) = \|\boldsymbol{f}_A - \boldsymbol{f}_B\| = \Big[\sum_{i=1}^{11} |\psi_i(A) - \psi_i(B)|^2\Big]^{1/2}. \qquad (10.52)$$

Concrete distances between the five sample shapes are listed in Table 10.2. Since the moment vectors are rotation and scale invariant,[16] shape comparisons should remain unaffected by such transformations. Note, however, that the magnitude of the individual moments varies over a very large range. Thus, if the Euclidean distance is used as we have just suggested, the comparison (matching) of shapes is typically dominated by a few moments (or even a single moment) of relatively large magnitude, while the small-valued moments play virtually no role in the distance calculation. This is because the Euclidean distance treats the multi-dimensional feature space uniformly along all dimensions.

As a consequence, moment-based shape discrimination with the ordinary Euclidean distance is typically not very selective. A simple solution is to replace Eqn. (10.52) by a *weighted distance* measure of the form

$$\mathrm{d}'_{\mathrm{E}}(A, B) = \Big[\sum_{i=1}^{11} w_i \cdot |\psi_i(A) - \psi_i(B)|^2\Big]^{1/2}, \qquad (10.53)$$

with fixed weights $w_1, \ldots, w_{11} \geq 0$ assigned to each each moment feature to compensate for the differences in magnitude.

A more elegant approach is to use of the *Mahalanobis* distance [24, 157] for comparing the moment vectors, which accounts for the statistical distribution of each vector component and avoids large-magnitude components dominating the smaller ones. In this case,

[16] Although the invariance property holds perfectly for continuous shapes, rotating and scaling *discrete* binary images may significantly affect the associated region moments.

the distance calculation becomes

$$\mathrm{d_M}(A,B) = \left[(\boldsymbol{f}_A - \boldsymbol{f}_B)^\mathsf{T} \cdot \boldsymbol{\Sigma}^{-1} \cdot (\boldsymbol{f}_A - \boldsymbol{f}_B)\right]^{1/2}, \tag{10.54}$$

where $\boldsymbol{\Sigma}$ is the 11×11 *covariance matrix* for the moment vectors $\boldsymbol{f}$. Note that the expression under the root in Eqn. (10.54) is the dot product of a row vector and a column vector, that is, the result is a non-negative scalar value. The Mahalanobis distance can be viewed as a special form of the weighted Euclidean distance (Eqn. (10.53)), where the weights are determined by the variability of the individual vector components. See Sec. D.3 in the Appendix and Exercise 10.16 for additional details.

10.7 Projections

Image projections are 1D representations of the image contents, usually calculated parallel to the coordinate axis. In this case, the horizontal and vertical projections of a scalar-valued image $I(u,v)$ of size $M \times N$ are defined as

$$\mathsf{P}_{\mathrm{hor}}(v) = \sum_{u=0}^{M-1} I(u,v) \quad \text{for } 0 < v < N, \tag{10.55}$$

$$\mathsf{P}_{\mathrm{ver}}(u) = \sum_{v=0}^{N-1} I(u,v) \quad \text{for } 0 < u < M. \tag{10.56}$$

The *horizontal* projection $P_{\mathrm{hor}}(v_0)$ (Eqn. (10.55)) is the sum of the pixel values in the image *row* v_0 and has length N corresponding to the height of the image. On the other hand, a *vertical* projection P_{ver} of length M is the sum of all the values in the image *column* u_0 (Eqn. (10.56)). In the case of a binary image with $I(u,v) \in 0, 1$, the projection contains the count of the foreground pixels in the corresponding image row or column.

Program 10.4 gives a direct implementation of the projection calculations as the `run()` method for an ImageJ plugin, where projections in both directions are computed during a single traversal of the image.

