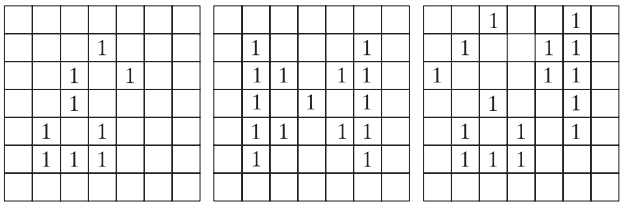


a b c

FIGURE 11.2 Examples of boundaries that can be processed by the boundary-following algorithm. (a) Closed boundary with a branch. (b) Self-intersecting boundary. (c) Multiple boundaries (processed one at a time).

a straightforward approach is to extract the holes (see Section 9.6) and treat them as 1-valued regions on a background of 0's. Applying the boundary-following algorithm to these regions will yield the inner boundaries of the original region.

We could have stated the algorithm just as easily based on following a boundary in the counterclockwise direction but you will find it easier to have just one algorithm and then reverse the order of the result to obtain a sequence in the opposite direction. We use both directions interchangeably (but consistently) in the following sections to help you become familiar with both approaches.

CHAIN CODES

Chain codes are used to represent a boundary by a connected sequence of straight-line segments of specified length and direction. We assume in this section that all curves are closed, simple curves (i.e., curves that are closed and not self intersecting).

Freeman Chain Codes

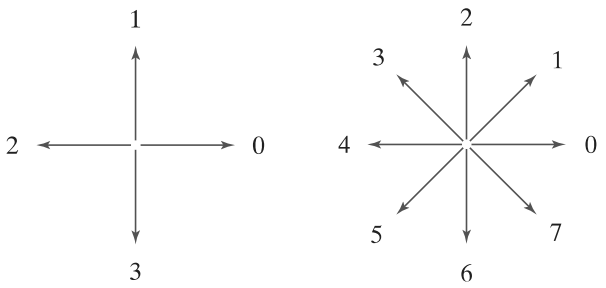
Typically, a chain code representation is based on 4- or 8-connectivity of the segments. The *direction* of each segment is coded by using a numbering scheme, as in Fig. 11.3. A boundary code formed as a sequence of such directional numbers is referred to as a *Freeman chain code*.

Digital images usually are acquired and processed in a grid format with equal spacing in the x- and y-directions, so a chain code could be generated by following a boundary in, say, a clockwise direction and assigning a direction to the segments connecting every pair of pixels. This level of detail generally is not used for two principal reasons: (1) The resulting chain would be quite long and (2) any small disturbances along the boundary due to noise or imperfect segmentation would cause changes in the code that may not be related to the principal shape features of the boundary.

An approach used to address these problems is to resample the boundary by selecting a larger grid spacing, as in Fig. 11.4(a). Then, as the boundary is traversed, a boundary point is assigned to a node of the coarser grid, depending on the proximity of the original boundary point to that node, as in Fig. 11.4(b). The resampled boundary obtained in this way can be represented by a 4- or 8-code. Figure 11.4(c) shows the coarser boundary points represented by an 8-directional chain code. It is a simple matter to convert from an 8-code to a 4-code and vice versa (see Problems 2.15, 9.27,

a b

FIGURE 11.3 Direction numbers for (a) 4-directional chain code, and (b) 8-directional chain code.



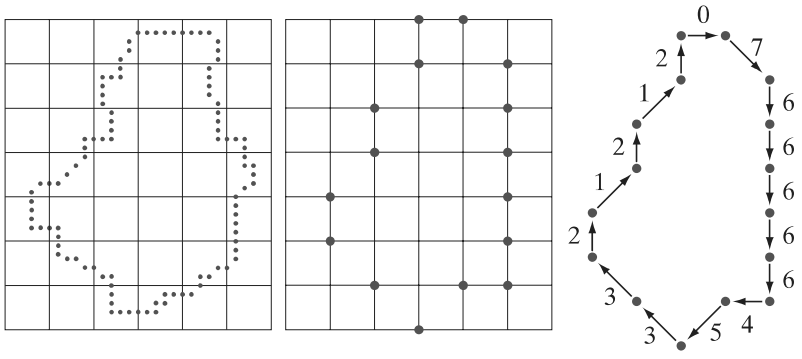
and 9.29). For the same reason mentioned when discussing boundary tracing earlier in this section, we chose the starting point in Fig. 11.4(c) as the uppermost-leftmost point of the boundary, which gives the chain code 0766…1212. As you might suspect, the spacing of the resampling grid is determined by the application in which the chain code is used.

