

MODULE 5

CHAPTER 9 - MORPHOLOGY IN IMAGE PROCESSING

Textbook

2. Rafael C G., Woods R E. and Eddins S L, Digital Image Processing, Pearson, 4th edition, 2019.

Mathematical Morphology and Set Theory:

Mathematical morphology is grounded in set theory and provides a unified framework for solving various image processing problems. In this context, image components are treated as sets, allowing for the application of set operations to manipulate and analyze image structures.

Representing Images as Sets:

In binary images, objects are modeled as sets of pixels within the 2D integer space Z^2 . Each set element is a 2D coordinate (tuple) corresponding to a foreground pixel. For grayscale images, sets are extended to Z^3 , where each element includes pixel coordinates and its intensity value. Higher-dimensional sets can also incorporate other attributes like color or temporal changes.

Objects and Structuring Elements (SEs):

Morphological operations rely on two types of sets: the object set and the structuring element (SE). Objects are defined as collections of foreground pixels. Structuring elements, on the other hand, can include both foreground and background pixels, and may contain "don't care" elements (denoted by $\times$). These elements can be ignored or dynamically assigned values during matching, depending on the application.

Embedding Sets in Arrays:

Since digital images are stored as rectangular arrays, sets used in morphological operations must be embedded into such arrays. Background pixels are assigned default values to fill non-object areas. This embedding helps bridge the gap between abstract set theory and practical digital image representation. For instance, images typically show a border of background pixels around objects, whereas structuring elements do not include such padding.

Morphological Operations and Convolution Similarity:

Though morphological operations differ from standard convolution, they share similar processing steps like padding and sliding the structuring element across the image. Structuring elements function in a way that resembles convolution kernels, particularly in how they are applied over the image.

Set Reflection and Translation in Morphology:

Two essential concepts used with structuring elements are set reflection and translation. The reflection of a structuring element, denoted as

$$\hat{B} = \{w \mid w = -b, \text{ for } b \in B\}$$

is obtained by mirroring the element about its origin. This reflected version plays a key role in operations like dilation and erosion, as it determines how the SE interacts with the image during processing.

Set Reflection in Morphology:

If B is a set of points in two-dimensional space, its reflection, denoted $\hat{B}$, is obtained by replacing each point with coordinates (x,y) by its reflected coordinates $(-x,-y)$. This reflection is a fundamental operation in mathematical morphology and is visually represented in Figure 9.2. The origin of the structuring element (SE) is typically marked by a dot. Reflecting an SE is equivalent to rotating it 180 degrees about its origin. This rotation applies to all parts of the

SE, including the foreground, background, and any “don’t care” elements ($\times$), ensuring a complete and consistent transformation.

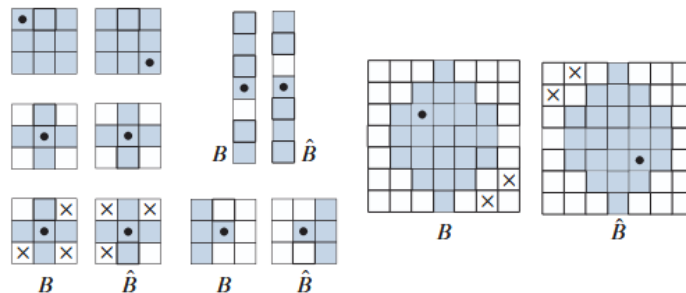
Set Translation in Morphology:

Translation of a set B by a point $z=(z_1,z_2)$ is denoted by B_z . This operation shifts every point in the set B by the vector z . In other words, each point (x,y) in B becomes $(x+z_1,y+z_2)$. Translation is essential for positioning the structuring element at various locations over the image during morphological operations such as dilation and erosion.

$$(B)_z = \{c \mid c = b + z, \text{ for } b \in B\}$$

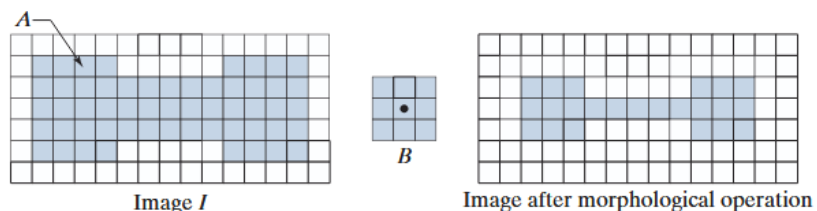
That is, if B is a set of pixels in 2-D, then $(B)_z$ is the set of pixels in B whose (x,y) Coordinates have been replaced by $(x+ z_1, y+ z_2)$.

FIGURE 9.2
Structuring elements and their reflections about the origin (the $\times$'s are don't care elements, and the dots denote the origin). Reflection is rotation by 180° of an SE about its origin.



a b c

FIGURE 9.3
(a) A binary image containing one object (set), A . (b) A structuring element, B . (c) Image resulting from a morphological operation (see text).



INTRODUCTION TO MORPHOLOGICAL OPERATIONS

Morphological operations in image processing involve interactions between a binary image and a structuring element (SE). Consider a binary image I , where the object of interest is denoted as set A (shaded), and a 3×3 structuring element B , composed entirely of foreground pixels (1s), is used.

To perform a morphological operation such as erosion, a new image of the same size as I is initialized with background values (0s). The structuring element B is then translated across the image. At every position, if the SE is entirely contained within the object A , the origin of B is marked as a foreground pixel in the new image. If not, it remains a background point.

When the origin of B lies at the boundary of A , part of B falls outside A , causing erosion at that location. The final image thus reflects a shrunk version of the original object, demonstrating the erosion effect. To keep visual representations compact, the minimum padding required for the SE is used—only one pixel wide in the case of a 3×3 SE.

In morphological terminology, saying " B is contained in A " specifically implies that all foreground elements of B overlap only with foreground elements of A . This concept becomes more nuanced when the SE includes background and “don’t care” ($\times$) elements.

In upcoming sections (9.2 to 9.7), binary morphology will be explored in detail. Grayscale morphological operations and structuring elements will be introduced in Section 9.8.

9.2 Erosion and Dilation – Introduction

Morphological image processing begins with two basic operations: **erosion** and **dilation**. These operations serve as the foundation for many advanced morphological techniques. Morphological expressions are defined using a **set of foreground pixels (A)** and a **structuring element (B)**.

Definition of Erosion

Erosion of a set A by a structuring element B , denoted by $A|B$, is defined as the set of all points z such that B , translated by z , is entirely contained within A . This operation removes boundary pixels from objects, effectively **shrinking** them.

Erosion in Terms of Images

In practice, morphological operations are applied on full images, not just sets. A more complete form of erosion, denoted $I|B$, accounts for both foreground and background pixels. It states that erosion includes only those positions where the translated structuring element lies completely within the object set A embedded in image I .

Use of Notation

When using erosion:

- Use $A|B$ if only foreground elements are involved.
- Use $I|B$ only when background or “don’t care” elements are included.
- Avoid using $|$ with mixed-value SEs; alternate notations are used in such cases.

Alternate View of Erosion

An equivalent definition of erosion is that the intersection of the translated structuring element B_z and the background (complement of A) must be empty:

$$A|B = \{z : B_z \cap A^c = \emptyset\}$$

This means no part of the SE can overlap with background pixels during erosion.

Visual Interpretation

In a binary image, erosion results in the **inner region** of the object, where the structuring element fits entirely. The result of erosion is influenced by the **shape and size** of the structuring element. A larger or differently shaped SE can erode objects into thinner or differently shaped remnants.

Practical Considerations

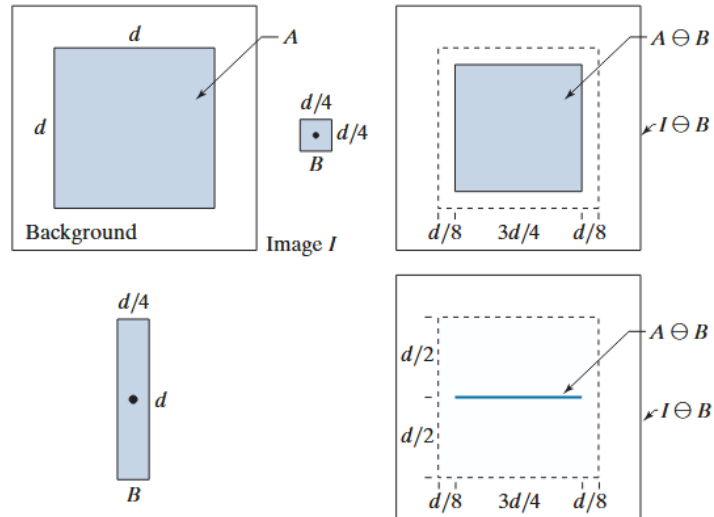
To apply erosion correctly:

- Ensure the image is padded to accommodate the full movement of the structuring element.
- Crop the output to match the size of the original image.
- This is similar to padding and cropping in **spatial convolution**.

a b c
d e

FIGURE 9.4

(a) Image I , consisting of a set (object) A , and background.
(b) Square SE, B (the dot is the origin).
(c) Erosion of A by B (shown shaded in the resulting image).
(d) Elongated SE.
(e) Erosion of A by B . (The erosion is a line.) The dotted border in (c) and (e) is the boundary of A , shown for reference.



DILATION

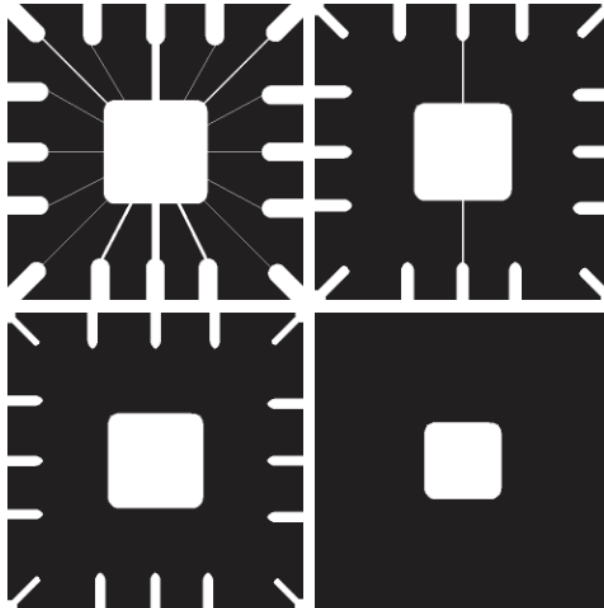
With A and B as sets in Z^2 , the dilation of A by B , denoted as $A \oplus B$, is defined as

$$A \oplus B = \{z \mid (\hat{B})_z \cap A \neq \emptyset\} \quad (9-6)$$

a b
c d

FIGURE 9.5

Using erosion to remove image components.
(a) A 486×486 binary image of a wire-bond mask in which foreground pixels are shown in white.
(b)–(d) Image eroded using square structuring elements of sizes 11×11 , 15×15 , and 45×45 elements, respectively, all valued 1.



DILATION

Dilation is the second fundamental morphological operation. It is based on **reflecting** the structuring element BBB about its origin and **translating** it by a vector zzz, similar to the process in erosion.

Definition of Dilation

The dilation of a set A by a structuring element BBB, denoted as $A \oplus B$, is defined as the set of all points z such that the **reflected and translated structuring element** $B^\wedge_z$ overlaps **at least one foreground pixel** in set A.

Mathematical Expression

This can be expressed as:

$$A \oplus B = \{z : (\hat{B})_z \cap A \neq \emptyset\}$$

Here, z represents the **displacement of the origin** of the reflected structuring element $B^\wedge$. The dilation includes all such positions where there is **any intersection** between $B^\wedge$ and A.

Interpretation

While erosion requires the structuring element to fit **completely inside** the object, dilation is more **inclusive**, allowing partial overlap. This causes the object to **grow** or **expand**, depending on the size and shape of the structuring element.