Projections in the direction of the coordinate axis are often utilized to quickly analyze the structure of an image and isolate its component parts; for example, in document images it is used to separate graphic elements from text blocks as well as to isolate individual lines (see the example in Fig. 10.22). In practice, especially to account for document skew, projections are often computed along the major axis of an image region Eqn. (10.28). When the projection vectors of a region are computed in reference to the centroid of the region along the major axis, the result is a rotation-invariant vector description (often referred to as a "signature") of the region.

10.8 Topological Region Properties

Topological features do not describe the shape of a region in continuous terms; instead, they capture its structural properties. Topological

```
1  public void run(ImageProcessor I) {
2    int M = I.getWidth();
3    int N = I.getHeight();
4    int[] pHor = new int[N]; // = P_hor(v)
5    int[] pVer = new int[M]; // = P_ver(u)
6    for (int v = 0; v < N; v++) {
7      for (int u = 0; u < M; u++) {
8        int p = I.getPixel(u, v);
9        pHor[v] += p;
10       pVer[u] += p;
11     }
12   }   // use projections pHor, pVer now
13   // ...
14 }
```

Prog. 10.4
Calculation of horizontal and vertical projections. The `run()` method for an ImageJ plugin (`ip` is of type `ByteProcessor` or `ShortProcessor`) computes the projections in x and y directions simultaneously in a a single traversal of the image. The projections are represented by the one-dimensional arrays `horProj` and `verProj` with elements of type `int`.

Fig. 10.22
Horizontal and vertical projections of a binary image.

properties are typically invariant even under strong image transformations. The convexity of a region, which can be calculated from the convex hull (Sec. 10.4.2), is also a topological property.

A simple and robust topological feature is the *number of holes* $N_L(\mathcal{R})$ in a region. This feature is easily determined while finding the inner contours of a region, as described in Sec. 10.2.2.

A useful topological feature that can be derived directly from the number of holes is the so-called *Euler number* N_E, which is the difference between the number of connected regions N_R and the number of their holes N_L, that is,

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_L(\mathcal{R}). \tag{10.57}$$

In the case of a single connected region this is simply $1 - N_L$. For a picture of the number "**8**", for example, $N_E = 1 - 2 = -1$ and for the letter "**D**" we get $N_E = 1 - 1 = 0$.

Topological features are often used in combination with numerical features for classification. A classic example of this combination is OCR (optical character recognition) [38]. Figure 10.23 shows an

Fig. 10.23
Visual identification markers composed of recursively nested regions [22].

interesting use of topological structures for coding optical markers used in augmented reality applications [22].[17] The recursive nesting of outer and inner regions is equivalent to a tree structure that allows fast and unique identification of a larger number of known patterns (see also Exercise 10.21).

10.9 Java Implementation

Most algorithms described in this chapter are implemented as part of the `imagingbook` library.[18] The key classes are `BinaryRegion` and `Contour`, the abstract class `RegionLabeling` and its concrete subclasses `RecursiveLabeling`, `BreadthFirstLabeling`, `DepthFirstLabeling` (Alg. 10.1) and `SequentialLabeling` (Alg. 10.2). The combined region labeling and contour tracing method (Algs. 10.3 and 10.4) is implemented by class `RegionContourLabeling`. Additional details can be found in the online documentation.

Example

A complete example for the use of this API is shown in Prog. 10.5. Particularly useful is the facility for visiting all positions of a specific region using the iterator returned by method `getRegionPoints()`, as demonstrated by this code segment:

```
RegionLabeling segmenter =  ....
// Get the largest region:
BinaryRegion r = segmenter.getRegions(true).get(0);
// Loop over all points of region r:
for (Point p : r.getRegionPoints()) {
  int u = p.x;
  int v = p.y;
  // do something with position u, v
}
```

10.10 Exercises

Exercise 10.1. Manually simulate the execution of both variations (*depth-first* and *breadth-first*) of the flood-fill algorithm using the image in Fig. 10.24 and starting at position $(5, 1)$.

