8
Morphological processing

8.1 Introduction

The word morphology signifies the study of *form* or *structure*. In image processing we use mathematical morphology as a means to identify and extract meaningful image descriptors based on properties of form or shape within the image. Key areas of application are segmentation together with automated counting and inspection. Morphology encompasses a powerful and important body of methods which can be precisely treated mathematically within the framework of *set theory*. While this set-theoretic framework does offer the advantages associated with mathematical rigour, it is not readily accessible to the less mathematically trained reader and the central ideas and uses of morphology can be much more easily grasped through a practical and intuitive discussion. We will take such a pragmatic approach in this chapter.

Morphological operations can be applied to images of all types, but the primary use for morphology (or, at least, the context in which most people will *first* use it) is for processing *binary* images and the key morphological operators are the relatively simple ones called *dilation* and *erosion*. It is in fact possible to show that many more sophisticated morphological procedures can be reduced to a sequence of dilations and erosions. We will, however, begin our discussion of morphological processing on binary images with some preliminary but important definitions.

8.2 Binary images: foreground, background and connectedness

A binary image is an image in which each pixel assumes one of only two possible discrete, logical values: 1 or 0 (see Section 3.2.3). The logical value 1 is variously described as 'high', 'true' or 'on', whilst logical value 0 is described as 'low', 'false' or 'off'. In image processing, we refer to the pixels in a binary image having logical value 1 as the image *foreground* pixels, whilst those pixels having logical value 0 are called the image *background* pixels. An *object* in a binary image consists of any group of *connected* pixels. Two definitions of connection between pixels are commonly used. If we require that a given foreground pixel must have at least one neighbouring foreground pixel to the north, south, east or west of

Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab Chris Solomon and Toby Breckon
 ISBN: 978-0-470-84472-4

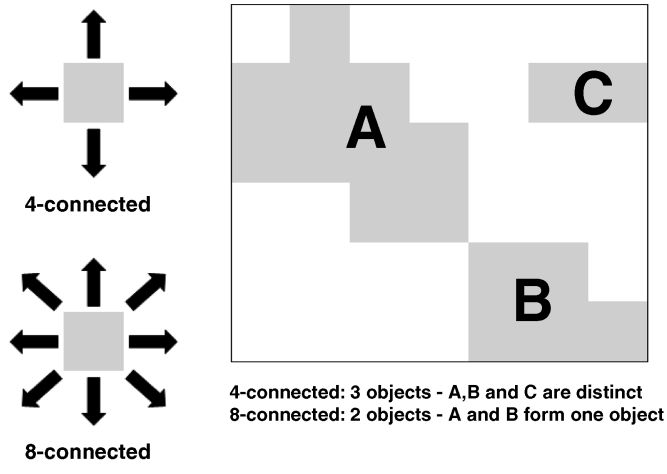

Figure 8.1 The binary image depicted above contains two objects (groups of connected pixels) under 8-connectedness but three objects under 4-connectedness

itself to be considered as part of the same object, then we are using 4-connection. If, however, a neighbouring foreground pixel to the north-east, north-west, south-east or south-west is sufficient for it to be considered as part of the same object, then we are using 8-connection. These simple concepts are illustrated in Figure 8.1.

By their very nature, binary images have no textural (i.e. grey-scale or colour) content; thus, the only properties of interest in binary images are the *shape*, *size* and *location* of the objects in the image. Morphological operations can be extended to grey-scale and colour images, but it is easier, at least initially, to think of morphological operations as *operating on a binary image input to produce a modified binary image output*. From this perspective, the effect of any morphological processing reduces simply to the *determination of which foreground pixels become background and which background pixels become foreground*.

Speaking quite generally, whether or not a given foreground or background pixel has its value changed depends on three things. Two of them are the *image* and the *type of morphological operation* we are carrying out. These two are rather obvious. The third factor is called the structuring element and is a key element in any morphological operation. The structuring element is the entity which determines exactly which image pixels surrounding the given foregound/background pixel must be considered in order to make the decision to change its value or not. The particular choice of structuring element is central to morphological processing.

