Chapter 7

Shape Representation

After detecting the shape of an object, contour, or connected component, we should represent it in a concise and informative way. The simplest shape representation is a list of pixel coordinates. It can be easily obtained from the detection algorithm and includes precise spatial information. However, it is not ideal for further processing by higher-level algorithms for purposes such as object recognition or scene understanding. It does not characterize the shape in a useful way or indicate any shape features such as curvatures, slopes, or angles. Furthermore, this type of representation only describes a shape at a specific position, orientation, and scale. In summary, a list of coordinates describes a shape in a way that is too low-level and local to be immediately useful for high-level processing.

We know that the early stages of the human visual processing stream use simple, local features for representing visual input. As information is passed on to higher levels, representations become more and more high-level, abstract, and location invariant. Ideally, our computer vision systems should do the same. It is highly inefficient to stick with pixel-based representations throughout the processing hierarchy and then attempt to accomplish the highest-level task based on such input. The highest level has to solve the most intricate problems and needs the lower levels to prepare the visual information in such a way that its tasks become computationally feasible.

In this chapter, we will explore a variety of methods for representing shape at higher levels. Each of these techniques operates at a different level and emphasizes particular shape features. When choosing a representation for a computer vision system, we should of course always keep in mind the purpose of the system. Before starting any implementation, we should have developed a plan for all processing stages, their functionality, and the interfaces between them.
7.1 Data Structures

7.1.1 Matrices

The simplest way of representing a two-dimensional shape such as an image region is the use of a matrix. Often, this is the initial data structure for storing shape data prior to their conversion into a more appropriate representation for the given purpose. Such matrices typically have the same size as the input image so that each of its entries corresponds to exactly one of the pixels in the image. A matrix can hold information about an individual region by containing 1s for pixels included in the regions and 0s otherwise. Multiple regions in the same image can be represented by separate matrices or by using a single matrix whose entries are region labels, with each region being assigned an individual label. If, in contrast to our definition in the previous Chapter, regions can overlap, we can still represent them in a single matrix if we choose powers of 2 as region labels. Each entry then contains the sum of the labels for all regions to which the corresponding pixel belongs. For example, a pixel contained in both region 2 and region 16 is indicated by value 18 in the matrix. Using an array of 32-bit integers as the underlying data structure, we could store up to 32 overlapping regions in such a matrix.

An advantage of matrix representation is the one-to-one spatial correspondence between its entries and image pixels. Furthermore, matrices can be efficiently stored and processed in digital computers. However, matrices do not immediately provide any information about the shape or other features of objects that is needed for higher-level processing. Whenever we need to know shape characteristics such as its area, straight or round contour sections, neighboring objects, or similarity with other regions, computations involving a large number of individual entries have to be performed. Therefore, for most applications, it is useful to use additional means of representation that contain higher-level information represented in a more flexible way.

7.1.2 Adjacency graphs

An important piece of information for understanding a given scene is the adjacency of regions. The occurrence of two adjacent regions could either indicate two objects with one partially occluding the other, or two connected surfaces, among other possibilities. Adjacency is also an important concept for
algorithms such as image segmentation through split-and-merge. Furthermore, the notion of adjacency is related to that of inclusion, the situation when a region is completely included in another one. It can be caused by one object partially occluding another, larger object. Alternatively, it could indicate the smaller object being a part of the larger one. The is-part-of relationship forms a hierarchy in our description and understanding of real-world objects. For example, the engine and the wheels are parts of a car. In turn, the cylinders and pistons are parts of the engine, and the hubcap and tire are parts of each wheel.

Both adjacency and inclusion can be represented in an adjacency graph. The idea is straightforward: We create one vertex for each region in the image and connect those vertices with edges that correspond to adjacent regions. In the example in Fig. 7.1a, region A is adjacent to regions B, E, F, and I. Region B, in turn, is adjacent to regions C and D, which are also adjacent to each other. Finally, region F is adjacent to region G, which is also adjacent to region H. The corresponding graph is shown in Fig. 7.1b. While the underlying mathematical concept of a simple graph only represents adjacency, we can add inclusion information by showing included “child” regions below their including “parent.” In this layout scheme, adjacency without inclusion is indicated by horizontal connections. Fig. 7.1b uses exactly this scheme. Since regions B, E, F, and I are included in region A, their nodes are shown below node A. Similarly, nodes C and D are below B, G is below F, and finally H is below G.

![Fig. 7.1 Adjacency graph. (a) Sample image partitioned into regions A to I; (b) Corresponding adjacency graph showing adjacent regions connected by edges and included regions below the including ones.](image)

The inclusion information can be explicitly added to this graph by labeling the edges to indicate a directed parent-child relationship wherever it occurs. The
resulting data structure is useful for both low-level operations such as region merging and high-level processes for understanding the hierarchical composition of an image.

### 7.1.3 Image pyramids

Some computer vision problems require the analysis of an image at different scales. For instance, a global analysis of the image could reveal the overall context or scene gist, which can guide the detection of specific objects in a more fine-grained examination of the scene. Another example is a face recognition system that first detects faces at the global scale, then localized face features such as eyes and mouth at an intermediate scale, and finally analyzes the individual face characteristics at a high resolution. In order to perform the global analyses efficiently, it should not be based on the individual pixels of the input image. Each computation of this kind could require the traversal of millions of pixels.

In these cases, an image pyramid may be the most suitable data structure to hold the image data. Such a pyramid works best for square images whose vertical and horizontal size is a power of two. If the input image has a different size, we can pad it with additional rows and columns until it reaches the required $2^k \times 2^k$ pixels for an integer $k$. The image itself then forms the base of the pyramid, also referred to as level 0. The next higher level, level 1, consists of a downsampled version of the input of size $2^k \times 2^{k-1}$, that is half the height and width of the input image. Level 2 contains a downsampled version of level 1 of size $2^{k-1} \times 2^{k-2}$, and so on, until we reach level $k$, which is of size $2^0 \times 2^0$, i.e., consists of a single pixel. The final data structure, illustrated in Fig. 7.2, thus resembles a pyramid. Any algorithm could be used for performing the downscaling. The simplest approach for grayscale images is to compute the average intensity for each $2 \times 2$ patch of pixels to determine the intensity of the pixel associated with it in the level above. Another possibility is the use of resampling algorithms discussed in Chapter 2.
Regardless of the downsampling method, there is a clear spatial correspondence between each pixel at level 1 and above and a group of 2×2 pixels at the level immediately below it. This arrangement enables efficient transitions across levels for multi-resolution algorithms. Because the size of each level is only one quarter of that below it, pyramids do not require substantially more memory than their input images alone. For example, the input image at level 0 in Fig. 7.2 needs 16×16 = 256 bytes of memory, whereas the entire pyramid requires 256 + 64 + 16 + 4 + 1 = 341 bytes. The memory demands thus only increase by approximately one third, a proportion that remains unchanged even for very large images. This can be demonstrated using the equation for the sum of an infinite geometric series, which converges for common ratios $r$ with $-1 < r < 1$:

$$a + ar + ar^2 + ar^3 + ... = \frac{a}{1-r}$$ (7.1)

In our case, $a = 1$ and $r = \frac{1}{4}$, which gives us:
\[ 1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \ldots = \frac{1}{1-\frac{1}{4}} = \frac{4}{3} \quad (7.2) \]

In other words, even for very large images, the proportion of extra memory required for turning them into pyramids never exceeds one third. However, as noted above, some input images may need padding with additional rows and columns, which further increases memory demand.