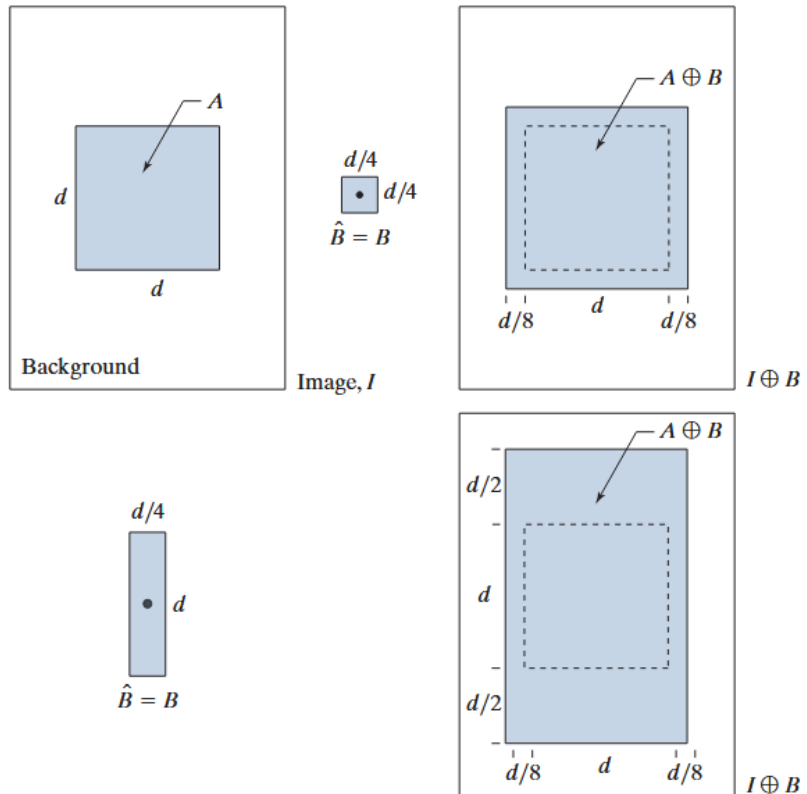
Dilation is the opposite of erosion. Instead of shrinking objects, dilation **grows or thickens** them. This growth is controlled by the **size and shape** of the structuring element (SE) used. In the dilation process, the background of the image is often expanded to accommodate the object's growth. When using a **symmetric structuring element**, reflection has no effect. As the structuring element slides over the image, all locations where any part of it **overlaps** the object lead to the addition of foreground pixels in the output.

The shape of the structuring element determines the direction and extent of dilation. For example, a vertically elongated SE leads to **more vertical expansion**, as seen in the illustrative results. The dilation includes all points where the origin of the SE is within a region that causes **non-empty intersection** with the object.

a b c
d e

FIGURE 9.6

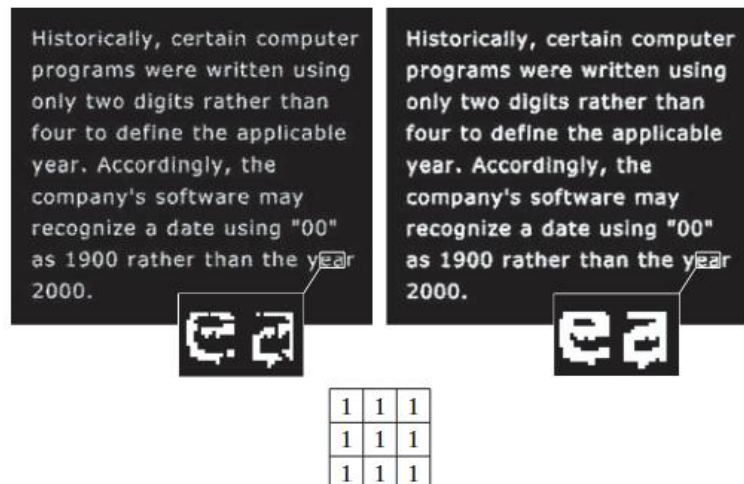
(a) Image I , composed of set (object) A and background.
(b) Square SE (the dot is the origin).
(c) Dilation of A by B (shown shaded).
(d) Elongated SE.
(e) Dilation of A by this element. The dotted line in (c) and (e) is the boundary of A , shown for reference.



a c
b

FIGURE 9.7

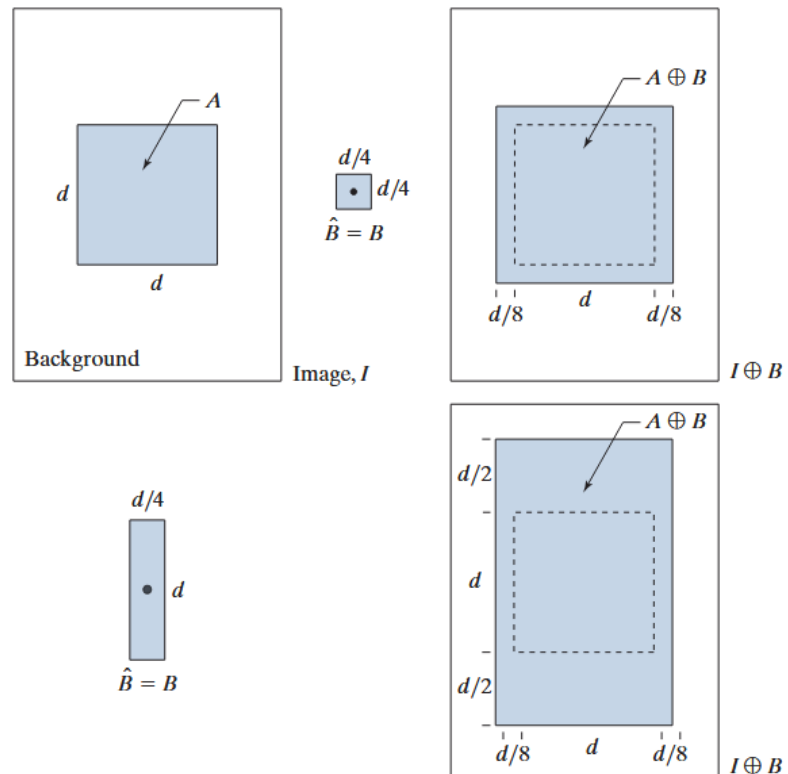
(a) Low-resolution text showing broken characters (see magnified view).
(b) Structuring element.
(c) Dilation of (a) by (b). Broken segments were joined.



a b c
d e

FIGURE 9.6

(a) Image I , composed of set (object) A and background.
(b) Square SE (the dot is the origin).
(c) Dilation of A by B (shown shaded).
(d) Elongated SE.
(e) Dilation of A by this element. The dotted line in (c) and (e) is the boundary of A , shown for reference.



DUALITY

Erosion and dilation are *duals* of each other with respect to set complementation and reflection. That is,

$$(A \ominus B)^c = A^c \oplus \hat{B} \quad (9-8)$$

and

$$(A \oplus B)^c = A^c \ominus \hat{B} \quad (9-9)$$

Equation (9-8) indicates that erosion of A by B is the complement of the dilation of A^c by $\hat{B}$ and vice versa.

We proceed to prove formally the validity of Eq. (9-8) in order to illustrate a typical approach for establishing the validity of morphological expressions. Starting with the definition of erosion, it follows that

$$(A \ominus B)^c = \left\{ z \mid (B)_z \subseteq A \right\}^c$$

If set $(B)_z$ is contained in A , then it follows that $(B)_z \cap A^c = \emptyset$, in which case the preceding expression becomes

$$(A \ominus B)^c = \left\{ z \mid (B)_z \cap A^c = \emptyset \right\}^c$$

But the *complement* of the set of z 's that satisfy $(B)_z \cap A^c = \emptyset$ is the set of z 's such that $(B)_z \cap A^c \neq \emptyset$. Therefore,

$$\begin{aligned} (A \ominus B)^c &= \left\{ z \mid (B)_z \cap A^c \neq \emptyset \right\} \\ &= A^c \oplus \hat{B} \end{aligned}$$

where the last step follows from the definition of dilation in Eq. (9-6) and its equivalent form in Eq. (9-7). This concludes the proof. A similar line of reasoning can be used to prove Eq. (9-9) (see Problem 9.16).

9.3 Opening and Closing – Combined Morphological Operations

Opening and closing are two essential operations in morphological image processing that are built upon erosion and dilation.

Opening involves an erosion followed by a dilation using the same structuring element. It is primarily used to **smooth object contours**, **remove thin protrusions**, and **break narrow connections** (called isthmuses) between objects. This operation is particularly useful in eliminating small noise while preserving the overall shape and size of larger objects.

Closing, on the other hand, consists of a dilation followed by an erosion. It tends to **fill small holes**, **close small gaps**, and **connect nearby objects**. This makes closing effective for strengthening object boundaries and fusing narrow breaks.

Both operations maintain the general structure of objects but are applied in opposite contexts — opening removes small foreground features, while closing removes small background features.

The *opening* of set A by structuring element B , denoted by $A \circ B$, is defined as

$$A \circ B = (A \ominus B) \oplus B \quad (9-10)$$

Thus, the opening A by B is the erosion of A by B , followed by a dilation of the result by B .

Similarly, the *closing* of set A by structuring element B , denoted $A \bullet B$, is defined as

$$A \bullet B = (A \oplus B) \ominus B \quad (9-11)$$

which says that the closing of A by B is simply the dilation of A by B , followed by erosion of the result by B .

The interpretation that the opening of A by B is the union of all the translations of B such that B fits entirely within A can be written in equation form as

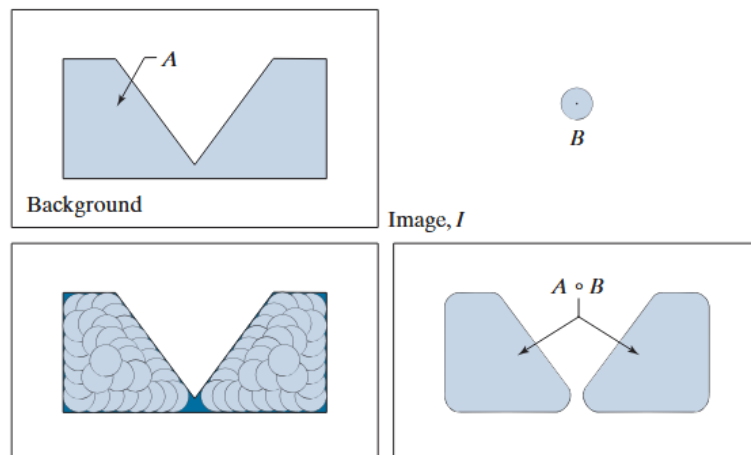
$$A \circ B = \bigcup \{ (B)_z \mid (B)_z \subseteq A \} \quad (9-12)$$

where $\bigcup$ denotes the union of the sets inside the braces.

a b
c d

FIGURE 9.8

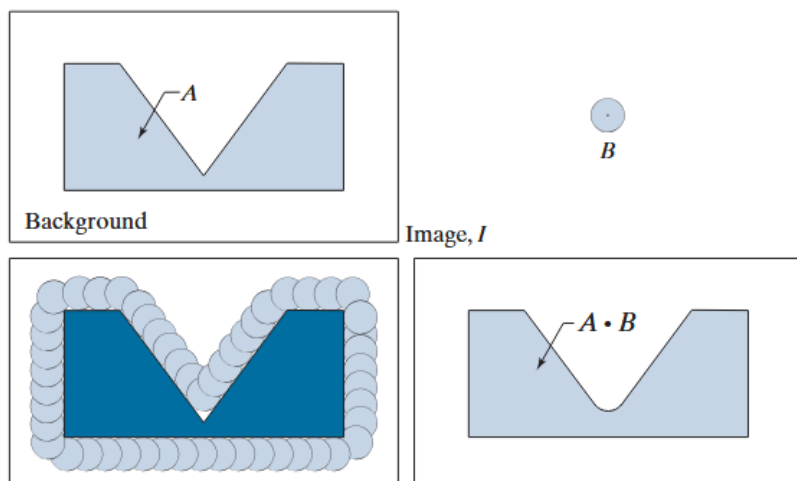
- (a) Image I , composed of set (object) A and background.
(b) Structuring element, B .
(c) Translations of B while being contained in A . (A is shown dark for clarity.)
(d) Opening of A by B .



a b
c d

FIGURE 9.9

- (a) Image I , composed of set (object) A , and background.
(b) Structuring element B .
(c) Translations of B such that B does not overlap any part of A . (A is shown dark for clarity.)
(d) Closing of A by B .



