8 Finding Simple Curves: The Hough Transform

In Chapter 6 we demonstrated how to use appropriately designed filters to detect edges in images. These filters compute both the edge strength and orientation at every position in the image. In the following sections, we explain how to decide (e.g., 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 hence are missing from the map (false negatives).

8.1 Salient Image Structures

An intuitive approach to locating large image structures is to first select an arbitrary edge point, systematically examine its neighboring pixels and add 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 [128] (see also Sec. 10.2).

A completely different approach is to search for globally apparent structures that consist of certain simple shape features. As an example, Fig. 8.1 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

© Springer-Verlag London 2016
W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_8

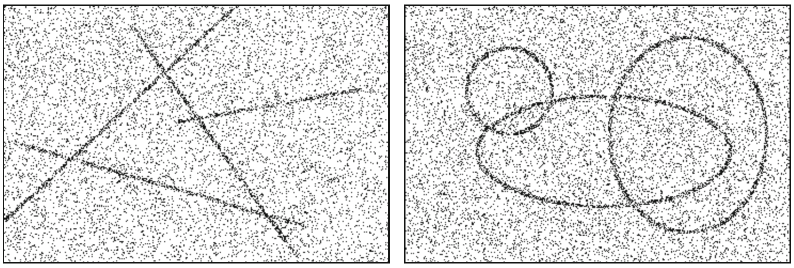


Fig. 8.1
The human visual system is capable of instantly recognizing prominent image structures even under difficult conditions.

recognizes four lines or three ellipses in Fig. 8.1 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.

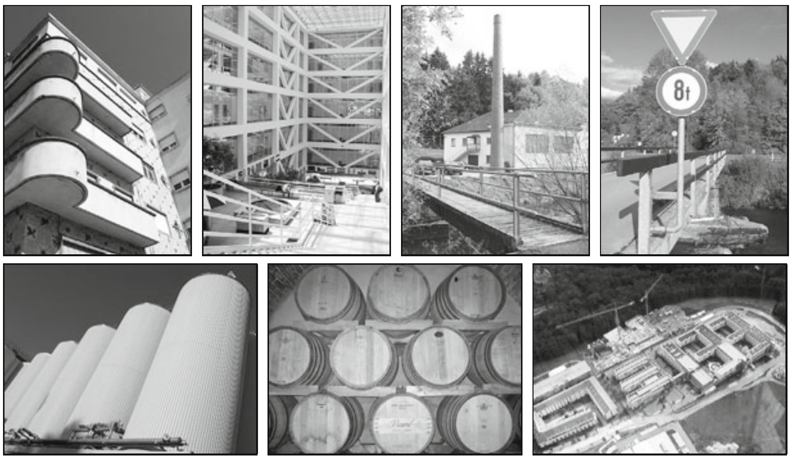
8.2 The Hough Transform

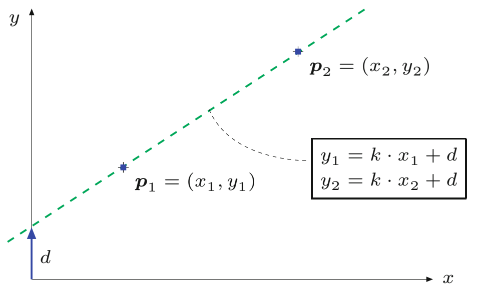
The method from Paul Hough—originally published as a US Patent [111] and often referred to as the "Hough transform" (HT)—is a general approach to localizing any shape that can be defined parametrically within a distribution of points [64, 117]. For example, many geometrical shapes, such as lines, circles, 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. 8.2).

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

$$y = k \cdot x + d, \tag{8.1}$$

Fig. 8.2
Simple geometrical forms such as sections of lines, circles, and ellipses are often found in man-made objects.





y
x
d
$\boldsymbol{p}_2 = (x_2, y_2)$
$\boldsymbol{p}_1 = (x_1, y_1)$
$y_1 = k \cdot x_1 + d$
$y_2 = k \cdot x_2 + d$

Fig. 8.3
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.

where k is the slope and d the intercept—that is, the height at which the line would intercept the y axis (Fig. 8.3). 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

$$y_1 = k \cdot x_1 + d \qquad \text{and} \qquad y_2 = k \cdot x_2 + d, \tag{8.2}$$

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.

8.2.1 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 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

$$L_j \colon y_0 = k_j x_0 + d_j \tag{8.3}$$

for suitable values k_j, d_j. Equation 8.3 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. 8.4). Note that for a given k_j, the solution for d_j in Eqn. (8.3) is

$$d_j = -x_0 \cdot k_j + y_0, \tag{8.4}$$

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. (8.4) 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. (8.4) describes the line

$$M_i : d = -x_i \cdot k + y_i \tag{8.5}$$

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

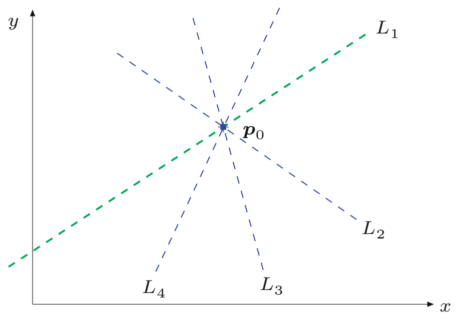


Fig. 8.4
A set of lines passing through an image point. For all possible lines L_j passing through the point $\boldsymbol{p}_0 = (x_0, y_0)$, the equation $y_0 = k_j x_0 + d_j$ holds for appropriate values of the parameters k_j, d_j.

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

	Image Space (x, y)		*Parameter Space* (k, d)	
Point	$\boldsymbol{p}_i = (x_i, y_i)$	$\longleftrightarrow$	$M_i : d = -x_i \cdot k + y_i$	Line
Line	$L_j : y = k_j \cdot x + d_j$	$\longleftrightarrow$	$\boldsymbol{q}_j = (k_j, d_j)$	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. 8.5 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*.