If the sampling grid used to obtain a connected digital curve is a uniform quadrilateral (see Fig. 2.19) all points of a Freeman code based on Fig. 11.3 are guaranteed to coincide with the points of the curve. The same is true if a digital curve is subsampled using the same type of sampling grid, as in Fig. 11.4(b). This is because the samples of curves produced using such grids have the same arrangement as in Fig. 11.3, so all points are reachable as we traverse a curve from one point to the next to generate the code.

The numerical value of a chain code depends on the starting point. However, the code can be normalized with respect to the starting point by a straightforward procedure: We simply treat the chain code as a circular sequence of direction numbers and redefine the starting point so that the resulting sequence of numbers forms an integer of minimum magnitude. We can normalize also for rotation (in angles that are integer multiples of the directions in Fig. 11.3) by using the *first difference* of the chain code instead of the code itself. This difference is obtained by counting the number of direction changes (in a counterclockwise direction in Fig. 11.3) that separate two adjacent elements of the code. If we treat the code as a circular sequence to normalize it with respect to the starting point, then the first element of the difference is computed by using the transition between the last and first components of the chain.

a b c

FIGURE 11.4 (a) Digital boundary with resampling grid superimposed. (b) Result of resampling. (c) 8-directional chain-coded boundary.



For instance, the first difference of the 4-directional chain code 10103322 is 3133030. Size normalization can be achieved by altering the spacing of the resampling grid.

The normalizations just discussed are exact only if the boundaries themselves are invariant to rotation (again, in angles that are integer multiples of the directions in Fig. 11.3) and scale change, which seldom is the case in practice. For instance, the same object digitized in two different orientations will have different boundary shapes in general, with the degree of dissimilarity being proportional to image resolution. This effect can be reduced by selecting chain elements that are long in proportion to the distance between pixels in the digitized image, and/or by orienting the resampling grid along the principal axes of the object to be coded, as discussed in Section 11.3, or along its eigen axes, as discussed in Section 11.5.

EXAMPLE 11.1: Freeman chain code and some of its variations.

Figure 11.5(a) shows a 570×570-pixel, 8-bit gray-scale image of a circular stroke embedded in small, randomly distributed specular fragments. The objective of this example is to obtain a Freeman chain code, the corresponding integer of minimum magnitude, and the first difference of the outer boundary of the stroke. Because the object of interest is embedded in small fragments, extracting its boundary would result in a noisy curve that would not be descriptive of the general shape of the object. As you know, smoothing is a routine process when working with noisy boundaries. Figure 11.5(b) shows the original image smoothed using a box kernel of size 9×9 pixels (see Section 3.5 for a discussion of spatial smoothing), and Fig. 11.5(c) is the result of thresholding this image with a global threshold obtained using Otsu's method. Note that the number of regions has been reduced to two (one of which is a dot), significantly simplifying the problem.

Figure 11.5(d) is the outer boundary of the region in Fig. 11.5(c). Obtaining the chain code of this boundary directly would result in a long sequence with small variations that are not representative of the global shape of the boundary, so we resample it before obtaining its chain code. This reduces insignificant variability. Figure 11.5(e) is the result of using a resampling grid with nodes 50 pixels apart (approximately 10% of the image width) and Fig. 11.5(f) is the result of joining the sample points by straight lines. This simpler approximation retained the principal features of the original boundary.