Closing is conceptually similar to dilation but involves translating the structuring element B **outside** the set A. The closing of A by B is defined as the **complement** of the union of all translations of B that **do not overlap** with A.

This means the boundary of the closing is determined by the **furthest extent** that BBB can reach without intersecting A. Visually, this operation tends to **fill small holes** and **connect nearby objects** by expanding A and then restoring its original shape.

Mathematically, the closing can be written as:

$$A \bullet B = (A^c \oplus B)^c = \{z : (B)_z \cap A \neq \emptyset\}^c$$

where A^c is the complement of A, and $\oplus$ denotes dilation.

As with erosion and dilation, opening and closing are duals of each other with respect to set complementation and reflection:

$$(A \circ B)^c = (A^c \bullet \hat{B}) \quad (9-14)$$

and

$$(A \bullet B)^c = (A^c \circ \hat{B}) \quad (9-15)$$

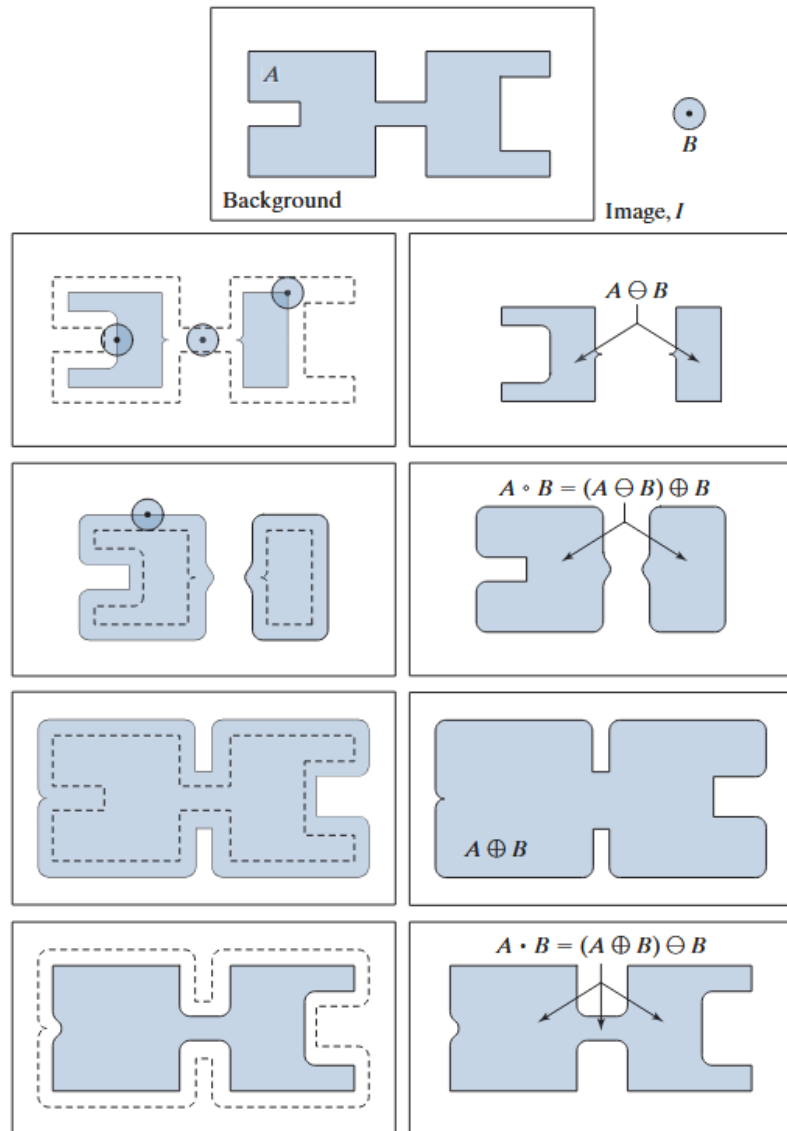
We leave the proof of these equations as an exercise (see Problem 9.20).

a
b c
d e
f g
h i

FIGURE 9.10

Morphological opening and closing.

(a) Image I , composed of a set (object) A and background; a solid, circular structuring element is shown also. (The dot is the origin.)
(b) Structuring element in various positions.
(c)-(i) The morphological operations used to obtain the opening and closing.



Morphological opening has the following properties:

- (a) $A \circ B$ is a subset of A .
- (b) If C is a subset of D , then $C \circ B$ is a subset of $D \circ B$.
- (c) $(A \circ B) \circ B = A \circ B$.

Similarly, closing satisfies the following properties:

- (a) A is a subset of $A \bullet B$.
- (b) If C is a subset of D , then $C \bullet B$ is a subset of $D \bullet B$.
- (c) $(A \bullet B) \bullet B = A \bullet B$.

Note from condition (c) in both cases that multiple openings or closings of a set have no effect after the operation has been applied once.

9.4 Hit-or-Miss Transform (HMT)

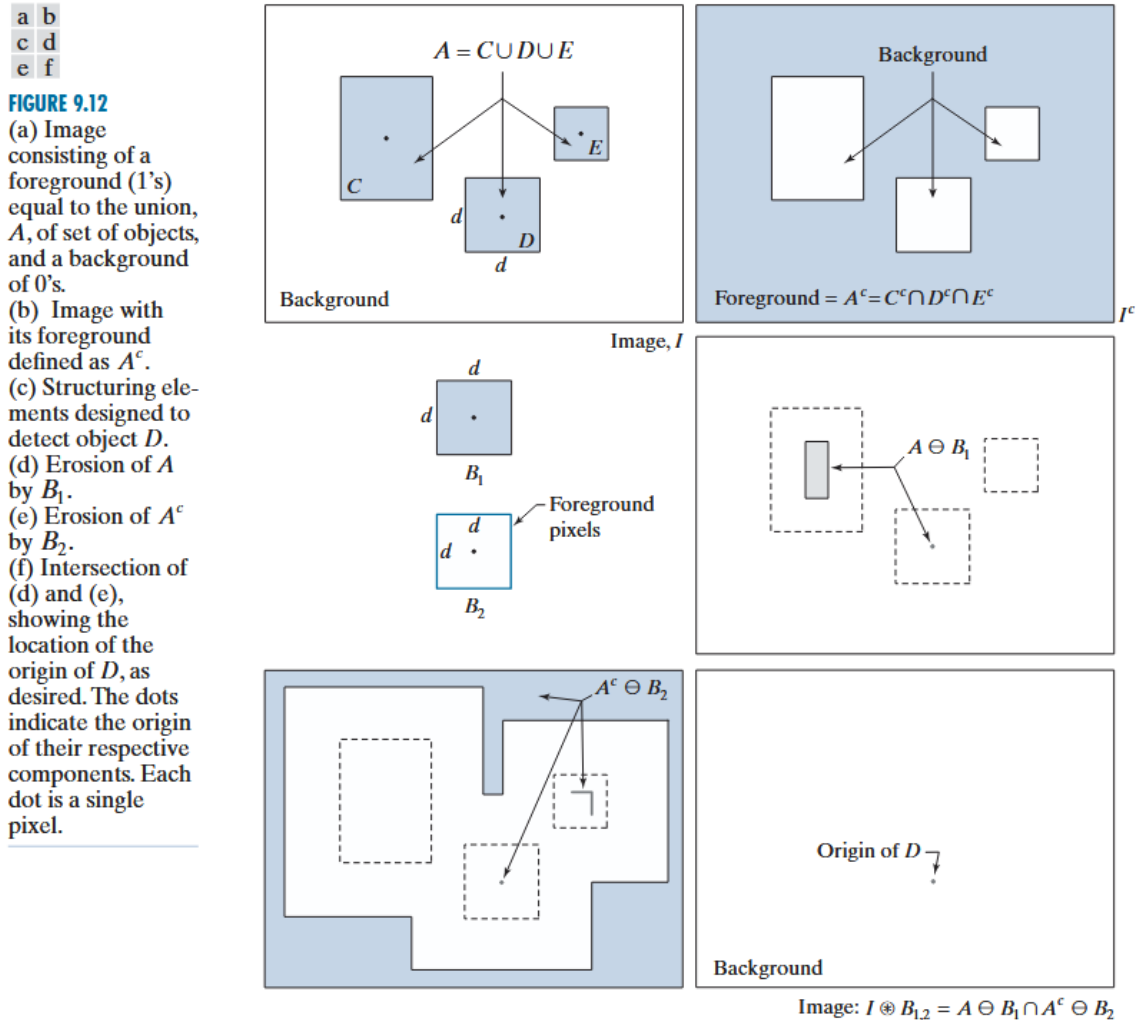
The **Hit-or-Miss Transform (HMT)** is a fundamental morphological operation used primarily for **shape detection** in binary images. In this context, a binary image I is composed of foreground pixels represented by a set A , and background pixels. Unlike previous morphological operations that focus only on foreground structures, HMT employs **two structuring elements** simultaneously—one that matches the foreground and another that corresponds to the background. This dual-element approach enables precise identification of specific configurations or patterns within the image, making HMT particularly effective for detecting shapes such as corners, line endings, or other defined structures.

structuring elements: B_1 , for detecting shapes in the foreground, and B_2 , for detecting shapes in the background. The HMT of image I is defined as

$$\begin{aligned} I \circledast B_{1,2} &= \left\{ z \mid (B_1)_z \subseteq A \text{ and } (B_2)_z \subseteq A^c \right\} \\ &= (A \ominus B_1) \cap (A^c \ominus B_2) \end{aligned} \tag{9-16}$$

The morphological **Hit-or-Miss Transform (HMT)** is mathematically defined using erosion operations, where one structuring element B_1 is matched with the foreground pixels A , and a second element B_2 is matched with the background, or the complement of A , denoted A^c . The transform identifies positions z where both conditions are met **simultaneously**— B_1 is fully contained within A ,

and B_2 is fully contained within A^c . The use of the word "miss" highlights that B_2 matching the background is equivalent to **missing** the foreground. This concept is illustrated by detecting the specific position of an object D , a subset of the foreground set A , in an image I , using an appropriately designed pair of structuring elements to match both the object and its surrounding background.



The **Hit-or-Miss Transform (HMT)** is a morphological technique used for **shape detection** in binary images by identifying specific patterns through a **pair of structuring elements**. One structuring element, B_1 , matches a shape within the **foreground** (set A), while the other, B_2 , matches its corresponding configuration in the **background** (A^c). This dual approach is necessary because the **geometric context** of an object is defined not only by its shape but also by the background surrounding it. In Fig. 9.12, B_1 is designed to match the object D , while B_2 detects the border around D within the background of the complement image I^c . The **intersection** of the results of erosions with B_1 and B_2 yields the precise location of D . To simplify this, a **single composite structuring element** B , containing foreground, background, and "don't care" pixels ($\times$), can be used in Eq. (9-17) to match patterns directly in I . This approach allows processing of **foreground and background pixels simultaneously**,

as detecting **holes**, **corners**, and **edges**. The inclusion of don't care elements allows a structuring element to flexibly adapt, matching multiple configurations without the need for multiple SEs, thereby increasing its versatility in morphological image analysis.