### 7.1.4 Quad trees

While image pyramids enable multi-resolution representation of images with only little memory overhead, it is even possible to save memory by storing images in a hierarchical data structure. The most common structure of this kind is called a *quad tree*, which is typically used with binary images, such as the example in Fig. 7.3a. The input image is represented by the root node of the quad tree (Fig. 7.3b). We then divide the image into its quadrants, each of which is represented by a child node of the root, in the order shown in Fig. 7.3c. If a quadrant contains only black or only white pixels, its node is marked as “black” or “white,” respectively, and becomes a leaf of the tree, i.e., has no children. Otherwise, we mark the node as “gray” and repeat the process recursively: The area represented by the node is split into quadrants, and the node receives four children, each representing one of the quadrants (Fig. 7.3d). The algorithm continues until the entire image is represented by the quad tree, which sometimes requires division all the way to the level of individual pixels (Fig. 7.3e).
Fig. 7.3 Quad trees. (a-e) Constructing a quad tree for a contiguous object. The resulting tree is an efficient representation of the input image; (f-g) The same process for an image with multiple connected components. Here, the quad tree representation is less efficient and requires more nodes than there are pixels in the image.

Fig. 7.3e illustrates the amount of data compression that can be achieved. The resulting quad tree represents the 64-pixel image using only 17 nodes. However, this amount of data compression can only be achieved for large connected components with little noise in the image. As shown in Figs. 7.3f and g, in the absence of large black and white areas in the input, we may actually need more nodes (77) than pixels (64). Similar to image pyramids (Eq. 7.2), in the worst case we require about one third more nodes than there are pixels in the image.

Quad trees can also represent images that contain multiple objects, with the pixels belonging to each object having the same index value. In that case, we recursively split the image until all pixels in a given square region have the same value, which is then used to label the corresponding node. Regardless of the number of objects represented by a quad tree, their features such as area, center of gravity, or shape can be efficiently analyzed. Starting from the root, we can traverse those nodes whose labels indicate pixels of the object to be analyzed. For large objects, this process is much more efficient than the traversal of individual pixels in the original image.

7.2 Scalar Shape Descriptors

7.2.1 Moments

We can interpret a grayscale image as a 2D probability density function describing the shape and position of an object. In the case of a binary image, these shape and position are precisely defined. If there is some uncertainty about these characteristics, then we can use a grayscale image with each pixel value being proportional to the estimated probability that it belongs to the object. In both cases, we can use moments to describe the statistical properties of the object. There is a variety of 2D moments that can be computed, distinguished by their parameters $p$ and $q$ that define the treatment of the vertical and horizontal pixel coordinates, respectively. For an $m \times n$ image $A$, the moment $m_{pq}$ is defined as:
\[ m_{pq} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} i^p j^q A[i, j] \] (7.3)

We also refer to the sum \((p + q)\) as the order of the moment \(m_{pq}\). Clearly, for a binary image, the zeroth-order moment \(m_{00}\) is simply the object’s area, because it sums all pixel values in the image. The first-order moments \(m_{10}\) and \(m_{01}\) describe the sum of the vertical and horizontal object pixel coordinates, respectively. Therefore, the object’s center of gravity \([i_c, j_c]\) can be computed as:

\[ i_c = \frac{m_{01}}{m_{00}}, \quad j_c = \frac{m_{01}}{m_{00}} \] (7.4)

Based on this center, we can define moments in a translation-invariant manner. The resulting central moments \(\mu_{pq}\) are calculated as follows:

\[ \mu_{pq} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (i - i_c)^p (j - j_c)^q A[i, j] \] (7.5)

If the object is elongated, we can use the central moments to compute the orientation \(\alpha\) of the object’s main axis, i.e., the axis of minimal squared distance to all object pixels:

\[ \alpha = \frac{1}{2} \arctan \left( \frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right) \] (7.6)

While the central moments \(\mu_{pq}\) could be used as shape descriptors, it should be noted that they are neither scale nor orientation invariant. We can derive scale invariant moments \(\nu_{pq}\) by dividing the central moments \(\mu_{pq}\) by an appropriately normalized value of \(\mu_{00}\):

\[ \nu_{pq} = \frac{\mu_{pq}}{\mu_{00}^{p+q+1}} \] (7.7)

From these scale invariant moments, numerous orientation invariant features can be computed. In a continuous 2D space, these features are unaffected by object location, size, or orientation. The discrete space of digital images can introduce small deviations, which are typically negligible for large objects without visibly pixelated contours. Here, we will only list the four features \(\phi_i\) to \(\phi_2\) of this kind, which can be computed most efficiently:
Chapter 7: Shape Representation

\[ \varphi_1 = v_{20} + v_{02} \quad (7.8) \]

\[ \varphi_2 = (v_{20} - v_{02})^2 + 4v_{11}^2 \quad (7.9) \]

\[ \varphi_3 = (v_{30} - 3v_{12})^2 + (3v_{21} - v_{03})^2 \quad (7.10) \]

\[ \varphi_4 = (v_{30} + v_{12})^2 + (v_{21} + v_{03})^2 \quad (7.11) \]

The shape characteristics described by these features are rather abstract and non-intuitive. Nevertheless, they can serve as robust shape descriptors. They yield similar values for objects that resemble each other in their large-scale shape, and in most cases produce substantially different values for objects with clearly mismatched shape.

### 7.2.2 Bending energy

One fundamental characteristic that distinguishes object shapes is whether their contour mainly consists of straight or curved elements. In order to quantify this characteristic, we can compute the bending energy that would be required to completely straighten the contour of an object. To compute curvature in a discrete image, we can look at each contour pixel and its two contour neighbors. The sequence of contour pixels can be determined using a boundary-following algorithm that will be discussed in Section 7.3.1. Since a pixel can have neighbors in only eight different directions, only few curvature values are possible. The scheme in Fig. 7.4a illustrates the computation of absolute curvature values for a contour pixel \( p \) shown in the center of the 3×3 grid. If the preceding contour pixel is located to the left of \( p \), then the numbers indicate the curvature values for each possible position of the contour pixel following \( p \). For example, if the next pixel is to the right of \( p \), then the local contour near \( p \) is a straight, horizontal line, and the curvature is 0. Turns of 45°, 90°, 135°, and 180° correspond to curvature values 1, 2, 3, and 4. Extreme turns of 180° are possible when a part of the object is as thin as one pixel so that each of its pixels is traversed twice when following the contour around the object once.
Fig. 7.4 Computation of curvature $\kappa$, bending energy $b$, and compactness $c$. (a) Calculation of absolute curvature values. When following the contour and approaching the central pixel from the left, the position of the next contour pixel determines the curvature value; (b, c) Bending energy and compactness computation for two different shapes. All contour pixels are marked by their curvature values.

