2
Regions in Binary Images

In binary images, a pixel 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 a figure like Fig. 2.1. As long as we continue

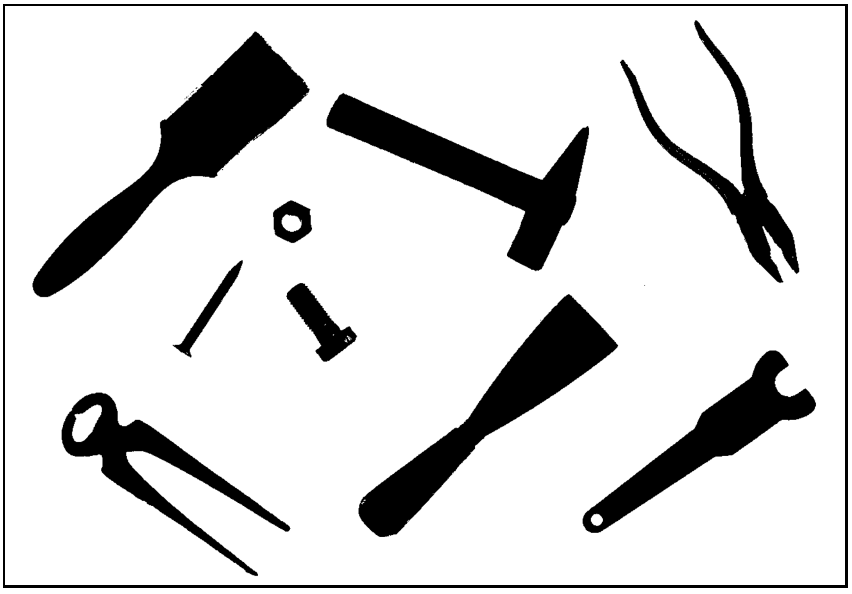


Figure 2.1 Binary image with nine objects. Each object corresponds to a connnected region of related foreground pixels.

W. Burger, M.J. Burge, *Principles of Digital Image Processing*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84800-195-4_2,

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*.

2.1 Finding 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. 2.2.2, we describe a third method that combines two useful processes, region labeling and contour finding, in a single algorithm.

Independent of which of the methods above we use, we must first settle on either the 4- or 8-connected definition of neighboring (see Vol. 1 [14, Fig. 7.5]) 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, i. e., the pixel values are

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

2.1.1 Region Labeling with 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 (Alg. 2.1). 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

Algorithm 2.1 Region marking using *flood filling* (Part 1). 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. The actual FloodFill() procedure is described in Alg. 2.2.

```
1:  RegionLabeling(I)
        I: binary image; I(u,v) = 0: background, I(u,v) = 1: foreground
        The image I is labeled (destructively modified) and returned.
2:    Let m ← 2                          ▷ value of the next label to be assigned
3:    for all image coordinates (u,v) do
4:      if I(u,v) = 1 then
5:        FloodFill(I, u, v, m)          ▷ use any version from Alg. 2.2
6:        m ← m + 1.
7:    return the labeled image I.
```

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. 2.2):

(A) **Recursive Flood Filling:** The recursive version (Alg. 2.2, lines 1–8) 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* [20] of this tree and results 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. 2.2, lines 9–20) 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

[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.

Algorithm 2.2 Region marking using *flood filling* (Part 2). Three variations of the FloodFill() procedure: *recursive*, *depth-first*, and *breadth-first*.

```
 1: FloodFill(I, u, v, label)                          ▷ Recursive Version
 2:     if (u, v) is inside the image and I(u, v) = 1 then
 3:         Set I(u, v) ← label
 4:         FloodFill(I, u+1, v, label)
 5:         FloodFill(I, u, v+1, label)
 6:         FloodFill(I, u, v−1, label)
 7:         FloodFill(I, u−1, v, label)
 8:     return.
 9: FloodFill(I, u, v, label)                          ▷ Depth-First Version
10:     Create an empty stack S
11:     Put the seed coordinate (u, v) onto the stack: Push(S, (u, v))
12:     while S is not empty do
13:         Get the next coordinate from the top of the stack:
                (x, y) ← Pop(S)
14:         if (x, y) is inside the image and I(x, y) = 1 then
15:             Set I(x, y) ← label
16:             Push(S, (x+1, y))
17:             Push(S, (x, y+1))
18:             Push(S, (x, y−1))
19:             Push(S, (x−1, y))
20:     return.
21: FloodFill(I, u, v, label)                          ▷ Breadth-First Version
22:     Create an empty queue Q
23:     Insert the seed coordinate (u, v) into the queue: Enqueue(Q, (u, v))
24:     while Q is not empty do
25:         Get the next coordinate from the front of the queue:
                (x, y) ← Dequeue(Q)
26:         if (x, y) is inside the image and I(x, y) = 1 then
27:             Set I(x, y) ← label
28:             Enqueue(Q, (x+1, y))
29:             Enqueue(Q, (x, y+1))
30:             Enqueue(Q, (x, y−1))
31:             Enqueue(Q, (x−1, y))
32:     return.
```

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. 2.2, lines 21–32). 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 corresponds practically 1:1 to its Java implementation. However, a normal Java runtime environment does not support more than about 10,000 recursive calls of the FloodFill() procedure (Alg. 2.2, line 1) 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 2.1 gives the complete Java implementation for both variants of the iterative FloodFill() procedure. In implementing the stack (S) in the iterative *depth-first* Version (B), we use the stack data structure provided by the Java class `Stack` (Prog. 2.1, line 1), which serves as a container for generic Java objects. For the queue data structure (Q) in the *breadth-first* variant (C), we use the Java class `LinkedList`[2] with the methods `addFirst()`, `removeLast()`, and `isEmpty()` (Prog. 2.1, line 18). We have specified `<Point>` as a type parameter for both generic container classes so they can only contain objects of type `Point`.[3]

Figure 2.2 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. 2.2 (using the implementation given Prog. 2.1), the stack in the *depth-first* variant

[2] The class `LinkedList` is a part of the *Java Collection Framework* (see also Vol. 1 [14, Appendix B.2]).

[3] Generic types and templates (i. e., the ability to specify a parameterization for a container) have only been available since Java 5 (1.5).

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

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

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

```
18 void floodFill(int x, int y, int label) {
19   LinkedList<Point> q = new LinkedList<Point>();
20   q.addFirst(new Point(x, y));
21   while (!q.isEmpty()) {
22     Point n = q.removeLast();
23     int u = n.x;
24     int v = n.y;
25     if ((u>=0) && (u<width) && (v>=0) && (v<height)
26           && ip.getPixel(u,v)==1) {
27       ip.putPixel(u, v, label);
28       q.addFirst(new Point(u+1, v));
29       q.addFirst(new Point(u, v+1));
30       q.addFirst(new Point(u, v-1));
31       q.addFirst(new Point(u-1, v));
32     }
33   }
34 }
```

Program 2.1 Flood filling (Java implementation). The standard class `Point` (defined in `java.awt`) represents a single pixel coordinate. The *depth-first* variant uses the standard stack operations provided by the methods `push()`, `pop()`, and `isEmpty()` of the Java class `Stack`. The *breadth-first* variant uses the Java class `LinkedList` (with access methods `addFirst()` for ENQUEUE() and `removeLast()` for DEQUEUE()) for implementing the queue data structure.

grows to a maximum of 28,822 elements, while the queue used by the *breadth-first* variant never exceeds a maximum of 438 nodes.

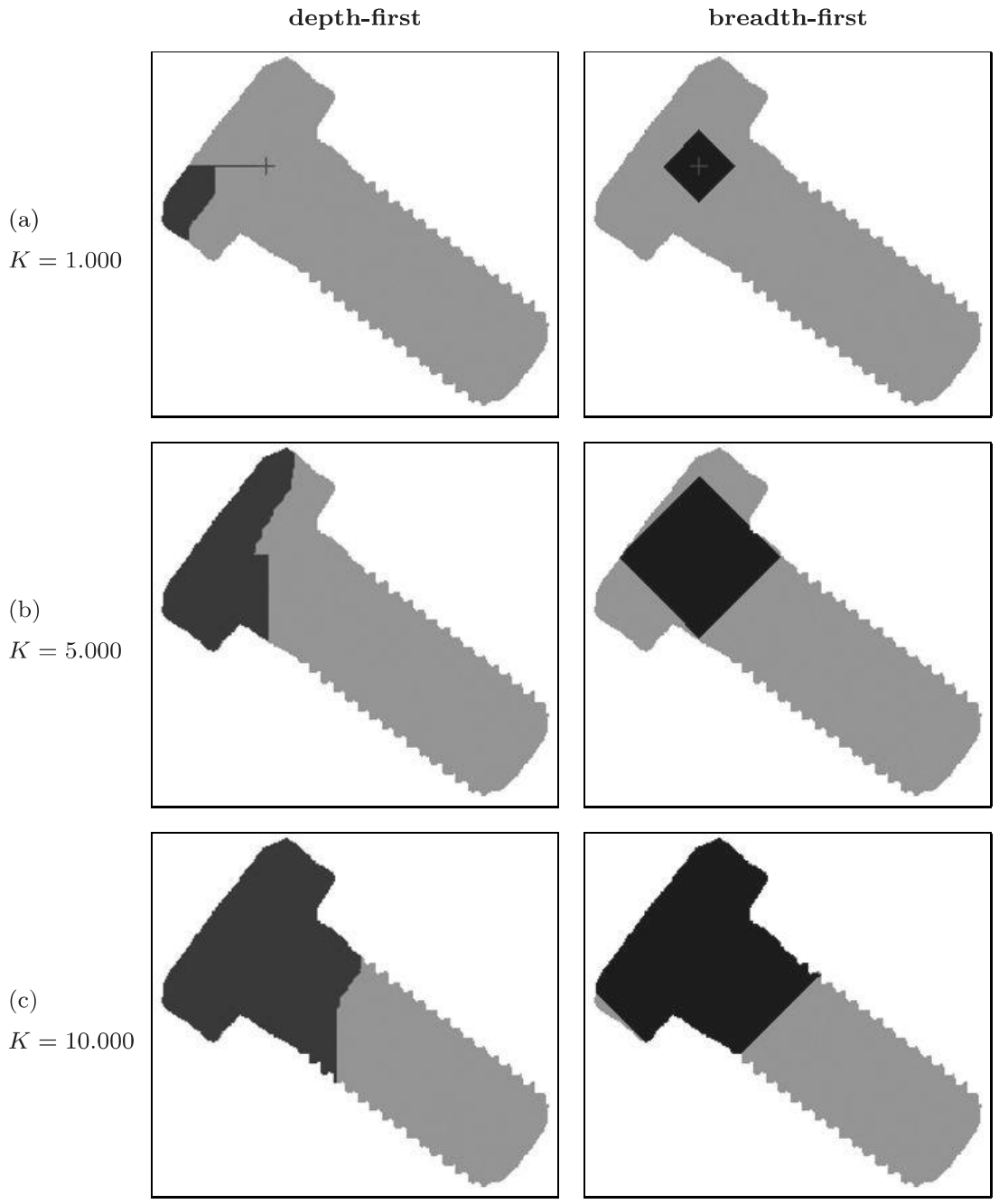


Figure 2.2 Iterative *flood filling*—comparison between the *depth-first* and *breadth-first* approach. The starting point, marked + in the top two image (a), was arbitrarily chosen. Intermediate results of the *flood fill* process after 1000 (a), 5000 (b), and 10,000 (c) marked pixels are shown. The image size is 250×242 pixels.

2.1.2 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) a 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. 2.3–2.4, with the following main steps:

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 ($\times$ marks the current pixel at the position (u, v)):

$$\mathcal{N}_4(u,v) = \begin{array}{|c|c|c|}\hline & N_2 & \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array} \qquad \text{or} \qquad \mathcal{N}_8(u,v) = \begin{array}{|c|c|c|}\hline N_2 & N_3 & N_4 \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array}$$

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.

Example

In the following example (Figs. 2.3–2.5), we use an 8-connected neighborhood and a very simple test image (Fig. 2.3 (a)) to demonstrate the sequential region labeling process.

Propagating 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. 2.3 (b, c)).

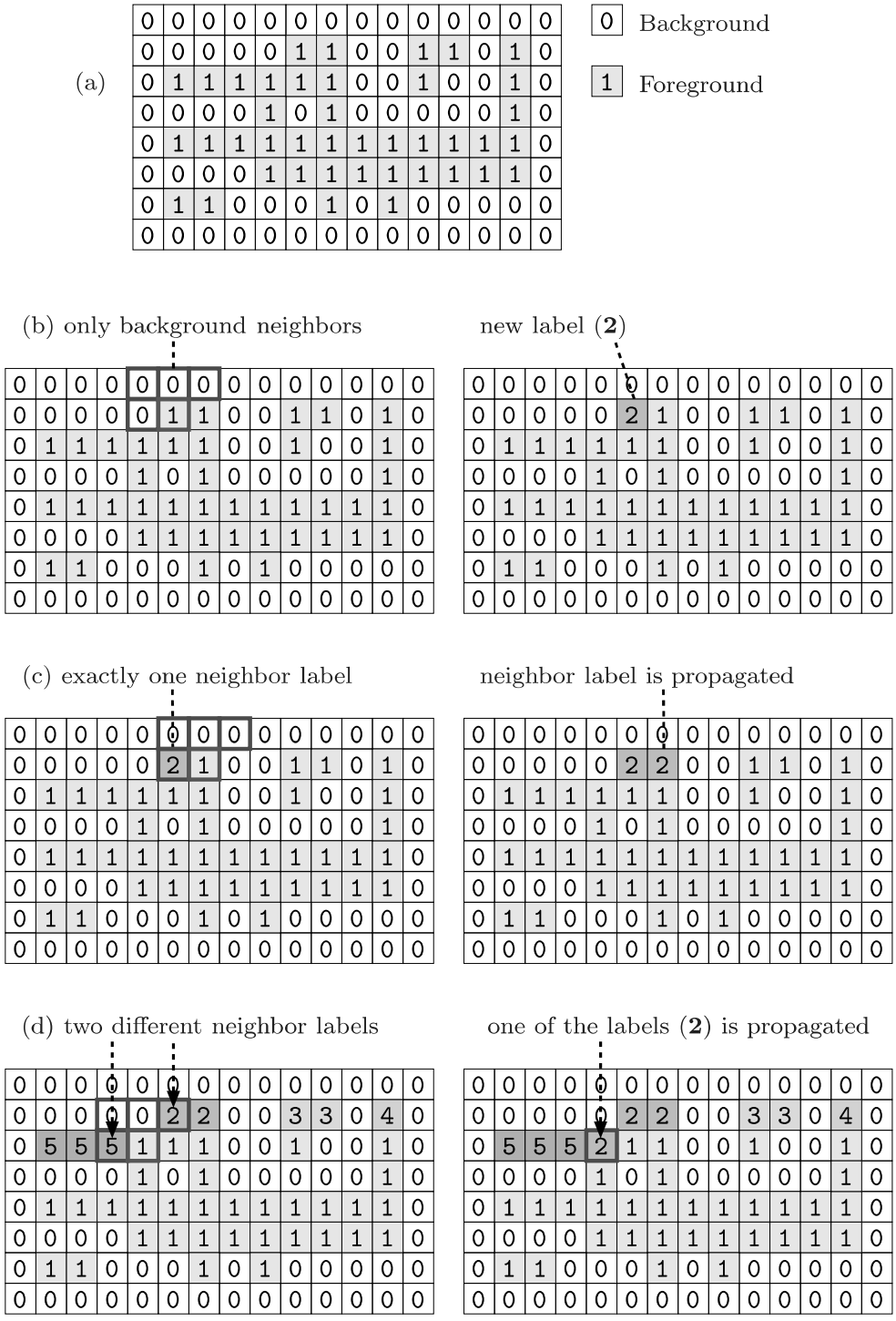


Figure 2.3 Sequential region labeling—label propagation. Original image (a). The first foreground pixel (marked **1**) is found in (b): all neighbors are background pixels (marked **0**), and the pixel is assigned the first label (**2**). In the next step (c), there is exactly *one* neighbor pixel marked 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 $\langle \mathbf{2}, \mathbf{5} \rangle$ is registered.

Algorithm 2.3 Sequential region labeling (Part 1). The binary input image I contains the values $I(u,v) = 0$ for background pixels and $I(u,v) = 1$ for foreground (region) pixels. The resulting region labels in I have the values $2 \ldots m-1$.

```
1:  SequentialLabeling(I)
        I: binary image; I(u,v) = 0: background, I(u,v) = 1: foreground
        The image I is labeled (destructively modified) and returned.
        m: number of assigned labels; C: set of label collisions.
2:      (m, C) ← AssignInitialLabels(I)
3:      R ← ResolveLabelCollisions(m, C)                    ▷ see Alg. 2.4
4:      RelabelImage(I, R)                                  ▷ see Alg. 2.4
5:      return I.
```

```
6:  AssignInitialLabels(I)
        Performs a preliminary labeling on image I (which is modified).
        Returns the number of assigned labels (m) and
        the set of detected label collisions (C).
7:      Initialize m ← 2 (the value of the next label to be assigned).
8:      C ← {}                                              ▷ empty set of collisions
9:      for v ← 0 ... H − 1 do                              ▷ H = height of image I
10:         for u ← 0 ... W − 1 do                          ▷ W = width of image I
11:             if I(u,v) = 1 then do one of:
12:                 if all neighbors of (u,v) are background pixels (all n_i = 0)
                        then
13:                     I(u,v) ← m
14:                     m ← m + 1
15:                 else if exactly one of the neighbors has a label value n_k > 1
                        then
16:                     set I(u,v) ← n_k
17:                 else if several neighbors of (u,v) have label values n_j > 1
                        then
18:                     Select one of them as the new label:
                            I(u,v) ← k ∈ {n_j}.
19:                     for all other neighbors of (u,v) with label values n_i > 1
                            and n_i ≠ k do
20:                         Create a new label collision: c_i = ⟨n_i, k⟩.
21:                         Record the collision: C ← C ∪ {c_i}
        Remark: The image I now contains label values 0, 2, ... m − 1.
22:     return (m, C).
```

continued in Alg. 2.4 ▷▷

Algorithm 2.4 Sequential region labeling (Part 2).