We can use a structuring element formed in such a way to restate the HMT as

$$I \circledast B = \{z \mid (B)_z \subseteq I\}$$

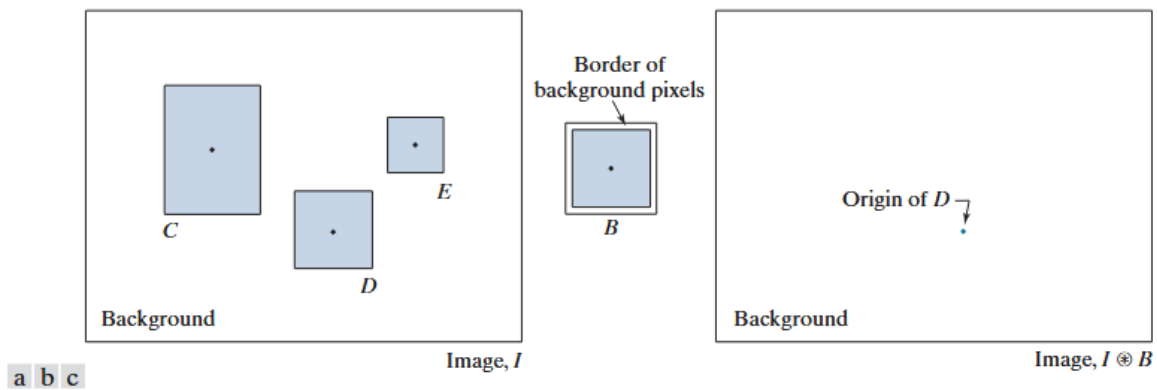
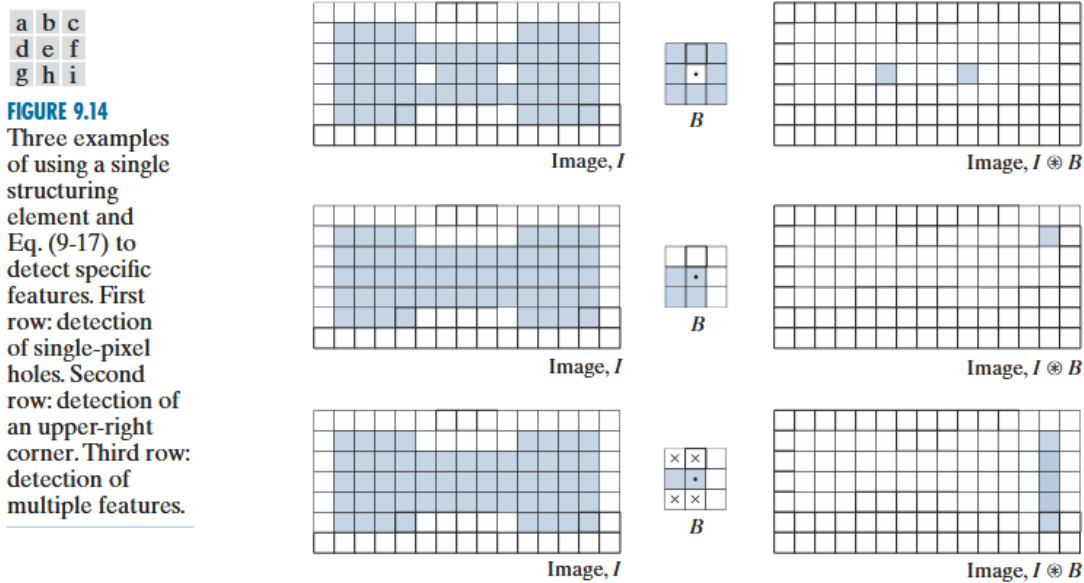


FIGURE 9.13 Same solution as in Fig. 9.12, but using Eq. (9-17) with a single structuring element.

9.5 SOME BASIC MORPHOLOGICAL ALGORITHMS

The field of **morphological image processing** offers a variety of practical techniques, especially when working with **binary images**. These techniques are essential for extracting and analyzing components within an image that contribute to the **representation and description of shape**. This section builds upon previously discussed morphological concepts to introduce several key algorithms. Specifically, it addresses methods for **boundary extraction**, **connected component labeling**, **convex hull computation**, and **skeletonization** of regions. Additionally, the section presents useful operations such as **region filling**, **thinning**, **thickening**, and **pruning**, which are commonly applied during pre- or post-processing stages. To enhance clarity, the explanations include simplified **mini-images** that visually illustrate the function and effect of each algorithm. These binary visuals maintain a consistent representation, with **foreground pixels (1's) shaded** and **background pixels (0's) left white**, providing

an intuitive understanding of the morphological transformations involved.



BOUNDARY EXTRACTION: is a fundamental morphological operation used to identify the border of objects in binary images. Given a set A of foreground pixels, its boundary, denoted $b(A)$, is computed by first performing **erosion** on A using a suitable **structuring element** B , and then taking the **set difference** between A and its erosion. Mathematically, this is represented as

$$\beta(A) = A - (A \ominus B)$$

where $\ominus$ denotes the erosion operation. The idea is to shrink the object slightly and subtract the result from the original to leave behind only the pixels at the edge. Commonly, a 3×3 structuring element is used, but larger elements like a 5×5 matrix of 1's can produce thicker boundaries. During the implementation, the image is typically padded with background pixels to accommodate the erosion process, and the final result is then cropped back to the original image size. This method is particularly useful for shape analysis and feature extraction in image processing tasks.

HOLE FILLING:

In morphological image processing, a **hole** is defined as a **background region completely surrounded by connected foreground pixels**. The objective of hole filling is to convert these enclosed background areas into foreground pixels (1's). The process is based on an **iterative morphological dilation** technique conditioned by **complementation and intersection** operations. Let A represent the foreground object boundaries, and assume that a **starting point** (or seed) within each hole is known. An initial image X_0 is formed, consisting of all zeros except at the seed points, which are set to one. Then, the iterative process

$X_k = (X_{k-1} \oplus B) \cap I^c \quad k = 1, 2, 3, \dots$ is applied, where B is a symmetric structuring element, and A^c is the complement of the original image. This operation continues until convergence, that is, when

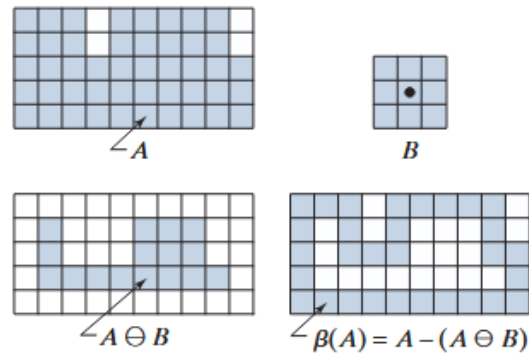
when $X_k = X_{k-1}$. The result, X_k ,

The result, X_k , represents the **filled hole regions**, and the union of X_k with the original image yields the final image with holes filled. This approach, known as **conditional dilation**, ensures that dilation occurs only within the background regions bounded by the object. Although the example provided in the figure illustrates a single hole, the algorithm can be applied to multiple holes, provided that a seed point is identified within each. This method serves as an effective demonstration of how morphological operations can be **controlled to satisfy specific image-processing objectives**.

a b
c d

FIGURE 9.15

- (a) Set, A , of foreground pixels.
- (b) Structuring element.
- (c) A eroded by B .
- (d) Boundary of A .



a b

FIGURE 9.16

- (a) A binary image.
- (b) Result of using Eq. (9-18) with the structuring element in Fig. 9.15(b).



a	b	c
d	e	f
g	h	i

FIGURE 9.17

Hole filling.

(a) Set A (shown shaded) contained in image I .

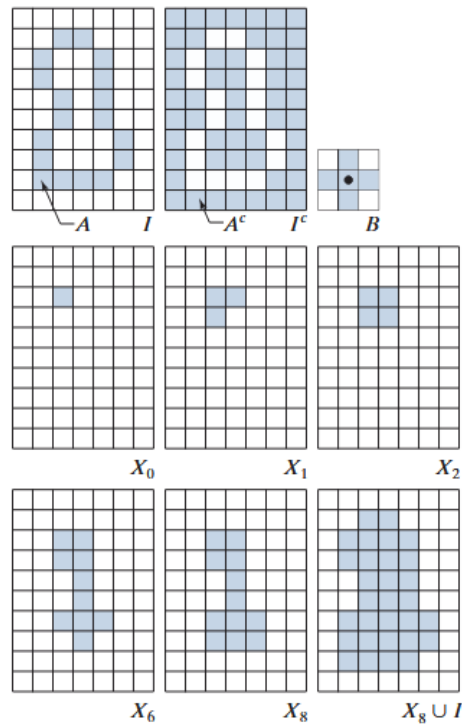
(b) Complement of I .

(c) Structuring element B . Only the foreground elements are used in computations

(d) Initial point inside hole, set to 1.

(e)–(h) Various steps of Eq. (9-19).

(i) Final result [union of (a) and (h)].



EXTRACTION OF CONNECTED COMPONENTS: The **extraction of connected components** is a key task in binary image analysis, forming the basis of many automated image interpretation techniques. Given a binary image III , let $\mathbf{A}$ denote the set of **foreground pixels** which may contain **multiple connected components**. To begin the extraction process, an initial image X_0 is created with all zero (background) values, except at **known seed points**, each corresponding to a foreground pixel within a connected component. The objective is to iteratively grow these seed points until the entire connected component is recovered. This is achieved using the following iterative formula:

$$X_k = (X_{k-1} \oplus B) \cap I \quad k = 1, 2, 3, \dots \quad (9-20)$$

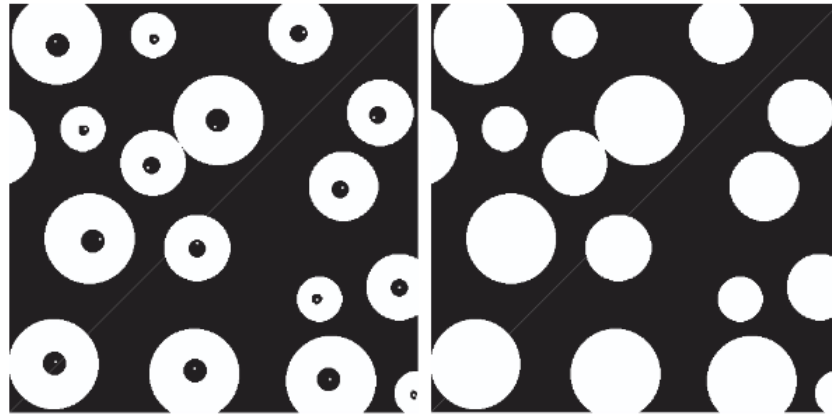
where B is a **structuring element** that reflects the chosen pixel connectivity—typically 8-connected for generality. The operation continues until convergence, that is, when

$$X_k = X_{k-1}.$$

The final set X_k then contains all foreground pixels that are **connected to the original seed points**, effectively extracting the connected components. This process employs **conditional dilation**, which restricts growth to the foreground region of III , contrasting with the **hole-filling algorithm**, which operates within the image complement I^c to find enclosed background regions. While the described method requires at least one **initial pixel per component**, completely automated techniques that eliminate this requirement are discussed elsewhere, such as in Problem 9.37.

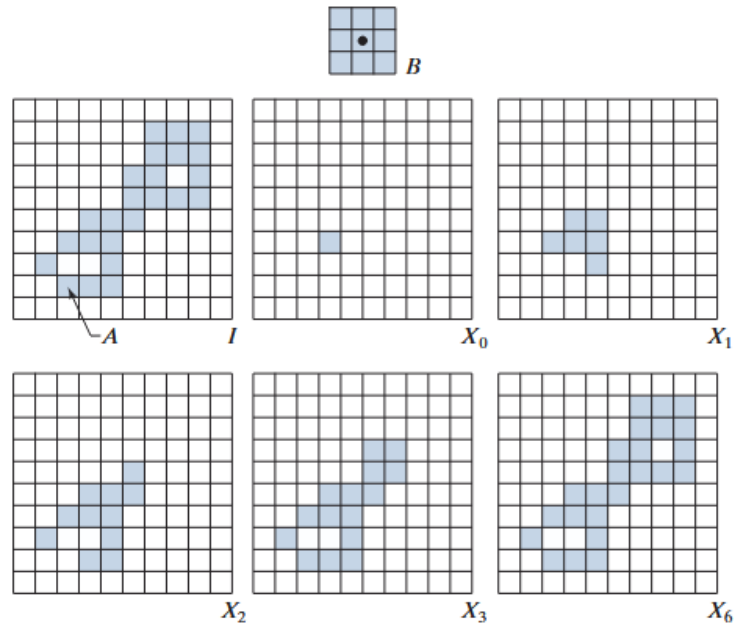
a b

FIGURE 9.18
(a) Binary image. The white dots inside the regions (shown enlarged for clarity) are the starting points for the hole-filling algorithm.
(b) Result of filling all holes.