[17] http://reactivision.sourceforge.net/.
[18] Package `imagingbook.pub.regions`.

```
1  ...
2  import imagingbook.pub.regions.BinaryRegion;
3  import imagingbook.pub.regions.Contour;
4  import imagingbook.pub.regions.ContourOverlay;
5  import imagingbook.pub.regions.RegionContourLabeling;
6  import java.awt.geom.Point2D;
7  import java.util.List;
8
9  public class Region_Contours_Demo implements PlugInFilter {
10
11   public int setup(String arg, ImagePlus im) {
12     return DOES_8G + NO_CHANGES;
13   }
14
15   public void run(ImageProcessor ip) {
16     // Make sure we have a proper byte image:
17     ByteProcessor bp = ip.convertToByteProcessor();
18
19     // Create the region labeler / contour tracer:
20     RegionContourLabeling segmenter =
21             new RegionContourLabeling(bp);
22
23     // Get the list of detected regions (sort by size):
24     List<BinaryRegion> regions =
25             segmenter.getRegions(true);
26     if (regions.isEmpty()) {
27       IJ.error("No regions detected!");
28       return;
29     }
30
31     // List all regions:
32     IJ.log("Detected regions: " + regions.size());
33     for (BinaryRegion r: regions) {
34       IJ.log(r.toString());
35     }
36
37     // Get the outer contour of the largest region:
38     BinaryRegion largestRegion = regions.get(0);
39     Contour oc =  largestRegion.getOuterContour();
40     IJ.log("Points on outer contour of largest region:");
41     Point2D[] points = oc.getPointArray();
42     for (int i = 0; i < points.length; i++) {
43       Point2D p = points[i];
44       IJ.log("Point " + i + ": " + p.toString());
45     }
46
47     // Get all inner contours of the largest region:
48     List<Contour> ics = largestRegion.getInnerContours();
49     IJ.log("Inner regions (holes): " + ics.size());
50   }
51 }
```

10.10 Exercises

Prog. 10.5 Complete example for the use of the `regions` API. The ImageJ plugin `Region_Contours_Demo` segments the binary (8-bit grayscale) image `ip` into connected components. This is done with an instance of class `RegionContourLabeling` (see line 21), which also extracts the regions' contours. In line 25, a list of regions (sorted by size) is produced which is subsequently traversed (line 33). The treatment of outer and inner contours as well as the iteration over individual contour points is shown in lines 38–49.

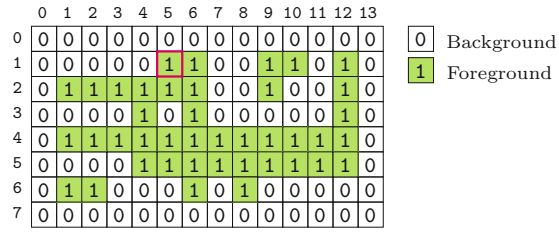


Fig. 10.24
Binary image for Exercise 10.1.

Exercise 10.2. The implementation of the flood-fill algorithm in Prog. 10.1 places all the neighboring pixels of each visited pixel into either the *stack* or the *queue* without ensuring they are foreground pixels and that they lie within the image boundaries. The number of items in the stack or the queue can be reduced by ignoring (not inserting) those neighboring pixels that do not meet the two conditions given. Modify the *depth-first* and *breadth-first* variants given in Prog. 10.1 accordingly and compare the new running times.

Exercise 10.3. The implementations of depth-first and breadth-first labeling shown in Prog. 10.1 will run significantly slower than the recursive version because the frequent creation of new `Point` objects is quite time consuming. Modify the *depth-first* version of Prog. 10.1 to use a stack with elements of a *primitive type* (e.g., `int`) instead. Note that (at least in Java)[19] it is not possible to specify a built-in list structure (such as `Deque` or `LinkedList`) for a primitive element type. Implement you own stack class that internally uses an `int`-array to store the (u, v) coordinates. What is the maximum number of stack entries needed for a given image of size $M \times N$? Compare the performance of your solution to the original version in Prog. 10.1.