1: ResolveLabelCollisions($m, \mathcal{C}$)

Resolves the label collisions contained in the set $\mathcal{C}$.
Returns $\mathcal{R}$, a vector of sets that represents a partitioning
of the complete label set into equivalent labels.

2: Let $\mathcal{L} = \{2, 3, \ldots m-1\}$ be the set of preliminary region labels.

3: Create a partitioning of $\mathcal{L}$ as a *vector of sets*, one set for each label value:
$\mathcal{R} \leftarrow [\mathcal{R}_2, \mathcal{R}_3, \ldots, \mathcal{R}_{m-1}] = [\{2\}, \{3\}, \{4\}, \ldots, \{m-1\}]$,
so $\mathcal{R}_i = \{i\}$ for all $i \in \mathcal{L}$.

4: **for all** collisions $\langle a, b \rangle \in \mathcal{C}$ **do**

5: Find in $\mathcal{R}$ the sets $\mathcal{R}_a$, $\mathcal{R}_b$:
$\mathcal{R}_a \leftarrow$ the set that currently contains label a
$\mathcal{R}_b \leftarrow$ the set that currently contains label b

6: **if** $\mathcal{R}_a \neq \mathcal{R}_b$ (a and b are contained in different sets) **then**

7: Merge sets $\mathcal{R}_a$ and $\mathcal{R}_b$ by moving all elements of $\mathcal{R}_b$ to $\mathcal{R}_a$:
$\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \mathcal{R}_b, \quad \mathcal{R}_b \leftarrow \{\}$

Remark: All *equivalent* label values (i. e., all labels of pixels in the same region) are now contained in the same set $\mathcal{R}_i$ within $\mathcal{R}$.

8: **return** $\mathcal{R}$.

9: RelabelImage($I, \mathcal{R}$)

Relabels the image I using the label partitioning in $\mathcal{R}$.
The image I is modified.

10: **for all** image locations (u, v) **do**

11: **if** $I(u, v) > 1$ **then** ▷ $I(u, v)$ contains a region label

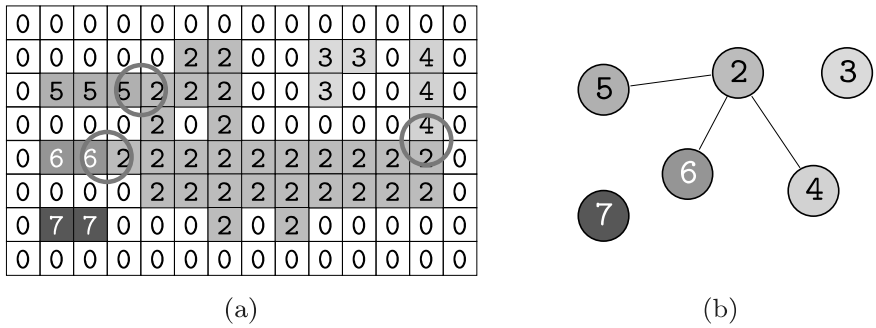
12: Find the set $\mathcal{R}_i$ in $\mathcal{R}$ that contains the label $I(u, v)$

13: Choose one unique representative element k from the set $\mathcal{R}_i$, e. g., the minimum value:
$k = \min(\mathcal{R}_i)$

14: Replace the image label:
$I(u, v) \leftarrow k$

15: **return**.

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



(a) (b)

Figure 2.4 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.

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. 2.3 (d)).

When two labels a, b collide, then we know that they are actually "equivalent"; i. e., 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. 2.4 illustrates the state upon completion of step 1: all foreground pixels have been assigned preliminary labels (Fig. 2.4 (a)), and the following collisions (depicted by circles) between the labels $\langle 2, 4\rangle$, $\langle 2, 5\rangle$, and $\langle 2, 6\rangle$ have been registered. The labels $\mathcal{L} = \{2, 3, 4, 5, 6, 7\}$ and collisions $\mathcal{C} = \{\langle 2, 4\rangle, \langle 2, 5\rangle, \langle 2, 6\rangle\}$ correspond to the nodes and edges of an undirected graph (Fig. 2.4 (b)).

Step 2: Resolving 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 different labels to be connected transitively (e. g., $\langle a, b\rangle \cap \langle b, c\rangle \Rightarrow \langle a, c\rangle$) 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 [20], 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. 2.4 (b)).

Step 3: Relabeling the image

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 (for example, chosing the smallest label number in the region), as shown in Fig. 2.5.

0	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

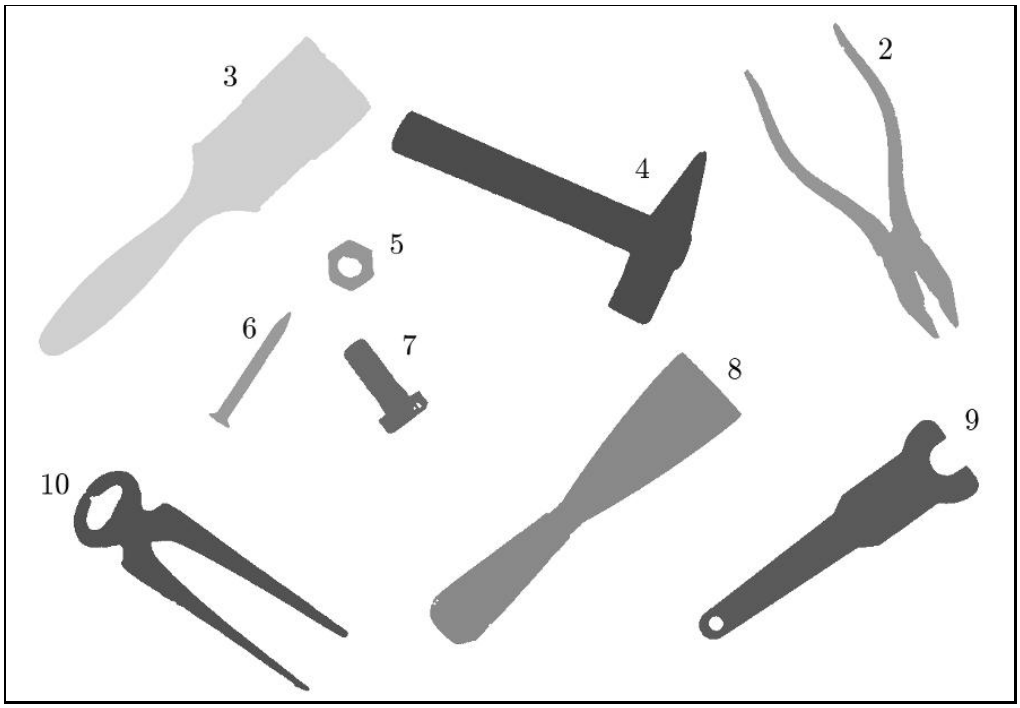
Figure 2.5 Sequential region labeling—final result after Step 3. All equivalent labels have been replaced by the smallest label within that region.

2.1.3 Region Labeling—Summary

In this section, we 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. 2.1.1) 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. 2.1.2) 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.

2.2 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.



Label	Area (*pixels*)	Bounding Box (*left*, *top*, *right*, *bottom*)	Center (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)

Figure 2.6 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 below (total image size is 1212×836).

2.2.1 External and Internal Contours

As we discussed in Vol. 1 [14, Sec. 7.2.7], the pixels along the edge of a binary region (that is, 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

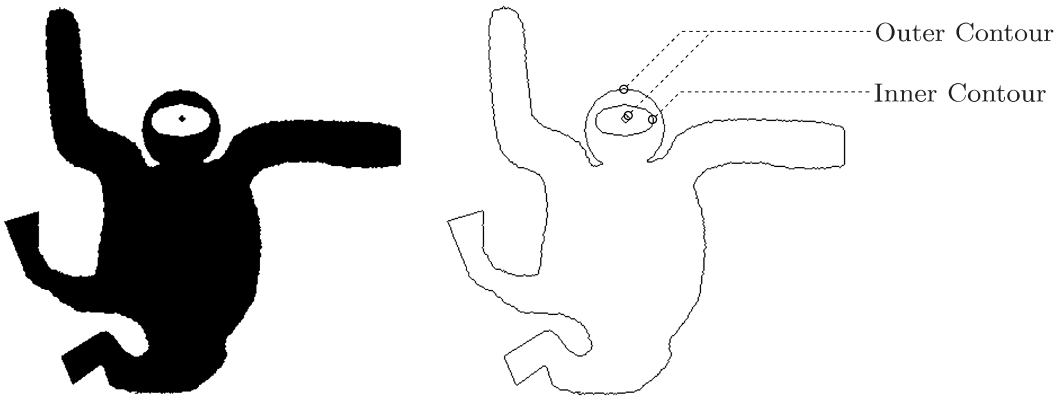


Figure 2.7 Binary image with outer and inner contours. The outer contour lies along the outside of the foreground region (dark). The inner contour surrounds the space within the region, which may contain further regions (holes), and so on.

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. 2.7).

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. 2.8). 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 the previous section (2.1); that is, 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 [61], [57, pp. 142–148], and [65, p. 296]. However, while the idea of contour tracing is simple in essence, the implementation requires careful record-keeping and is complicated by special cases such as the single-pixel bridges described in the previous section.

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

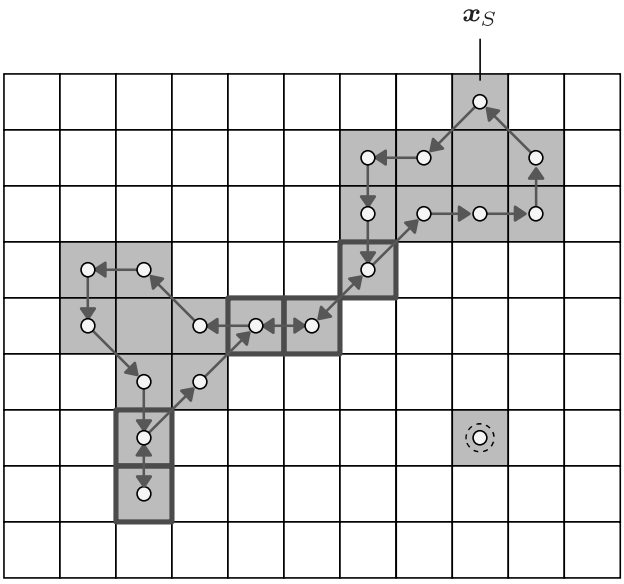


Figure 2.8 The path along a contour as an ordered sequence of pixel coordinates with a given start point $\boldsymbol{x}_S$. Individual pixels may occur (be visited) more than once within the path, and a region consisting of a single isolated pixel will also have a contour (bottom right).

2.2.2 Combining Region Labeling and Contour Finding

This method, based on [18], combines the concepts of sequential region labeling (Sec. 2.1) 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 very efficient when compared with other methods with similar capabilities. The key steps of this method are described below and illustrated in Fig. 2.9:

1. As in the sequential region labeling (Alg. 2.3), 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.

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

   Case A: The transition from a foreground pixel to a previously unmarked foreground pixel (A in Fig. 2.9 (a)) 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 Fig. 2.9 (a) and Alg. 2.5 (line 19)). Furthermore, all background pixels directly bordering the region are marked with the special label -1.

   Case B: The transition from a foreground pixel (B in Fig. 2.9 (b)) to an

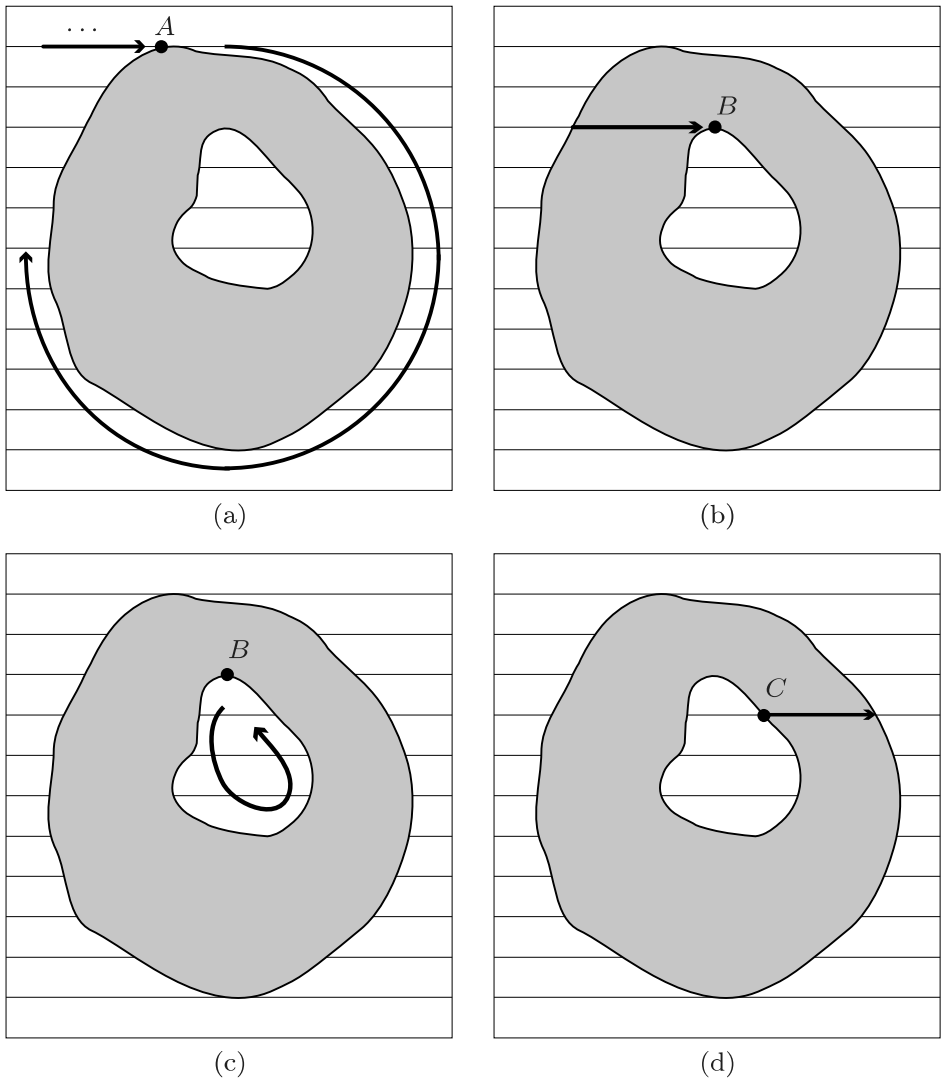


Figure 2.9 Combined region labeling and contour following (after [18]). The image is traversed from the top left to the lower right a row at a time. In (a), the first point A on the outer edge of the region is found. Starting from point A, the pixels on the edge along the outer contour are visited and labeled until A is reached again. In (b), the first point B on an inner contour is found. The pixels along the inner contour are visited and labeled until arriving back at B (c). In (d), an already labeled point C on an inner contour is found. Its label is propagated along the image row within the region.

unmarked background pixel means that this pixel lies on an *inner* contour. Starting from B, the inner contour is traversed and its pixels are marked with labels from the surrounding region (Fig. 2.9 (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. 2.9 (d)) and this label is propagated to the current pixel.

In Algorithms 2.5 and 2.6, the entire procedure is presented again and explained precisely. The method CombinedContourLabeling() traverses the image line-by-line and calls the method 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 the method FindNextPoint() in Alg. 2.6.

2.2.3 Implementation

While the main idea of the algorithm can be sketched out in a few simple steps, the actual implementation requires attention to a number of details, so we have provided the complete Java source for an ImageJ plugin implementation in Appendix B (pp. 283–293). The implementation closely follows the description in Algs. 2.5 and 2.6 but illustrates several additional details:[4]

- The task is performed by methods of the class `ContourTracer`. First the image I (`pixelArray`) and the associated label map L (`labelArray`) are enlarged by padding one layer of elements around their borders. The new pixels are marked as *background* (0) in the image I. This simplifies contour following and eliminates the need to handle a number of special situations.
- As contours are found they are turned into objects of class `Contour` and collected in two separate lists: `outerContours` and `innerContours`. Every contour consists of an ordered sequence of coordinate points of the standard class `Point` (defined in `java.awt`). The Java container class `ArrayList` (templated on the type `Point`) is used as a dynamic data structure for storing the point sequences of the outer and inner contours.
- The method `traceContour()` (see p. 289) traverses an outer or inner contour, beginning from the starting point $\boldsymbol{x}_S$ (`xS`, `yS`). It calls the method `findNextPoint()`, to determine the next contour point $\boldsymbol{x}_T$ (`xT`, `yT`) following $\boldsymbol{x}_S$:
  - In the case that no following point is found, then $\boldsymbol{x}_S = \boldsymbol{x}_T$ and the region (contour) consists of a single isolated pixel. The method `traceContour()` is finished.
  - In the other case the remaining contour points are found by repeatedly calling `findNextPoint()`, and for every successive pair of points the *current* point $\boldsymbol{x}_c$ (`xC`, `yC`) and the *previous* point $\boldsymbol{x}_p$ (`xP`, `yP`) are recorded. Only when *both* points correspond to the original starting

[4] In the following description the names in parentheses after the algorithmic symbols denote the corresponding identifiers used in the Java implementation.