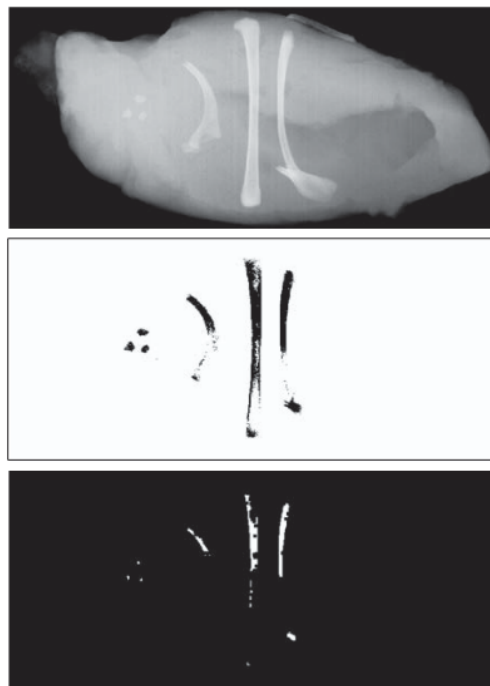
a
b c d
e f g

FIGURE 9.19
(a) Structuring element.
(b) Image containing a set with one connected component.
(c) Initial array containing a 1 in the region of the connected component.
(d)–(g) Various steps in the iteration of Eq. (9-20)



a
b
c d

FIGURE 9.20
(a) X-ray image of a chicken filet with bone fragments.
(b) Thresholded image (shown as the negative for clarity).
(c) Image eroded with a 5×5 SE of 1's.
(d) Number of pixels in the connected components of (c). (Image (a) courtesy of NTB Elektronische Geraete GmbH, Diepholz, Germany, www.ntbxray.com.)



Connected component	No. of pixels in connected comp
01	11
02	9
03	9
04	39
05	133
06	1
07	1
08	743
09	7
10	11
11	11
12	9
13	9
14	674
15	85

CHAPTER 11

FEATURE EXTRACTION

11.1 Background

In image processing, although there's no strict definition of what an “image feature” is, we generally understand a feature as a distinctive attribute that helps us label or differentiate objects. The key purposes of features are to help us **identify** (label) and **distinguish** (differentiate) between image elements. These features may refer to specific objects within an image or to entire images themselves. Essentially, features are important properties or descriptors that help us tell images or image parts apart.

Feature extraction in images has two main components: **feature detection** and **feature description**. Detection refers to the process of locating the features in an image, while description refers to assigning values or attributes to those features. For example, if we consider corners of objects as features, then detection means finding those corners, and description involves measuring their orientation, location, or relation to other corners. Simply detecting a feature isn't enough; we also need descriptive information to help compare and classify objects or images.

To be effective in differentiating between image elements or full images, the features we extract must have certain **important characteristics**. Ideally, features should be **independent of position (location), size (scale), and orientation (rotation)**. Additionally, features should be robust against changes in lighting or the viewpoint from which the image was captured. This means a good feature should remain consistent even if the image is taken from a different angle or under different lighting conditions.

Preprocessing plays a crucial role in improving feature extraction. By using techniques like **histogram equalization**, we can adjust the brightness and contrast of an image to minimize the effects of lighting changes, making it easier to detect and describe features accurately. Using known information about the images in advance (a priori information) can help us apply appropriate preprocessing before extracting features.

Finally, when we say a feature is **independent**, it usually means it is either **invariant** or **covariant** under certain transformations. A feature descriptor is **invariant** if its value does not change after a transformation (like rotation, translation, or reflection). It is **covariant** if the transformation applied to the object also reflects similarly in the feature's description. For example, the **area of an elliptical region** remains the same under rotation, reflection, or translation—making area an invariant feature with respect to those transformations.

Feature Descriptors

One of the key ideas in image analysis is the use of features to describe image content. A commonly used feature descriptor is **area**. It is **invariant** with respect to some transformations, meaning its value does not change when those transformations are applied. However, when we introduce **scaling**, the area becomes **covariant**. This means the descriptor value changes proportionally with the transformation. For example, **scaling a region** increases its area, and thus the area descriptor scales accordingly.

Similarly, the **direction of the principal axis** of a region is also a covariant feature. When the region is **rotated**, the direction value also rotates by the same angle.

Most **feature descriptors** are generally covariant — they might be **invariant to certain transformations** but **sensitive to others**. To handle this, we often apply **normalization techniques** to eliminate as many covariances as possible. For instance, to make features **rotation-invariant**, we can calculate the actual direction of each region and rotate the region so that its **principal axis aligns** with a predefined direction.

Local vs. Global Features

Features are also categorized as **local** or **global**. However, this classification depends heavily on the **application context**.

Consider the example of a **production line** that checks how full bottles are. Images of 10 bottles are captured, where the **liquid appears bright**, and the **background is dark**. The **area** of the bright region in each bottle directly reflects the **amount of liquid**, making it a **local feature**, as it applies to **individual regions** in an image.

Now, if we change the problem to computing the **total amount of liquid in one image**, area becomes a **global feature** because it applies to the **entire image**.

Taking it further, suppose the task now is to compute the **total liquid per day**. In this case, the **area per image** is treated as a **local feature**, and the **total across all images** becomes a **global descriptor**. This can go on indefinitely, depending on how we define the “**member**” and “**set**.”

So, in these notes:

- A **local feature** applies to a **member** of a set (e.g., one bottle in an image).
- A **global feature** applies to the **whole set** (e.g., all bottles or all images in a day).

— **Purpose of Feature Descriptors**

Feature descriptors are rarely intended for **direct human interpretation**, except in specialized cases like **interactive image processing**. Often, feature extraction methods generate **dozens to thousands of descriptors**, which may seem meaningless if viewed individually.

Instead, feature descriptors are used as a **preprocessing step** in more advanced image processing tasks, such as:

- Image **registration**
- **Object recognition**
- **Scene understanding**

These features allow machines to understand visual data and make intelligent decisions based on them.

In various computer vision tasks like **automated inspection**, **pattern recognition** (such as detecting **faces** or **fingerprints** in image databases), and **autonomous systems** (including **robotic and vehicle navigation**), it is essential to describe image content numerically. This is usually done using a **feature vector**, which is essentially a $1 \times n$ or $n \times 1$ **matrix** where each element represents a specific **descriptor**.

A basic example is an **RGB image**, where each **pixel** is described using a **3-dimensional vector**:

$$x = [x_1, x_2, x_3]$$

Here, x_1 represents the **red intensity**, x_2 the **green intensity**, and x_3 the **blue intensity** at a given point. When using **color as a feature**, a particular region in an RGB image can be viewed as a group of these vectors positioned in a **3D feature space**.

If we use **n descriptors**, then our feature vector becomes **n-dimensional**, and the collection of all such vectors lies in an **n-dimensional feature space**. You can imagine these vectors as forming a “**hypercloud**” of points in **n-dimensional Euclidean space**, which gives a spatial perspective to complex data.

The features are broadly classified into three major categories: **boundary features**, **region features**, and **whole image features**. This classification helps in logically grouping features based on the image property they describe. For instance, when referring to the “**length of a boundary**,” it makes sense only when talking about a **region’s boundary**, not the entire image. On the other hand, some features may apply to **both boundaries and regions**, and a few can even describe **whole images**.

The classification is not about limitations of the techniques, but about the meaningful application of features to specific aspects of an image. This approach will help us better understand and analyze various image characteristics in the following sections.

11.2 BOUNDARY PREPROCESSING

In image segmentation, the output often consists of raw data in the form of **pixels located along object boundaries** or **within segmented regions**. To make the computation of descriptors more efficient and structured, it is a common practice to compress this data into more compact and useful representations. This section introduces several **boundary preprocessing techniques** that help in achieving this goal.

One fundamental approach is called **Boundary Following (or Tracing)**. Many algorithms require the **boundary points** of a region to be in a specific order—either **clockwise** or **counterclockwise**. Thus, it is essential to use an algorithm that produces an **ordered sequence of boundary points**. The method discussed here operates under the assumptions that the image is **binary** (with object pixels labeled as 1 and background pixels as 0) and that the image has been padded with a **border of 0s**. This padding ensures that objects do not merge with the image borders. Although the explanation is limited to a single region, the method can be extended to handle **multiple disjoint regions** by processing each region separately.

The algorithm, commonly known as the **Moore Boundary Tracing Algorithm**, follows these steps:

1. Identify the **starting point**, denoted as b_0 , which is the uppermost-leftmost pixel in the image labeled 1. Let c_0 be the **west neighbor** of b_0 . Since c_0 lies outside the object, it will always be a background pixel. Starting at c_0 , examine the **8-neighbors** of b_0 in a **clockwise direction**. The first neighbor encountered with a value of 1 is labeled as b_1 , and the **preceding pixel** in this scan (a background pixel) is labeled as c_1 . The coordinates of b_0 are stored for use in a later step.
2. Set $b = b_0$ and $c = c_0$.
3. Again, examine the **8-neighbors** of b , beginning at c and scanning **clockwise**. Let $n_1, n_2, \dots, n_8$ represent these neighbors. Find the first pixel labeled 1 and denote it as n_k .
4. Set $b = n_k$ and let c be the pixel immediately preceding n_k in the clockwise scan.
5. Repeat steps 3 and 4 until b equals b_0 . The sequence of points collected during this process represents the **ordered boundary points** of the object.

An important detail is that c , the background pixel in each step, always precedes the newly found object pixel. This method effectively traces the outer boundary of a region and is reliable for **complex shapes**, including **branched** or **self-intersecting boundaries**. It can also be applied to images with **multiple regions**, processing one boundary at a time.

If the input is a **binary region** (as opposed to a boundary), the algorithm extracts the **outer boundary**. While the result is usually a **one-pixel-wide boundary**, it may vary in some cases.

To trace **interior boundaries** (such as holes within regions), these holes must be extracted and treated as **separate 1-valued regions** against a 0-valued background. Applying the same algorithm will then yield the **inner boundaries** of the original region.

Although this tracing process can be defined in either the **clockwise** or **counterclockwise** direction, it is typically simpler to use one consistent direction and reverse the result when needed. In later discussions, both directions will be used interchangeably to help students become familiar with each approach.

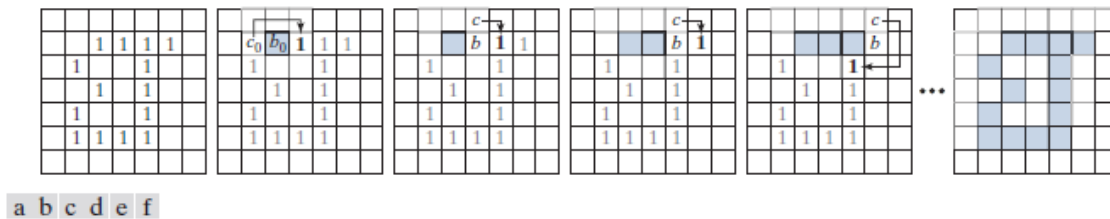


FIGURE 11.1 Illustration of the first few steps in the boundary-following algorithm. The point to be processed next is labeled in bold, black; the points yet to be processed are gray; and the points found by the algorithm are shaded. Squares without labels are considered background (0) values.

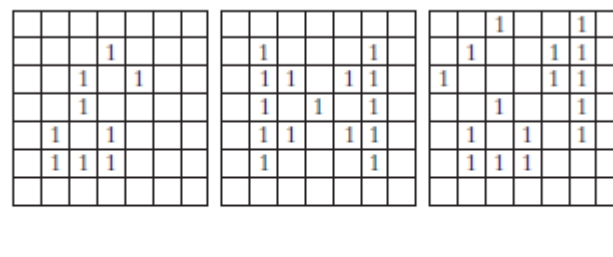


FIGURE 11.2 Examples of boundaries that can be processed by the boundary-following algorithm. (a) Closed boundary with a branch. (b) Self-intersecting boundary. (c) Multiple boundaries (processed one at a time).

11.2.2 CHAIN CODES

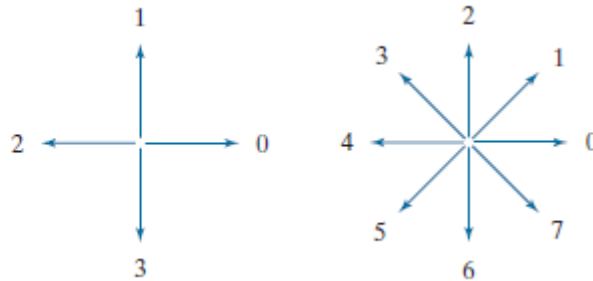
Chain codes are used to describe a **boundary** through a connected sequence of **straight-line segments** of defined **length and direction**. In this context, we typically assume that all curves are **simple and closed**—meaning they form a loop without self-intersection.