Exercise 10.4. Implement an ImageJ plugin that encodes a given binary image by run length encoding (Sec. 10.3.2) and stores it in a file. Develop a second plugin that reads the file and reconstructs the image.

Exercise 10.5. Calculate the amount of memory required to represent a contour with 1000 points in the following ways: (a) as a sequence of coordinate points stored as pairs of `int` values; (b) as an 8-chain code using Java `byte` elements, and (c) as an 8-chain code using only 3 bits per element.

Exercise 10.6. Implement a Java class for describing a binary image region using chain codes. It is up to you, whether you want to use an absolute or differential chain code. The implementation should be able to encode closed contours as chain codes and also reconstruct the contours given a chain code.

Exercise 10.7. The *Graham Scan* method [91] is an efficient algorithm for calculating the convex hull of a 2D point set (of size n), with time complexity $\mathcal{O}(n \cdot \log(n))$.[20] Implement this algorithm and show that it is sufficient to consider only the outer contour points of a region to calculate its convex hull.

[19] Other languages like C# allow this.
[20] See also http://en.wikipedia.org/wiki/Graham_scan.

Exercise 10.8. While computing the convex hull of a region, the maximal diameter (maximum distance between two arbitrary points) can also be simply found. Devise an alternative method for computing this feature without using the convex hull. Determine the running time of your algorithm in terms of the number of points in the region.

Exercise 10.9. Implement an algorithm for comparing contours using their shape numbers Eqn. (10.6). For this purpose, develop a metric for measuring the distance between two normalized chain codes. Describe if, and under which conditions, the results will be reliable.

Exercise 10.10. Sketch the contour equivalent to the *absolute* chain code sequence $\boldsymbol{c}'_{\mathcal{R}} = (6, 7, 7, 1, 2, 0, 2, 3, 5, 4, 4)$. (a) Choose an arbitrary starting point and determine if the resulting contour is closed. (b) Find the associated *differential* chain code $\boldsymbol{c}''_{\mathcal{R}}$ (Eqn. (10.5)).

Exercise 10.11. Calculate (under assumed 8-neighborhood) the *shape number* of base $b = 8$ (see Eqn. (10.6)) for the differential chain code $\boldsymbol{c}''_{\mathcal{R}} = (1, 0, 2, 1, 6, 2, 1, 2, 7, 0, 2)$ and all possible circular shifts of this code. Which shift yields the maximum arithmetic value?

Exercise 10.12. Using Eqn. (10.14) as the basis, develop and implement an algorithm that computes the area of a region from its 8-chain-encoded contour (see also [263], [127, Sec. 19.5]).

Exercise 10.13. Modify Alg. 10.3 such that the outer and inner contours are not returned as individual lists $(\mathsf{C}_{\text{out}}, \mathsf{C}_{\text{in}})$ but as a composite tree structure. An outer contour thus represents a region that may contain zero, one, or more inner contours (i.e., holes). Each inner contour may again contain other regions (i.e., outer contours), and so on.

Exercise 10.14. Sketch an example binary region where the centroid does not lie inside the region itself.

Exercise 10.15. Implement the binary region moment features proposed by *Hu* (Eqn. (10.47)) and/or *Flusser* (Eqn. (10.51)) and verify that they are invariant under image scaling and rotation. Use the test image in Fig. 10.25[21] (or create your own), which contains rotated and mirrored instances of the reference shapes, in addition to other (unknown) shapes.

Exercise 10.16. Implement the Mahalanobis distance calculation, as defined in Eqn. (10.54), for measuring the similarity between shape moment vectors.

A. Compute the covariance matrix $\boldsymbol{\Sigma}$ (see Sec. D.3 in the Appendix) for the $m = 11$ Flusser shape features $\psi_1, \ldots, \psi_{11}$ of the reference images in Table 10.1. Calculate and tabulate the inter-class Mahalanobis distances for the reference shapes, analogous to the example in Table 10.2.