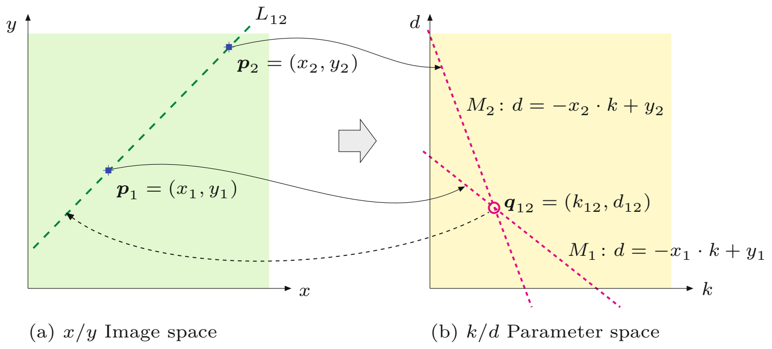
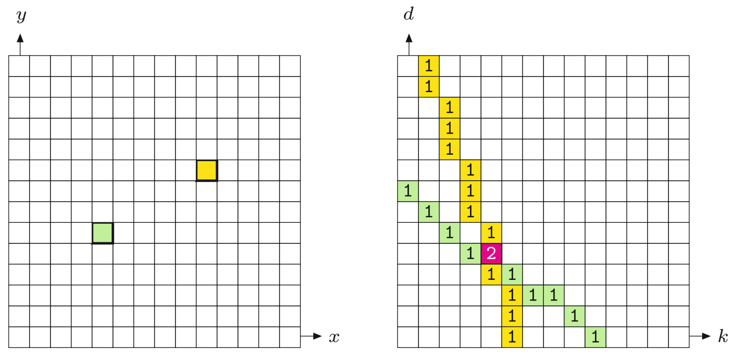

y
L_{12}
$\boldsymbol{p}_2 = (x_2, y_2)$
$\boldsymbol{p}_1 = (x_1, y_1)$
x
d
$M_2 : d = -x_2 \cdot k + y_2$
$\boldsymbol{q}_{12} = (k_{12}, d_{12})$
$M_1 : d = -x_1 \cdot k + y_1$
k
(a) x/y Image space
(b) k/d Parameter space

Fig. 8.5
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.

8.2.2 Accumulator Map

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



(a) Image space (b) Accumulator map

Fig. 8.6
The accumulator map 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).

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 2D 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. 8.6).

8.2.3 A Better Line Representation

The line representation in Eqn. (8.1) is not used in practice because for vertical lines the slope is infinite, that is, $k = \infty$. A more practical representation is the so-called *Hessian normal form* (HNF)[1] for representing lines,

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = r, \tag{8.6}$$

which does not exhibit such singularities and also provides a natural linear quantization for its parameters, the angle θ and the radius r (Fig. 8.7).

With the HNF representation, the parameter space is defined by the coordinates θ, r, and a point $\boldsymbol{p} = (x, y)$ in image space corresponds to the relation

$$r(\theta) = x \cdot \cos(\theta) + y \cdot \sin(\theta), \tag{8.7}$$

for angles in the range $0 \leq \theta < \pi$ (see Fig. 8.8). Thus, for a given image point $\boldsymbol{p}$, the associated radius r is simply a function of the angle θ. If we use the center of the image (of size $M \times N$),

$$\boldsymbol{x}_r = \begin{pmatrix} x_r \\ y_r \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} M \\ N \end{pmatrix}, \tag{8.8}$$

[1] The Hessian normal form is a normalized version of the general ("algebraic") line equation $Ax + By + C = 0$, with $A = \cos(\theta)$, $B = \sin(\theta)$, and $C = -r$ (see, e.g., [35, p. 194]).

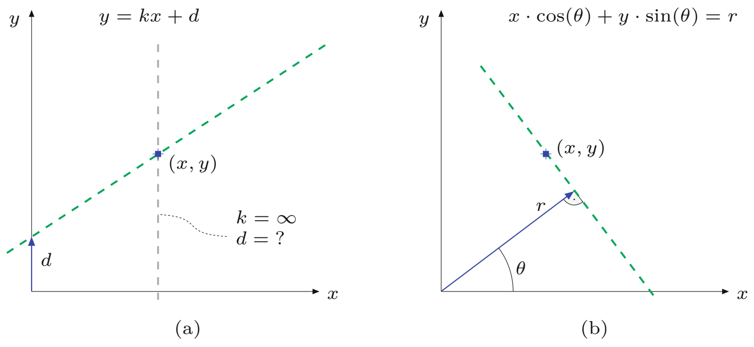


Fig. 8.7 Representation of lines in 2D. In the common k, d representation (a), vertical lines pose a problem because $k = \infty$. The Hessian normal form (b) avoids this problem by representing a line by its angle θ and distance r from the origin.

as the reference point for the x/y image coordinates, then it is possible to limit the range of the radius to half the diagonal of the image, that is,

$$-r_{\max} \leq r(\theta) \leq r_{\max}, \qquad \text{with} \qquad r_{\max} = \tfrac{1}{2}\sqrt{M^2 + N^2}. \tag{8.9}$$

We can see that the function $r(\theta)$ in Eqn. (8.7) is the sum of a cosine and a sine function on θ, each being weighted by the x and y coordinates of the image point (assumed to be constant for the moment). The result is again a sinusoidal function whose magnitude and phase depend only on the weights (coefficients) x, y. Thus, with the Hessian parameterization θ/r, an image point (x, y) does not create a straight line in the accumulator map $\mathsf{A}(i, j)$ but a unique sinusoidal curve, as shown in Fig. 8.8. Again, each image point adds a curve to the accumulator and each resulting cluster point corresponds to to a dominant line in the image with a proportional number of points on it.[2]