One of the most well-known chain code methods is the **Freeman Chain Code**, which represents direction based on **4-connectivity** or **8-connectivity**. Each segment's direction is encoded using a specific **numbering scheme**, such as the one shown in standard figures like Fig. 11.3. The resulting boundary code—a sequence of these numbers—is known as the Freeman chain code.

Since **digital images** are generally arranged in a **grid format** with equal spacing along both x and y axes, a chain code can be generated by following the boundary of an object, usually in a **clockwise direction**, and assigning a directional number to each segment between pixel pairs.

a b

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



However, using the full pixel-level detail often leads to two main issues: (1) The resulting chain becomes **excessively long**, and (2) **Minor disturbances** or **noise** can cause **unnecessary changes** in the code that do not reflect the actual shape of the object.

To overcome these challenges, a common strategy is to **resample the boundary** using a **coarser grid** (as shown in Fig. 11.4(a)). During traversal, each boundary point is mapped to the nearest **grid node** (as seen in Fig. 11.4(b)). This **simplified representation** can be encoded using either a **4-directional** or **8-directional** chain code. For instance, Fig. 11.4(c) shows how this is applied using the **8-directional scheme**. Converting between 4-code and 8-code is straightforward.

To maintain consistency in representation, the **starting point** for chain code generation is typically chosen as the **uppermost-leftmost point** on the boundary. For example, in Fig. 11.4(c), this results in a chain code sequence like **0766...1212**. The spacing of the resampling grid is **application-specific** and must be selected accordingly.

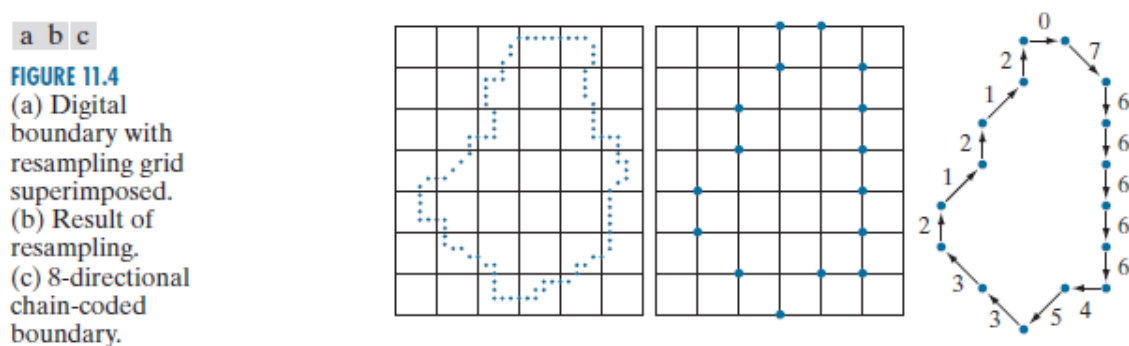
When the sampling grid is a **uniform quadrilateral**, as illustrated in standard references (e.g., Fig. 2.19), the resulting **Freeman chain code** aligns precisely with the digital curve. Similarly, when using a resampling approach with the same type of grid, all points in the code remain **reachable**, preserving accuracy in representation.

The numerical values in a chain code are dependent on the **chosen starting point**, but this dependency can be removed through a simple **normalization process**. By treating the chain code as a **circular sequence**, we can rotate the sequence to start from the point that yields the **lowest possible integer value**. Additionally, **rotation normalization** can be achieved using the **first difference** of the chain code. This is calculated by counting the **number of direction**

changes, in a **counterclockwise** manner, between each pair of adjacent elements in the code. For example, the first difference of the 4-directional chain code **10103322** is **3133030**.

We can also perform **size normalization** by modifying the **spacing of the resampling grid**. However, it is important to note that all the normalization techniques discussed are **exact only under ideal conditions**, i.e., when the object's boundaries remain unchanged under **rotation** (in fixed-angle increments) and **scaling**. In practice, this is rarely the case, as even the same object captured from different angles or at different resolutions will produce **distinct boundary shapes**. The extent of variation increases with **image resolution**.

To minimize these effects, one can use **longer chain elements** relative to the distance between pixels and align the **resampling grid** along the **principal axes** of the object being analyzed.



FREEMAN CHAIN CODE AND BOUNDARY REPRESENTATION:

Figure 11.5(a) presents a 570×570 -pixel, 8-bit grayscale image containing a **circular stroke** embedded within small, **randomly scattered specular fragments**. The primary goal of this example is to derive the **Freeman chain code**, determine the **corresponding integer of minimum magnitude**, and compute the **first difference** for the outer boundary of the stroke. Due to the presence of noise in the form of small fragments, directly extracting the object's boundary would result in a **noisy curve** that poorly represents the actual shape. As previously discussed, **smoothing** is an essential step when dealing with such **noisy boundaries**. Figure 11.5(b) illustrates the image after it has been smoothed using a 9×9 pixel box filter. (Refer to Section 3.5 for details on **spatial smoothing**.)

Following the smoothing step, **Figure 11.5(c)** shows the result of applying **Otsu's thresholding method**, which significantly simplifies the image by reducing it to **two regions** (one being just a dot). This reduction in complexity aids in isolating the stroke more effectively.

In Figure 11.5(d), we observe the **outer boundary** of the region identified in the previous step. If we were to extract the **chain code** from this boundary directly, the outcome would be a long sequence with **minor variations** that fail to capture the **overall shape**. To address this, we perform **resampling** before computing the chain code, thereby reducing unnecessary variations.

Figure 11.5(e) demonstrates the result of using a **resampling grid** with nodes spaced **50 pixels apart** (approximately 10% of the image width). In Figure 11.5(f), these sample points are joined by **straight lines**, producing a **simplified approximation** that retains the main structure of the original boundary.

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

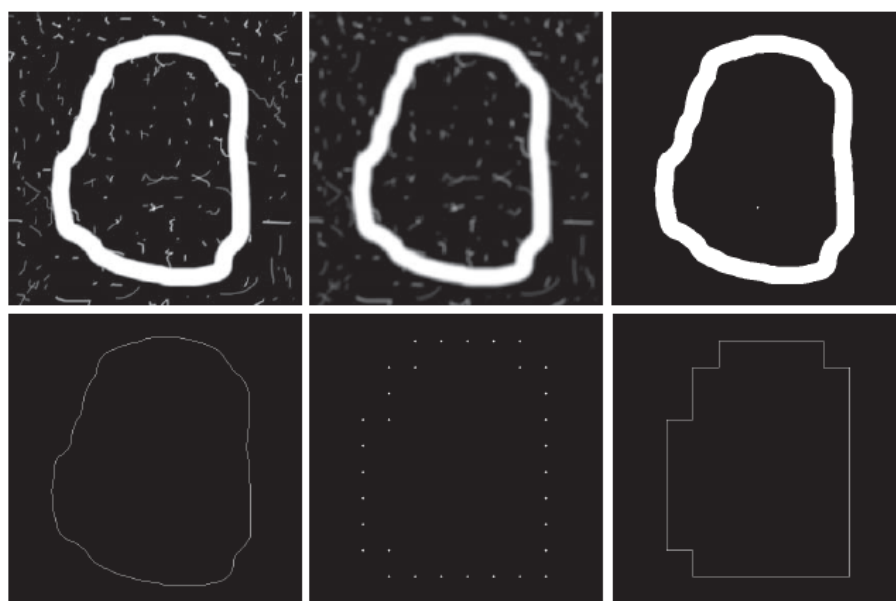
0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2

The **starting point** of the boundary is at coordinates **(2, 5)** in the **subsampled grid**. As noted in Figure 2.19, the **origin** of an image is located at the **top-left corner**, making this the **uppermost-leftmost** point in Figure 11.5(f).

In this case, the **integer of minimum magnitude** derived from the code is the same as the chain code itself: 0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2

The **first difference** of the code, which helps reduce redundancy and enables efficient analysis, 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 coding method offers a substantial reduction in the amount of data required to represent the boundary.



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 (SCCs):

An alternative to the traditional Freeman chain codes for boundary representation is the use of **Slope Chain Codes (SCCs)**, as proposed by **Bribiesca (1992, 2013)**. Freeman chain codes typically require a **resampling of the boundary** by defining a **grid structure**, where boundary points are assigned to their **nearest grid neighbors** to minimize small variations. In contrast, SCCs approach the problem by placing **equal-length straight-line segments** along the boundary of a 2D curve. These segments touch the curve at their endpoints, forming a simplified geometric representation.

The key step in generating an SCC is to compute the **slope changes between consecutive line segments**. These slope changes are then **normalized** into the continuous **open interval $(-1, 1)$** . Unlike Freeman codes, SCCs do not require grid alignment, simplifying the implementation. Additionally, SCCs offer **better rotation invariance** because they are not limited to the **eight directional codes** used by Freeman's approach. They also share **translation invariance** and can be normalized to account for **scaling effects**.

To begin generating an SCC, the **length of the line segment** must be chosen. Next, a **starting point** is defined—typically one of the endpoints of the curve in the case of an open shape. A segment is placed such that one of its ends is at this origin point, and the other touches the curve. This touched point becomes the starting position for the next segment. The process continues until the other end of the open curve is reached. This method can be visualized as using **identical circles**, where the **radius equals the length of the segment**, moving along the curve. The **intersection points** between these circles and the curve define the points of connection for the line segments.

After establishing the points of intersection, the **slope changes between consecutive segments** are calculated. These changes are normalized such that **positive and zero values fall within $[0, 1)$** and **negative changes fall within $(-1, 0)$** . Importantly, slope values of ± 1 are excluded to avoid ambiguities arising from segments of the same slope but in **opposite directions**.

The resulting series of normalized slope changes forms the SCC, which represents the approximation of the curve. For instance, a curve might be encoded with the sequence: **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 slope measurements determines the **size of the symbol alphabet**. In the example shown in Fig. 11.6(d), an accuracy of 10^{-2} produces **199 possible slope values**, while an accuracy of 10^{-1} would produce **19 symbols**.

One limitation of SCCs is that the **final point of the encoded curve** may not exactly coincide with the **curve's true end point**. This discrepancy can often be reduced by either **decreasing**

the segment length or **increasing the angular resolution**, though computations are still constrained by rounding to **integer coordinates**.

The **inverse of an SCC** is created by **reversing the order and signs** of the slope values. To obtain a **mirror image**, one simply reverses the signs while keeping the order from the origin. These principles apply equally to **closed curves**, where one may start at any arbitrary point (often the **top-leftmost point**) and proceed around the curve in either a **clockwise** or **counterclockwise** direction until returning to the starting point.

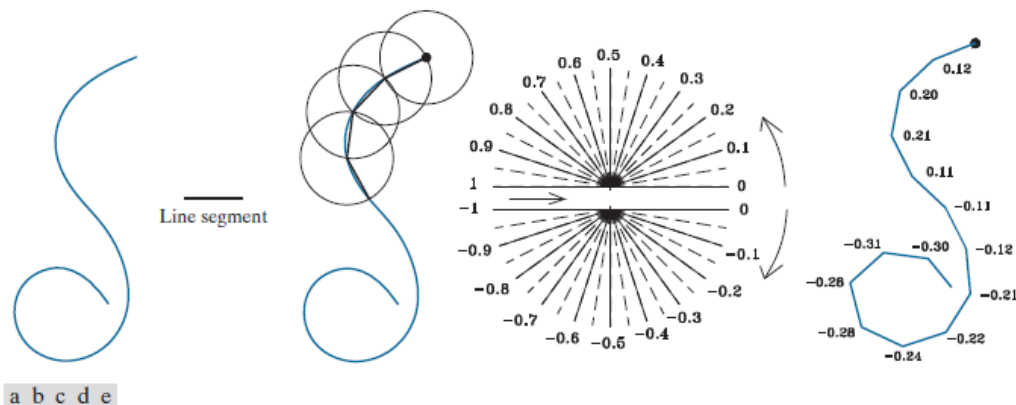


FIGURE 11.6 (a) An open curve. (b) A straight-line segment. (c) Traversing the curve using circumferences to determine slope changes; the dot is the origin (starting point). (d) Range of slope changes in the open interval $(-1, 1)$ (the arrow in the center of the chart indicates direction of travel). 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.)