Algorithm 2.5 Combined contour tracing and region labeling (Part 1). Given a binary image I, the method CombinedContourLabeling() 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 the method TraceContour() (line 19 and line 26). TraceContour() itself is described in Alg. 2.6.

```
1:  CombinedContourLabeling (I)
        I: binary image.
        Returns the sets of outer and inner contours and a label map.
2:      C_outer ← {},  C_inner ← {}          ▷ create two empty sets of contours
3:      Create a label map L of the same size as I and initialize:
4:      for all image locations (u, v) do
5:          L(u, v) ← 0                                        ▷ label map L
6:      R ← 0                                                  ▷ region counter R

        Scan the image from left to right and top to bottom:
7:      for v ← 0 ... N−1 do
8:          l ← 0                          ▷ set the current label l to "none"
9:          for u ← 0 ... M−1 do
10:             if I(u, v) is a foreground pixel then
11:                 if (l ≠ 0) then                      ▷ continue inside region
12:                     L(u, v) ← l
13:                 else
14:                     l ← L(u, v)
15:                     if (l = 0) then                  ▷ hit a new outer contour
16:                         R ← R + 1
17:                         l ← R
18:                         x_S ← (u, v)
19:                         c ← TraceContour(x_S, 0, l, I, L)
20:                         C_outer ← C_outer ∪ {c}      ▷ collect outer contour
21:                         L(u, v) ← l
22:             else                                     ▷ I(u, v) is a background pixel
23:                 if (l ≠ 0) then
24:                     if (L(u, v) = 0) then            ▷ hit new inner contour
25:                         x_S ← (u−1, v)
26:                         c ← TraceContour(x_S, 1, l, I, L)
27:                         C_inner ← C_inner ∪ {c}      ▷ collect inner contour
28:                     l ← 0
29:     return (C_outer, C_inner, L).        ▷ return the contour sets and label map
```

continued in Alg. 2.6 ▷▷

Algorithm 2.6 Combined contour finding and region labeling (Part 2, continued from Alg. 2.5). 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 FindNextPoint() to determine the next point along the contour (line 10). The function Delta() returns the next coordinate in the sequence, taking into account the search direction d.

```
1:  TraceContour(x_S, d_S, l, I, L)
        x_S: start position,
        d_S: initial search direction (0 for outer, 1 for inner contours),
        l: label for this contour, I: original image, L: label map.
        Traces and returns the contour starting at x_S.
2:      (x_T, d_next) ← FindNextPoint(x_S, d_S, I, L)
3:      c ← [x_T]                          ▷ create a contour starting with x_T
4:      x_p ← x_S                          ▷ previous position x_p = (u_p, v_p)
5:      x_c ← x_T                          ▷ current position x_c = (u_c, v_c)
6:      done ← (x_S ≡ x_T)                 ▷ isolated pixel?
7:      while (¬done) do
8:          L(u_c, v_c) ← l
9:          d_search ← (d_next + 6) mod 8
10:         (x_n, d_next) ← FindNextPoint(x_c, d_search, I, L)
11:         x_p ← x_c
12:         x_c ← x_n
13:         done ← (x_p ≡ x_S ∧ x_c ≡ x_T)   ▷ back at start point?
14:         if (¬done) then
15:             Append(c, x_n)             ▷ add point x_n to contour c
16:     return c.                          ▷ return this contour
17: FindNextPoint(x_c, d, I, L)
        x_c: start point, d: search direction,
        I: original image, L: label map.
18:     for i ← 0...6 do                   ▷ search in 7 directions
19:         x' ← x_c + Delta(d)            ▷ x' = (u', v')
20:         if I(u', v') is a background pixel then
21:             L(u', v') ← −1             ▷ mark background as visited (−1)
22:             d ← (d + 1) mod 8
23:         else                           ▷ found a nonbackground pixel at x'
24:             return (x', d)
25:     return (x_c, d).                   ▷ found no next point, return start point
```

26: Delta$(d) = (\Delta x, \Delta y)$, 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

```
1 import java.util.List;
2 ...
3 public class Trace_Contours implements PlugInFilter {
4   public void run(ImageProcessor ip) {
5     ContourTracer tracer = new ContourTracer(ip);
6     // extract contours and regions
7     List<Contour> outerContours = tracer.getOuterContours();
8     List<Contour> innerContours = tracer.getInnerContours();
9     List<BinaryRegion> regions = tracer.getRegions();
10    ...
11  }
12 }
```

Program 2.2 Example of using the class `ContourTracer`. See Appendix B.1 for a listing of the complete implementation.

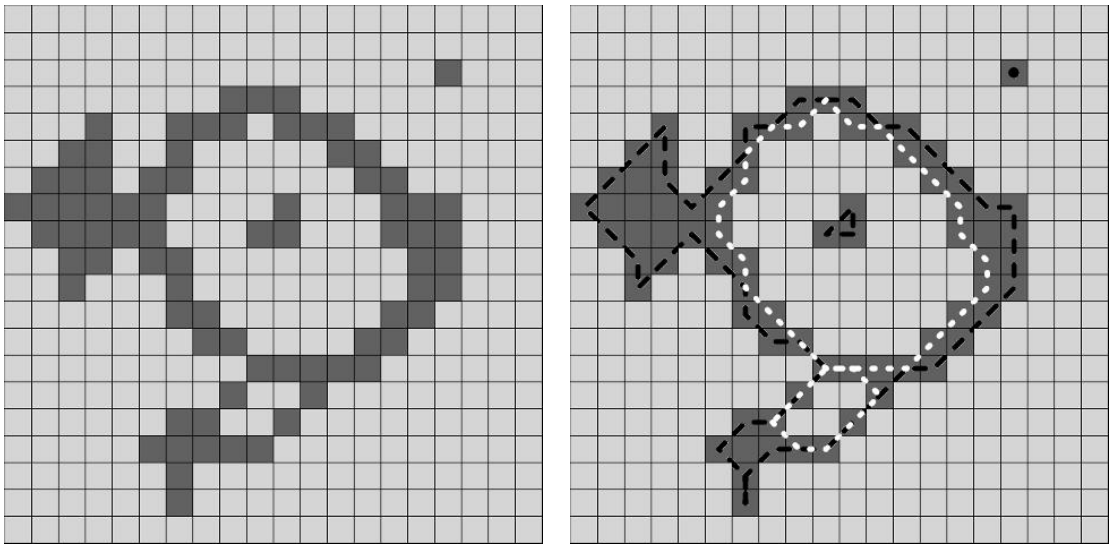
points on the contour, $\boldsymbol{x}_p = \boldsymbol{x}_S$ and $\boldsymbol{x}_c = \boldsymbol{x}_T$, we know that the contour has been completely traversed.

– The method `findNextPoint()` (see p. 290) determines which point on the contour follows the current point $\boldsymbol{x}_c$ (`xC`, `yC`) by searching in the *direction* d (`dir`), depending upon the position of the previous contour point. Starting in the first search direction, up to seven neighboring pixels (all neighbors except the previous contour point) are searched in clockwise direction until the next contour point is found. At the same time, all background pixels in the *label map* L (`labelArray`) are marked with the value -1 to prevent them from being searched again. If no valid contour point is found among the seven possible neighbors, then `findNextPoint()` returns the original point $\boldsymbol{x}_c$ (`xC`, `yC`).

In this implementation the core of the algorithm is contained in the class `ContourTracer` (pp. 287–292). Program 2.2 provides an example of its usage within the `run()` method of an ImageJ plugin. An interesting detail is the class `ContourOverlay` (pp. 292–293) that is used to display the resulting contours by a vector graphics overlay. In this way graphic structures that are smaller and thinner than image pixels can be visualized on top of ImageJ's raster images at arbitrary magnification (zooming).

2.2.4 Example

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 2.10 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



(a) (b)

Figure 2.10 Combined contour and region marking: original image in gray (a), located contours (b) with black lines for out and white lines for inner contours. The contour consisting of singe isolated pixels (for example, in the upper-right of (b)) are marked by a single circle in the appropriate color.

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 2.11 shows the results for a larger section taken from a real image (Vol. 1 [14, Fig. 7.12]).

2.3 Representing Image Regions

2.3.1 Matrix Representation

A natural representation for images is a matrix (that is, 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. 2.12). Since Boolean values can be represented by a single bit, such

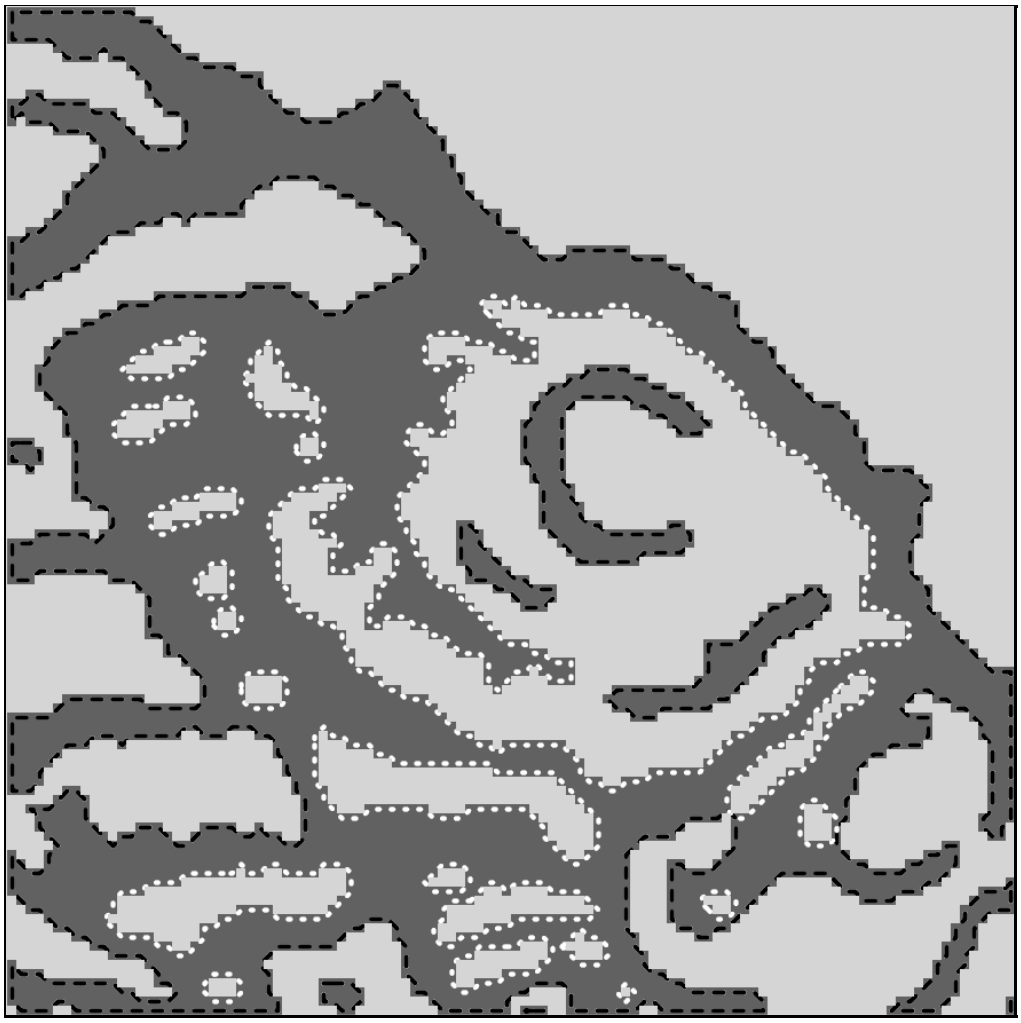


Figure 2.11 Example of a complex contour (in a section cut from Fig. 7.12 in Vol. 1 [14]). Outer contours are marked in black and inner contours in white.

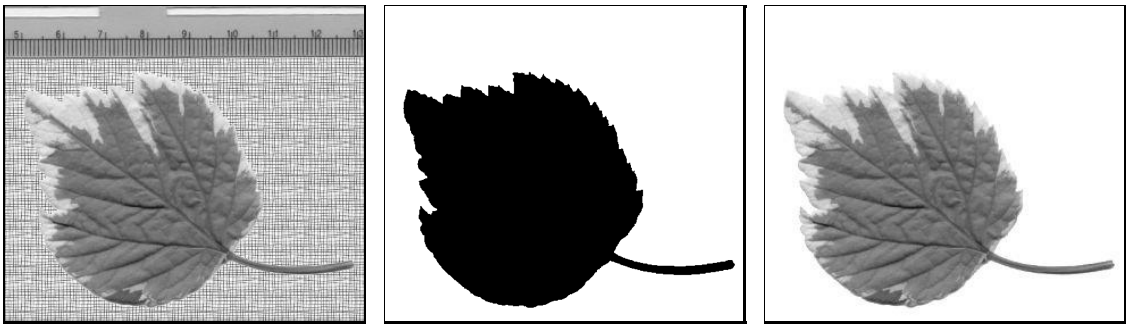
a matrix is often referred to as a "bitmap".[5]

2.3.2 Run Length Encoding

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

$$Run_i = \langle \mathsf{row}_i, \mathsf{column}_i, \mathsf{length}_i \rangle,$$

[5] In Java, variables of the type `boolean` are represented internally within the Java virtual machine (JVM) as 32-bit `int`s. There is currently no direct way to implement genuine bitmaps in Java.



(a) (b) (c)

Figure 2.12 Use of a binary mask to specify a region of an image: original image (a), logical (bit) mask (b), and masked image (c).

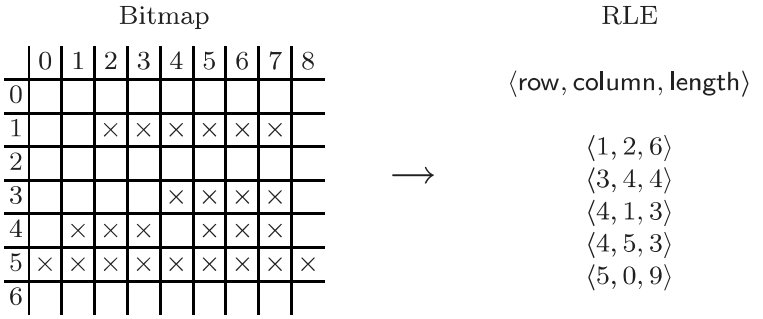


Figure 2.13 Run length encoding in row direction. A run of pixels can be represented by its starting point $(1, 2)$ and its length (6).

two to represent the starting pixel (row, column) and a third for the length of the run as illustrated in Fig. 2.13. 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.

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. 2.4.3).

2.3.3 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 [25], are a classical method of contour encoding. In this encoding, the contour beginning

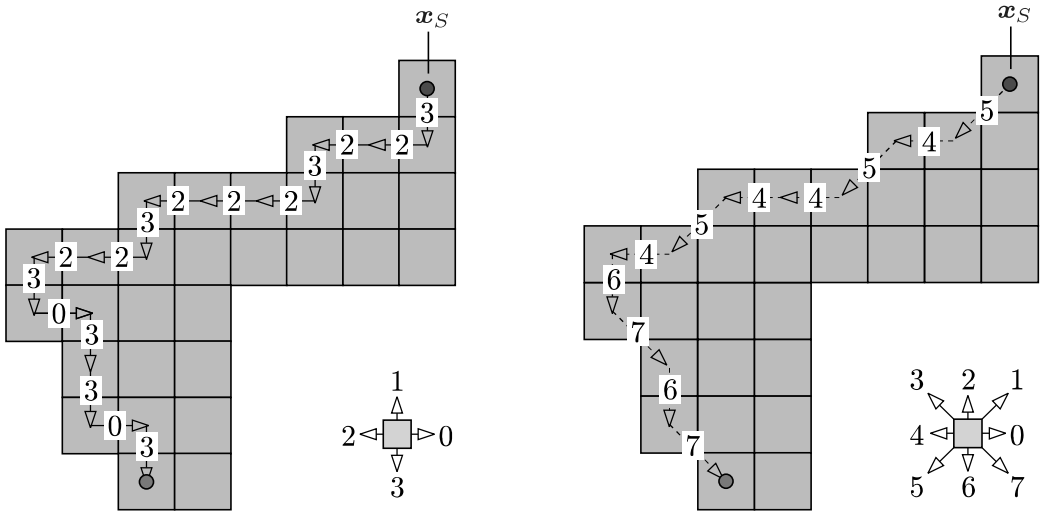


Figure 2.14 Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point $\boldsymbol{x}_S$. Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) 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.

at a given start point $\boldsymbol{x}_S$ is represented by the sequence of directional changes it describes on the discrete image raster (Fig. 2.14).