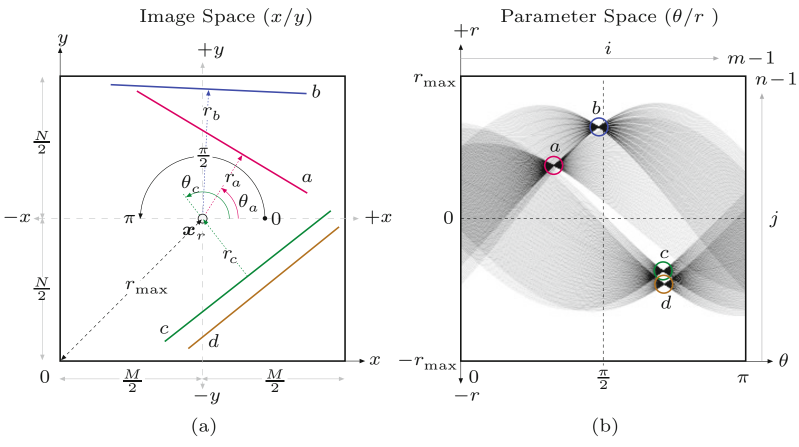


Fig. 8.8 Image space and parameter space using the HNF representation. The image (a) of size $M \times N$ contains four straight lines $L_a, \ldots, L_d$. Each point on an image line creates a sinusoidal curve in the θ/r parameter space (b) and the corresponding line parameters are indicated by the clearly visible cluster points in the accumulator map. The reference point $\boldsymbol{x}_r$ for the x/y coordinates lies at the center of the image. The line angles θ_i are in the range $[0, \pi)$ and the associated radii r_i are in $[-r_{\max}, r_{\max}]$ (the length $r_{\max}$ is half of the image diagonal). For example, the the angle θ_a of line L_a is approximately $\pi/3$, with the (positive) radius $r_a \approx 0.4\, r_{\max}$. Note that, with this parameterization, line L_c has the angle $\theta_c \approx 2\pi/3$ and the *negative* radius $r_c \approx -0.4\, r_{\max}$.

[2] Note that, in Fig. 8.8(a), the positive direction of the y-coordinate runs *upwards* (unlike our usual convention for image coordinates) to stay in line with the previous illustrations (and high school geometry). In practice, the consequences are minor: only the rotation angle runs in the opposite direction and thus the accumulator image in Fig. 8.8(b) was mirrored horizontally for proper display.

8.3 Hough Algorithm

The fundamental Hough algorithm using the HNF line representation (Eqn. (8.6)) is given in Alg. 8.1. Starting with a binary image $I(u, v)$ where the edge pixels have been assigned a value of 1, the first stage creates a 2D accumulator array and then iterates over the image to fill it. The resulting increments are

$$d_\theta = \pi/m \qquad \text{and} \qquad d_r = \sqrt{M^2 + N^2}/n \tag{8.10}$$

for the angle θ and the radius r, respectively. The discrete indices of the accumulators cells are denoted i and j, with $j_0 = n \div 2$ as the center index (for $r = 0$).

For each relevant image point (u, v), a sinusoidal curve is added to the accumulator map by stepping over the discrete angles $\theta_i = \theta_0, \ldots, \theta_{m-1}$, calculating the corresponding radius[3]

$$r(\theta_i) = (u - x_r) \cdot \cos(\theta_i) + (v - y_r) \cdot \sin(\theta_i) \tag{8.11}$$

(see Eqn. (8.7)) and its discrete index

$$j = j_0 + \text{round}\left(\frac{r(\theta_i)}{d_r}\right), \tag{8.12}$$

and subsequently incrementing the accumulator cell $\mathsf{A}(i, j)$ by one (see Alg. 8.1, lines 10–17). The line parameters θ_i and r_j for a given accumulator position (i, j) can be calculated as

$$\theta_i = i \cdot d_\theta \qquad \text{and} \qquad r_j = (j - j_0) \cdot d_r. \tag{8.13}$$

In the second stage of Alg. 8.1, the accumulator array is searched for local peaks above a given minimum Values $a_{\min}$. For each detected peak, a line object is created of the form

$$L_k = \langle \theta_k, r_k, a_k \rangle, \tag{8.14}$$

consisting of the angle θ_k, the radius r_k (relative to the reference point $\boldsymbol{x}_r$), and the corresponding accumulator value a_k. The resulting sequence of lines $\mathcal{L} = (L_1, L_2, \ldots)$ is then sorted by descending a_k and returned.

Figure 8.9 shows the result of applying the Hough transform to a very noisy binary image, which obviously contains four straight lines. They appear clearly as cluster points in the corresponding accumulator map in Fig. 8.9 (b). Figure 8.9 (c) shows the reconstruction of these lines from the extracted parameters. In this example, the resolution of the discrete parameter space is set to 256×256.[4]

[3] The frequent (and expensive) calculation of $\cos(\theta_i)$ and $\sin(\theta_i)$ in Eqn. (8.11) and Alg. 8.1 (line 15) can be easily avoided by initially tabulating the function values for all m possible angles $\theta_i = \theta_0, \ldots, \theta_{m-1}$, which should yield a significant performance gain.

[4] Note that *drawing* a straight line given in Hessian normal form is not really a trivial task (see Excercises 8.1–8.2 for details).