IMAGE PATTERN CLASSIFICATION

12.1 Background

Humans possess the most advanced pattern recognition abilities in the known biological world. In contrast, machine recognition systems still fall short when compared to routine human tasks, such as interpreting complex visual information or generalizing from past experiences. Nonetheless, these machines play a vital role in daily life. Consider the importance of systems that can read barcodes, process bank checks, inspect manufacturing quality, read fingerprints, sort mail, and recognize speech—all examples of pattern recognition in action.

In image pattern recognition, a pattern is defined as a spatial arrangement of features, and a pattern class is a collection of patterns that share similar characteristics. Pattern recognition by machine involves techniques for automatically assigning patterns to their respective classes. In simple terms, the goal is to assign a class label to each input pattern based on its characteristics. There are four fundamental stages in the pattern recognition process:

1. Sensing – Capturing data in a spatial (2-D) or higher-dimensional format.
2. Preprocessing – Performing tasks such as noise reduction, image enhancement, restoration, and segmentation.
3. Feature Extraction – Identifying and isolating important attributes from the image.
4. Classification – Using the extracted features to assign class labels to unknown image patterns.

This chapter focuses primarily on classification, the final step in the recognition process. We will explore three main approaches to image pattern classification:

1. Prototype-based classification – Matching unknown patterns with predefined prototypes.
2. Statistical classification – Using decision-theoretic models, such as the Bayes classifier, to optimize classification.
3. Neural networks – Employing systems that can learn features and patterns from raw data, sometimes even without manual feature design.

The difference between these approaches lies in the amount of engineering effort required to prepare the data. In prototype-based classification, the features are designed to be so distinctive that classification becomes straightforward. For example, bank-check processors use specific font styles to enhance recognition accuracy.

In statistical classification, the problem is framed using probability and statistics, and the goal is to identify parameters that offer the best classification performance. This approach gives equal importance to both feature selection and classifier design.

In the case of neural networks, especially deep learning models, the system can automatically learn feature representations from raw data. These models do not require manually designed features and can perform highly accurate classification based solely on training data.

All three approaches rely on parameters that must be either predefined or learned from examples. Data used for this purpose can be labeled (where class labels are known) or unlabeled (where the goal is to discover patterns without prior labels). A classic example of labeled data is character recognition, where each character is tagged with a label (e.g., ‘a’ to ‘z’ or ‘0’ to ‘9’). Unlabeled data is often used for tasks like clustering, where the system identifies groups or patterns without prior knowledge.

When working with labeled data, the dataset is usually split into three parts:

- A training set (typically 50%) for adjusting model parameters.
- A validation set (about 25%) for tuning and comparing model designs.
- A test set (about 25%) for evaluating performance on unseen data.

If the system performs well during training and validation but fails on the test set, it may be overfitted—meaning the model learned the training data too well and cannot generalize to new data. This signals the need to rethink the model architecture or the data used.

When a system is trained using labeled data, the process is called supervised learning. If the system learns from unlabeled data, it undergoes unsupervised learning. This chapter deals primarily with supervised learning, which includes methods ranging from models with fixed features designed by humans, to advanced deep learning systems that automatically learn features from large datasets.

12.2 Classes and Pattern Vectors

In image pattern classification, patterns are generally divided into two main types: **quantitative** and **structural**.

- **Quantitative patterns** are represented using **pattern vectors**, which are arrays of numerical feature values.
- **Structural patterns** consist of symbols and are arranged as **strings**, **trees**, or sometimes **graphs**. While this chapter primarily focuses on pattern vectors, structural patterns will be discussed briefly at the end of this section and in Section 12.3.

✓ Pattern Vectors

Pattern vectors are typically denoted by lowercase letters such as $\mathbf{x}$, $\mathbf{y}$, or $\mathbf{z}$. These vectors take the form:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

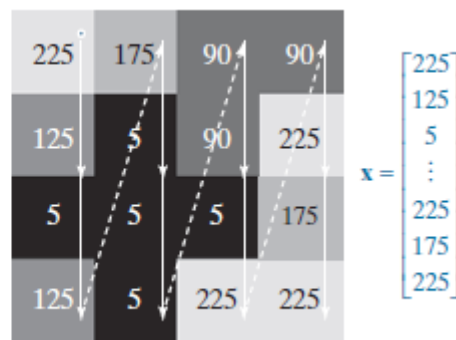
This format is known as a **column vector**, where each element x_i represents a specific **feature descriptor**, and n is the total number of such descriptors. The same vector can also be written as a row vector:

$$\mathbf{x} = (x_1, x_2, \dots, x_n)^t$$

Here, the superscript **T** denotes the **transpose** operation, converting the row vector into a column vector.

a b

FIGURE 12.1
Using linear indexing to vectorize a grayscale image.



A pattern vector can be imagined as a **point in n-dimensional space**, where each axis represents a feature. A **pattern class** is then a **cluster or “hypercloud”** of these points in this space. Ideally, we want the clusters to be **tight** (intra-class similarity) and **well-separated** (inter-class distinction).

✓ Creating Pattern Vectors

Pattern vectors can be formed directly from **pixel values** of an image by converting the 2D image into a 1D vector using **linear indexing**. However, a more common and efficient approach is to use **image features** as vector components.

One historical example is the work of **R. A. Fisher** in 1936, where he used **discriminant analysis** to classify three species of iris flowers (setosa, versicolor, and virginica). Each flower was described using **four features**: petal length and width, and sepal length and width. This resulted in **4-dimensional vectors** for classification.

High-Level Feature-Based Vectors

Features extracted from image **boundaries**, **regions**, or both can also form pattern vectors. For instance:

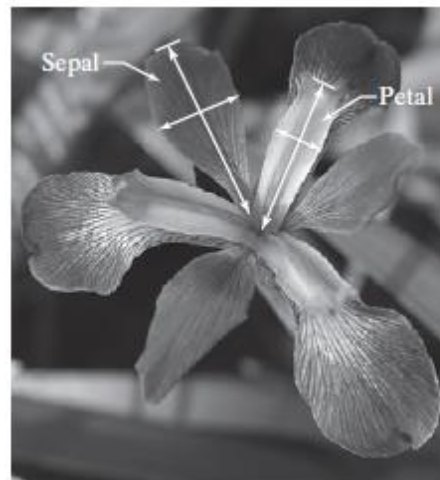
- **Boundary signatures** (1D curves representing shape outlines) can be sampled at regular intervals to form vectors.
- Instead of raw samples, feature functions like **statistical moments** (as introduced in Chapter 11) are often computed and used in the vectors.
- Objects with both boundary and region properties can be represented by **3D or higher-dimensional vectors**.

For example, in binary images, objects can be represented by features like **compactness**, **elongation**, and **area**, forming compact pattern vectors.

✓ Texture and Invariant Features

Pattern vectors may also include **texture measures** (e.g., coarseness, contrast) or **transformation-invariant features**. These features are especially helpful when dealing with varying **scales**, **rotations**, or **illumination** conditions, ensuring the classifier remains robust.

FIGURE 12.2
Petal and sepal width and length measurements (see arrows) performed on iris flowers for the purpose of data classification. The image shown is of the *Iris virginica* gender. (Image courtesy of USDA.)



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

x_1 = Petal width
 x_2 = Petal length
 x_3 = Sepal width
 x_4 = Sepal length

✓ Multispectral and High-Dimensional Applications

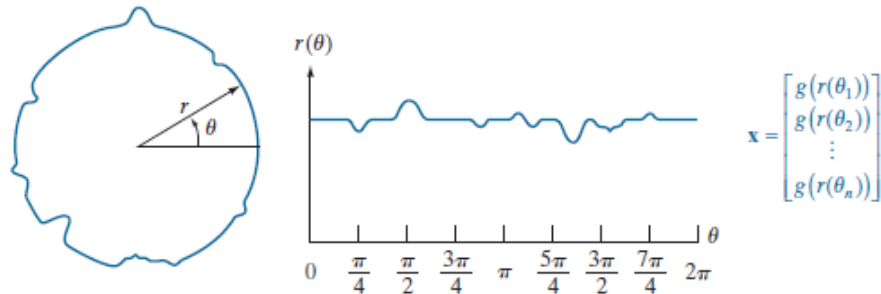
When working with **registered image sequences**, such as in **multispectral imaging**, pattern vectors are formed from corresponding **pixels** across images. This enables consistent recognition based on the same spatial locations across different spectral bands.

In cases where **entire images** are used, much higher-dimensional vectors are necessary. Traditional methods like **SIFT** rely on engineered features, but **deep convolutional neural networks (CNNs)** now offer more powerful alternatives by learning relevant features directly from raw image data.

a b

FIGURE 12.3

(a) A noisy object boundary, and (b) its corresponding signature.



STRUCTURAL PATTERNS:

While most pattern classification in image processing uses pattern vectors, **structural patterns** are essential for applications where **shape and relational features** dominate.

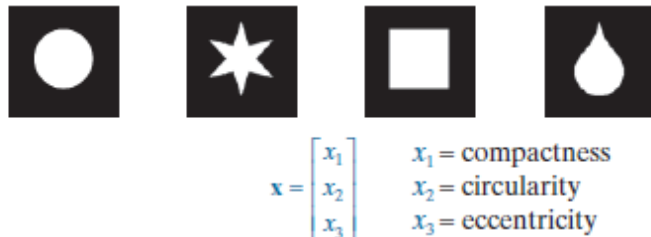
What Are Structural Patterns?

Unlike pattern vectors that rely on numerical feature descriptors, **structural patterns** use symbolic representations—often in the form of **strings**, **trees**, or occasionally **graphs**. These representations are particularly useful for describing **object shapes**, **hierarchical relationships**, or **compositions**.

a b c d

FIGURE 12.4

Pattern vectors whose components capture both boundary and regional characteristics.



✓ String-Based Structural Patterns

An example of a structural pattern is illustrated in Fig. 12.8. In this example, the **boundary of a bottle** is approximated using a polygon. The process involves:

- Subdividing the boundary into **line segments**.
- Calculating the **interior angle** (denoted as **u**) at each junction between segments.
- Encoding the boundary traversal as a **sequence of symbols**, forming a **string**.

The string describes how the shape changes as the boundary is followed in a **counterclockwise** direction. These symbol strings can be **matched** to predefined **prototype strings**, allowing classification based on shape structure. This approach is especially useful when geometric precision is difficult but **symbolic patterns** can be reliably matched.

✓ Tree-Based Structural Patterns

Another common structural representation is the **tree**, which is suitable for **hierarchical image descriptions**. A tree structure allows for a **top-down decomposition** of an image into its **component regions**.

Consider the satellite image in Fig. 12.9. A possible hierarchical breakdown might look like:

- Root node (\$): Represents the **entire image**.
- First-level branches: Classify the image into **downtown** and **residential** areas.
- Subsequent levels:
 - **Residential areas** are further broken down into **housing**, **highways**, and **shopping malls**.
 - **Housing** might include apartments, single-family homes, etc.
 - **Highways** might include overpasses, intersections, and feeder roads.

Each **node** in the tree represents a component of the image, and each **branching** reflects the "composed of" relationship. This type of representation is powerful for applications in **scene understanding**, **object hierarchy**, or **semantic segmentation**.

✓ Applications of Structural Patterns

Structural approaches are particularly relevant in:

- **Shape recognition** (e.g., industrial inspection)
- **Scene analysis** (e.g., satellite imagery)
- **Document layout understanding**
- **Biological structure modelling** (e.g., anatomical trees in medical imaging)

Although less common in general-purpose image classification tasks, structural patterns offer rich descriptive power where **relationships** between parts are as important as the parts themselves.

12.3 – PATTERN CLASSIFICATION BY PROTOTYPE MATCHING

Pattern classification by **prototype matching** is a fundamental approach in pattern recognition. It involves comparing an **unknown pattern** with a set of **known prototypes**, then assigning the unknown to the class of the **most similar** prototype.

Each **prototype** corresponds to a **pattern class**, and there may be **multiple prototypes per class** to better capture class variability. The key factor that differentiates various matching techniques is the **similarity (or distance) measure** used.