[21] Images are available on the book's website.

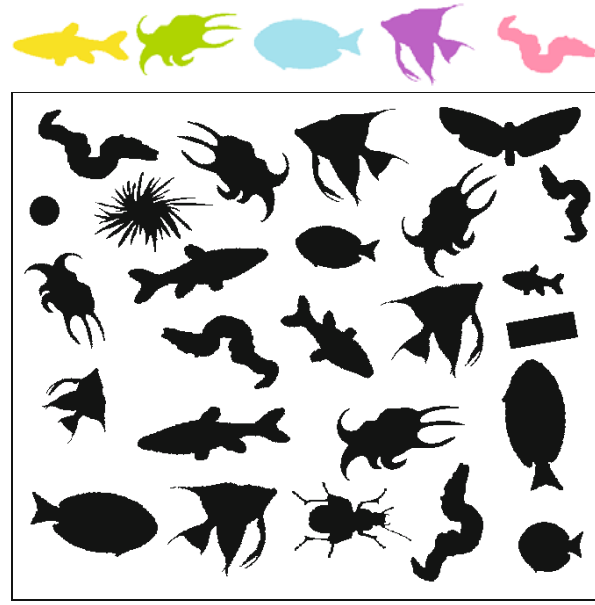


Fig. 10.25
Test image for moment-based shape matching. Reference shapes (top) and test image (bottom) composed of rotated and/or scaled shapes from the Kimia database and additional (unclassified) shapes.

B. Extend your analysis to a larger set of 500–1000 shapes (e.g., from the Kimia dataset [134], which contains more than 20 000 binary shape images). Calculate the normalized moment features and the covariance matrix $\mathbf{\Sigma}$ for the entire image set. Calculate the inter-class distance matrices for (a) the Euclidean and (b) the Mahalanobis distance. Display the distance matrices as grayscale images (`FloatProcessor`) and interpret them.

Exercise 10.17. There are alternative definitions for the *eccentricity* of a region Eqn. (10.34); for example [128, p. 394],

$$\mathsf{Ecc}_2(\mathcal{R}) = \frac{[\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})]^2 + 4 \cdot \mu_{11}^2(\mathcal{R})}{[\mu_{20}(\mathcal{R}) + \mu_{02}(\mathcal{R})]^2}. \tag{10.58}$$

Implement this version as well as the one in Eqn. (10.34) and contrast the results using suitably designed regions. Determine the numeric range of these quantities and test if they are really rotation and scale-invariant.

Exercise 10.18. Write an ImageJ plugin that (a) finds (labels) all regions in a binary image, (b) computes the orientation and eccentricity for each region, and (c) shows the results as a direction vector and the equivalent ellipse on top of each region (as exemplified in Fig. 10.19). Hint: Use Eqn. (10.39) to develop a method for drawing ellipses at arbitrary orientations (not available in ImageJ).

Exercise 10.19. The Java method in Prog. 10.4 computes an image's horizontal and vertical projections. The scheme described in Sec. 10.6.3 and illustrated in Fig. 10.20 can be used to calculate projections along arbitrary directions θ. Develop and implement such a process and display the resulting projections.

Exercise 10.20. Text recognition (OCR) methods are likely to fail if the document image is not perfectly axis-aligned. One method for estimating the skew angle of a text document is to perform binary segmentation and connected components analysis (see Fig. 10.26):

- *Smear* the original binary image by applying a disk-shaped morphological dilation with a specified radius (see Chapter 9, Sec. 9.2.3). The aim is to close the gaps between neighboring glyphs without closing the space between adjacent text lines (Fig. 10.26(b))
- Apply region segmentation to the resulting image and calculate the orientation $\theta(\mathcal{R})$ and the eccentricity $E(\mathcal{R})$ of each region $\mathcal{R}$ (see Secs. 10.6.1 and 10.6.2). Ignore all regions that are either too small or not sufficiently elongated.
- Estimate the global skew angle by averaging the regions' orientations θ_i. Note that, since angles are *circular*, they cannot be averaged in the usual way (see Chapter 15, Eqn. (15.14) for how to calculate the mean of a circular quantity). Consider using the eccentricity as a weight for the contribution of the associated region to the global average.
- Obviously, this scheme is sensitive to *outliers*, that is, against angles that deviate strongly from the average orientation. Try to improve this estimate (i.e., make it more robust and accurate) by iteratively removing angles that are "too far" from the average orientation and then recalculating the result.