Alg. 8.1
Hough algorithm for detecting straight lines. The algorithm returns a sorted list of straight lines of the form $L_k = \langle \theta_k, r_k, a_k \rangle$ for the binary input image I of size $M \times N$. The resolution of the discrete Hough accumulator map (and thus the step size for the angle and radius) is specified by parameters m and n, respectively. $a_{\min}$ defines the minimum accumulator value, that is, the minimum number of image point on any detected line. The function IsLocalMax() used in line 20 is the same as in Alg. 7.1 (see p. 151).

```
1:  HoughTransformLines($I, m, n, a_{\min}$)
        Input: $I$, a binary image of size $M \times N$; $m$, angular accumulator
        steps; $n$, radial accumulator steps; $a_{\min}$, minimum accumulator
        count per line. Returns a sorted sequence $\mathcal{L} = (L_1, L_2, \ldots)$ of the
        most dominant lines found.
2:      $(M, N) \leftarrow \mathsf{Size}(I)$
3:      $(x_r, y_r) \leftarrow \frac{1}{2} \cdot (M, N)$          ▷ reference point $\boldsymbol{x}_r$ (image center)
4:      $d_\theta \leftarrow \pi / m$                              ▷ angular step size
5:      $d_r \leftarrow \sqrt{M^2 + N^2} / n$                      ▷ radial step size
6:      $j_0 \leftarrow n \div 2$                                  ▷ map index for $r = 0$

        Step 1 – set up and fill the Hough accumulator:
7:      Create map $\mathsf{A}\colon [0, m-1] \times [0, n-1] \mapsto \mathbb{Z}$   ▷ accumulator
8:      for all accumulator cells $(i, j)$ do
9:          $\mathsf{A}(i, j) \leftarrow 0$                        ▷ initialize accumulator

10:     for all $(u, v) \in M \times N$ do                         ▷ scan the image
11:         if $I(u, v) > 0$ then                                  ▷ $I(u, v)$ is a foreground pixel
12:             $(x, y) \leftarrow (u - x_r, v - y_r)$             ▷ shift to reference
13:             for $i \leftarrow 0, \ldots, m-1$ do               ▷ angular coordinate $i$
14:                 $\theta \leftarrow d_\theta \cdot i$           ▷ angle, $0 \leq \theta < \pi$
15:                 $r \leftarrow x \cdot \cos(\theta) + y \cdot \sin(\theta)$   ▷ see Eqn. 8.7
16:                 $j \leftarrow j_0 + \mathrm{round}(r / d_r)$   ▷ radial coordinate $j$
17:                 $\mathsf{A}(i, j) \leftarrow \mathsf{A}(i, j) + 1$   ▷ increment $\mathsf{A}(i, j)$

        Step 2 – extract the most dominant lines:
18:     $\mathcal{L} \leftarrow (\,)$                              ▷ start with empty sequence of lines
19:     for all accumulator cells $(i, j)$ do                      ▷ collect local maxima
20:         if $(\mathsf{A}(i, j) \geq a_{\min}) \wedge \mathsf{IsLocalMax}(\mathsf{A}, i, j)$ then
21:             $\theta \leftarrow i \cdot d_\theta$               ▷ angle $\theta$
22:             $r \leftarrow (j - j_0) \cdot d_r$                 ▷ radius $r$
23:             $a \leftarrow \mathsf{A}(i, j)$                    ▷ accumulated value $a$
24:             $L \leftarrow \langle \theta, r, a \rangle$        ▷ create a new line $L$
25:             $\mathcal{L} \leftarrow \mathcal{L} \smile (L)$    ▷ add line $L$ to sequence $\mathcal{L}$
26:     $\mathsf{Sort}(\mathcal{L})$                               ▷ sort $\mathcal{L}$ by descending accumulator count $a$
27:     return $\mathcal{L}$
```

8.3.1 Processing the Accumulator Array

The reliable detection and precise localization of peaks in the accumulator map $\mathsf{A}(i, j)$ is not a trivial problem. As can readily be seen in Fig. 8.9(b), even in the case where the lines in the image are geometrically "straight", the parameter space curves associated with them do not intersect 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 by the discrete coordinate grid used for the accumulator array. Since the maximum points are really maximum *areas* in the accumulator array, simply traversing the array and returning the positions of its largest values is not sufficient. Since this is a critical step in the algorithm, we examine two different approaches below (see Fig. 8.10).

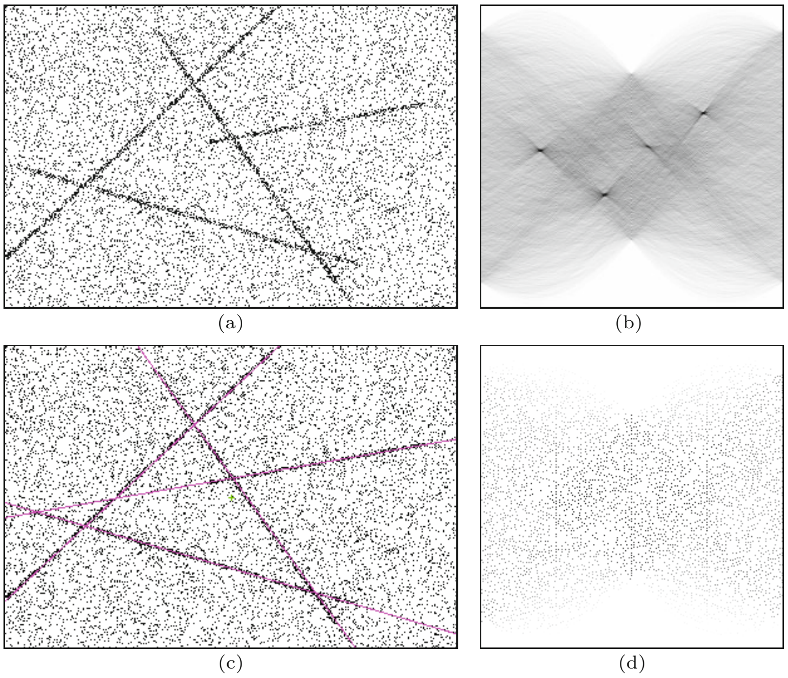


Fig. 8.9
Hough transform for straight lines. The dimensions of the original image (a) are 360×240 pixels, so the maximal radius (measured from the image center) is $r_{\max} \approx 216$. For the parameter space (b), a step size of 256 is used for both the angle $\theta = 0, \ldots, \pi$ (horizontal axis) and the radius $r = -r_{\max}, \ldots, r_{\max}$ (vertical axis). The four (dark) clusters in (b) surround the maximum values in the accumulator array, and their parameters correspond to the four lines in the original image. Intensities are shown inverted in all images to improve legibility.

Approach A: Thresholding

First the accumulator is thresholded to the value of t_a by setting all accumulator values $\mathsf{A}(i,j) < t_a$ to 0. The resulting scattering of points, or point clouds, are first coalesced into regions (Fig. 8.10(b)) using a technique such as a morphological *closing* operation (see Sec. 9.3.2). Next the remaining regions must be localized, for instance using the region-finding technique from Sec. 10.1, and then each region's centroid (see Sec. 10.5) 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.[5] This is carried out by determining for every accumulator cell $\mathsf{A}(i,j)$ 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. 8.10(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 considered. The result for Fig. 8.9(a) is shown in Fig. 8.10(d).

[5] Nonmaximum suppression is also used in Sec. 7.2.3 for isolating corner points.