◆ Minimum-Distance Classifier

One of the most commonly used methods for prototype matching is the **Minimum-Distance Classifier**. This approach operates as follows:

1. Each pattern class c_j is represented by a **prototype vector** $\mathbf{m}_j$, typically the **mean vector** of all training samples from that class:

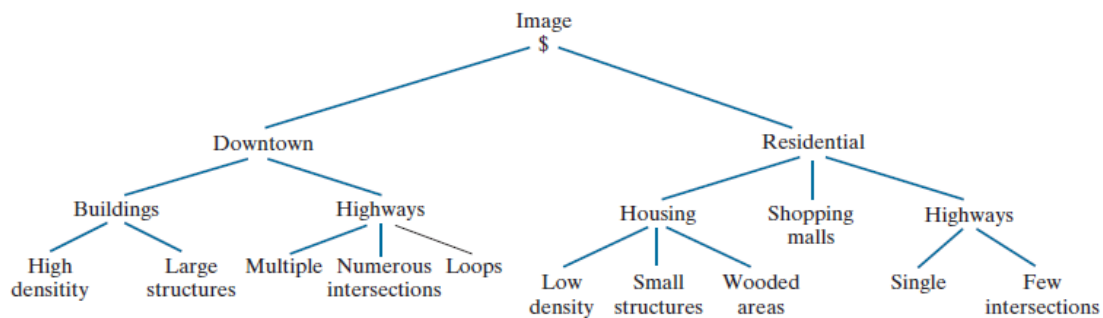


FIGURE 12.9 Tree representation of a satellite image showing a heavily built downtown area (Washington, D.C.) and surrounding residential areas. (Original image courtesy of NASA.)

Minimum-Distance Classifier

One of the most commonly used methods for prototype matching is the **Minimum-Distance Classifier**. This approach operates as follows:

1. Each pattern class c_j is represented by a **prototype vector** $\mathbf{m}_j$, typically the mean vector of all training samples from that class:

$$\mathbf{m}_j = \frac{1}{n_j} \sum_{\mathbf{x} \in c_j} \mathbf{x} \quad (12-2)$$

where:

- n_j : Number of training vectors in class c_j
- N_c : Total number of classes

2. To classify a new input vector $\mathbf{x}$, compute the Euclidean distance between $\mathbf{x}$ and each prototype:

$$D_j(\mathbf{x}) = \|\mathbf{x} - \mathbf{m}_j\| \quad (12-3)$$

where $\|\cdot\|$ denotes the Euclidean norm.

3. Assign $\mathbf{x}$ to the class c_i with the smallest distance:

$$D_i(\mathbf{x}) < D_j(\mathbf{x}) \quad \text{for all } j \neq i$$

Ties (i.e., equal distances) are resolved arbitrarily.

Discriminant Functions

Instead of computing distances directly, classification can also be performed using discriminant functions, which are algebraically equivalent:

$$d_j(\mathbf{x}) = \mathbf{m}_j^T \mathbf{x} - \frac{1}{2} \mathbf{m}_j^T \mathbf{m}_j \quad (12-4)$$

The decision rule is then:

$$\text{Assign } \mathbf{x} \text{ to class } c_i \text{ if } d_i(\mathbf{x}) > d_j(\mathbf{x}) \text{ for all } j \neq i \quad (12-5)$$

Decision Boundaries

The decision boundary between two classes c_i and c_j is defined by:

$$d_i(\mathbf{x}) = d_j(\mathbf{x}) \quad (12-6)$$

Or equivalently:

$$d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0 \quad (12-7)$$

Substituting Eq. (12-4) into Eq. (12-7) gives:

$$(\mathbf{m}_i - \mathbf{m}_j)^T \mathbf{x} - \frac{1}{2} (\mathbf{m}_i^T \mathbf{m}_i - \mathbf{m}_j^T \mathbf{m}_j) = 0 \quad (12-8)$$

This equation represents a hyperplane that bisects the line joining $\mathbf{m}_i$ and $\mathbf{m}_j$ perpendicularly.

- For $n = 2$: The boundary is a line
- For $n = 3$: A plane
- For $n > 3$: A hyperplane

This forms the basis for the minimum-distance classifier's decision boundaries, enabling geometrical separation of pattern classes in n -dimensional space.

EXAMPLE 12.1: Illustration of the minimum-distance classifier for two classes in 2-D.

Figure 12.10 shows scatter plots of petal width and length values for the classes Iris versicolor and Iris setosa. As mentioned in the previous section, pattern vectors in the iris database consists of four measurements for each flower. We show only two here so that you can visualize the pattern classes and the decision boundary between them. We will work with the complete database later in this chapter.

We denote the Iris versicolor and setosa data as classes c_1 and c_2 , respectively. The means of the two classes are $\mathbf{m}_1 = (4.3, 1.3)^T$ and $\mathbf{m}_2 = (1.5, 0.3)^T$. It then follows from Eq. (12-4) that

$$\begin{aligned} d_1(\mathbf{x}) &= \mathbf{m}_1^T \mathbf{x} - \frac{1}{2} \mathbf{m}_1^T \mathbf{m}_1 \\ &= 4.3x_1 + 1.3x_2 - 10.1 \end{aligned}$$

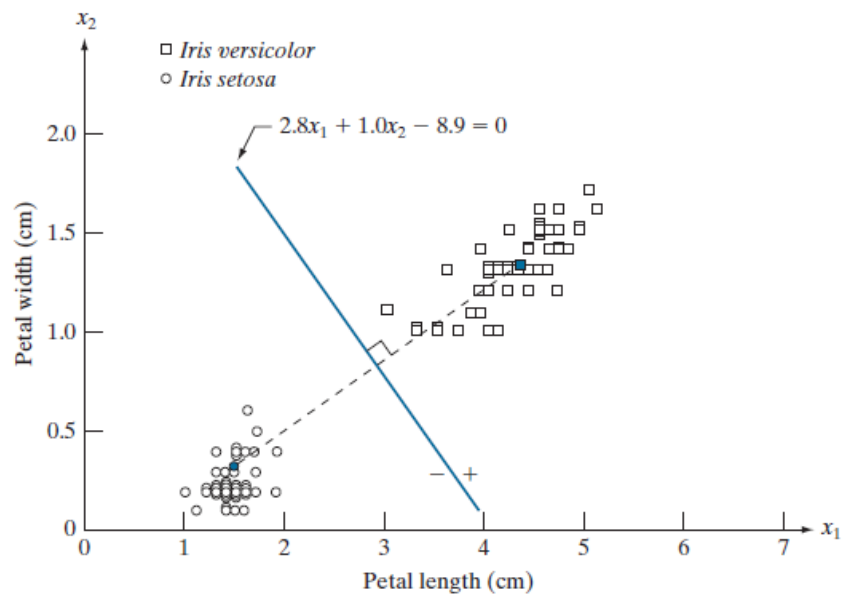
and

$$\begin{aligned} d_2(\mathbf{x}) &= \mathbf{m}_2^T \mathbf{x} - \frac{1}{2} \mathbf{m}_2^T \mathbf{m}_2 \\ &= 1.5x_1 + 0.3x_2 - 1.17 \end{aligned}$$

From Eq. (12-8), the equation of the boundary is

$$\begin{aligned} d_{12}(\mathbf{x}) &= d_1(\mathbf{x}) - d_2(\mathbf{x}) \\ &= 2.8x_1 + 1.0x_2 - 8.9 = 0 \end{aligned}$$

FIGURE 12.10
Decision boundary of a minimum distance classifier (based on two measurements) for the classes of Iris versicolor and Iris setosa. The dark dot and square are the means of the two classes.



PATTERN CLASSIFICATION USING MINIMUM-DISTANCE CLASSIFIER

The **minimum-distance classifier** performs effectively when the **distance between the mean vectors** of pattern classes is significantly greater than the **variation or spread** within each class around its mean. This implies that the classes are **tightly clustered** and **well separated**, leading to improved recognition accuracy.

In cases where each class's distribution forms a **spherical "hypercloud"** in an **n-dimensional pattern space**, the classifier achieves **optimal performance**. This means that under such conditions, the **average misclassification loss** is minimized, making this method particularly suitable for pattern recognition tasks.

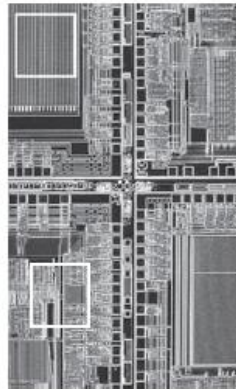
A critical factor influencing the classifier's success is the **quality of the features** used for classification. **Well-chosen features** that offer high discrimination between classes significantly enhance the classifier's accuracy. For the minimum-distance method, this specifically translates to having **clearly separated class means** and **minimal within-class variation**.

A classic application of effective feature engineering with a simple classifier is demonstrated by the **Banker's Association E-13B font system**. In the **1940s**, bank checks were processed **manually**, which was inefficient and error-prone. With rising volumes in the **1950s**, banks sought automation. The E-13B font, developed in the **mid-1950s**, became the standard solution.

This font consists of **14 stylized characters** arranged in a **9×7 grid**, designed specifically to **maximize inter-character differences**. While being **compact and human-readable**, the primary goal was **machine readability with high speed and accuracy**. The success of this system illustrates how **engineered features**, even with a **simple classifier**, can lead to **high-performance recognition systems**.

FIGURE 12.5

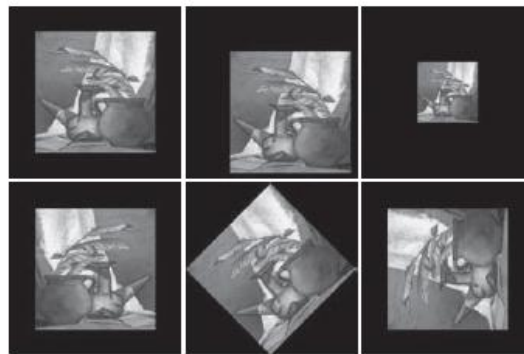
An example of pattern vectors based on properties of subimages. See Table 11.3 for an explanation of the components of $\mathbf{x}$.



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

x_1 = max probability
 x_2 = correlation
 x_3 = contrast
 x_4 = uniformity
 x_5 = homogeneity
 x_6 = entropy

FIGURE 12.6 Feature vectors with components that are invariant to transformations such as rotation, scaling, and translation. The vector components are moment invariants.



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \\ \phi_6 \\ \phi_7 \end{bmatrix}$$

The ϕ 's are moment invariants

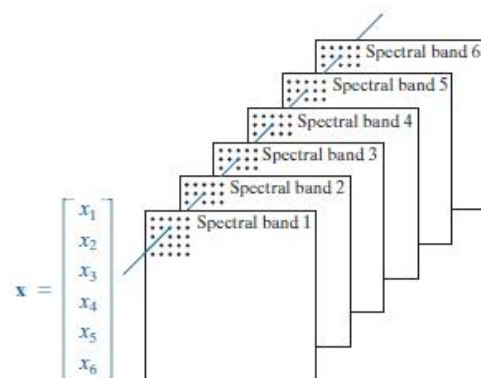
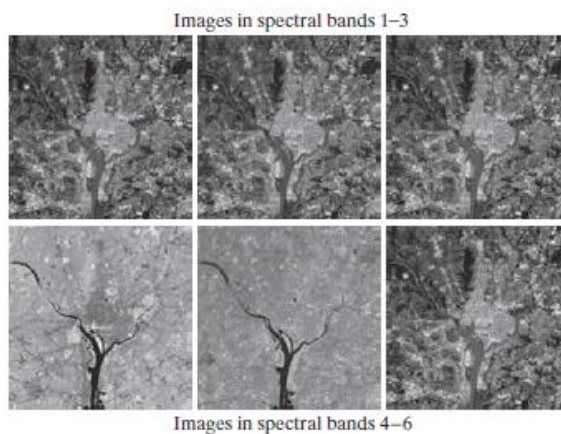


FIGURE 12.7 Pattern (feature) vectors formed by concatenating corresponding pixels from a set of registered images. (Original images courtesy of NASA.)

FIGURE 12.8
Symbol string
generated from
a polygonal
approximation of
the boundaries of
medicine bottles.