The 8-directional Freeman chain code of the simplified boundary is

0 0 0 0 6 0 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2

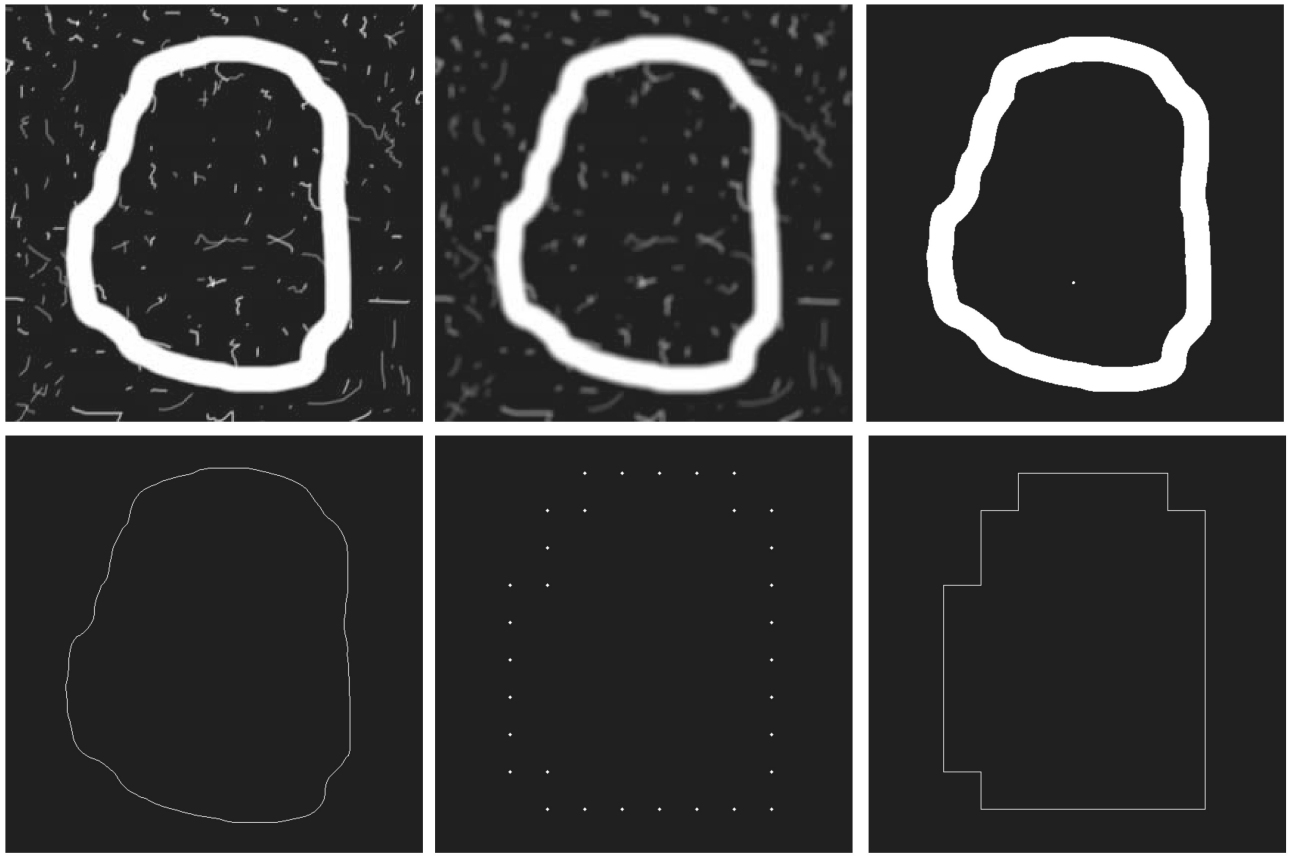
The starting point of the boundary is at coordinates (2, 5) in the subsampled grid (remember from Fig. 2.19 that the origin of an image is at its top, left). This is the uppermost-leftmost point in Fig. 11.5(f). The integer of minimum magnitude of the code happens in this case to be the same as the chain code:

0 0 0 0 6 0 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2

The first difference of the code is

0 0 0 6 2 6 0 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0 0 0 6 2 0 6 2 6

Using this code to represent the boundary results in a significant reduction in the amount of data needed to store the boundary. In addition, working with code numbers offers a unified way to analyze the shape of a boundary, as we discuss in Section 11.3. Finally, keep in mind that the subsampled boundary can be recovered from any of the preceding codes.



a b c
d e f

FIGURE 11.5 (a) Noisy image of size 570×570 pixels. (b) Image smoothed with a 9×9 box kernel. (c) Smoothed image, thresholded using Otsu's method. (d) Longest outer boundary of (c). (e) Subsampled boundary (the points are shown enlarged for clarity). (f) Connected points from (e).