Exercise 10.21. Draw the tree structure, defined by the recursive nesting of outer and inner regions, for each of the markers shown in Fig. 10.23. Based on this graph structure, suggest an algorithm for matching pairs of markers or, alternatively, for retrieving the best-matching marker from a database of markers.

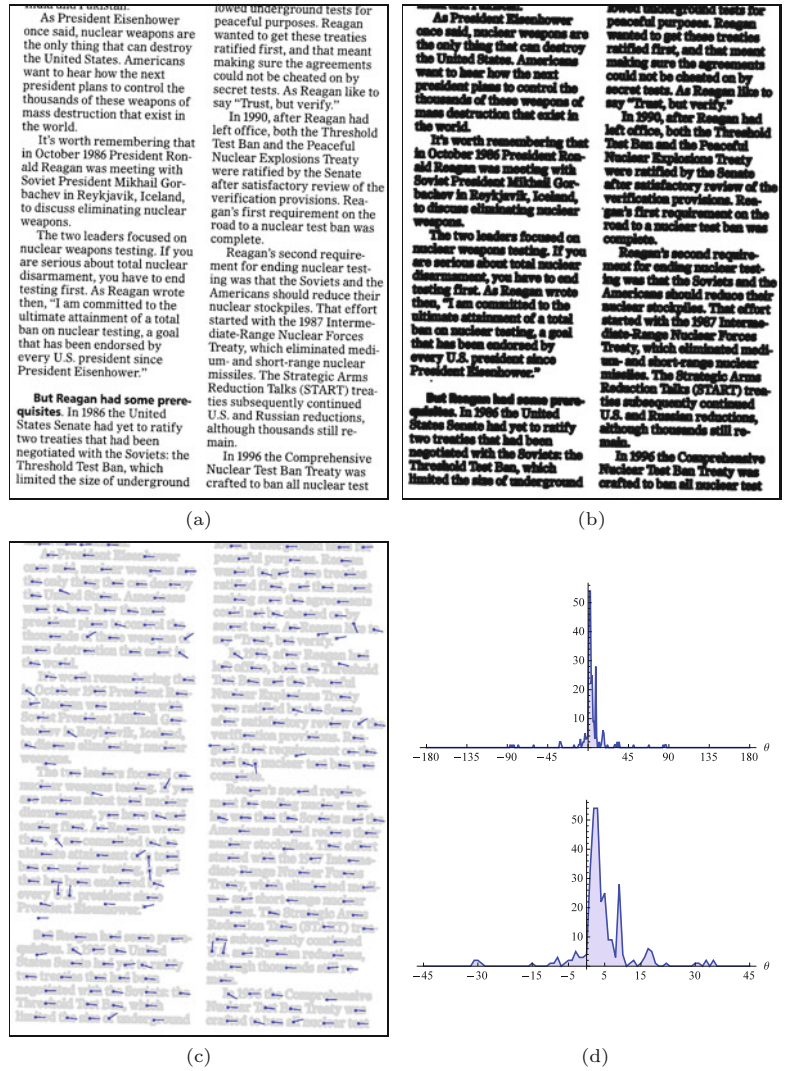


Fig. 10.26
Document skew estimation example (see Exercise 10.20). Original binary image (a); result of applying a disk-shaped morphological dilation with radius 3.0 (b); region orientation vectors (c); histogram of the orientation angle θ (d). The real skew angle in this scan is approximately 1.1°.