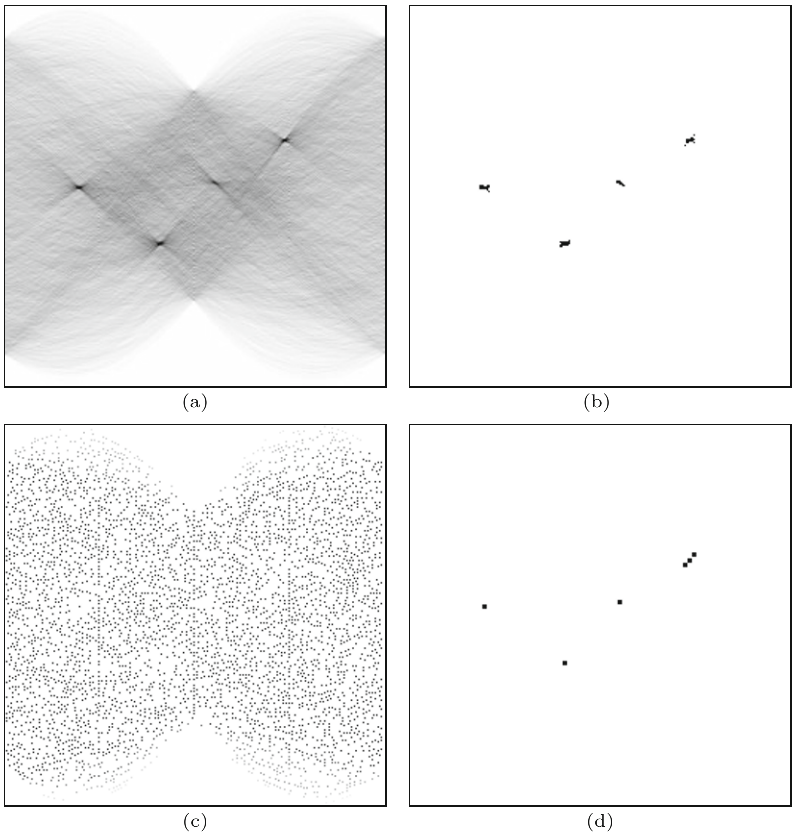


Fig. 8.10
Finding local maximum values in the accumulator array. Original distribution of the values in the Hough accumulator (a). **Variant A:** *Threshold operation* using 50% of the maximum value (b). The remaining regions represent the four dominant lines in the image, and the coordinates of their centroids are a good approximation to the line parameters. **Variant B:** Using *non-maximum suppression* results in a large number of local maxima (c) that must then be reduced using a threshold operation (d).

Mind the vertical lines!

Special consideration should be given to *vertical* lines (once more!) when processing the contents of the accumulator map. The parameter pairs for these lines lie near $\theta = 0$ and $\theta = \pi$ at the left and right borders, respectively, of the accumulator map (see Fig. 8.8(b)). Thus, to locate peak clusters in this part of the parameter space, the horizontal coordinate along the θ axis must be treated circularly, that is, modulo m. However, as can be seen clearly in Fig. 8.8(b), the sinusoidal traces in the parameter space do not continue smoothly at the transition $\theta = \pi \rightarrow 0$, but are vertically mirrored! Evaluating such neighborhoods near the borders of the parameter space thus requires special treatment of the vertical (r) accumulator coordinate.

8.3.2 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 method.

Modified accumulation

The purpose of the accumulator map is to locate the intersections of multiple 2D curves. Due to the discrete nature of the image and accumulator coordinates, rounding errors usually cause the parameter curves not to intersect in a single accumulator cell, even when the associated image lines are exactly straight. A common remedy is, for a given angle $\theta = i_\theta \cdot \Delta_\theta$ (Alg. 8.1), to increment not only the *main* accumulator cell $\mathsf{A}(i,j)$ but also the *neighboring* cells $\mathsf{A}(i,j-1)$ and $\mathsf{A}(i,j+1)$, possibly with different weights. This makes the Hough transform more tolerant against inaccurate point coordinates and rounding errors.

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 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, that is,

$$\mathsf{A}(i,j) \leftarrow \mathsf{A}(i,j) + E(u,v). \tag{8.15}$$

In this way, strong edge points will contribute more to the accumulated values than weak ones (see also Exercise 8.6).

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, e.g., [125, p. 483]).

Bias compensation

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. 8.11). 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

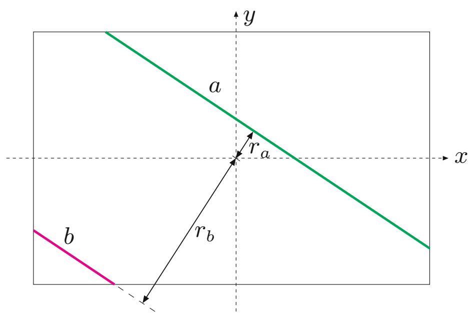


Fig. 8.11
Hough transform bias problem. When an image represents only a finite section of an object, then those lines nearer the center (smaller r values) will have higher values than those farther away (larger r values). As an example, the maximum value of the accumulator for line a will be higher than that of line b.

this inherent bias is to compute for each accumulator entry $\mathsf{A}(i,j)$ the maximum number of image points $\mathsf{A}_{\max}(i,j)$ possible for a line with the corresponding parameters and then normalize the result, for example, in the form

$$\mathsf{A}(i,j) \leftarrow \frac{\mathsf{A}(i,j)}{\max(1, \mathsf{A}_{\max}(i,j))}. \tag{8.16}$$

The normalization map $\mathsf{A}_{\max}(i,j)$ can be determined analytically (by calculating the intersecting length of each line) or by simulation; 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 four addition coordinates to

$$\mathsf{A}(i,j) = (a, u_{\min}, v_{\min}, u_{\max}, v_{\max}), \tag{8.17}$$

where component a denotes the original accumulator value and $u_{\min}$, $v_{\min}$, $u_{\max}$, $v_{\max}$ are the coordinates of the line's bounding box. After the additional coordinates are initialized, they are updated simultaneously with the positions along the parameter trace for every image point (u, v). After completion of the process, the accumulator cell (i, j) contains the bounding box for all image points that contributed it. When finding the maximum values in the second stage, care should be taken so that the merged cells contain the correct endpoints (see also Exercise 8.4).

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.