Bending energy $b$ is then defined as the mean squared curvature $\kappa[p]$ across all contour pixels $p = 1..P$:

$$b = \frac{1}{P} \sum_{p=1}^{P} \kappa^2[p] \quad (7.12)$$

As expected, $b$ is much smaller for an object with mostly straight contour elements and soft corners (Fig. 7.4b) than for a jagged contour (Fig. 7.4c). Bending energy is a useful contour descriptor, but the way we defined it, it is not scale invariant, and its orientation invariance is limited to rotations of $90^\circ$, $180^\circ$, and $270^\circ$. For objects larger than those in Fig. 7.4, we can make $b$ nearly invariant to both scale and orientation by modifying the definition of curvature. There are numerous techniques for estimating continuous curvature descriptors. They consider not only the immediate contour neighbors of $p$ but an entire contour neighborhood, often with greater weight given to points closer to $p$.

### 7.2.3 Compactness

Another commonly measured object characteristic is its compactness $c$. It is defined for a given perimeter length $l$ and object area $a$ as follows:
Chapter 7: Shape Representation

\[ c = \frac{l^2}{a} \]  

(7.13)

This definition is a bit counterintuitive, because the more compact an object is, i.e., the more area is wrapped by a perimeter of the same length, the *smaller* is \( c \). For example, in a continuous 2D space, a square with edges of length \( s \) has a perimeter \( l = 4s \) and an area \( a = s^2 \). Thus, its compactness is given by

\[ c_{\text{square}} = \frac{(4s)^2}{s^2} = \frac{16s^2}{s^2} = 16, \]  

(7.14)

regardless of its size. A rectangle with a height of \( s \) and a width of \( \lambda s \) has a perimeter \( l = (2 + 2\lambda)s \) and an area \( a = \lambda s^2 \), which gives us

\[ c_{\text{rect}}(\lambda) = \frac{[(2 + 2\lambda)s]^2}{\lambda s^2} = \frac{(4 + 8\lambda + 4\lambda^2)s^2}{\lambda s^2} = \frac{4}{\lambda} + 8 + 4\lambda \]  

(7.15)

Eq. 7.15 shows that \( c \) is minimal for a square (\( \lambda = 1 \)) and increases with greater elongation of the rectangle, that is, greater deviation of \( \lambda \) from 1. For example, a rectangle that is twice as tall as it is wide (\( \lambda = 0.5 \) or \( \lambda = 2 \)) has a compactness of 18, and for a factor of 4 (\( \lambda = 0.25 \) or \( \lambda = 4 \)), the compactness is 25. Intuitively, we expect that the most compact 2D shape is a circle. A circle with radius \( r \) has a perimeter \( l = 2\pi r \) and an area \( a = \pi r^2 \), and thus we get:

\[ c_{\text{circle}} = \frac{(2\pi r)^2}{\pi r^2} = \frac{4\pi^2 r^2}{\pi r^2} = 4\pi \approx 12.6 \]  

(7.16)

The circle is indeed the most compact shape, which means that there is no 2D shape with \( c < 4\pi \). In order to compute \( c \) in digital images, \( l \) can be measured as the number of contour pixels as determined by the boundary-following algorithm, and \( a \) is the number of object pixels, which is identical to \( \mu_{00} \). For the two objects in Fig. 7.4, it is not surprising that the near-square shape in panel (b) is found to be more compact than the jagged one in panel (c).

Similar to bending energy, compactness is not invariant to scale or orientation other than rotations of 90°, 180°, and 270°. However, for large objects we can make our compactness measure nearly independent of scale and orientation if we slightly modify the way in which we calculate the perimeter length \( l \). To do this, we add a length of 1 for each vertical or horizontal step between contour pixels and \( \sqrt{2} \) for diagonal steps. To further improve the compactness measure, we
could fit straight line and curve segments to the contour as discussed in Chapter 6 and compute \( l \) as the sum of the segment lengths.

### 7.3 Contour Representation by One-Dimensional Functions

#### 7.3.1 A boundary-following algorithm

The previous section already showed the need for a consistent way of determining and ordering the pixels that belong to a contour. As we will see, in both this and the following section this issue will become even more crucial. For this reason, we will now introduce a very useful algorithm that computes a well-defined solution for any given 2D shape represented by a connected component consisting of at least 2 pixels. It is aptly named the *boundary-following algorithm*. Its output is a list of the coordinates of all contour pixels, starting with the leftmost pixel in the top row of the object. The order of pixels follows a counterclockwise trip along the object’s entire contour that terminates right before visiting the starting point once again. The choice of counterclockwise direction is completely arbitrary, and many other textbooks use clockwise direction instead. In this book, we will consistently use counterclockwise contour descriptions, which is the direction of increasing angles in a 2D coordinate system. The algorithm can be described as follows:

**Algorithm 7.1 Boundary-following algorithm**

1. Determine the initial boundary pixel \( p \) of the object by systematically scanning the image from top to bottom, and within each row from left to right.
2. Let \( q \) be the background pixel to the left of \( p \). Set the step index \( s = 0 \).
3. Store the coordinates of \( p \) in pixel list element \( L[s] \). Increment \( s \).
4. Let the 8-neighbors of \( p \), starting with \( q \) in counterclockwise order, be \( n_1, n_2, ..., n_8 \). Find the least \( j \) so that \( n_j \) is an object pixel.
5. Set \( p = n_j \) and \( q = n_{j+1} \).
6. Repeat steps 3 to 5 until \( L[s] = L[0] \) and \( p = L[1] \).
7. The list \( L[0], ..., L[s-1] \) contains the ordered list of boundary pixels.
Note that some descriptions of this algorithm in the literature only require $L[0] = L[i]$ as the termination criterion in step 6. However, such a criterion is insufficient and can result in incomplete pixel lists if parts of the object are only one pixel wide. Fig. 7.5 illustrates the progression of the algorithm on a small object. The final panel (i) shows the sequence of boundary pixels as determined by the algorithm.

Fig. 7.5 Illustration of the boundary-following algorithm. (a-h) The algorithm proceeds by finding the first object pixel neighbor of the current pixel $p$ in a counterclockwise sequence whose starting point $q$ depends on the previous iteration of the algorithm; (i) Numbers indicating the sequence of boundary pixels determined by the algorithm. As pixels 10 and 11 are identical to pixels 0 and 1, respectively, the
algorithm terminates.

For the discussions in the following sections, it is beneficial to get the output of the boundary-following algorithm into a specific form. The first step is to translate the pixel list \( L \) into two one-dimensional functions \( v[s] \) and \( h[s] \) describing the vertical and horizontal coordinates along the contour. The input to both functions is the same parameter \( s \). We define \( s \) as the distance from starting point \( p \) when traveling along the contour in counterclockwise direction. By varying \( s \), we can obtain any point on the contour.