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 $\boldsymbol{c}'_\mathcal{R} = [c'_0, c'_1, \ldots c'_{M-1}]$ by

$$c'_i = \textsc{Code}(\Delta u_i, \Delta v_i), \tag{2.1}$$

$$\text{where} \quad (\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \le i < M-1 \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1, \end{cases}$$

and $\textsc{Code}(\Delta u, \Delta v)$ being defined by the following table:[6]

Δu	1	1	0	−1	−1	−1	0	1
Δv	0	1	1	1	0	−1	−1	−1
$\textsc{Code}(\Delta u, \Delta v)$	0	1	2	3	4	5	6	7

[6] Assuming an 8-connected neighborhood.

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

$$c''_i = \begin{cases} (c'_{i+1} - c'_i) \bmod 8 & \text{for } 0 \le i < M-1 \\ (c'_0 - c'_i) \bmod 8 & \text{for } i = M-1, \end{cases} \tag{2.2}$$

again under the assumption of an 8-connected neighborhood.[7] 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. 2.14 (b), the results are

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

Given the starting point $\boldsymbol{x}_S$ 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 [2, 28] is to interpret the elements c''_i in the differential chain code as the digits of

[7] See Vol. 1 [14, Appendix B.1.2] for implementing the mod operator used in Eqn. (2.2).

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

$$\begin{aligned} \textsc{Val}(\boldsymbol{c}''_{\mathcal{R}}) &= c''_0 \cdot b^0 + c''_1 \cdot b^1 + \ \ldots \ + c''_{M-1} \cdot b^{M-1} \\ &= \sum_{i=0}^{M-1} c''_i \cdot b^i. \end{aligned} \tag{2.3}$$

Then the sequence $\boldsymbol{c}''_{\mathcal{R}}$ is shifted cyclically 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 cyclically shifted by k positions to the right,[8] such as (for $k = 2$)

$$\begin{aligned} \boldsymbol{c}''_{\mathcal{R}} &= [\,0, 1, 3, 2, \ldots 9, 3, 7, 4\,] \\ \boldsymbol{c}''_{\mathcal{R}} \rhd 2 &= [\,7, 4, 0, 1, 3, 2, \ldots 9, 3\,] \end{aligned}$$

and

$$k_{\max} = \arg \max_{0 \le k < M} \textsc{Val}(\boldsymbol{c}''_{\mathcal{R}} \rhd k) \tag{2.4}$$

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

$$\boldsymbol{s}_{\mathcal{R}} = \boldsymbol{c}''_{\mathcal{R}} \rhd k_{\max}, \tag{2.5}$$

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 $\textsc{Val}()$ in Eqn. (2.3) produces values that are in general too large to be actually computed, in practice the relation

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

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 2.4 presents a number of tools that are more appropriate in these types of cases.

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

Fourier descriptors

An elegant approach to describing contours are so-called Fourier descriptors, which interpret the two-dimensional contour $\boldsymbol{c}_{\mathcal{R}} = [\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1}]$ with $\boldsymbol{x}_i = (u_i, v_i)$ as a sequence of values $[z_0, z_1 \ldots z_{M-1}]$ in the complex plane, where

$$z_i = (u_i + \mathrm{i} \cdot v_i) \in \mathbb{C}. \tag{2.6}$$

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 one-dimensional *Fourier spectrum* (see Sec. 7.3) 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 [28, 30, 46, 47, 69].

2.4 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 [21, 55, 72].

2.4.1 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.

2.4.2 Geometric Features

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

$$\mathcal{R} = \{\boldsymbol{x}_0, \boldsymbol{x}_1 \dots \boldsymbol{x}_{N-1}\} = \{(u_0, v_0), (u_1, v_1) \dots (u_{N-1}, v_{N-1})\}.$$

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. 2.1, 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. 2.14, 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 $\boldsymbol{c}'_{\mathcal{R}} = [c'_0, c'_1, \dots c'_{M-1}]$, the perimeter of the region is arrived at by

$$\mathsf{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \mathrm{length}(c'_i), \tag{2.7}$$

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

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 satisfactory even for relatively small regions:

$$P(\mathcal{R}) \approx \mathsf{Perimeter}_{\mathrm{corr}}(\mathcal{R}) = 0.95 \cdot \mathsf{Perimeter}(\mathcal{R}). \tag{2.8}$$

Area

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

$$A(\mathcal{R}) = |\mathcal{R}| = N. \tag{2.9}$$

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:

$$A(\mathcal{R}) \approx \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} \big(u_i \cdot v_{(i+1) \bmod M} \; - \; u_{(i+1) \bmod M} \cdot v_i\big) \right|. \tag{2.10}$$

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 using Eqn. (2.10) by expanding $\boldsymbol{c}'_{\mathcal{R}}$ into a sequence of contour points, using an arbitrary starting point (e. g., $(0, 0)$).

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 described below, it is possible to specify combined features that are *invariant* to translation, rotation, *and* scaling as well.

Compactness and roundness

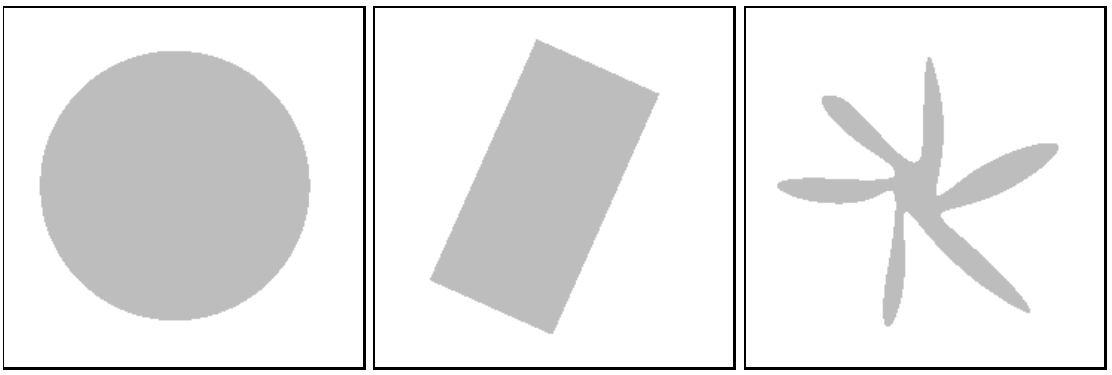
Compactness is understood as the relation between a region's area and its perimeter. We can use the fact that a region's perimeter P increases linearly with the enlargement factor while the area A increases quadratically to see that, for a particular shape, the ratio A/P^2 should be the same at any scale. 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,

$$\mathsf{Circularity}(\mathcal{R}) = \; 4\pi \cdot \frac{A(\mathcal{R})}{P^2(\mathcal{R})}, \tag{2.11}$$

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. 2.15). If an absolute value for a region's roundness is required, the corrected perimeter estimate (Eqn. (2.8)) should be employed:

$$\mathsf{Circularity}(\mathcal{R}) \approx \; 4\pi \cdot \frac{A(\mathcal{R})}{\mathsf{Perimeter}^2_{\mathrm{corr}}(\mathcal{R})}. \tag{2.12}$$

Figure 2.15 shows the circularity values of different regions as computed with the formulation in Eqn. (2.12).



1.001 0.672 0.086

Figure 2.15 Circularity values for different shapes. Shown are the corresponding estimates for $\mathsf{Circularity}(\mathcal{R})$ as defined in Eqn. (2.12).

Bounding box

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

$$\mathsf{BoundingBox}(\mathcal{R}) = \langle u_{\min}, u_{\max}, v_{\min}, v_{\max} \rangle, \tag{2.13}$$

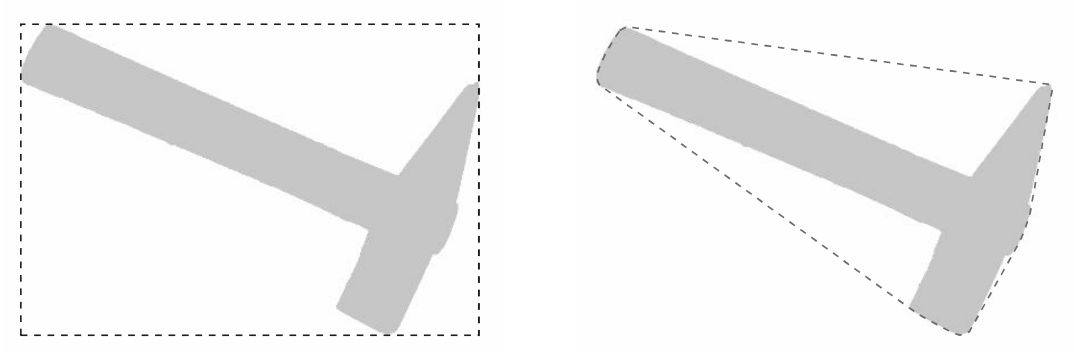
where $u_{\min}, u_{\max}$ and $v_{\min}, v_{\max}$ are the minimal and maximal coordinate values of all points $(u_i, v_i) \in \mathcal{R}$ in the x and y directions, respectively (Fig. 2.16 (a)).

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 (Fig. 2.16 (b)). The convex hull can be computed for N contour points in time $\mathcal{O}(N \log V)$, where V is the number vertices in the polygon of the resulting convex hull [3].[9]

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.

[9] For $\mathcal{O}()$ complexity notation, see Vol. 1 [14, Appendix A.3].



(a) (b)

Figure 2.16 Example bounding box (a) and convex hull (b) of a binary image region.

2.4.3 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*.

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.[10]

The centroid $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ of a binary (not necessarily connected) region is the arithmetic mean of the coordinates in the x and y directions,

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u \qquad \text{and} \qquad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} v \,. \tag{2.14}$$

[10] 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.

Moments

The formulation of the region's centroid in Eqn. (2.14) is only a special case of the more general statistical concept of a *moment*. Specifically, the expression

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot u^p v^q \tag{2.15}$$

describes the (ordinary) moment of the 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 computed directly from the coordinates of the contour points [64, 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. (2.15) can be simplified to

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} u^p v^q. \tag{2.16}$$

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

$$A(\mathcal{R}) = |\mathcal{R}| = \sum_{(u,v)\in\mathcal{R}} 1 = \sum_{(u,v)\in\mathcal{R}} u^0 v^0 = m_{00}(\mathcal{R}), \tag{2.17}$$

and similarly the *centroid* $\bar{\boldsymbol{x}}$ Eqn. (2.14) as

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^1 v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})}, \tag{2.18}$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^0 v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})}. \tag{2.19}$$

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.

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:

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot (u-\bar{x})^p \cdot (v-\bar{y})^q. \tag{2.20}$$

For a binary image (with $I(u,v) = 1$ within the region $\mathcal{R}$), Eqn. (2.20) can be simplified to

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} (u-\bar{x})^p \cdot (v-\bar{y})^q. \tag{2.21}$$

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

$$s^{(p+q+2)}. \tag{2.22}$$

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

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq}(\mathcal{R}) \cdot \Big(\frac{1}{\mu_{00}(\mathcal{R})}\Big)^{(p+q+2)/2} \tag{2.23}$$

for $(p+q) \geq 2$ [46, p. 529].

Program 2.3 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, for example, [48]).

2.4.4 Moment-Based Geometrical Properties

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

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. 2.18 (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

```
1  import ij.process.ImageProcessor;
2
3  public class Moments {
4    static final int BACKGROUND = 0;
5
6    static double moment(ImageProcessor ip,int p,int q) {
7      double Mpq = 0.0;
8      for (int v = 0; v < ip.getHeight(); v++) {
9        for (int u = 0; u < ip.getWidth(); u++) {
10         if (ip.getPixel(u,v) != BACKGROUND) {
11           Mpq += Math.pow(u, p) * Math.pow(v, q);
12         }
13       }
14     }
15     return Mpq;
16   }
17   static double centralMoment(ImageProcessor ip,int p,int q)
18     {
19     double m00 = moment(ip, 0, 0); // region area
20     double xCtr = moment(ip, 1, 0) / m00;
21     double yCtr = moment(ip, 0, 1) / m00;
22     double cMpq = 0.0;
23     for (int v = 0; v < ip.getHeight(); v++) {
24       for (int u = 0; u < ip.getWidth(); u++) {
25         if (ip.getPixel(u,v) != BACKGROUND) {
26           cMpq +=
27             Math.pow(u - xCtr, p) *
28             Math.pow(v - yCtr, q);
29         }
30       }
31     }
32     return cMpq;
33   }
34   static double normalCentralMoment
35                         (ImageProcessor ip,int p,int q) {
36     double m00 = moment(ip, 0, 0);
37     double norm = Math.pow(m00, (double)(p + q + 2) / 2);
38     return centralMoment(ip, p, q) / norm;
39   }
40
41 } // end of class Moments
```

Program 2.3 Example of directly computing moments in Java. The methods `moment()`, `centralMoment()`, and `normalCentralMoment()` compute for a binary image the moments m_{pq}, μ_{pq}, and $\bar{\mu}_{pq}$ (Eqns. (2.16), (2.21), and (2.23)).

pencil between your hands and twist it around its major axis (that is, around the lead), the pencil exhibits the least mass inertia (Fig. 2.17). 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

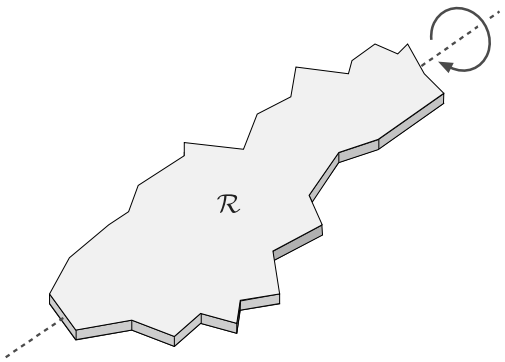


Figure 2.17 Major axis of a region. Rotating an elongated region $\mathcal{R}$, interpreted as a physical body, around its major axis requires less effort (least moment of inertia) than rotating it around any other axis.

$$\tan(2\,\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \tag{2.24}$$

and therefore

$$\theta_{\mathcal{R}} = \frac{1}{2}\tan^{-1}\left(\frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})}\right) \tag{2.25}$$

$$= \frac{\mathrm{Arctan}\big(2 \cdot \mu_{11}(\mathcal{R}),\ \mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})\big)}{2}. \tag{2.26}$$

The resulting angle $\theta_{\mathcal{R}}$ is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.[11] Orientation measurements based on region moments are very accurate in general.

Computing orientation vectors. When visualizing region properties, a frequent task is to plot the region's orientation as a line or arrow, that are usually anchored at the center of gravity $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$; for example, by a parametric line of the form

$$\boldsymbol{x} = \bar{\boldsymbol{x}} + \lambda \cdot \boldsymbol{x}_d = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \lambda \cdot \begin{pmatrix} \cos(\theta_{\mathcal{R}}) \\ \sin(\theta_{\mathcal{R}}) \end{pmatrix}, \tag{2.27}$$

for some length $\lambda > 0$. To find the unit orientation vector $\boldsymbol{x}_d = (\cos\theta, \sin\theta)^T$, we could first compute the inverse tangent to get 2θ (Eqn. (2.25)) 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. (2.24) as

$$\tan(2\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} = \frac{A}{B} = \frac{\sin(2\theta_{\mathcal{R}})}{\cos(2\theta_{\mathcal{R}})} \tag{2.28}$$

[11] See Appendix A.1 for the computation of angles with the Arctan() (inverse tangent) function and Vol. 1 [14, Appendix B.1.6] for the corresponding Java method `Math.atan2()`.

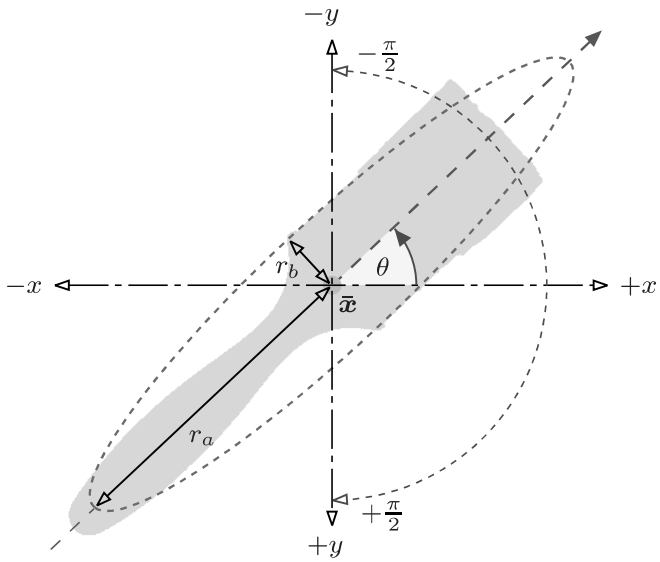


Figure 2.18 Region orientation and eccentricity. The major axis of the region extends through its center of gravity $\bar{\boldsymbol{x}}$ at the orientation θ. Note that angles are in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ and increment in the *clockwise* direction because the y axis of the image 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.

we get (by Pythagoras' theorem)

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

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 region's orientation vector $\boldsymbol{x}_d = (x_d, y_d)^T$ as

$$x_d = \cos(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } A = B = 0 \\ \left[\frac{1}{2}\left(1+\frac{B}{\sqrt{A^2+B^2}}\right)\right]^{\frac{1}{2}} & \text{otherwise,} \end{cases} \tag{2.29}$$

$$y_d = \sin(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } A = B = 0 \\ \left[\frac{1}{2}\left(1-\frac{B}{\sqrt{A^2+B^2}}\right)\right]^{\frac{1}{2}} & \text{for } A \geq 0 \\ -\left[\frac{1}{2}\left(1-\frac{b}{\sqrt{A^2+B^2}}\right)\right]^{\frac{1}{2}} & \text{for } A < 0, \end{cases} \tag{2.30}$$

straight from the central region moments $\mu_{11}(\mathcal{R})$, $\mu_{20}(\mathcal{R})$, and $\mu_{02}(\mathcal{R})$, as defined in Eqn. (2.28). The horizontal component (x_d) in Eqn. (2.29) is always positive, while the case clause in Eqn. (2.30) corrects the sign of the vertical component (y_d) to map to the same angular range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ as Eqn. (2.25). 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 below.