8.3 Structuring elements and neighbourhoods

A structuring element is a rectangular array of pixels containing the values either 1 or 0 (akin to a small binary image). A number of example structuring elements are depicted in Figure 8.2. Structuring elements have a designated *centre pixel*. This is located at the true

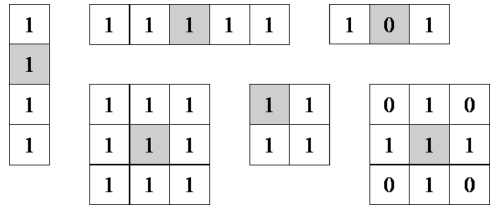

Figure 8.2 Some examples of morphological structuring elements. The centre pixel of each structuring element is shaded

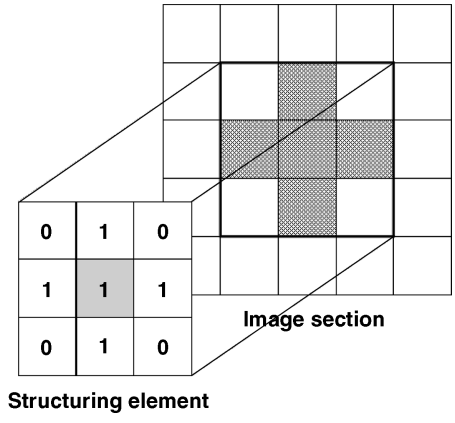

Figure 8.3 The local neighbourhood defined by a structuring element. This is given by those shaded pixels in the image which lie beneath the pixels of value 1 in the structuring element

centre pixel when both dimensions are odd (e.g. in 3×3 or 5×5 structuring elements). When either dimension is even, the centre pixel is chosen to be that pixel north, north-west or west (i.e. *above and/or to the left)* of the geometric centre (thus, a 4×3, 3×4 and a 4×4 structuring element would all have centre pixels at location [2,2]).[1] If we visualize the centre pixel of the structuring element being placed directly above the pixel under consideration in the image, then the *neighbourhood* of that pixel is determined by those pixels which lie underneath those pixels having value 1 in the structuring element. This is illustrated in Figure 8.3.

In general, structuring elements may consist of ones and zeros so as to define any neighbourhood we please, but the practicalities of digital image processing mean that they must be padded with zeros in an appropriate fashion to make *them rectangular in shape* overall. As we shall see in the examples and discussion that follow, much of the art in morphological processing is to *choose the structuring element so as to suit the particular application or aim* we have in mind.

[1] In Matlab, the coordinates of the structuring element centre pixel are defined by the expression ***floor((size (nhood) + 1)/2)***, where ***nhood*** is the structuring element.

8.4 Dilation and erosion

The two most important morphological operators are dilation and erosion. All other morphological operations can be defined in terms of these primitive operators. We denote a general image by A and an arbitrary structuring element by B and speak of the erosion/dilation of A by B.

Erosion To perform erosion of a binary image, we successively place the centre pixel of the structuring element on each foreground pixel (value 1). If *any* of the neighbourhood pixels are background pixels (value 0), then the foreground pixel is switched to background. Formally, the erosion of image A by structuring element B is denoted $A \ominus B$.
Dilation To perform dilation of a binary image, we successively place the centre pixel of the structuring element on each background pixel. If *any* of the neighbourhood pixels are foreground pixels (value 1), then the background pixel is switched to foreground. Formally, the dilation of image A by structuring element B is denoted $A \oplus B$.