As noted above, for a more accurate representation of \( s \) we need to consider that vertical and horizontal transitions between neighboring pixels cover a Euclidean distance of 1, whereas for diagonal transitions the distance is \( \sqrt{2} \). We can easily modify \( v[s] \) and \( h[s] \) to account for this issue. However, it is more practical for our purposes to have \( v[s] \) and \( h[s] \) be discrete functions, i.e., defined only for integer values of \( s \). This allows us to efficiently store them in arrays and use them as input to some algorithms that require it in discrete form.

To discretize the functions, we can use linear interpolation. Let us assume that the sequence \( \{s_0, s_1, \ldots, s_{n-1}\} \) contains all distances from \( p \) that are associated with contour pixel coordinates, i.e, for any \( s_i \) with \( 0 \leq i < (n-1) \) either \( s_{i+1} - s_i = 1 \) or \( s_{i+1} - s_i = \sqrt{2} \). In order to determine the value of \( v[d] \) for an integer \( d \), we first find \( j \) such that \( s_j \leq d \leq s_{j+1} \). We can then use linear interpolation to determine \( v[d] \) as follows:

\[
v[d] = \frac{(s_{j+1} - d)v[s_j] + (d - s_j)v[s_{j+1}]}{s_{j+1} - s_j}
\]  

(7.17)

Obviously, Eq. 7.17 can be modified to compute \( h[d] \) as well. Let us take a look at an example object to illustrate the process of deriving discrete functions \( v \) and \( h \) from it. Fig. 7.6a shows a binary image depicting the silhouette of a car. The result of applying the boundary-following algorithm to this mage is illustrated in Fig. 7.6b. The algorithm determines \( p \) as its starting point and then follows the shape boundary in counterclockwise direction until it returns to \( p \).
Fig. 7.6 (a) Object shape represented by binary image; (b) Result of applying the boundary-following algorithm to the object. The starting point is at position $p$, and the arrow indicates the (counterclockwise) direction of the algorithm’s path.

The algorithm determines that, using Euclidean distances as discussed above, the contour is 5092 pixel units long. Using linear interpolation, functions $v[s]$ and $h[s]$ are generated in such a way that they are defined for all integers $s$ from 0 to 5091. Fig. 7.7 shows both functions in a combined plot.

![Diagram](image-url)

**Fig. 7.7** Description of the object contour in Fig. 7.6b using separate one-dimensional functions $v[s]$ and $h[s]$ for the vertical and horizontal coordinates. The horizontal axis shows the distance $s$ from the starting point $p$ in counterclockwise direction.

It should be mentioned that by scanning the inside of objects and switching object and background in Algorithm 7.1, we can also detect and describe the boundaries of holes in objects in the same way as the outer boundary of objects. Generally speaking, the functions $v[s]$ and $h[s]$ constitute a more useful object description for higher-level processing than does the raw pixel matrix. Boundary descriptions are more compact without loss of information and make important information about an object’s shape more easily accessible. The methods presented in the following subsections will demonstrate the usefulness of
standardized boundary information and how it can be transformed to immediately serve higher-level functions such as object recognition.

### 7.3.2 Slope representation

While the description of object shape with the help of the boundary-following algorithm is definitely a step upward from lower- to higher-level representation, it is not in a useful shape for object recognition purposes yet. As shown in Fig. 7.7, the output of functions \( v[s] \) and \( h[s] \) indicates specific pixel positions and distances and is completely orientation dependent. If two objects of the same shape were shown in different positions, sizes, or orientations, their functions would look very different. Especially differences in orientation would affect both \( v[s] \) and \( h[s] \) in complex ways. These are undesirable features of shape descriptors for most applications.

To derive a more suitable shape description, instead of giving positional information as a function of \( s \), we provide the slope of the contour as a function of \( s \). Here, slope refers to the orientation of the tangent vector, i.e., the direction in which we are moving at a given point while following the contour. We are still doing this for one complete traversal of the contour in counterclockwise direction. The resulting slope function \( \psi[s] \) is given by

\[
\psi[s] = \arctan\left(\frac{h'[s]}{v'[s]}\right)
\]

Note that, as usual, the \texttt{atan2} function in C should be used in order to obtain the full \( 360^\circ \) range of orientations and avoid a division by zero. If, for computing \( v'[s] \) and \( h'[s] \), we only consider the neighboring values \( v[s-1], h[s-1], v[s+1], \) and \( h[s+1] \), only few orientation values are possible. This dependency on individual data points would make the function \( \psi[s] \) change considerably with the size and orientation of the shape. One simple solution is to smooth the functions \( v[s] \) and \( h[s] \) with a one-dimensional Gaussian convolution filter, whose optimal size depends on the size and curvature characteristics of the contour. After smoothing, we can compute \( v'[s] = v[s+1] - v[s-1] \) and \( h'[s] = h[s+1] - h[s-1] \) and obtain good results. There are various, more complex methods aimed at finding the best tangent orientation estimate for digital curves. Nevertheless, Fig. 7.8, showing the slope of the car contour from Fig. 7.6 as a function of \( s \), demonstrates that our simple method performs quite well. In this type of data plot, horizontal lines indicate straight lines in the input image. While
the slope values for straight lines that are neither vertical nor horizontal show some noise, the maximum deviations are small. Importantly, the wheels of the car, represented by perfect but pixelated semicircles, generate straight lines in the plot (starting at around $s = 1150$ and $s = 2300$), indicating a constant amount of slope change per length unit during the transition from $0^\circ$ to $180^\circ$. We should be aware, however, that stronger smoothing leads to softer corners and a deterioration of the shape. Here, the functions were smoothed using an 11-element Gaussian filter with standard deviation $\sigma = 2.7$, which is small for a contour that is 5092 pixel units long.

![Diagram](image)

**Fig. 7.8** Slope representation of the car contour from Fig. 7.6. The slope $\psi$ is shown as a function of distance $s$ from the starting point $p$. This type of diagram is also known as a $\psi$-$s$ plot.

Diagrams of the type shown in Fig. 7.8 are also referred to as $\psi$-$s$-plots. They can be used to represent and compare the shapes of different objects. When comparing two such plots, we should remind ourselves that $\psi$ is a clock variable, and thus values near $0^\circ$ and $360^\circ$, shown as most distant in Fig. 7.8, are actually very close to each other. This fact explains the extreme slope variations in the plot for the vertical contour parts before $s = 1000$. Therefore, when computing the difference $\delta$ between two slopes $\psi_1$ and $\psi_2$, we should pick the minimum angular distance, i.e.:
A positive characteristic of the slope representation is that it is unambiguous, that is, given the representation, we can always correctly reconstruct the shape that it represents. Obviously, the slope representation is invariant to absolute position. Furthermore, if we scale $s$ so that its maximum value is 1, then the slope representation also becomes size invariant. While it is not orientation invariant, a change in shape orientation will only lead to a consistent, vertical shift of the entire plot. Unfortunately, rotating a shape may also result in the choice of a different starting point $p$, which induces a consistent, horizontal shift of the entire plot. Orientation invariance can be achieved more easily with a related concept, called the shape’s **signature**.