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. (2.25)), 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. 2.4.2), and they can also handle nonconnected regions or point clouds. Several different formulations of region eccentricity can be found in the literature [2,46,47] (see also Exercise 2.11). We adopt the following definition because of its simple geometrical interpretation:

$$\mathsf{Ecc}(\mathcal{R}) = \frac{a_1}{a_2} = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}\,, \tag{2.31}$$

where $a_1 = 2\lambda_1$, $a_2 = 2\lambda_2$ are multiples of the eigenvalues λ_1, λ_2 of the symmetric 2×2 matrix

$$\boldsymbol{A} = \begin{pmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{pmatrix}$$

formed by the central moments μ_{pq} of the region $\mathcal{R}$. 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. Ecc itself is invariant to the region's orientation and size. However, the values a_1, a_2 contain information about the spatial extent 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. (2.25). The lengths of the ellipse's major and minor axes, r_a and r_b, are

$$r_a = 2 \cdot \left(\frac{\lambda_1}{|\mathcal{R}|}\right)^{\frac{1}{2}} = \left(\frac{2\,a_1}{|\mathcal{R}|}\right)^{\frac{1}{2}}, \tag{2.32}$$

$$r_b = 2 \cdot \left(\frac{\lambda_2}{|\mathcal{R}|}\right)^{\frac{1}{2}} = \left(\frac{2\,a_2}{|\mathcal{R}|}\right)^{\frac{1}{2}}, \tag{2.33}$$

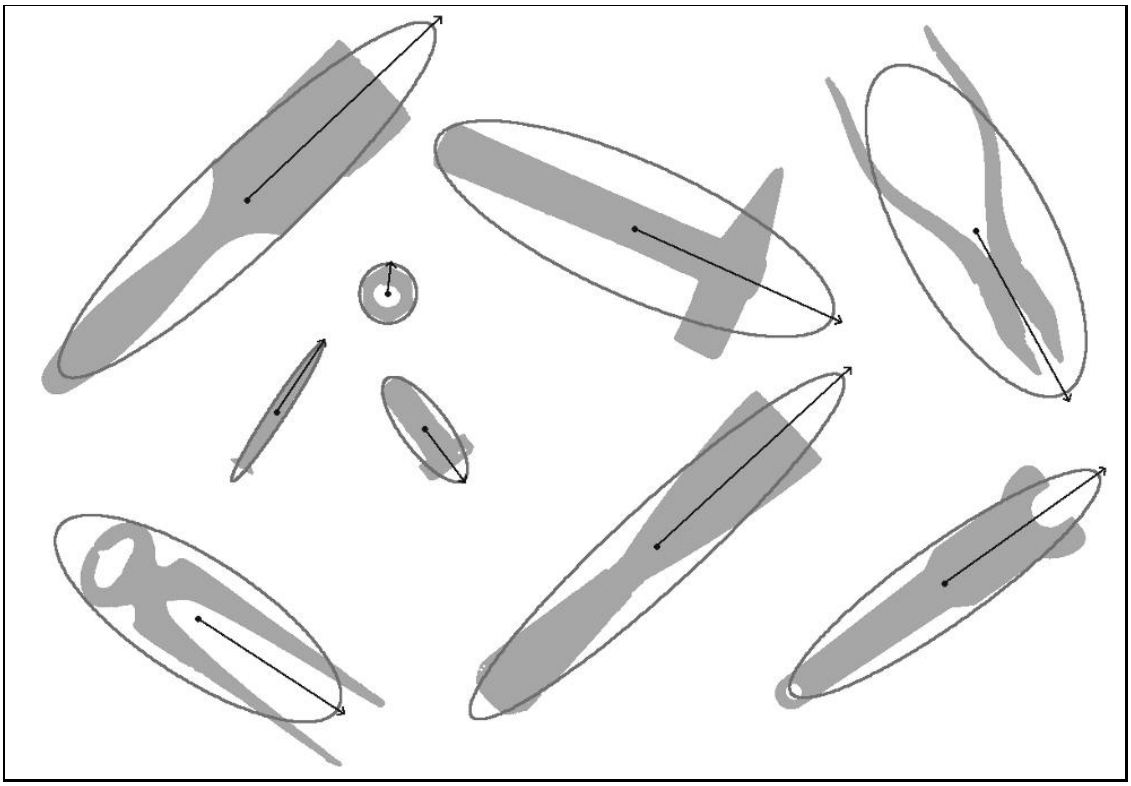


Figure 2.19 Orientation and eccentricity examples. The orientation θ (Eqn. (2.25)) is displayed for each connected region as a vector with the length proportional to the region's eccentricity value $\mathsf{Ecc}(\mathcal{R})$ (Eqn. (2.31)). Also shown are the ellipses (Eqns. (2.32) and (2.33)) corresponding to the orientation and eccentricity parameters.

respectively, with a_1, a_2 as defined in Eqn. (2.31) and $|\mathcal{R}|$ being the number of pixels in the region. The resulting parametric equation of the equivalent ellipse is

$$\begin{aligned}\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} &= \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} r_a \cdot \cos(t) \\ r_b \cdot \sin(t) \end{pmatrix} \\ &= \begin{pmatrix} \bar{x} + \cos(\theta) \cdot r_a \cdot \cos(t) - \sin(\theta) \cdot r_b \cdot \sin(t) \\ \bar{y} + \sin(\theta) \cdot r_a \cdot \cos(t) + \cos(\theta) \cdot r_b \cdot \sin(t) \end{pmatrix} \end{aligned} \tag{2.34}$$

for $0 \leq t < 2\pi$. If entirely *filled*, the region described by this ellipse would have the same (first and second order) central moments as the original region $\mathcal{R}$. Figure 2.19 shows a set of regions with overlaid orientation and eccentricity results.

Invariant 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. A classical solution to this problem is a clever

combination of simpler features known as "Hu's Moments" [37]:[12]

$$
\begin{aligned}
H_1 &= \bar{\mu}_{20} + \bar{\mu}_{02}, \\
H_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\,\bar{\mu}_{11}^2, \\
H_3 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12})^2 + (3\,\bar{\mu}_{21} - \bar{\mu}_{03})^2, \\
H_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2, \\
H_5 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2\right] \\
&\quad + (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \left[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\right], \\
H_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\right] \\
&\quad + 4\,\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}), \\
H_7 &= (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3\,(\bar{\mu}_{21} + \bar{\mu}_{03})^2\right] \\
&\quad + (3\,\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \left[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\right].
\end{aligned}
\tag{2.35}
$$

In practice, the logarithm of the results (that is, $\log(H_k)$) is used since the raw values can 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 grayscale images; for further information, see [28, p. 517].

2.4.5 Projections

Image projections are one-dimensional representations of the image contents, usually computed parallel to the coordinate axis; in this case, the horizontal, as well as the vertical, projection of an image $I(u, v)$, with $0 \le u < M$, $0 \le v < N$, defined as

$$
P_{\text{hor}}(v_0) = \sum_{u=0}^{M-1} I(u, v_0) \qquad \text{for } 0 \le v_0 < N, \tag{2.36}
$$

$$
P_{\text{ver}}(u_0) = \sum_{v=0}^{N-1} I(u_0, v) \qquad \text{for } 0 \le u_0 < M. \tag{2.37}
$$

The *horizontal* projection $P_{\text{hor}}(v_0)$ (Eqn. (2.36)) 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. (2.37)). 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.

[12] In order to improve the legibility of Eqn. (2.35) 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.

```
1  public void run(ImageProcessor ip) {
2    int M = ip.getWidth();
3    int N = ip.getHeight();
4    int[] horProj = new int[N];
5    int[] verProj = new int[M];
6    for (int v = 0; v < N; v++) {
7      for (int u = 0; u < M; u++) {
8        int p = ip.getPixel(u, v);
9        horProj[v] += p;
10       verProj[u] += p;
11     }
12   }
13   // use projections horProj, verProj now
14   // ...
15 }
```

Program 2.4 Computation 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`.

Program Prog. 2.4 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. 2.20). In practice, especially to account for document skew, projections are often computed along the major axis of an image region Eqn. (2.25). 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.

2.4.6 Topological Properties

Topological features do not describe the shape of a region in continuous terms; instead, they capture its structural properties. They are typically invariant even under extreme image transformations. Two simple and robust topological features are the number of regions $N_R(\mathcal{R})$ and the number of holes $N_L(\mathcal{R})$ in those regions. $N_L(\mathcal{R})$ can be easily computed while finding the inner contours of a region, as described in Sec. 2.2.2.

A feature that can be derived from the number of holes is the so-called *Euler number* N_E, which is the difference between the number of connected

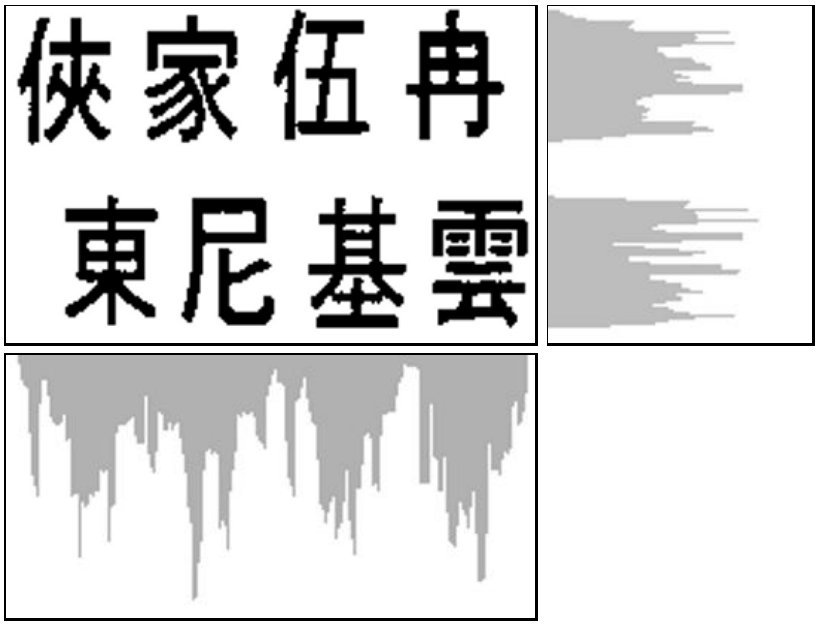


Figure 2.20 Example of the horizontal projection $P_{\text{hor}}(v)$ (right) and vertical projection $P_{\text{ver}}(u)$ (bottom) of a binary image.

regions N_R and the number of their holes N_H,

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_H(\mathcal{R}). \tag{2.38}$$

For a single connected region, the above formula simplifies to $1 - N_H$, so, for example, for an image of the number "**8**", $N_E = 1 - 2 = -1$, while for an image of the letter "**D**", $N_E = 1 - 1 = 0$.

Topological features are often used in combination with numerical features for classification, for example in optical character recognition (OCR) [12].

2.5 Exercises

Exercise 2.1

Trace, by hand, the execution of both variations (*depth-first* and *breadth-first*) of the flood-fill algorithm using the image shown in Fig. 2.21 and starting at coordinates $(5, 1)$.

Exercise 2.2

The implementation of the flood-fill algorithm in Prog. 2.1 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

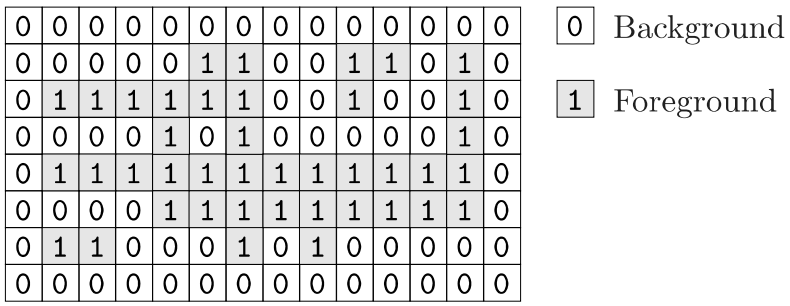


Figure 2.21 Binary image for Exercise 2.1.

meet the two conditions given above. Modify the *depth-first* and *breadth-first* variants given in Prog. 2.1 accordingly and compare the new running times.

Exercise 2.3
Implement an ImageJ plugin that encodes a grayscale image using run length encoding (Sec. 2.3.2) and stores it in a file. Develop a second plugin that reads the file and reconstructs the image.

Exercise 2.4
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 2.5
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 2.6
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 2.7
Implement an algorithm for comparing contours using their shape numbers Eqn. (2.3). 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 2.8
Using Eqn. (2.10) as the basis, develop and implement an algorithm that computes the area of a region from its 8-chain code encoded contour. What type of discrepancy from the region's actual area (the number of pixels it contains) do you expect?

Exercise 2.9
Sketch an example binary region where the centroid lies outside of the region.

Exercise 2.10
Implement the moment features developed by Hu (Eqn. (2.35)) and show that they are invariant under scaling and rotation for both binary and grayscale images.

Exercise 2.11
There are alternative definitions for the eccentricity of a region Eqn. (2.31); for example,

$$\mathsf{Ecc}_2(\mathcal{R}) = \frac{\big(\mu_{20} - \mu_{02}\big)^2 + 4 \cdot \mu_{11}^2}{\big(\mu_{20} + \mu_{02}\big)^2} \qquad \text{[47, p. 394]},$$

$$\mathsf{Ecc}_3(\mathcal{R}) = \frac{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}}{m_{00}} \qquad \text{[46, p. 531]},$$

$$\mathsf{Ecc}_4(\mathcal{R}) = \frac{\sqrt{\mu_{20} - \mu_{02}} + 4 \cdot \mu_{11}}{m_{00}} \qquad \text{[2, p. 255]}.$$

Implement all four variations (including the one in Eqn. (2.31)) and contrast the results using suitably designed regions. Determine how these measures work and what their range of values is, and propose a geometrical interpretation for each.

Exercise 2.12
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. 2.19). Hint: Use Eqn. (2.34) to develop a method for drawing ellipses at arbitrary orientations (not available in ImageJ).

Exercise 2.13
The Java method in Prog. 2.4 computes an image's horizontal and vertical projections. For document image processing, projections in the diagonal directions are also useful. Implement these projections and consider what role they play in document image analysis.

3
Detecting Simple Curves

In Volume 1 we demonstrated how to use appropriately designed filters to detect edges in images [14, Chap. 6]. These filters compute both the edge strength and orientation at every position in the image. In the following sections, we explain how to decide (for example, by using a threshold operation on the edge strength) if a curve is actually present at a given image location. The result of this process is generally represented as a binary *edge map*. Edge maps are considered preliminary results since with an edge filter's limited ("myopic") view it is not possible to accurately ascertain if a point belongs to a true edge. Edge maps created using simple threshold operations contain many edge points that do not belong to true edges (false positives), and, on the other hand, many edge points are not detected and so are missing from the map (false negatives).[1] In general, edge maps contain many irrelevant structures, while at the same time many important structures are completely missing. The theme of this chapter is how, given a binary edge map, one can find relevant and possibly significant structures based on their forms.

3.1 Salient Structures

An intuitive approach to locating large image structures is to first select an arbitrary edge point, systematically examine its neighboring pixels and add

[1] Typically thresholding is performed at a level that decreases false negatives at the expense of introducing false positives, the reasoning being that it is much simpler to remove false positives during higher-level processing than it is to, in essence, fill in the missing elements eliminated during low-level processing.

W. Burger, M.J. Burge, *Principles of Digital Image Processing*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84800-195-4_3,

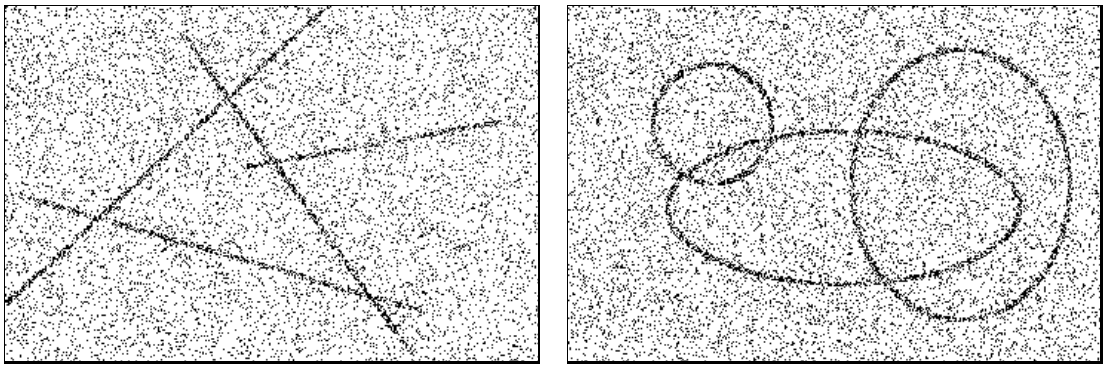


Figure 3.1 The human visual system is capable of instantly recognizing prominent image structures even under difficult conditions.

them if they belong to the object's contour, and repeat. In principle, such an approach could be applied to either a continuous edge map consisting of edge strengths and orientations or a simple binary *edge map*. Unfortunately, with either input, such an approach is likely to fail due to image noise and ambiguities that arise when trying to follow the contours. Additional constraints and information about the type of object sought are needed in order to handle pixel-level problems such as branching, as well as interruptions. This type of local sequential *contour tracing* makes for an interesting optimization problem [47] (see also Sec. 2.2).