The mechanics of dilation and erosion operate in a very similar way to the convolution kernels employed in spatial filtering. The structuring element slides over the image so that its centre pixel successively lies on top of each foreground or background pixel as appropriate. The new value of each image pixel then depends on the values of the pixels in the neighbourhood defined by the structuring element. Figure 8.4 shows the results of dilation

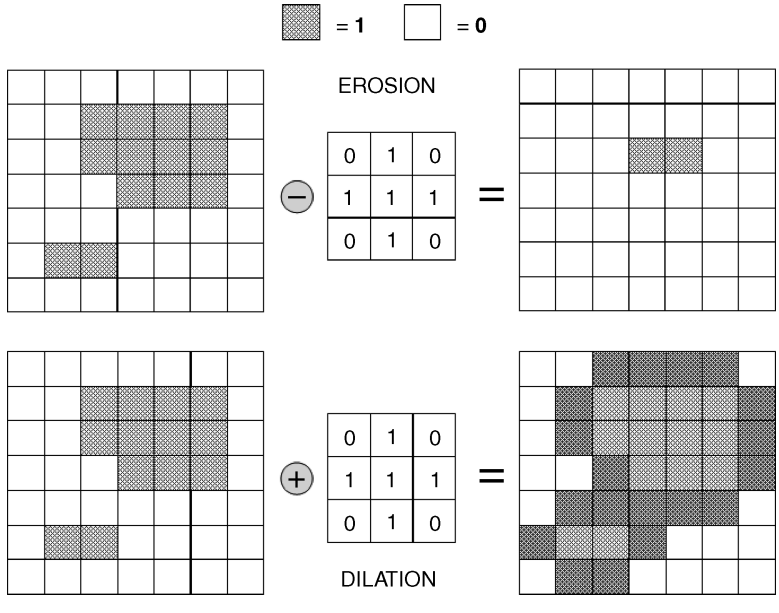

Figure 8.4 The erosion and dilation of a simple binary image. *Erosion*: a foreground pixel only remains a foreground pixel if the 1s in the structuring element (in this example, a cross) are *wholly contained* within the image foreground. If not, it becomes a background pixel. *Dilation*: a background pixel only remains a background pixel if the 1s in the structuring element are wholly contained within the image background. If not, it becomes a foreground pixel. The foreground pixels are shaded and the background pixels are clear. In the diagram demonstrating dilation, the newly created foreground pixels are shaded *darker* to differentiate them from the original foreground pixels

and erosion on a simple binary image. The foreground pixels are shaded and the background pixels are clear. In the diagram demonstrating dilation, the newly created foreground pixels are shaded darker to differentiate them from the original foreground pixels. Note that whenever the structuring element goes over the boundary of the image, we only consider that part of the neighbourhood that lies within the boundary of the image.

There is a powerful and simple way of visualizing the outcome of applying erosion or dilation to an image. For erosion, consider all those locations within the image at which the structuring element can be placed and still remain wholly contained within the image foreground. Only the pixels at these positions will survive the erosion and thus constitute the eroded image. We can consider dilation in an entirely analogous way; namely, we consider those locations at which the structuring element can be placed so as to remain entirely within the image background. Only these pixels will remain background pixels after the dilation and form the dilated image.

8.5 Dilation, erosion and structuring elements within Matlab

In Matlab, we can carry out image erosion and dilation using the Image Processing Toolbox functions ***imerode*** and ***imdilate***. The following simple examples, Examples 8.1 and 8.2, illustrate their use.

Example 8.1

Matlab code	**What is happening?**
>>bw = imread('text.png');	%Read in binary image
>>se=[0 1 0; 1 1 1; 0 1 0];	%Define structuring element
>>bw_out=imdilate(bw,se);	%Erode image
>>subplot(1,2,1), imshow(bw);	%Display original
>>subplot(1,2,2), imshow(bw_out);	%Display dilated image

Comments

Matlab functions: ***imdilate***.

In this example, the structuring element is defined explicitly as a 3×3 array.

The basic syntax requires the image and structuring element as input to the function and the dilated image is returned as output.