Line intersections

It may be useful in certain applications not to find the lines themselves but their intersections, for example, 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 (that is, as pairs $L_k = \langle\theta_k, r_k\rangle$). To compute the point of intersection $\boldsymbol{x}_{12} = (x_{12}, y_{12})^\intercal$ for two lines $L_1 = \langle\theta_1, r_1\rangle$ and $L_2 = \langle\theta_2, r_2\rangle$ we need to solve the system of linear equations

$$\begin{aligned} x_{12} \cdot \cos(\theta_1) + y_{12} \cdot \sin(\theta_1) &= r_1, \\ x_{12} \cdot \cos(\theta_2) + y_{12} \cdot \sin(\theta_2) &= r_2, \end{aligned} \tag{8.18}$$

for the unknowns x_{12}, y_{12}. The solution is

$$\begin{aligned} \begin{pmatrix} x_{12} \\ y_{12} \end{pmatrix} &= \frac{1}{\cos(\theta_1)\sin(\theta_2) - \cos(\theta_2)\sin(\theta_1)} \cdot \begin{pmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{pmatrix} \\ &= \frac{1}{\sin(\theta_2 - \theta_1)} \cdot \begin{pmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{pmatrix}, \end{aligned} \tag{8.19}$$

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 $\theta_1 \equiv \theta_2$).

Figure 8.12 shows an illustrative example using *ARToolkit*[6] markers. After automatic thresholding (see Ch. 11) the straight line segments along the outer boundary of the largest binary region are analyzed with the Hough transform. Subsequently, the corners of the marker are calculated precisely as the intersection points of the involved line segments.

8.4 Java Implementation

The complete Java source code for the straight line Hough transform is available online in class `HoughTransformLines`.[7] Detailed usage of this class is shown in the ImageJ plugin `Find_Straight_Lines` (see also Prog. 8.1 for a minimal example).[8]

`HoughTransformLines` (class)

This class is a direct implementation of the Hough transform for straight lines, as outlined in Alg. 8.1. The sin/cos function calls (see Alg. 8.1, line 15) are substituted by precalculated tables for improved efficiency. The class defines the following constructors:

`HoughTransformLines (ImageProcessor I, Parameters params)`

I denotes the input image, where all pixel values > 0 are assumed to be relevant (edge) points; `params` is an instance of the (inner) class `HoughTransformLines.Parameters`, which allows to specify the accumulator size (`nAng`, `nRad`) etc.

[6] Used for augmented reality applications, see www.hitl.washington.edu/artoolkit/.

[7] Package `imagingbook.pub.hough`.

[8] Note that the current implementation has no bias compensation (see Sec. 8.3.2, Fig. 8.11).

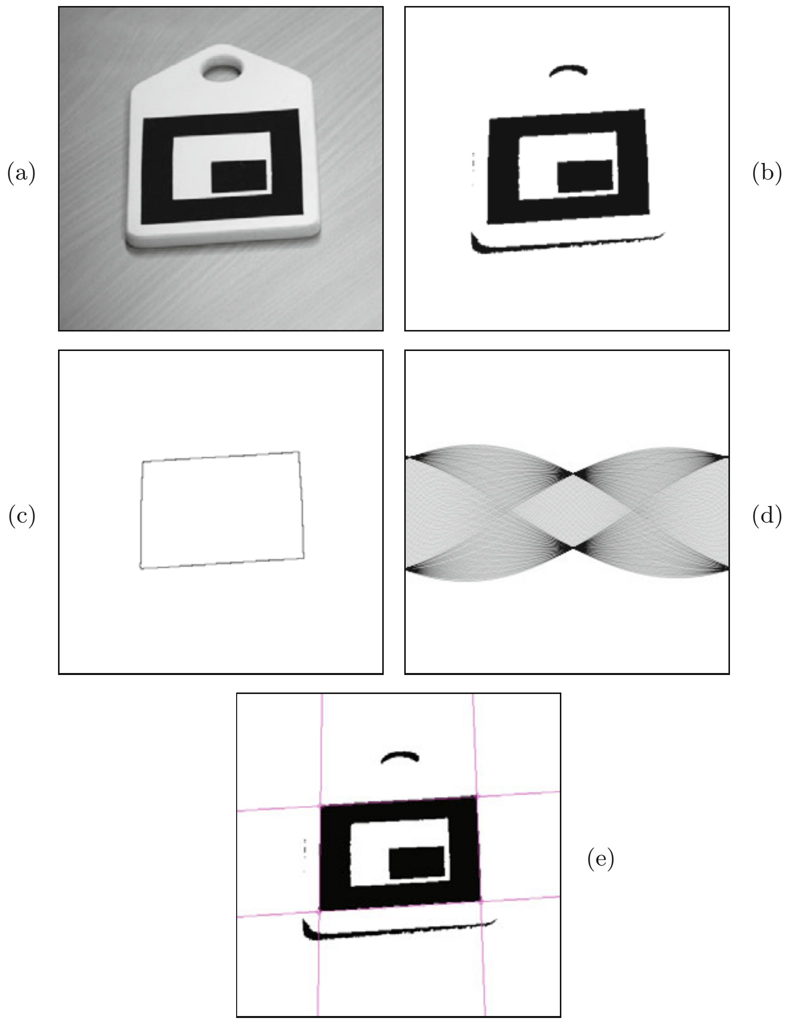


Fig. 8.12
Hough transform used for precise calculation of corner points. Original image showing a typical ARToolkit marker (a), result after automatic thresholding (b). The outer contour pixels of the largest binary region (c) are used as input points to the Hough transform. Hough accumulator map (d), detected lines and marked intersection points (e).

`HoughTransformLines (Point2D[] points, int M, int N, Parameters params)`
: In this case the Hough transform is calculated for a sequence of 2D points (`points`); `M`, `N` specify the associated coordinate frame (for calculating the reference point $\boldsymbol{x}_r$), which is typically the original image size; `params` is a parameter object (as described before).

The most important public methods of the class ClassHoughTransformLines are:

`HoughLine[] getLines (int amin, int maxLines)`
: Returns a sorted sequence of line objects[9] whose accumulator value is `amin` or greater. The sequence is sorted by accumulator values and contains up to `maxLines` elements

`int[][] getAccumulator ()`
: Returns a reference to the accumulator map A (of size $m \times n$ for angles and radii, respectively).