A completely different approach is to search for globally apparent structures that consist of certain simple shape features. As an example, Fig. 3.1 shows that certain structures are readily apparent to the human visual system, even when they overlap in noisy images. The biological basis for why the human visual system spontaneously recognizes four lines or three circles in Fig. 3.1 instead of a larger number of disjoint segments and arcs is not completely known. At the cognitive level, theories such as "Gestalt" grouping have been proposed to address this behavior. The next sections explore one technique, the Hough transform, that provides an algorithmic solution to this problem.

3.2 Hough Transform

The method from Paul Hough—originally published as a US Patent [36] and often referred to as the "Hough transform" (HT)—is a general approach to localizing any shape that can be defined parametrically within a distribution of points [21, 39]. For example, many geometrical shapes, such as lines, circles, and ellipses, can be readily described using simple equations with only a few parameters. Since simple geometric forms often occur as part of man-made objects, they are especially useful features for analysis of these types of images (Fig. 3.2).

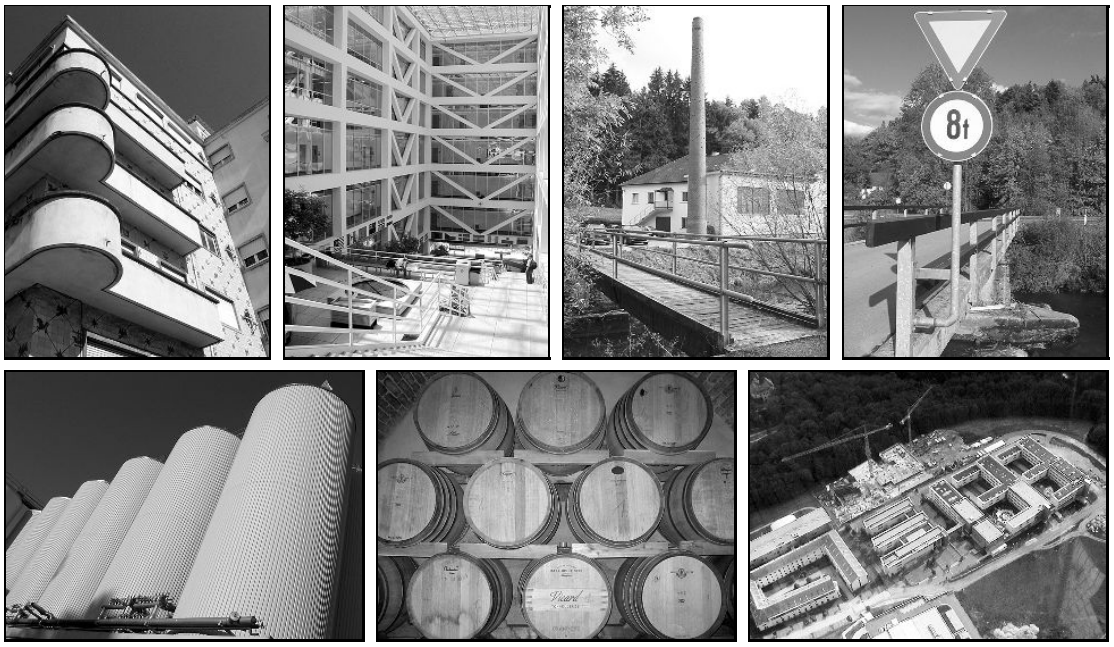


Figure 3.2 Simple geometrical forms such as sections of lines, circles, and ellipses are often found in man-made objects.

The Hough transform is perhaps most often used for detecting line segments in edge maps. A line segment in 2D can be described with two real-valued parameters using the classic slope-intercept form

$$y = kx + d, \tag{3.1}$$

where k is the slope and d the intercept—that is, the height at which the line would intercept the y axis (Fig. 3.3). A line segment that passes through two given edge points $\boldsymbol{p}_1 = (x_1, y_1)$ and $\boldsymbol{p}_2 = (x_2, y_2)$ must satisfy the conditions

$$y_1 = kx_1 + d \qquad \text{and} \qquad y_2 = kx_2 + d \tag{3.2}$$

for $k, d \in \mathbb{R}$. The goal is to find values of k and d such that as many edge points as possible lie on the line they describe; in other words, the line that fits the most edge points. But how can you determine the number of edge points that lie on a given line segment? One possibility is to exhaustively "draw" every possible line segment into the image while counting the number of points that lie exactly on each of these. Even though the discrete nature of pixel images (with only a finite number of different lines) makes this approach possible in theory, generating such a large number of lines is infeasible in practice.

3.2.1 Parameter Space

The Hough transform approaches the problem from another direction. It examines all the possible line segments that run through a single given point in

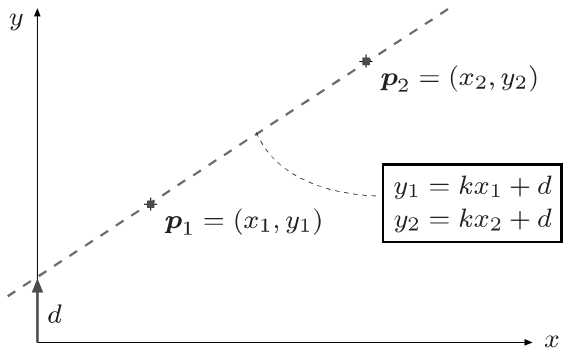


Figure 3.3 Two points, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, lie on the same line when $y_1 = kx_1 + d$ and $y_2 = kx_2 + d$ for a particular pair of parameters k and d.

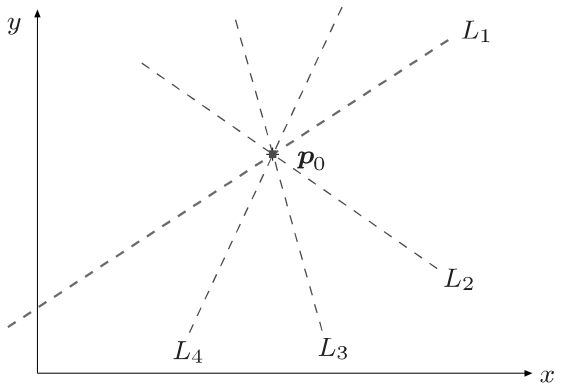


Figure 3.4 Set of lines passing through an image point. For all possible lines L_j passing through the point $\boldsymbol{p}_0 = (x_0, y_0)$, the equation $y_0 = k_j x_0 + d_j$ holds for appropriate values of the parameters k_j, d_j.

the image. Every line $L_j = \langle k_j, d_j \rangle$ that runs through a point $\boldsymbol{p}_0 = (x_0, y_0)$ must satisfy the condition

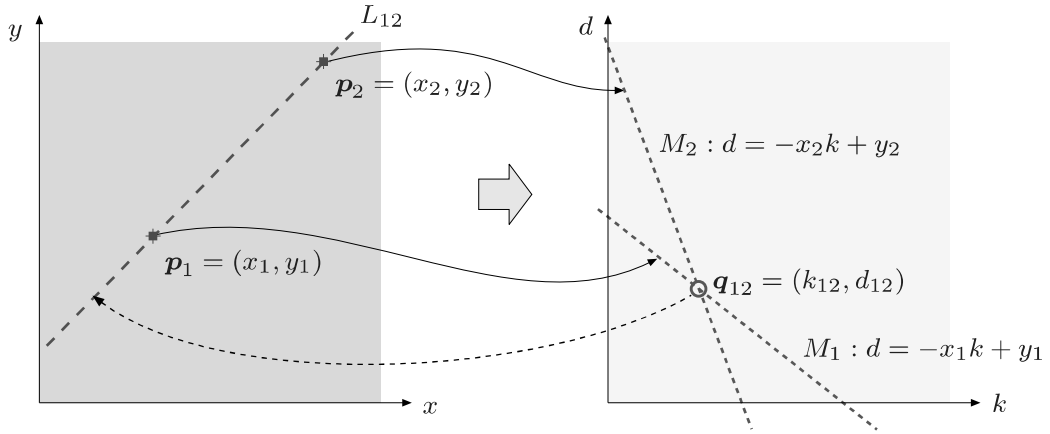
$$L_j : y_0 = k_j x_0 + d_j \tag{3.3}$$

for some suitable pair of values k_j, d_j. Equation 3.3 is underdetermined and the possible solutions for k_j, d_j correspond to an infinite set of lines passing through the given point $\boldsymbol{p}_0$ (Fig. 3.4). Note that for a given k_j, the solution for d_j in Eqn. (3.3) is

$$d_j = -x_0 k_j + y_0, \tag{3.4}$$

which is another equation for a line, where now k_j, d_j are the *variables* and x_0, y_0 are the constant *parameters* of the equation. The solution set $\{(k_j, d_j)\}$ of Eqn. (3.4) describes the parameters of all possible lines L_j passing through the image point $\boldsymbol{p}_0 = (x_0, y_0)$. For an *arbitrary* image point $\boldsymbol{p}_i = (x_i, y_i)$, Eqn. (3.4) describes the line

$$M_i : d = -x_i k + y_i \tag{3.5}$$



(a) x/y image space (b) k/d parameter space

Figure 3.5 Relationship between image space and parameter space. The parameter values for all possible lines passing through the image point $\boldsymbol{p}_i = (x_i, y_i)$ in image space (a) lie on a single line M_i in parameter space (b). This means that each point $\boldsymbol{q}_j = (k_j, d_j)$ in parameter space corresponds to a single line L_j in image space. The intersection of the two lines M_1, M_2 at the point $\boldsymbol{q}_{12} = (k_{12}, d_{12})$ in parameter space indicates that a line L_{12} through the two points k_{12} and d_{12} exists in the image space.

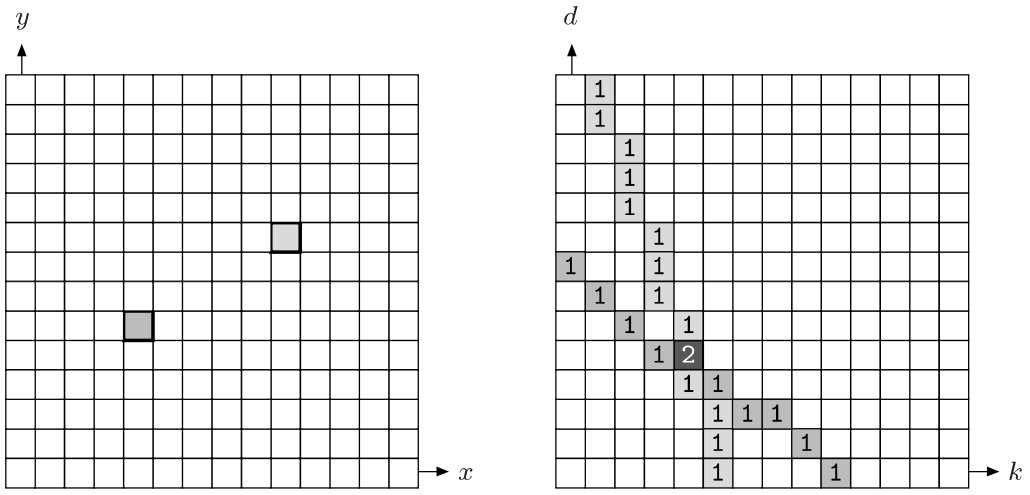
with the parameters $-x_i, y_i$ in the so-called *parameter* or *Hough* space, spanned by the coordinates k, d.

The relationship between (x, y) *image* space and (k, d) *parameter* space can be summarized as follows:

Image Space (x, y)		*Parameter Space* (k, d)	
Point	$\boldsymbol{p}_i = (x_i, y_i)$	$M_i : d = -x_ik + y_i$	Line
Line	$L_j : y = k_jx + d_j$	$\boldsymbol{q}_j = (k_j, d_j)$	Point

Each image point $\boldsymbol{p}_i$ and its associated line bundle correspond to exactly one line M_i in parameter space. Therefore we are interested in those places in the parameter space where lines *intersect*. The example in Fig. 3.5 illustrates how the lines M_1 and M_2 intersect at the position $\boldsymbol{q}_{12} = (k_{12}, d_{12})$ in the parameter space, which means (k_{12}, d_{12}) are the parameters of the line in the image space that runs through both image points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. The more lines M_i that intersect at a single point in the parameter space, the more image space points lie on the corresponding line in the image! In general, we can state:

> If N lines intersect at position (k', d') in *parameter space*, then N image points lie on the corresponding line $y = k'x + d'$ in *image space*.



(a) Image Space (b) Accumulator Array

Figure 3.6 Main idea of the Hough transform. The accumulator array is a discrete representation of the parameter space (k, d). For each image point found (a), a discrete line in the parameter space (b) is drawn. This operation is performed *additively* so that the values of the array through which the line passes are incremented by 1. The value at each cell of the accumulator array is the number of parameter space lines that intersect it (in this case 2).

3.2.2 Accumulator Array

Finding the dominant lines in the image can now be reformulated as finding all the locations in parameter space where a significant number of lines intersect. This is basically the goal of the HT. In order to compute the HT, we must first decide on a discrete representation of the continuous parameter space by selecting an appropriate step size for the k and d axes. Once we have selected step sizes for the coordinates, we can represent the space naturally using a two-dimensional array. Since the array will be used to keep track of the number of times parameter space lines intersect, it is called an "accumulator" array. Each parameter space line is painted into the accumulator array and the cells through which it passes are incremented, so that ultimately each cell accumulates the total number of lines that intersect at that cell (Fig. 3.6).

3.2.3 A Better Line Representation

The line representation in Eqn. (3.1) is not used in practice because for vertical lines the slope is infinite, i. e., $k = \infty$. A more practical representation is the so-called *Hessian normal form* (HNF, [11, p. 195]) for representing lines,

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = r, \tag{3.6}$$

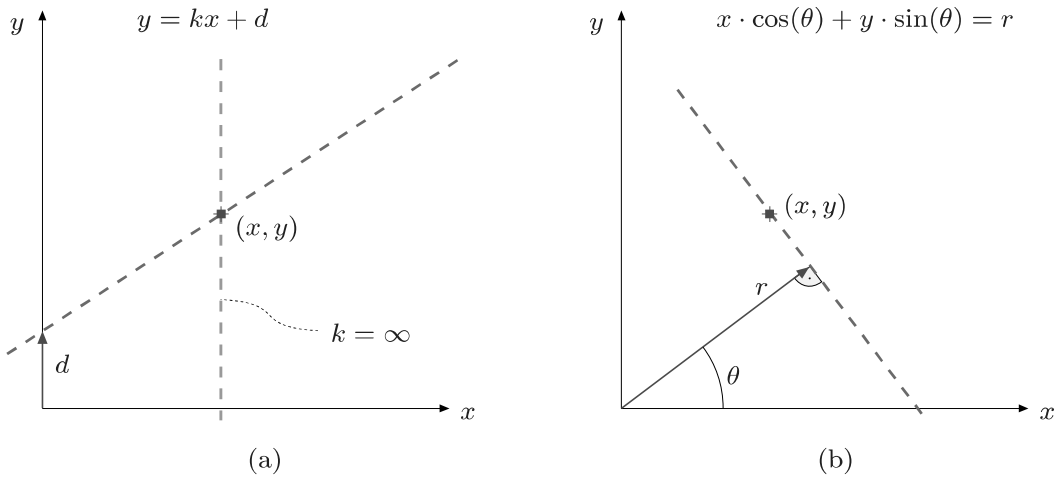


Figure 3.7 Representation of lines in 2D. In the normal k, d representation (a), vertical lines pose a problem because $k = \infty$. The Hessian normal form (b) avoids this problem by representing a line by its angle θ and distance r from the origin.

which does not exhibit such singularities and also provides a natural linear quantization for its parameters, the angle θ and the radius r (Fig. 3.7). With the HNF[2] representation, the parameter space is defined by the coordinates θ, r, and a point $\boldsymbol{p} = (x, y)$ in image space corresponds to the function

$$r_{x,y}(\theta) = x \cdot \cos(\theta) + y \cdot \sin(\theta) \tag{3.7}$$

for angles in the range $0 \leq \theta < \pi$ (Fig. 3.8). If we use the center of the image as the reference point for the x/y image space, then it is possible to limit the range of the radius to half the diagonal of the image,

$$-r_{\max} \leq r_{x,y}(\theta) \leq r_{\max}, \quad \text{where} \quad r_{\max} = \tfrac{1}{2}\sqrt{M^2 + N^2}\,, \tag{3.8}$$

for an image of width M and height N.