Example 8.2

Matlab code	**What is happening?**
>>bw = imread('text.png');	%Read in binary image
>>se=ones(6,1);	%Define structuring element
>>bw_out=imerode(bw,se);	%Erode image
>>subplot(1,2,1), imshow(bw);	%Display original
>>subplot(1,2,2), imshow(bw_out);	%Display eroded image

Comments

Matlab functions: ***imerode.***

In this example, the structuring element is defined explicitly as a 6×1 array.

Explicitly defining the structuring element as in Examples 8.1 and 8.2 is a perfectly acceptable way to operate. However, the best way to define the structuring element is to use the Image Processing Toolbox function ***strel***. The basic syntax is

$$\gg \text{se} = \text{strel}(\text{'}\textit{shape}\text{'}, \text{'}\textit{parameters}\text{'})$$

where *shape* is a string that defines the required shape of the structuring element and *parameters* is a list of parameters that determine various properties, including the size. Example 8.3 shows how structuring elements generated using ***strel*** are then supplied along with the target image as inputs to ***imdilate*** and ***imerode***.

Example 8.3

Matlab code	**What is happening?**
bw = imread('text.png');	%Read in binary image
se1 = strel('square',4)	%4-by-4 square
se2 = strel('line',5,45)	%line, length 5, angle 45 degrees
bw_1=imdilate(bw,se1);	%Dilate image
bw_2=imerode(bw,se2);	%Erode image
subplot(1,2,1), imshow(bw_1);	%Display dilated image
subplot(1,2,2), imshow(bw_2);	%Display eroded image

Comments

Matlab functions: ***strel***.

This example illustrates the use of the Image Processing Toolbox function ***strel*** to define two different types of structuring element and demonstrates their effects by dilating and eroding an image.

Type *doc strel* at the Matlab prompt for full details of the structuring elements that can be generated.

Use of the image processing toolbox function ***strel*** is recommended for two reasons. First, all the most common types of structuring element can be specified directly as input and it is rare that one will encounter an application requiring a structuring element that cannot be specified directly in this way. Second, it also ensures that the actual computation of the dilation or erosion is carried out in the *most efficient way*. This optimal efficiency relies on the principle that dilation by large structuring elements can often actually be achieved by successive dilation with a sequence of smaller structuring elements, a process known as *structuring element decomposition*. Structuring element decomposition results in a computationally more efficient process and is performed automatically whenever function ***strel*** is called.

8.6 Structuring element decomposition and Matlab

It is important to note from these examples that the function ***strel*** *does not return a normal Matlab array*, but rather an entity known as a ***strel*** *object*. The ***strel*** object is used to enable

the *decomposition* of the structuring element to be stored together with the desired structuring element.

The example below illustrates how Matlab displays when a ***strel*** object is created:

```
>> se3 = strel('disk',5');     % A disk of radius 5

se3 =

Flat STREL object containing 69 neighbors.

Decomposition: 6 STREL objects containing a total of 18
  neighbors

Neighborhood:

0  0  1  1  1  1  1  0  0
0  1  1  1  1  1  1  1  0
1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1
0  1  1  1  1  1  1  1  0
0  0  1  1  1  1  1  0  0
```

This is basically telling us two things:

- The neighbourhood is the structuring element that you have created and will effectively be applied; it has 69 nonzero elements (neighbours).

- The decomposition of this neighbourhood (to improve the computational efficiency of dilation) results in six smaller structuring elements with a total of only 18 nonzero elements. *Thus, successive dilation by each of these smaller structuring elements will achieve the same result with less computation*. Since the computational load is proportional to the number of nonzero elements in the structuring element, we expect the execution time to be approximately 18/69 of that using the original form. This is the whole purpose of structuring element decomposition.