[9] Of type `HoughTransformLines.HoughLine`.

```
 1 import imagingbook... .HoughTransformLines;
 2 import imagingbook... .HoughTransformLines.HoughLine;
 3 import imagingbook... .HoughTransformLines.Parameters;
 4 ...
 5
 6   public void run(ImageProcessor ip) {
 7     Parameters params = new Parameters();
 8     params.nAng = 256;     // = m
 9     params.nRad = 256;     // = n
10
11     // compute the Hough Transform:
12     HoughTransformLines ht =
13           new HoughTransformLines(ip, params);
14
15     // retrieve the 5 strongest lines with min. 50 accumulator votes
16     HoughLine[] lines = ht.getLines(50, 5);
17
18     if (lines.length > 0) {
19       IJ.log("Lines found:");
20       for (HoughLine L : lines) {
21         IJ.log(L.toString()); // list the resulting lines
22       }
23     }
24     else
25       IJ.log("No lines found!");
26   }
```

Prog. 8.1
Minimal example for the usage of class `HoughTransformLines` (`run()` method for an ImageJ plugin of type `PlugInFilter`). First (in lines 7–9) a parameter ob ect is created and configured; `nAng` (= m) and `nRad` (= n) specify the number of discrete angular and radial steps in the Hough accumulator map. In lines 12-13 an instance of `HoughTransformLines` is created for the image `ip`. The accumulator map is calculated in this step. In line 16, `getLines()` is called to retrieve the sequence of the 5 strongest detected lines, with at least 50 image points each. Unless empty, this sequence is subsequently listed.

`int[][] getAccumulatorMax()`
Returns a copy of accumulator array in which all non-maxima are replaced by zero values.

`FloatProcessor getAccumulatorImage()`
Returns a floating-point image of the accumulator array, analogous to `getAccumulator()`. Angles θ_i run horizontally, radii r_j vertically.

`FloatProcessor getAccumulatorMaxImage()`
Returns a floating-point image of the accumulator array with suppressed non-maximum values, analogous to `getAccumulatorMax()`.

`double angleFromIndex(int i)`
Returns the angle $\theta_i \in [0, \pi)$ for the given index `i` in the range $0, \ldots, m-1$.

`double radiusFromIndex(int j)`
Returns the radius $r_j \in [-r_{\max}, r_{\max}]$ for the given index j in the range $0, \ldots, n-1$.

`Point2D getReferencePoint()`
Returns the (fixed) reference point $\boldsymbol{x}_r$ for this Hough transform instance.

`HoughLine` (class)

`HoughLine` represents a straight line in Hessian normal form. It is implemented as an inner class of `HoughTransformLines`. It offers no public constructor but the following methods:

`double getAngle ()`
: Returns the angle $\theta \in [0, \pi)$ of this line.

`double getRadius ()`
: Returns the radius $r \in [-r_{\max}, r_{\max}]$ of this line, relative to the associated Hough transform's reference point $\boldsymbol{x}_r$.

`int getCount ()`
: Returns the Hough transform's accumulator value (number of registered image points) for this line.

`Point2D getReferencePoint ()`
: Returns the (fixed) reference point $\boldsymbol{x}_r$ for this line. Note that all lines associated with a given Hough transform share the same reference point.

`double getDistance (Point2D p)`
: Returns the Euclidean distance of point `p` to this line. The result may be positive or negative, depending on which side of the line `p` is located.

8.5 Hough Transform for Circles and Ellipses

8.5.1 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

$$C = \langle \bar{x}, \bar{y}, r \rangle,$$

where $\bar{x}$, $\bar{y}$ are the coordinates of the center and ρ is the radius of the circle (Fig. 8.13).

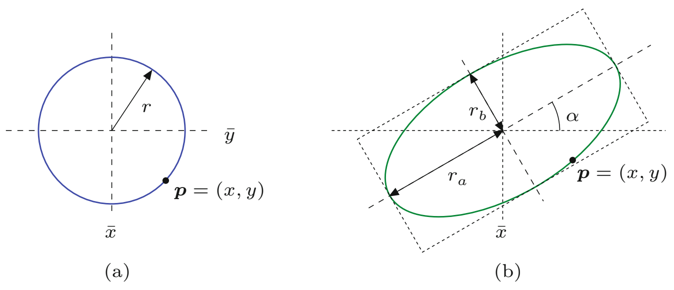


Fig. 8.13 Representation of circles and ellipses in 2D. A circle (a) requires three parameters (e.g., $\bar{x}, \bar{y}, r$). An arbitrary ellipse (b) takes five parameters (e.g., $\bar{x}, \bar{y}, r_a, r_b, \alpha$).

A point $\boldsymbol{p} = (x, y)$ lies exactly on the circle C if the condition

$$(x - \bar{x})^2 + (x - \bar{y})^2 = r^2 \qquad (8.20)$$

holds. Therefore the Hough transform for circles requires a 3D parameter space $\mathsf{A}(i, j, k)$ 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 $(\bar{x}, \bar{y}, r)$ that satisfies Eqn. (8.20) for a given image point (x, y)? A "brute force" is to a exhaustively test all cells of the parameter space to see if the relation in Eqn. (8.20) holds, which is computationally quite expensive, of course.

If we examine Fig. 8.14, 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 3D parameter space for each image point. Instead we need only increase the cell values along the edge of the appropriate circle on each r 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 [33] is particularly well-suited.

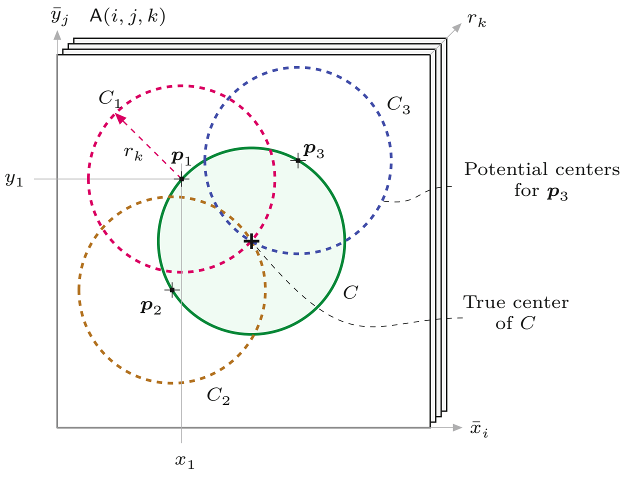


Fig. 8.14
Hough transform for circles. The illustration depicts a single slice of the 3D accumulator array $\mathsf{A}(i, j, k)$ at a given circle radius r_k. The center points of all the circles running through a given image point $\boldsymbol{p}_1 = (x_1, y_1)$ form a circle C_1 with a radius of r_k 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 r_k are traversed and their values in the accumulator array incremented. The cell in the accumulator array contains a value of 3 where the circles intersect at the true center of the image circle C.