### 7.3.3 Signatures

In order to compute the signature of an object, we prepare the contour information in exactly the same way as for the slope representation. This time, however, we do not determine the direction of the tangent at the starting point $p$ but the direction of the normal, which is perpendicular to the tangent. We now draw a straight line through $p$ in the same orientation as the normal until we hit another point on the contour. The shortest distance $d$ along this line between $p$ and any other point on the contour is the measure that we are interested in (Fig. 7.9). We follow the contour, as usual in counterclockwise direction, and determine $d$ for every point on the contour, until we are back at point $p$. The resulting function $d[s]$ is the signature of the object.

\[
\delta(\psi_1, \psi_2) = \min(\left| \psi_1 - \psi_2 \right|, 360^\circ - \left| \psi_1 - \psi_2 \right|) \tag{7.19}
\]
determined and then the contour point \( c \) that is closest to \( p \) when measured in the direction of the normal. Shifting \( p \) along the contour for a full cycle and plotting the Euclidean distance \( d \) between \( p \) and \( c \) as a function of the distance \( s \) covered by \( p \) yields the signature of the contour.

For describing a closed contour, we can arbitrarily choose a starting point \( p \) and define \( s \) as the distance from \( p \) when traveling along the contour in counterclockwise direction. Since the signature for complex shapes can be difficult to interpret, let us examine this concept using some simple, geometric objects as shown in Fig. 7.10. One problem with such ideal shapes is that their tangent vector is undefined at their corners. However, since this only concerns a few infinitesimal points on the contour, let us be practical and accept the resulting gaps in the signature. For an equilateral triangle (upper left panel), moving along each of the sides creates a signature that resembles the triangle itself. As a consequence, the complete signature looks like three copies of the triangle side by side.

Unfortunately, the correspondence between a shape and its signature is not always as clear as it is for the triangle. When recording the signature for the polygon in the upper right panel of Fig. 7.10, we have to perform many changes...
in direction in which we measure $d$, and therefore, the result looks rather intricate, and it is not obvious from its appearance what shape it represents. Unfortunately, it is not generally possible to infer a shape from its signature, because multiple shapes can have the same signature. This fact is illustrated by the examples of circle and square (lower left and right panels, respectively). For these shapes, $d$ does not vary at all, leading to a horizontal line as their signature.

On the positive side, after normalizing the range of $s$, signatures are invariant to changes in position, size, or orientation. Only the choice of starting point $p$ still causes variation in the signature in the form of horizontal shifts of the entire function graph. This problem can be avoided by using histograms instead of functions of $s$ to describe shapes, as demonstrated below.

### 7.3.4 Slope and curvature histograms

When generating a slope histogram of a given shape, we still travel along its boundary in counterclockwise direction for one complete cycle as we did before. Again, we first compute the orientation of the tangent vector at each point. Afterwards, we create a histogram of these orientations that indicates how frequently each orientation occurred. To obtain best results, we should choose a size for the orientation bins that provides sufficient angular resolution and at the same time contains enough samples per bin. Typical bin widths range from $1^\circ$ to $10^\circ$.

As with the signatures, it is most instructive to examine slope histograms using geometric shapes in a continuous 2D space. Fig. 7.11 shows four examples of this kind. For a square (upper left panel), the slope histogram reveals identical peaks at $0^\circ$, $90^\circ$, $180^\circ$, and $270^\circ$ and is zero otherwise. Note that the peak shown for $360^\circ$ is of course the same as the one for $0^\circ$. If we rotate the square by $45^\circ$ and thereby turn it into a diamond (upper right panel), then there are still four peaks, but they are shifted by $45^\circ$ as compared to the square. Clearly, rotating the shape by a certain angle causes a horizontal shift of the entire slope histogram by exactly that angle.
Fig. 7.11  Slope histograms for four different geometric shapes. The vertical axes have no particular scale, because the histograms shown here only have qualitative character to illustrate the basic concept.

If the shape to be described is a circle (lower left panel), the histogram shows the same value for each angle, because all tangent orientations occur equally often. An ellipse (lower right panel) also contains all orientations, and thus the slope histogram does not include any frequencies of zero. However, as the given ellipse is taller than wide, the angles near 0° and 180° occur more often than those near 90° and 270°, leading to a wave-shaped histogram. It is easy to see that if we rotate the ellipse, the histogram will once again shift horizontally. In fact, for any shape, rotating it will have this effect.

We can exploit this fact for efficient shape matching. For two given shapes, we can compute their slope histograms and then compute the one-dimensional correlation of these histograms (see Chapter 3). The maximum value in the correlation vector indicates how similar the two shapes are, and the position of the maximum within the vector tells us the relative orientation of the shapes, i.e., by what angle we have to rotate one of them to match it with the other. By performing this correlation, shape matching based on slope histograms becomes invariant to position, size, and orientation of the objects, as well as the starting point of the contour description. However, different objects can have the same slope histogram. For example, if we cut a square into small segments and put them back together in any way we like, without changing their orientation, the new shape will have exactly the same signature as the square.

If we are not interested in the orientation of shapes and just want to compute their similarity as efficiently as possible, we can consider using curvature histograms. To do this, instead of picking slope values along the contour to build the histograms, we use curvature values. They can be computed as the angular difference between two consecutive slope values and should be normalized to
indicate the change in orientation for a given proportion of contour length. Curvature values are negative for right turns, positive for left turns, and zero for straight line segments. As the curvature does not depend on absolute orientations, curvature histograms do not vary with the position, size, or orientation of shapes and can be compared immediately to measure shape similarity. A drawback of this method is that the measurement of curvature is more strongly affected by noise than the measurement of orientation. Furthermore, for shapes mainly consisting of straight lines, almost all curvature values are near zero. A weighting scheme for such data needs to be employed in order to devise a useful matching algorithm.

7.4 Fourier Descriptors of Object Boundaries

7.4.1 Transforming the coordinates of contour pixels

Let us consider once again the car contour shown in Fig. 7.6 and its description by the two functions \( v[s] \) and \( h[s] \) in Fig. 7.7. In this function plot, \( s \) is shown in the range from 0 to 5092, i.e., for one complete trip along the contour from \( p \) back to \( p \). What happens if we let \( s \) exceed the value of 5092? Since we have a closed contour, further increasing \( s \) will take us on another trip around the contour, and then another, and so on. In other words, both \( v \) and \( h \) are periodic functions with a period of 5092 pixels in this specific case. This fact gives us an idea – we could use the Fourier transform to derive a frequency domain representation of each function. Based on our previous experience with the frequency domain, it is likely that we can discard very high frequencies, which would lead to a more compact description. Moreover, it is possible that we can develop a contour descriptor in the frequency domain that is invariant to spatial translation, rotation, and scale and would thus lend itself to the tasks of contour recognition and classification.