Slope Chain Codes

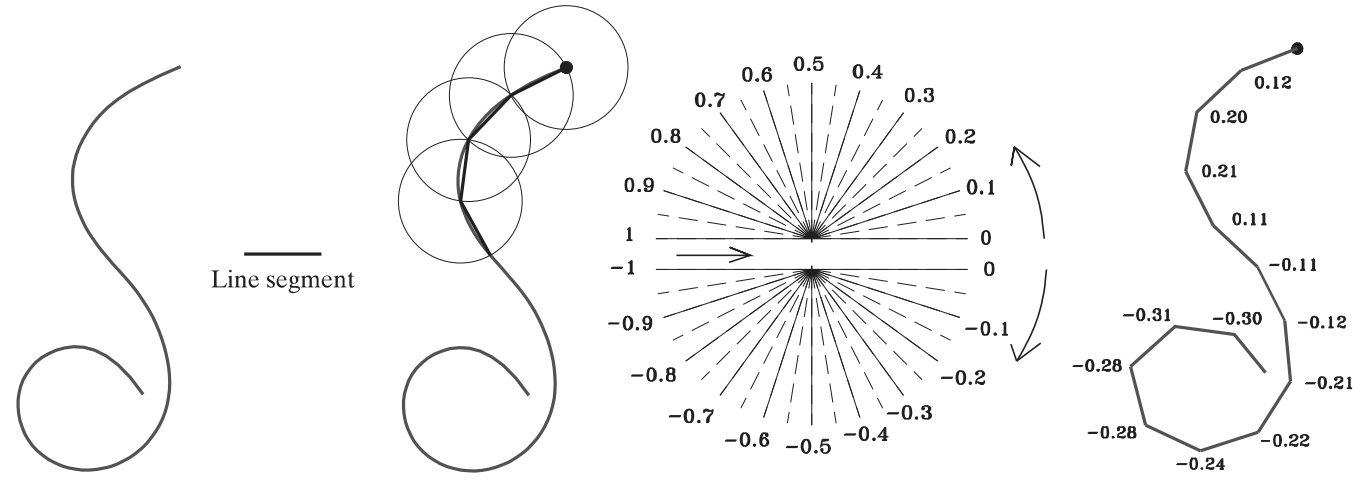
Using Freeman chain codes generally requires resampling a boundary to smooth small variations, a process that implies defining a grid and subsequently assigning all boundary points to their closest neighbors in the grid. An alternative to this approach is to use *slope chain codes* (SCCs) (Bribiesca [1992, 2013]). The SCC of a 2-D curve is obtained by placing straight-line segments of equal length around the curve, with the end points of the segments touching the curve.

Obtaining an SSC requires calculating the *slope changes* between contiguous line segments, and normalizing the changes to the *continuous* (open) interval $(-1, 1)$. This approach requires defining the length of the line segments, as opposed to Freeman codes, which require defining a grid and assigning curve points to it—a much more elaborate procedure. Like Freeman codes, SCCs are independent of rotation, but a larger range of possible slope changes provides a more accurate representation under rotation than the rotational independence of the Freeman codes, which is limited to the eight directions in Fig. 11.3(b). As with Freeman codes, SCCs are independent of translation, and can be normalized for scale changes (see Problem 11.8).

Figure 11.6 illustrates how an SCC is generated. The first step is to select the length of the line segment to use in generating the code [see Fig. 11.6(b)]. Next, a starting point (the origin) is specified (for an open curve, the logical starting point is one of its end points). As Fig. 11.6(c) shows, once the origin has been selected, one end of a line segment is placed at the origin and the other end of the segment is set to coincide with the curve. This point becomes the starting point of the next line segment, and we repeat this procedure until the starting point (or end point in the case of an open curve) is reached. As the figure illustrates, you can think of this process as a sequence of identical circles (with radius equal to the length of the line segment) traversing the curve. The intersections of the circles and the curve determine the nodes of the straight-line approximation to the curve.