Figure 8.15 shows the spatial structure of the 3D parameter space for circles. For a given image point $\boldsymbol{p}_m = (u_m, v_m)$, at each plane along the r axis (for $r_k = r_{\min}, \ldots, r_{\max}$), a circle centered at (u_m, v_m) with the radius r_k is traversed, ultimately creating a 3D 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. 8.3.2) also occurs in 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.

8.5.2 Ellipses

In a perspective image, most circular objects originating in our real, 3D 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

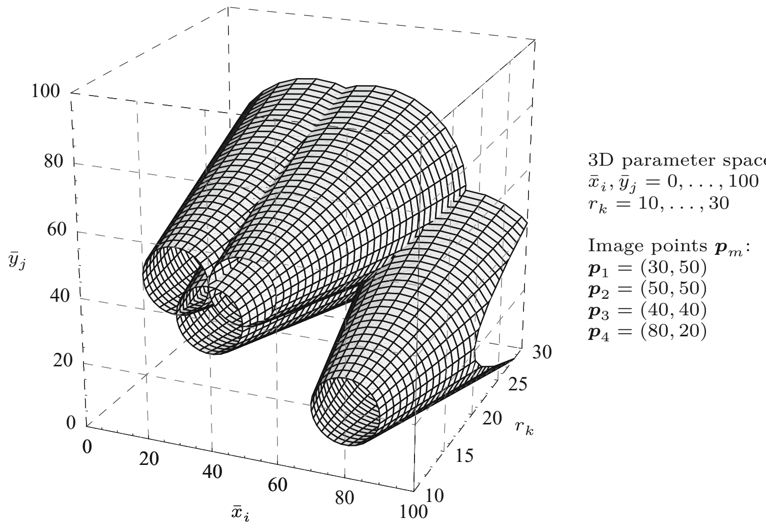


Fig. 8.15
3D parameter space for circles. For each image point $\boldsymbol{p} = (u, v)$, the cells lying on a cone (with its axis at (u, v) and varying radius r_k) in the 3D accumulator $\mathsf{A}(i, j, k)$ are traversed and incremented. The size of the discrete accumulator is set to $100 \times 100 \times 30$. Candidate center points are found where many of the 3D surfaces intersect.

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, for example,

$$E = \langle \bar{x}, \bar{y}, r_a, r_b, \alpha \rangle, \tag{8.21}$$

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. 8.13).[10] In order to find ellipses of any size, position, and orientation using the Hough transform, a 5D 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 every dimension results in 2^{35} accumulator cells, and implementing these using 4-byte `int` values thus requires 2^{37} bytes (128 gigabytes) of memory. Moreover, the amount of processing required for filling and evaluating such a huge parameter space makes this method unattractive for real applications.

An interesting alternative in this case is the *generalized Hough transform*, which in principle can be used for detecting any arbitrary 2D shape [15,117]. 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 4D space, which is smaller than that of the Hough method for ellipses described earlier.

[10] See Chapter 10, Eqn. (10.39) for a parametric equation of this ellipse.

8.6 Exercises

Exercise 8.1. Drawing a straight line given in Hessian normal (HNF) form is not directly possible because typical graphics environments can only draw lines between two specified end points.[11] An HNF line $L = \langle\theta, r\rangle$, specified relative to a reference point $\boldsymbol{x}_r = (x_r, y_r)$, can be drawn into an image I in several ways (implement both versions):

Version 1: Iterate over all image points (u, v); if Eqn. (8.11), that is,

$$r = (u - x_r) \cdot \cos(\theta) + (v - y_r) \cdot \sin(\theta), \tag{8.22}$$

is satisfied for position (u, v), then mark the pixel $I(u, v)$. Of course, this "brute force" method will only show those (few) line pixels whose positions satisfy the line equation *exactly*. To obtain a more "tolerant" drawing method, we first reformulate Eqn. (8.22) to

$$(u - x_r) \cdot \cos(\theta) + (v - y_r) \cdot \sin(\theta) - r = d. \tag{8.23}$$

Obviously, Eqn. (8.22) is only then exactly satisfied if $d = 0$ in Eqn. (8.23). If, however, Eqn. (8.22) is *not* satisfied, then the magnitude of $d \neq 0$ equals the distance of the point (u, v) from the line. Note that d itself may be positive or negative, depending on which side of the line (u, v) is located. This suggests the following version.

Version 2: Define a constant $w > 0$. Iterate over all image positions (u, v); whenever the inequality

$$|(u - x_r) \cdot \cos(\theta) + (v - y_r) \cdot \sin(\theta) - r| \leq w \tag{8.24}$$

is satisfied for position (u, v), mark the pixel $I(u, v)$. For example, all line points should show with $w = 1$. What is the geometric meaning of w?

Exercise 8.2. Develop a less "brutal" method (compared to Exercise 8.1) for drawing a straight line $L = \langle\theta, r\rangle$ in Hessian normal form (HNF). First, set up the HNF equations for the four border lines of the image, A, B, C, D. Now determine the intersection points of the given line L with each border line $A, \ldots, D$ and use the built-in `drawLine()` method or a similar routine to draw L by connecting the intersection points. Consider which special situations may appear and how they could be handled.

Exercise 8.3. Implement (or extend) the Hough transform for straight lines by including measures against the bias problem, as discussed in Sec. 8.3.2 (Eqn. (8.16)).

Exercise 8.4. Implement (or extend) the Hough transform for finding lines that takes into account line endpoints, as described in Sec. 8.3.2 (Eqn. (8.17)).

Exercise 8.5. Calculate the pairwise intersection points of all detected lines (see Eqns. (8.18)–(8.19)) and show the results graphically.

[11] For example, with `drawLine(x1, y1, x2, y2)` in ImageJ.

Exercise 8.6. Extend the Hough transform for straight lines so that updating the accumulator map takes into account the intensity (edge magnitude) of the current pixel, as described in Eqn. (8.15).

Exercise 8.7. Implement a *hierarchical* Hough transform for straight lines (see p. 172) capable of accurately determining line parameters.

Exercise 8.8. Implement the Hough transform for finding circles and circular arcs with varying radii. Make use of a fast algorithm for drawing circles in the accumulator array, such as described in Sec. 8.5.