Clearly, we have to apply a one-dimensional Fourier transform to each of the functions \( v \) and \( h \). Since we made sure that these functions take integers as inputs, we can use the discrete Fourier transform with which we are already familiar (see Chapter 4). Once again, we will indicate one-dimensional frequency using the letter \( l \), and \( n \) will indicate the period of our functions. In the present example, \( n = 5092 \). Then function \( v[s] \) can be Fourier transformed into the frequency domain function \( \hat{v}[l] \) in a way analogous to Eq. 4.25 using the complex notation:
Chapter 7: Shape Representation

\[ \hat{v}[l] = \frac{1}{n} \sum_{s=0}^{n-1} v[s] \cdot e^{-2\pi \frac{ls}{n}} \quad \text{for } l = 0..(n-1) \quad (7.20) \]

Obviously, function \( h[s] \) can be transformed into the frequency domain function \( \hat{h}[l] \) in the same way. Given a frequency domain function \( \hat{v}[l] \), we can obtain the corresponding spatial domain function \( v[s] \) via the inverse discrete Fourier transform as follows:

\[ v[s] = \sum_{l=0}^{n-1} \hat{v}[l] \cdot e^{2\pi \frac{ls}{n}} \quad \text{for } s = 0..(n-1) \quad (7.21) \]

Again, the same applies to the horizontal coordinates. We are now able to translate a given contour into its frequency domain representation and transform any such representation back into its spatial form to, for example, draw the contour. It is not immediately clear, however, how we can attain the benefits of the frequency domain descriptions as hypothesized above. This will require some further exploration of the frequency domain representation, which will be undertaken in the following section.

7.4.2 Characteristics of contour descriptors in the frequency domain

As a first step, let us investigate to what extent the frequency domain representation can provide more concise contour descriptions. In Chapter 4 it was discussed that frequencies above \( n/2 \) actually mirror the lower frequencies and are thus redundant. However, it needs to be noted that the frequency domain functions are complex, with real and imaginary components that indicate the contributions of cosine and sine basis functions, respectively, to the spatial contour coordinates. Therefore, even when discarding the frequencies above \( n/2 \), the size of the frequency domain descriptors is still \( n \) scalars per dimension (vertical and horizontal). Based on our experience with the frequency domain of images, we could imagine that we could even remove the highest frequencies below \( n/2 \) without visible loss of information.

To test this hypothesis, we will take the contour from Fig. 7.6 and Fourier transform it as described in Eq. 7.20. We will then delete all frequencies above a maximum frequency \( l_{\text{max}} \) and transform the shortened descriptor back into the spatial domain using Eq. 7.21. For \( l_{\text{max}} = n/2 \), the result will precisely match the original contour. According to our hypothesis, lowering \( l_{\text{max}} \) even further should not immediately deteriorate the result. Obviously, at some point the reduction in
information should visibly affect the back-transformed contour. Testing different values of $l_{\text{max}}$ will thus help us estimate the extent to which the frequency domain allows for more concise contour descriptions than the spatial domain.

Fig. 7.12a shows the result of this test. Even at high resolution, $l_{\text{max}} = 200$ does not reveal any deterioration. For $l_{\text{max}} = 100$, the contours start showing some low-amplitude, high-frequency waves that are only visible at high resolution. It takes until $l_{\text{max}} = 25$ for some important parts of the contour to be visibly distorted, such as the lights of the car. Even at $l_{\text{max}} = 6$ we could still guess that the contour might belong to a car. Given that the spatial domain description includes $2 \cdot 5092 = 10184$ scalars, the data compression achieved by the present technique is significant. The description with $l_{\text{max}} = 100$ is sufficient for almost all purposes. It uses 4 scalars per frequency (cosine and sine contributions for the vertical and horizontal coordinates) so that its total description length is 400 scalars. This is less than 4% of the original description length.
Fig. 7.12 Results of reducing the description length of contours in the frequency domain by limiting the range of frequencies to $l_{\text{max}}$. Each panel shows the outcome of discarding the frequencies above a specific value of $l_{\text{max}}$ and transforming the contour back into the spatial domain. (a) Simple “car” contour based on 5092 pixels; (b) Complex “ice crystal” contour based on 23149 pixels.

It is conceivable, though, that the conciseness of the frequency domain representation depends on the complexity of the contour. The car contour is simple with only few details. As another example, let us examine the extremely complex contour of a stylized ice crystal as shown in Fig. 7.12b. The spatial description of its contour uses 23149 coordinate pairs. This value clearly exceeds that for the car contour, even though the diameters of the two contours are similar.

Setting $l_{\text{max}} = 1000$ does not cause any noticeable loss of fidelity regarding the crystal contour (see Fig. 7.12b). However, at $l_{\text{max}} = 300$, the most fine-grained structures of the crystal are clearly eroding. For $l_{\text{max}} = 100$, the entire level of finest detail has disappeared. The next level of detail is eliminated at $l_{\text{max}} = 25$, and while the six-pronged basic shape remains visible until $l_{\text{max}} = 6$, all other characteristic information disappears. The level of accuracy that we considered to be sufficient for the car contour at $l_{\text{max}} = 100$ roughly corresponds to $l_{\text{max}} = 700$ in the crystal contour. The latter representation uses $4 \times 700 = 2800$ scalars, reduced from the original spatial description based on $2 \times 23149 = 46298$ scalars. In other words, the data of the crystal were compressed to approximately 6% of their original description length without significant loss of fidelity. While the compression rates between the car and crystal contours are similar, the absolute description length of the crystal contour is seven times as long as that for the car contour, reflecting the greater complexity and level of detail of the crystal contour.

The experiment above shows that the Fourier transform leads to more concise contour descriptions. Furthermore, these descriptions reveal the relative contributions of structures from large scales (low frequencies) to small scales (high frequencies) to the contour. What still needs to be done is to investigate the hypothesized benefits of the frequency domain representation with regard to its invariance to translation, rotation, and scale.

Translation invariance can easily be achieved. As we remember from Chapter 4, the average value of a function in the spatial domain is identical to its cosine
contribution for frequency zero. Since the sine contribution at frequency zero is meaningless, we can simply remove frequency zero from our contour descriptors. The resulting contour representation always places the center of gravity of the spatial contour points at \([0, 0]\), regardless of where it was located in the original contour.

Invariance to rotation can be accomplished based on the basic idea that the absolute contribution of individual frequencies to a given contour (i.e., its magnitude spectrum) should not change if we rotate that contour. Such a rotation should mainly alter the phase spectrum instead. However, the magnitudes in the vertical and horizontal dimensions can be affected differently. Imagine a contour of a tall, thin object such as a lamppost. Its Fourier description must contain larger magnitudes in the vertical than in the horizontal direction. If we rotate this contour by 90 degrees, the situation changes – now the horizontal magnitudes are larger than the vertical ones.