3.3 Implementing the Hough Transform

The fundamental Hough algorithm using the HNF line representation (Eqn. (3.6)) is given in Alg. 3.1. Starting with a binary image $I(u, v)$ where the edge pixels have been assigned a value of 1, the first stage creates a two-dimensional accumulator array and then iterates over the image to fill it. In the second stage, the accumulator array is searched (FindMaxLines()) for maximum values, and a list of parameter pairs for the K strongest lines

$$\mathit{MaxLines} = [\langle\theta_1, r_1\rangle, \langle\theta_2, r_2\rangle, \ldots \langle\theta_K, r_K\rangle]$$

is computed. The next sections explain these two stages in detail.

[2] The Hessian normal form is a constrained variant of the general line equation $ax + by + c = 0$, with $a = \cos(\theta)$, $b = \sin(\theta)$, and $c = -r$ (see [11, p. 194]).

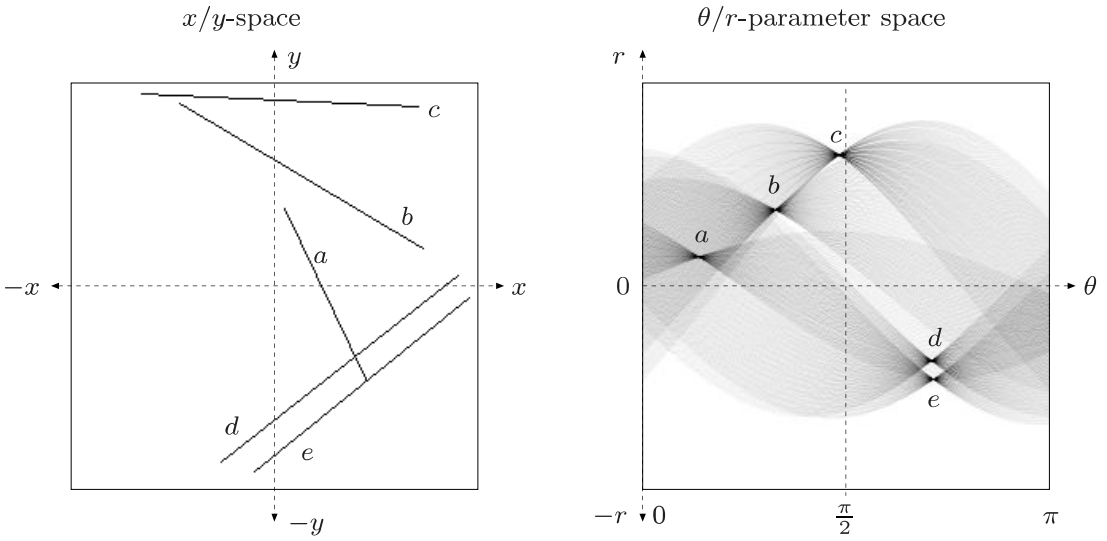


Figure 3.8 Image space and parameter space using the HNF representation.

3.3.1 Filling the Accumulator Array

A direct implementation of the first phase of Alg. 3.1 is given in the Java class `LinearHT` Prog. 3.1.[3] The accumulator array (`houghArray`) is defined as a two-dimensional `int` Array. The HT is computed from the original image `ip` by creating a new instance of the class `LinearHT`, for example,

```
LinearHT ht = new LinearHT(ip, 256, 256);
```

The binary image is passed as an `ImageProcessor` (`ip`), wherein any value greater than 0 is interpreted as an edge pixel. The other two parameters, `nAng` (256) and `nRad` (256), specify the number of discrete steps to use for the angle (N_θ steps for $\theta_i = 0$ to π) and the radius (N_r steps for $r_i = -r_{\max}$ to $r_{\max}$). The resulting increments for the angle and radius are thus

$$\Delta_\theta = \frac{\pi}{N_\theta} \qquad \text{and} \qquad \Delta_r = \frac{2 \cdot r_{\max}}{N_r}$$

(see lines 17 and 21 in Prog. 3.1, respectively). The output of this program for a very noisy edge image is given in Fig. 3.9.

3.3.2 Analyzing the Accumulator Array

The second phase is localizing the maximum values in the accumulator array $Acc(i_\theta, i_r)$. As can readily be seen in Fig. 3.9 (b), even in the case where

[3] The complete implementation of the Hough transform for straight lines can be found in the source code section of this book's Website.

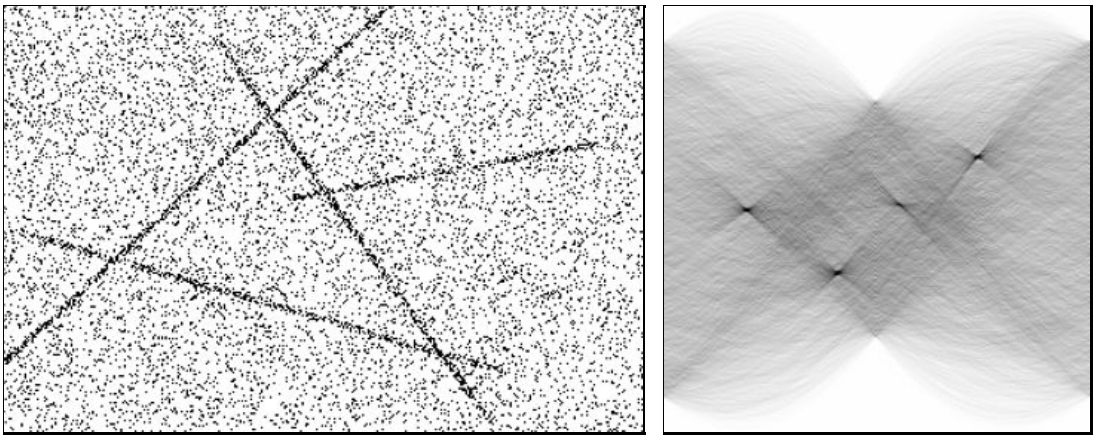
Algorithm 3.1 Simple Hough algorithm for detecting straight lines. It returns a list containing the parameters $\langle \theta, r \rangle$ of the K strongest lines in the binary edge image I.

1: HOUGHLINES(I, N_θ, N_r, K)
Computes the Hough transform to detect straight lines in the binary image I (of size $M \times N$), using N_θ, N_r discrete steps for the angle and radius, respectively. Returns the list of parameter pairs $\langle \theta_i, r_i \rangle$ for the K strongest lines found.
2: $(u_c, v_c) \leftarrow (\frac{M}{2}, \frac{N}{2})$ ▷ image center
3: $r_{\max} \leftarrow \sqrt{u_c^2 + v_c^2}$ ▷ max. radius is half the image diagonal
4: $\Delta_\theta \leftarrow \frac{\pi}{N_\theta}$ ▷ angular increment
5: $\Delta_r \leftarrow \frac{2 \cdot r_{\max}}{N_r}$ ▷ radial increment
6: Create the accumulator array $Acc(i_\theta, i_r)$ of size $N_\theta \times N_r$
7: **for all** accumulator cells (i_θ, i_r) **do**
8: $Acc(i_\theta, i_r) \leftarrow 0$ ▷ initialize the accumulator array
9: **for all** image coordinates (u, v) **do** ▷ scan the image
10: **if** $I(u, v)$ is an edge point **then**
11: $(x, y) \leftarrow (u - u_c, v - v_c)$ ▷ coordinate relative to center
12: **for** $i_\theta \leftarrow 0 \ldots N_\theta - 1$ **do** ▷ angular index i_θ
13: $\theta \leftarrow \Delta_\theta \cdot i_\theta$ ▷ real angle, $0 \le \theta < \pi$
14: $r \leftarrow x \cdot \cos(\theta) + y \cdot \sin(\theta)$ ▷ real radius (pos./neg.)
15: $i_r \leftarrow \frac{N_r}{2} + \text{round}(\frac{r}{\Delta_r})$ ▷ radial index i_r
16: $Acc(i_\theta, i_r) \leftarrow Acc(i_\theta, i_r) + 1$ ▷ increment $Acc(i_\theta, i_r)$
Find the parameters pairs $\langle \theta_j, r_j \rangle$ for the K strongest lines:
17: $MaxLines \leftarrow$ FINDMAXLINES(Acc, K)
18: **return** $MaxLines$.

the lines in the image are geometrically "straight", the parameter space curves associated with them do not intercept at *exactly* one point in the accumulator array but rather their intersection points are distributed within a small area. This is primarily caused by the rounding errors introduced due to the discrete coordinate grid used in the accumulator array. Since the maximum points are really maximum areas in the accumulator array, simply traversing the array and returning its K largest values is not sufficient. Since this is a critical step in the algorithm, we will examine two different approaches (Fig. 3.10) in the following.

```
1  class LinearHT {
2    ImageProcessor ip; // reference to the original image I
3    int xCtr, yCtr;    // x/y-coordinates of image center (u_c, v_c)
4    int nAng;          // N_θ steps for the angle (θ = 0 ... π)
5    int nRad;          // N_r steps for the radius (r = −r_max ... r_max)
6    int cRad;          // center of radius axis (r = 0)
7    double dAng;       // increment of angle Δ_θ
8    double dRad;       // increment of radius Δ_r
9    int[][] houghArray; // Hough accumulator Acc(i_θ, i_r)
10
11   //constructor method:
12   LinearHT(ImageProcessor ip, int nAng, int nRad) {
13     this.ip = ip;
14     this.xCtr = ip.getWidth()/2;
15     this.yCtr = ip.getHeight()/2;
16     this.nAng = nAng;
17     this.dAng = Math.PI / nAng;
18     this.nRad = nRad;
19     this.cRad = nRad / 2;
20     double rMax = Math.sqrt(xCtr * xCtr + yCtr * yCtr);
21     this.dRad = (2.0 * rMax) / nRad;
22     this.houghArray = new int[nAng][nRad];
23     fillHoughAccumulator();
24   }
25
26   void fillHoughAccumulator() {
27     int h = ip.getHeight();
28     int w = ip.getWidth();
29     for (int v = 0; v < h; v++) {
30       for (int u = 0; u < w; u++) {
31         if (ip.get(u, v) > 0) {
32           doPixel(u, v);
33         }
34       }
35     }
36   }
37
38   void doPixel(int u, int v) {
39     int x = u - xCtr, y = v - yCtr;
40     for (int ia = 0; ia < nAng; ia++) {
41       double theta = dAng * ia;
42       int ir = cRad + (int) Math.rint
43          ((x*Math.cos(theta) + y*Math.sin(theta)) / dRad);
44       if (ir >= 0 && ir < nRad) {
45         houghArray[ia][ir]++;
46       }
47     }
48   }
49
50 } // end of class LinearHT
```

Program 3.1 Hough transform for localizing straight lines (partial implementation). The complete Java implementation can be found in the source code section of the book's Website.



(a) (b)

Figure 3.9 Hough transform for straight lines. The dimensions of the original image (a) are 360×240 pixels, so the maximal radius (measured from the image center (u_c, v_c)) is $r_{\max} \approx 216$. For the parameter space (b), a step size of 256 is used for both the angle $\theta = 0 \ldots \pi$ (horizontal axis) and the radius $r = -r_{\max} \ldots r_{\max}$ (vertical axis). The four darkest spots in (b) mark the maximum values in the accumulator array, and their parameters correspond to the four lines in the original image. In (b), intensities have been inverted to improve legibility.

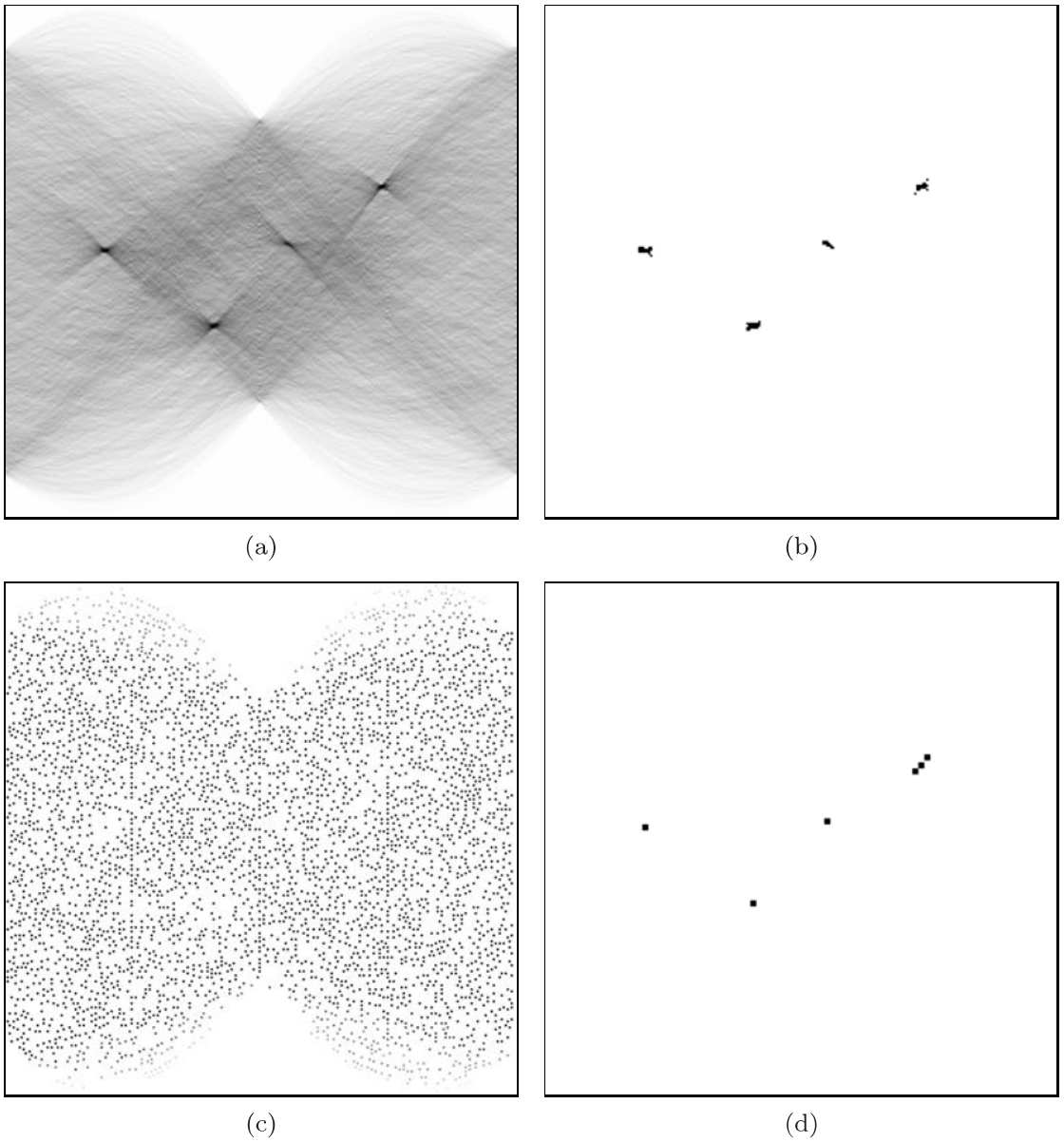
Approach A: Thresholding

First the accumulator is thresholded to the value of t_a by setting all accumulator values $Acc(i_\theta, i_r) < t_a$ to 0. The resulting scattering of points, or point clouds, are first coalesced into regions (Fig. 3.10 (b)) using a technique such as a morphological *closing* operation (see Vol. 1 [14, Sec. 7.3.2]). Next the remaining regions must be localized, for instance using the region-finding technique from Sec. 2.1, and then each region's centroid (see Sec. 2.4.3) can be utilized as the (noninteger) coordinates for the potential image space line. Often the sum of the accumulator's values within a region is used as a measure of the strength (number of image points) of the line it represents.

Approach B: Nonmaximum suppression

In this method, local maxima in the accumulator array are found by suppressing nonmaximal values.[4] This is carried out by determining for every cell in $Acc(\theta, r)$ whether the value is higher than the value of all of its neighboring cells. If this is the case, then the value remains the same; otherwise it is set to 0 (Fig. 3.10 (c)). The (integer) coordinates of the remaining peaks are potential line parameters, and their respective heights correlate with the strength of the image space line they represent. This method can be used in conjunction with a threshold operation to reduce the number of candidate points that must be

[4] Nonmaximum suppression is also used in Sec. 4.2.3 for isolating corner points.



(a) (b) (c) (d)

Figure 3.10 Determining the local maximum values in the accumulator array. Original distribution of the values in the Hough accumulator (a). **Variant A:** *Threshold operation* using 50% of the maximum value (b). The remaining regions represent the four dominant lines in the image, and the coordinates of their centroids are a good approximation to the line parameters. **Variant B:** Using *nonmaximum suppression* results in a large number of local maxima (c) that must then be reduced using a threshold operation (d).

considered. The result for Fig. 3.9 (a) is shown in Fig. 3.10 (d).

3.3.3 Hough Transform Extensions

So far, we have presented the Hough transform only in its most basic formulation. The following is a list of some of the more common methods of improving and refining the algorithm.