Once the intersections of the circles are known, we determine the slope changes between contiguous line segments. Positive and zero slope changes are normalized to the open half interval $[0, 1)$, while negative slope changes are normalized to the open interval $(-1, 0)$. Not allowing slope changes of ± 1 eliminates the implementation issues that result from having to deal with the fact that such changes result in the same line segment with opposite directions.

The sequence of slope changes is the chain that defines the SCC approximation to the original curve. For example, the code for the curve in Fig. 11.6(e) is 0.12, 0.20, 0.21, 0.11, −0.11, −0.12, −0.21, −0.22, −0.24, −0.28, −0.28, −0.31, −0.30. The accuracy of the slope changes defined in Fig. 11.6(d) is 10^{-2}, resulting in an "alphabet" of 199 possible symbols (slope changes). The accuracy can be changed, of course. For instance, and accuracy of 10^{-1} produces an alphabet of 19 symbols (see Problem 11.6). Unlike a Freeman code, there is no guarantee that the last point of the coded curve will coincide with the last point of the curve itself. However, shortening the line



a b c d e

FIGURE 11.6 (a) An open curve. (b) A straight-line segment. (c) Traversing the curve using circumferences to determine slope changes; the dot is the origin (starting point). (d) Range of slope changes in the open interval $(-1, 1)$ (the arrow in the center of the chart indicates direction of travel). There can be ten subintervals between the slope numbers shown.(e) Resulting coded curve showing its corresponding numerical sequence of slope changes. (Courtesy of Professor Ernesto Bribiesca, IIMAS-UNAM, Mexico.)

length and/or increasing angle resolution often resolves the problem, because the results of computations are rounded to the nearest integer (remember we work with integer coordinates).

The *inverse* of an SCC is another chain of the same length, obtained by reversing the order of the symbols and their signs. The *mirror image* of a chain is obtained by starting at the origin and reversing the signs of the symbols. Finally, we point out that the preceding discussion is directly applicable to closed curves. Curve following would start at an arbitrary point (for example, the uppermost-leftmost point of the curve) and proceed in a clockwise or counterclockwise direction, stopping when the starting point is reached. We will illustrate an use of SSCs in Example 11.6.

BOUNDARY APPROXIMATIONS USING MINIMUM-PERIMETER POLYGONS

For an open curve, the number of segments of an exact polygonal approximation is equal to the number of points minus 1.

A digital boundary can be approximated with arbitrary accuracy by a polygon. For a closed curve, the approximation becomes exact when the number of segments of the polygon is equal to the number of points in the boundary, so each pair of adjacent points defines a segment of the polygon. The goal of a polygonal approximation is to capture the essence of the shape in a given boundary using the fewest possible number of segments. Generally, this problem is not trivial, and can turn into a time-consuming iterative search. However, approximation techniques of modest complexity are well suited for image-processing tasks. Among these, one of the most powerful is representing a boundary by a *minimum-perimeter polygon* (MPP), as defined in the following discussion.

Foundation

An intuitive approach for computing MPPs is to enclose a boundary [see Fig. 11.7(a)] by a set of concatenated cells, as in Fig. 11.7(b). Think of the boundary as a rubber band contained in the gray cells in Fig. 11.7(b). As it is allowed to shrink, the rubber band will be constrained by the vertices of the inner and outer walls of the region of the gray cells. Ultimately, this shrinking produces the shape of a polygon of minimum perimeter (with respect to this geometrical arrangement) that circumscribes the region enclosed by the cell strip, as in Fig. 11.7(c). Note in this figure that all the vertices of the MPP coincide with corners of either the inner or the outer wall.

The size of the cells determines the accuracy of the polygonal approximation. In the limit, if the size of each (square) cell corresponds to a pixel in the boundary, the maximum error in each cell between the boundary and the MPP approximation would be $\sqrt{2}d$, where d is the minimum possible distance between pixels (i.e., the distance between pixels established by the resolution of the original sampled boundary). This error can be reduced in half by forcing each cell in the polygonal approximation to be centered on its corresponding pixel in the original boundary. The objective is to use the largest possible cell size acceptable in a given application, thus producing MPPs with the fewest number of vertices. Our objective in this section is to formulate a procedure for finding these MPP vertices.

The cellular approach just described reduces the shape of the object enclosed by the original boundary, to the area circumscribed by the gray walls in Fig. 11.7(b).