Intuitively, it seems that the two-dimensional magnitudes, composed of those in the vertical and horizontal dimensions, should be invariant to rotation. This is indeed correct, and we can use the familiar Euclidean distance to compute such two-dimensional magnitudes. Noting that the absolute value of a complex Fourier descriptor for frequency \(l\) equals its magnitude, the two-dimensional magnitude \(\hat{m}[l]\) is given by:

\[
\hat{m}[l] = \sqrt{\left|\hat{v}[l]\right|^2 + \left|\hat{h}[l]\right|^2} \tag{7.22}
\]

While \(\hat{m}[l]\) is invariant to rotation of the contour, it still depends on its scale. Since the Fourier transform is a linear function, all values of \(\hat{m}[l]\) change proportionally with the scale of the contour. Thus, we can compute the relative 2D magnitude \(\hat{m}_r[l]\) by linearly scaling \(\hat{m}[l]\) such that its sum across all considered frequencies is 1:

\[
\hat{m}_r[l] = \frac{\hat{m}[l]}{\sum_{f=1}^{l_{\text{max}}} \hat{m}[f]} \tag{7.23}
\]

Fig. 7.13 shows the relative 2D magnitudes for the car and crystal contours for \(l_{\text{max}} = 15\). The 15 values clearly differ between the two contours. Whereas the car contour descriptor is dominated by low frequencies, the crystal contour also involves significant contributions of specific higher frequencies (5, 7, and 11). These elevated frequencies likely correspond to the distinct levels of structures in
Chapter 7: Shape Representation

the contour. Importantly, as shown in the figure, the descriptors do not change at all when each of the contours is rotated, translated, and scaled by random parameters. This result illustrates that the relative 2D magnitudes are invariant to such transformations and are thus suitable contour descriptors for recognition and classification purposes. It should be noted, however, that different contours can have identical relative 2D magnitudes.

Fig. 7.13 Contours and corresponding frequency-domain descriptors based on the relative 2D magnitude for the 15 lowest frequencies. The descriptors differ across the car (top panels) and ice crystal contours (bottom panels) but are invariant to
the translation, rotation, and scale of each (left vs. right panels).

In summary, as compared to the spatial descriptors discussed above, using Fourier descriptors for contours typically leads to more concise representations and are able to distinguish between contour features at different spatial scales. Furthermore, Fourier descriptors can be made invariant to translation, rotation, and scaling, which is desirable for a variety of applications. This approach is particularly recommended for closed contours that cannot easily be described by polygons or circular arcs.

7.5 Region Skeletons

7.5.1 The medial axis

As we found out above, describing the contour of a shape is useful for storing important information about the shape in a concise format and match it with other shapes. Unfortunately, by focusing on the contour characteristics, we do not have easy access to the global properties of the shape. Does it look like the letters X, O, or H? How many corners does it have? Does it include holes, and if so, how are they arranged? These are fundamental shape properties, and possibly the first ones we would use if we had to describe the shape to another person. If we look at the low-frequency Fourier descriptors, we may get a rough idea of the global shape parameters, but this information is still difficult to extract. We decided to use separate descriptors for each hole in a shape, which leads to a complete but non-intuitive description.

There is an alternative way for describing a shape, which is not based on the contour of the shape but on its medial axis. Let us assume – but only in our thoughts for safety reasons – that we set fire to every point on the shape’s boundary. Furthermore, let us say that the fire progresses at the same speed throughout the shape. Then, as the fire moves from the boundary toward the center of the shape, the firefronts will meet at various places and extinguish each other. If the material remains intact at these places but is burnt up otherwise, then whatever is left of the shape is its medial axis. To use a more mathematical description, the medial axis consists of the centers of all circles that are completely contained within the shape and touch its boundary in two or more points. Alternatively, we can define the medial axis as the centers of circles that are completely contained in the shape but are not contained in any other circle of
that kind. All three definitions can be more easily understood with the help of Fig. 7.14.

Fig. 7.14 Medial axis of a simple shape. The axis consists of the three straight lines in the shape, as in consists of all points at the centers of circles contained within the shape that touch its boundary in at least two different places.

Fig. 7.14 also illustrates the origin of the term *medial axis*. For an elongated shape, the medial axis runs through its center like a spine and fans out into each of its corners. It thus resembles a skeleton, and in fact, the term *region skeleton* has been used by many authors as a synonym for medial axis. This skeleton contains important information about the global structure of a shape. Importantly, it is topology conserving, that is, it does not change the connectivity of points in the shape. Therefore, a contiguous shape has a contiguous skeleton, and each hole in a shape causes a circuit in the skeleton. Before we discuss the characteristics of skeletons in detail, let us find out how to compute them in digital images.

### 7.5.2 A thinning algorithm for skeleton computation

While the shape of a skeleton is well-defined for continuous 2D shapes, this is unfortunately not the case for discrete 2D binary images. For binary images, we would like to approximate some of the desirable properties of skeletons, despite having to manage pixel-based contour descriptions and possible noise. Most importantly, we would to retain the topology conservation, mediality, and
rotation invariance of skeletons as much as possible. Furthermore, we would like our skeletons to be only one pixel wide.

How can we approach this problem? It seems that repeated application of the shrinking operation from Chapter 4 could thin a given shape down towards its skeleton. While the general idea is good, the shrinking operation would erode the entire shape including its corners, and it is unclear when the process should terminate. Instead, we should use a thinning operation as shown in Fig. 7.15. It uses four versions of two templates referred to as thinning templates (panel a) and restoring templates (panel b) that are created by rotations in steps of $90^\circ$. Each template is placed on the image so that its cross marker coincides with a pixel $p$. If all of the black pixels in a template land on objects pixels and all white ones on background pixels, then we have a match.

![Fig. 7.15 Templates used by the thinning algorithm. (a) Thinning templates; (b) Restoring templates. A pixel is removed if we center the templates on it and find a match for any thinning template and no match for any restoring template.](image)

In each iteration of the thinning algorithm, we remove all object pixels $p$ for which there is a match with any thinning template but no match with any restoring one. Note that all of these changes need to be performed on a copy of the original image, because the templates are not supposed to “see” any changes made within the same iteration of the algorithm. On the positive side, this independence allows a parallel implementation of the algorithm that applies the templates to all object pixels at the same time. These iterations continue, with each iteration “shaving off” one outer layer of object pixels. As soon as an iteration does not result in any deleted pixels, the algorithm terminates.

Fig. 7.16 shows the result of applying the thinning algorithm to a variety of shapes. Clearly, despite its simplicity, the algorithm produces a useful approximation of the medial axis. All object corners are represented in the computed skeletons, the axes are centered within the objects, and holes are
Chapter 7: Shape Representation

represented by circuits. As demonstrated by the rectangles on the right side of the figure, the skeletons are reasonably rotation invariant.

![Figure 7.16 Result of applying the thinning algorithm to various shapes. (a) Input image; (b) Computed skeletons superimposed on gray copies of the original shapes.](image)

It should be noted that the algorithm cannot always achieve the desired one-pixel wide skeleton lines. A post-processing algorithm is necessary to accomplish this. At the same time, post-processing can be used to translate the pixel-based skeleton into a graph. Endpoints and intersections in the skeleton are translated into vertices, and paths between them are represented by edges in the graph. Such a skeleton graph is the most concise way to describe the topology of a shape and, combined with location data, can serve as a useful input for higher-level computer vision algorithms.