Modified accumulator updating

The purpose of the accumulator array is to find the intersections of two-dimensional curves. Due to the discrete nature of the image and accumulator coordinates, rounding errors usually cause the parameter curves for multiple image points on the same line not to intersect in a single accumulator cell. A common remedy is, for a given angle $\theta = i_\theta \cdot \Delta_\theta$ (Alg. 3.1), to increment not only the corresponding accumulator cell $Acc(i_\theta, i_r)$ but also the *neighboring* cells $Acc(i_\theta, i_r - 1)$ and $Acc(i_\theta, i_r + 1)$. This makes the Hough transform more tolerant against inaccurate point coordinates and rounding errors.

Bias problem

Since the value of a cell in the Hough accumulator represents the number of image points falling on a line, longer lines naturally have higher values than shorter lines. This may seem like an obvious point to make, but consider when the image only contains a small section of a "long" line. For instance, if a line only passes through the corner of an image then the cells representing it in the accumulator array will naturally have lower values than a "shorter" line that lies entirely within the image (Fig. 3.11).

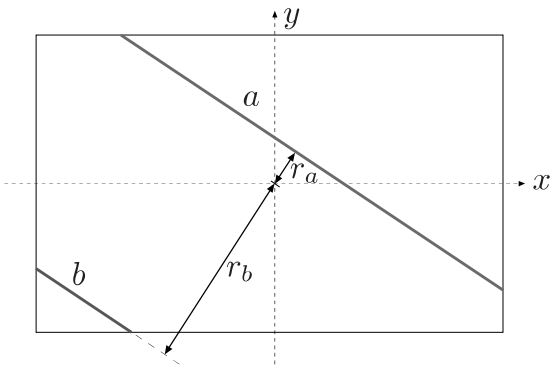


Figure 3.11 Bias problem. When an image represents only a finite section of an object, then those lines nearer the center (smaller r values) will have higher values than those farther away (larger r values). As an example, the maximum value of the accumulator for line a will be higher than that of line b.

It follows then that if we only search the accumulator array for maximal values, it is likely that we will completely miss short line segments. One way to compensate for this inherent bias is to compute for each accumulator entry $Acc(i_\theta, i_r)$ the maximum number of image points $MaxHits(i_\theta, i_r)$ possible for a line with the parameters θ, r and then normalize

$$Acc'(i_\theta, i_r) \leftarrow \frac{Acc(i_\theta, i_r)}{MaxHits(i_\theta, i_r)} \tag{3.9}$$

for $MaxHits(i_\theta, i_r) > 0$. The normalization term $MaxHits(i_\theta, i_r)$ can be determined, for example, by computing the Hough transform of an image with the same dimensions in which all pixels are edge pixels or by using a random image in which the pixels are uniformly distributed.

Line endpoints

Our simple version of the Hough transform determines the parameters of the line in the image but not their endpoints. These could be found in a subsequent step by determining which image points belong to any detected line (e. g., by applying a threshold to the perpendicular distance between the ideal line—defined by its parameters—and the actual image points). An alternative solution is to calculate the extreme point of the line during the computation of the accumulator array. For this, every cell of the accumulator array is supplemented with two additional coordinate pairs $\boldsymbol{x}_s = (x_s, y_s)$, $\boldsymbol{x}_e = (x_e, y_e)$, i. e.,

$$Acc(i_\theta, i_r) = \langle count, \boldsymbol{x}_s, \boldsymbol{x}_e \rangle.$$

Now the coordinates for the endpoints ($\boldsymbol{x}_s$, $\boldsymbol{x}_e$) of every line can be stored while filling in the accumulator array so that by the end of the process each cell contains the two endpoints that lie farthest from each other on the line it represents. When finding the maximum values in the second stage, care should be taken so that the merged cells contain the correct endpoints.

Line intersections

It may be useful in certain applications not to find the lines themselves but their intersections, e. g., for precisely locating the corner points of a polygon-shaped object. The Hough transform delivers the parameters of the recovered lines in Hessian normal form (i. e., as pairs $L_i = \langle \theta_i, r_i \rangle$). To compute the point of intersection $\boldsymbol{x}_0 = (x_0, y_0)^T$ for two lines

$$L_1 = \langle \theta_1, r_1 \rangle \qquad \text{and} \qquad L_2 = \langle \theta_2, r_2 \rangle,$$

we need to solve the system of linear equations

$$x_0 \cdot \cos(\theta_1) + y_0 \cdot \sin(\theta_1) = r_1, \tag{3.10}$$

$$x_0 \cdot \cos(\theta_2) + y_0 \cdot \sin(\theta_2) = r_2, \tag{3.11}$$

for the unknowns x_0, y_0. The solution is

$$\begin{aligned} \boldsymbol{x}_0 &= \frac{1}{\cos(\theta_1)\sin(\theta_2) - \cos(\theta_2)\sin(\theta_1)} \cdot \begin{bmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{bmatrix} \\ &= \frac{1}{\sin(\theta_2 - \theta_1)} \cdot \begin{bmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{bmatrix} \end{aligned} \tag{3.12}$$

for $\sin(\theta_2 - \theta_1) \neq 0$. Obviously $\boldsymbol{x}_0$ is undefined (no intersection point exists) if the lines L_1, L_2 are parallel to each other (i. e., if $\theta_1 \equiv \theta_2$).

Considering edge strength and orientation

Until now, the raw data for the Hough transform was typically an edge map that was interpreted as a binary image with ones at potential edge points. Yet edge maps contain additional information, such as the edge strength $E(u, v)$ and local edge orientation $\Phi(u, v)$ (see Vol. 1 [14, Sec. 6.3]), which can be used to improve the results of the HT.

The *edge strength* $E(u, v)$ is especially easy to take into consideration. Instead of incrementing visited accumulator cells by 1, add the strength of the respective edge:

$$Acc(i_\theta, i_r) \leftarrow Acc(i_\theta, i_r) + E(u, v).$$

In this way, strong edge points will contribute more to the accumulated value than weak points.

The local *edge orientation* $\Phi(u, v)$ is also useful for limiting the range of possible orientation angles for the line at (u, v). The angle $\Phi(u, v)$ can be used to increase the efficiency of the algorithm by reducing the number of accumulator cells to be considered along the θ axis. Since this also reduces the number of irrelevant "votes" in the accumulator, it increases the overall sensitivity of the Hough transform (see, for example, [45, p. 483]).

Hierarchical Hough transform

The accuracy of the results increases with the size of the parameter space used; for example, a step size of 256 along the θ axis is equivalent to searching for lines at every $\frac{\pi}{256} \approx 0.7°$. While increasing the number of accumulator cells provides a finer result, bear in mind that it also increases the computation time and especially the amount of memory required.

Instead of increasing the resolution of the entire parameter space, the idea of the hierarchical HT is to gradually "zoom" in and refine the parameter space. First, the regions containing the most important lines are found using a relatively low-resolution parameter space, and then the parameter spaces of those regions are recursively passed to the HT and examined at a higher resolution. In this way, a relatively exact determination of the parameters can be found using a limited (in comparison) parameter space.

3.4 Hough Transform for Circles and Ellipses

3.4.1 Circles and Arcs

Since lines in 2D have two degrees of freedom, they could be completely specified using two real-valued parameters. In a similar fashion, representing a circle in 2D requires *three* parameters, for example

$$Circle = \langle \bar{x}, \bar{y}, \rho \rangle,$$

where $\bar{x}$, $\bar{y}$ are the coordinates of the center and ρ is the radius of the circle (Fig. 3.12). A point $\boldsymbol{p} = (x, y)$ lies on this circle when the relation

$$(x - \bar{x})^2 + (y - \bar{y})^2 = \rho^2 \tag{3.13}$$

holds. Therefore the Hough transform requires a three-dimensional parameter space $Acc(\bar{x}, \bar{y}, \rho)$ to find the position and radius of circles (and circular arcs) in an image. Unlike the HT for lines, there does not exist a simple functional dependency between the coordinates in parameter space, so how can we find every parameter combination $(\bar{x}, \bar{y}, \rho)$ that satisfies Eqn. (3.13) for a given image point (u, v)? One solution is to apply a "brute force" method such as described in Alg. 3.2 that exhaustively tests each cell in the parameter space to see if the relation in Eqn. (3.13) holds.

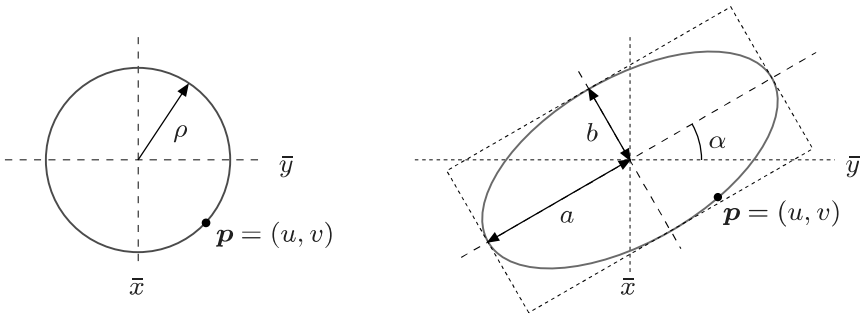


Figure 3.12 Representation of circles and ellipses in 2D.

If we examine Fig. 3.13, we can see that a better idea might be to make use of the fact that the coordinates of the center points also form a circle in Hough space. It is not necessary therefore to search the entire three-dimensional parameter space for each image point $\boldsymbol{p} = (u, v)$. Instead we need only increase the cell values along the edge of the appropriate circle on each ρ plane of the accumulator array. To do this, we can adapt any of the standard algorithms for generating circles. In this case, the integer math version of the well-known *Bresenham* algorithm [9] is particularly well-suited.

Figure 3.14 shows the spatial structure of the three-dimensional parameter space for circles. For a given image point $\boldsymbol{p}_k = (u_k, v_k)$, at each plane along

Algorithm 3.2 Exhaustive Hough algorithm for localizing circles.

HoughCircles(I)
Returns the list of parameters $\langle \bar{x}_i, \bar{y}_i, \rho_i \rangle$ corresponding to the strongest circles found in the binary image I.

```
Set up a three-dimensional array Acc(x̄, ȳ, ρ) and initialize to 0
for all image coordinates (u, v) do
    if I(u, v) is an edge point then
        for all (x̄_i, ȳ_i, ρ_i) in the accumulator space do
            if (u − x̄_i)² + (v − ȳ_i)² = ρ_i² then
                Increment Acc(x̄_i, ȳ_i, ρ_i)
MaxCircles ← FindMaxCircles(Acc)   ▷ a list of tuples ⟨x̄_j, ȳ_j, ρ_j⟩
return MaxCircles.
```

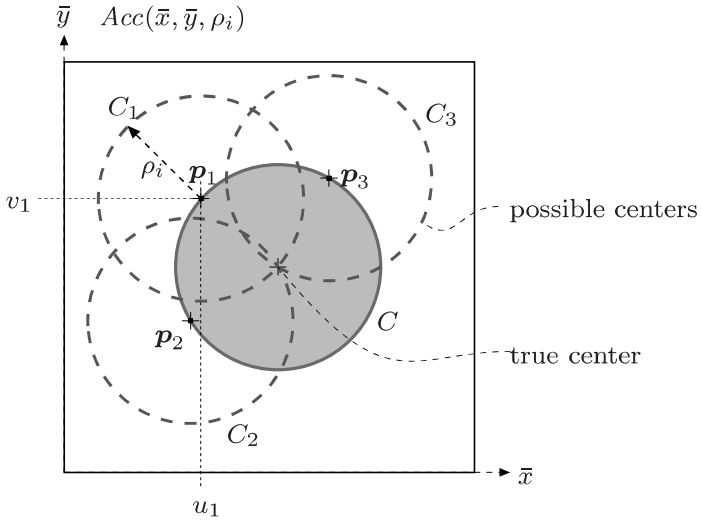


Figure 3.13 Hough transform for circles. The illustration depicts a slice of the three-dimensional accumulator array $Acc(\bar{x}, \bar{y}, \rho)$ at a given circle radius $\rho = \rho_i$. The center points of all the circles running through a given image point $\boldsymbol{p}_1 = (u_1, v_1)$ form a circle C_1 with a radius of ρ_i centered around $\boldsymbol{p}_1$, just as the center points of the circles that pass through $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ lie on the circles C_2, C_3. The cells along the edges of the three circles C_1, C_2, C_3 of radius ρ_i are traversed and their values in the accumulator array incremented. The cell in the accumulator array contains a value of three where the circles intersect at the true center of the image circle C.

the ρ axis (for $\rho_i = \rho_{\min} \ldots \rho_{\max}$), a circle centered at (u_k, v_k) with the radius ρ_i is traversed, ultimately creating a three-dimensional cone-shaped surface in the parameter space. The coordinates of the dominant circles can be found by searching the accumulator space for the cells with the highest values; that is, the cells where the most cones intersect.

Just as in the linear HT, the *bias* problem (see Sec. 3.3.3) also occurs in

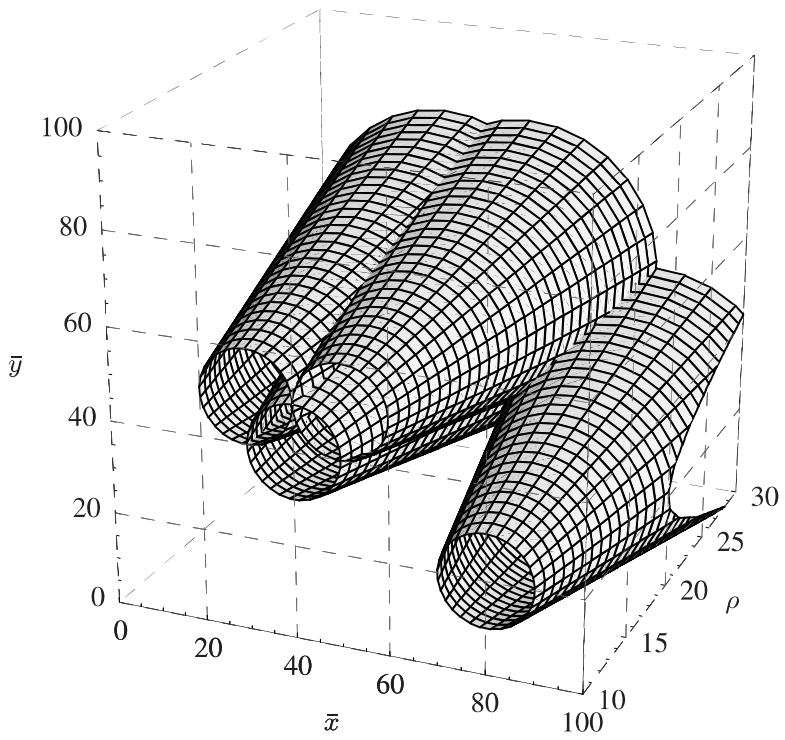


Figure 3.14 Three-dimensional parameter space for circles. For each image point $\boldsymbol{p}_k = (u_k, v_k)$, the cells lying along a cone in the three-dimensional accumulator array $Acc(\bar{x}, \bar{y}, \rho)$ are incremented.

the circle HT. Sections of circles (i. e., arcs) can be found in a similar way, in which case the maximum value possible for a given cell is proportional to the arc length.

3.4.2 Ellipses

In a perspective image, most circular objects originating in our real, three-dimensional world will actually appear in 2D images as ellipses, except in the case where the object lies on the optical axis and is observed from the front. For this reason, perfectly circular structures seldom occur in photographs. While the Hough transform can still be used to find ellipses, the larger parameter space required makes it substantially more expensive.

A general ellipse in 2D has five degrees of freedom and therefore requires five parameters to represent it,

$$\mathit{Ellipse} = \langle \bar{x}, \bar{y}, r_a, r_b, \alpha \rangle,$$

where $(\bar{x}, \bar{y})$ are the coordinates of the center points, (r_a, r_b) are the two radii,

and α is the orientation of the principal axis (Fig. 3.12).[5] In order to find ellipses of any size, position, and orientation using the Hough transform, a five-dimensional parameter space with a suitable resolution in each dimension is required. A simple calculation illustrates the enormous expense of representing this space: using a resolution of only $128 = 2^7$ steps in ever dimension results in 2^{35} accumulator cells, and implementing these using 4-byte `int` values thus requires 2^{37} bytes (128 gigabytes) of memory.

An interesting alternative in this case is the *generalized Hough transform*, which in principle can be used for detecting any arbitrary two-dimensional shape [2, 39]. Using the generalized Hough transform, the shape of the sought-after contour is first encoded point by point in a table and then the associated parameter space is related to the position (x_c, y_c), scale S, and orientation θ of the shape. This requires a four-dimensional space, which is smaller than that of the Hough method for ellipses described above.

3.5 Exercises

Exercise 3.1
Implement a version of the Hough transform for straight lines that incorporates the modified accumulator update, as suggested in Sec. 3.3.3. Analyze the extent to which the method improves the robustness with respect to inaccurate or noisy point positions.

Exercise 3.2
Implement a version of the Hough transform for finding lines that takes into account line endpoints as described in Sec. 3.3.3.

Exercise 3.3
Implement a *hierarchical* Hough transform for straight lines (see p. 63) capable of accurately determining line parameters.

Exercise 3.4
Implement the Hough transform for finding circles and circular arcs with varying radii. Make use of a fast algorithm for generating circles, such as described in Sec. 3.4, in the accumulator array.

[5] See Eqn. (2.34) on p. 43 for a parametric equation of this ellipse.