7.6 Keypoint Detection and Representation

In the preceding sections, we discussed ways of describing objects whose shape was determined by the set of all pixels belonging to the object. Unfortunately, when processing real-world images, it is not always possible to derive shape information of this kind. In such images, most objects are partially occluded by other ones, and it can be very difficult to decide which parts of the image belong to a certain object without recognizing that object beforehand. For these reasons, recent approaches to the description and matching of object shape have focused on local, informative, and robustly recognizable keypoints instead of complete
object boundaries. Arguably the most influential algorithm of this kind has been the scale-invariant feature transform (SIFT) that can both determine and match keypoints that are most robust with regard to changes in scale and orientation. SIFT is a patented, intricate algorithm that is beyond the scope of this book.

Fortunately, the success of SIFT has triggered the development of various other approaches to this problem, some of which, while still being powerful, can be computed more efficiently and are easier to understand and implement. In the following, we will take a look at the FAST algorithm for keypoint detection and the BRIEF technique for keypoint description and matching.

### 7.6.1 Detecting corners as keypoints with the FAST algorithm

How can we find points on an object that are most useful as local object descriptors, that is, are as invariant as possible to slight changes in viewing angle, scale, orientation, and lighting? A good choice are corners, as they can be located more precisely than straight or curved structures and do not change their appearance dramatically with small variations in perspective or scale. A very efficient way of finding corners in an image is the *Features from Accelerated Segment Test* (FAST) algorithm. To test whether a given pixel $c$ is a corner point, we imagine an approximated circle of radius 3, consisting of 16 pixels, around it as shown in Fig. 7.17b. Within this circle, we then determine the longest run, i.e., uninterrupted sequence, of pixels meeting an intensity criterion. This criterion is that either all of the pixels have to be brighter than $c$ by a minimum difference $\theta$ or all of them have to be darker than $c$ by the same difference $\theta$. If this run is at least 12 pixels long, then $c$ is considered a corner point.

An easy way of finding runs in a circular structure is to simply traverse the circle twice while looking for the longest run. This process can be sped up tremendously by quickly eliminating candidates. For this purpose, we first consider only pixels 1, 5, 9, 13 on the circle (see Fig. 7.17). In order for $c$ to be a corner point, at least three of these pixels have to meet the criterion.
Fig. 7.17 Corner detection with the FAST algorithm. (a) Cutout from the input image (Fig. 7.19); (b) Finding the maximum run of pixels in a circle of radius 3 around a given pixel $c$ that fulfill the intensity criterion. If the run is at least 12 pixels long, $c$ is considered a corner point. Here, the run is of length 13.

It often happens that several neighboring pixels are recognized as corner points, and it would be inefficient to use all of them as keypoints. To avoid this problem, we can define a minimum distance between neighboring corners. If two or more corners are closer to each other than allowed by this minimum distance, we only keep the one representing the strongest corner point. An appropriate measure of corner strength is the sum of absolute intensity differences between $c$ and the pixels in the run. Another improvement of the FAST technique can often be accomplished by smoothing the image with a small convolution filter prior to corner detection. This especially removes some noise from the crucial intensity of the center pixel $c$. A 3×3 Gaussian smoothing filter often has a positive effect.

Efficient description of keypoints with the BRIEF approach

After having detected robust keypoints with the FAST algorithm, we now need a method of describing them so that we can recognize and match them even if perspective, orientation, scale or lighting slightly change. Clearly, we should not only represent the corner pixels themselves but a sizeable neighborhood around them in order to facilitate their recognition in other images. An efficient method for this purpose, both in terms of computational complexity and memory requirements, is the Binary Robust Independent Elementary Features (BRIEF) technique. It is based on intensity differences between pairs of points that are randomly chosen from the neighborhood of the keypoints. First, we should smooth the input image by applying a 9×9 Gaussian filter to make the descriptor more robust with regard to small changes. Second, we have to create the random pairs, which are kept constant throughout the feature extraction and matching process. A typical BRIEF descriptor contains 256 pairs of points that are sampled
from a Gaussian random distribution with \( \sigma = 7 \) centered on a given keypoint. They should not exceed a square area with lateral length \( 5\sigma \), i.e., in this case, an area of 35×35 pixels. Fig. 7.18 shows an example set of pairs, referred to as a mask, for the same corner point shown in Fig. 7.17.

Fig. 7.18 Randomly chosen pairs of points for building a BRIEF keypoint descriptor. (a) Cutout of smoothed input image; (b) Selected pairs indicated by straight lines connecting its two points. Here, 256 pairs were chosen from a 2D Gaussian distribution with \( \sigma = 7 \) pixel units.

Third, in order to build the descriptor for a given keypoint, we center the mask on the keypoint and check for each of the 256 pixel pairs indicated by the mask whether the first pixel in the pair is brighter than the second one. If so, we add a 1 to our descriptor string, otherwise a 0. This process will result in a 256-element binary vector, which serves as the BRIEF keypoint descriptor. It can be efficiently stored and processed within 32 bytes of memory.

To compare two keypoints by means of their associated BRIEF descriptors, we simply compute their Hamming distance, that is, their number of mismatching bits in corresponding positions. For example, binary strings 100110 and 110100 have a Hamming distance of 2 because they differ in their second and fifth bits. Obviously, greater Hamming distance indicates more pronounced differences between the keypoints, making it less likely that they represent the same point on the given object.

The potential of the FAST and BRIEF techniques for feature matching between images is demonstrated in Fig. 7.19. For two different images, the 20
strongest corner points with an intensity threshold of 70 units and a minimum distance of 15 pixels between each other were computed and marked with letters \(a\) to \(t\). The left panels show the original images with the detected points, and the right panels show the same scenes, photographed from a slightly different perspective. Here, the best matches of each of the 20 BRIEF descriptors with any pixel in the image are marked. Note that the location of the original corners was not used in order to find the best match, but only the descriptors.

Fig. 7.19 Corner detection with the FAST algorithm and their matching using BRIEF descriptors. Left panels: Original scenes with the 20 strongest edge points computed by FAST. Right panels: Same scenes taken from a slightly different viewpoint, showing the locations of the best BRIEF descriptor match for each of the 20 points.

In the upper scene, all points are matched correctly except for point \(a\), which moves along the frame of the monitor, and point \(b\), which is located in an entirely different area of the image. These failures are mostly due to the reflections on the
screen that change rapidly between the two images. The lower scene presents mostly accurate matches except for point $k$, which moved between houses, and point $p$, which jumped to another step of the house front. An interesting case is presented by point $e$, which stayed fixed with the house to which it belonged but moved relative to the church steeple. Due to the relative change in position between house and steeple, such points at intersections can at best be matched with only one of their objects. Generally speaking, the FAST and BRIEF methods are efficient but still powerful and can be used for object recognition, motion detection, and stereo matching, as we will find out in later chapters.