If required, the function ***getsequence*** can be applied to find out what precisely these smaller structuring elements are. Try

```
>> decomp=getsequence(se3); whos decomp
```

The structuring elements of the decomposition can be accessed by indexing into decomp; e.g.:

```
>> decomp(1)
```

produces an output

```
ans =

Flat STREL object containing 3 neighbors.

Neighborhood:

1

1

1
```

```
>> decomp(2)
```

produces an output

```
ans =

Flat STREL object containing 3 neighbors.

Neighborhood:

1  0  0

0  1  0
```

and so on.

When you supply the strel object returned by the function ***strel*** to the functions ***imdilate*** and ***imopen***, the decomposition of the structuring element is automatically performed. Thus, in the majority of cases, there is actually no practical need to concern oneself with the details of the structuring element decomposition.

8.7 Effects and uses of erosion and dilation

It is apparent that erosion reduces the size of a binary object, whereas dilation increases it. Erosion has the effect of removing small isolated features, of breaking apart thin, joining regions in a feature and of reducing the size of solid objects by 'eroding' them at the boundaries. Dilation has an approximately reverse effect, broadening and thickening narrow regions and growing the feature around its edges. Dilation and erosion are the reverse of each other, although they are not inverses in the strict mathematical sense. This is because we cannot restore by dilation an object which has previously been completely removed by erosion. Whether it is dilation or erosion (or both) that is of interest, we stress that the appropriate choice of structuring element is often crucial and will depend strongly

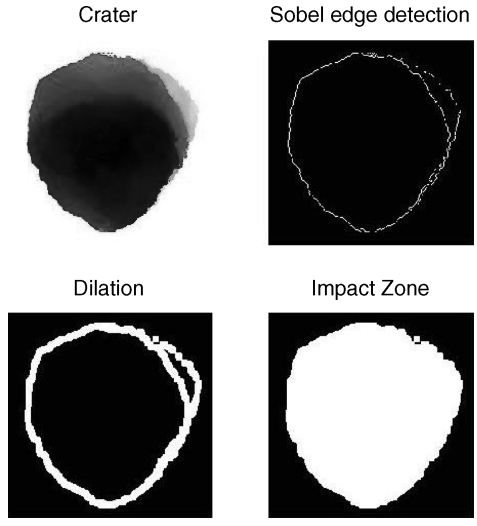

Figure 8.5 Illustrating a simple use of dilation to join small breaks in a defining contour (image courtesy of C.J. Solomon, M. Seeger, L. Kay and J. Curtis, 'Automated compact parametric representation of impact craters' *Int. J. Impact Eng.*, vol. 21, no. 10, 895–904 (1998))

on the application. In particular, *we generally seek to contrive structuring elements which are sensitive to specific shapes or structures* and, therefore, identify, enhance or delete them.

One of the simplest applications of dilation is to join together broken lines which form a contour delineating a region of interest. For example, the effects of noise, uneven illumination and other uncontrollable effects often limit the effectiveness of basic segmentation techniques (e.g. edge detection). In our chosen example, taken from the field of space debris studies, a 3-D depth map of an impact crater caused by a high-velocity, micro-sized aluminium particle has been obtained with a scanning electron microscope. This is displayed at the top left in Figure 8.5. The binary image (top right in Figure 8.5) resulting from edge detection using a Sobel kernel (Section 4.5.2) is reasonably good, but there are a number of breaks in the contour defining the impact region. In this particular example, the aim was to define the impact region of such craters *automatically* in order to express the 3-D depth map as an expansion over a chosen set of 2-D orthogonal functions. We can achieve this task in three simple steps.

(1) Dilate the edge map until the contour is closed.

(2) Fill in the background pixels enclosed by the contour. This is achieved by a related morphological method called *region filling* which is explained in Section 8.11.

(3) Erode the image (the same number of times as we originally dilated) to maintain the overall size of the delineated region.

Our second example is illustrated in Figure 8.6 (which is produced by the Matlab code produced in Example 8.4) and demonstrates the importance of choosing an appropriate structuring element for the given task at hand. Figure 8.6a depicts the layout of a printed

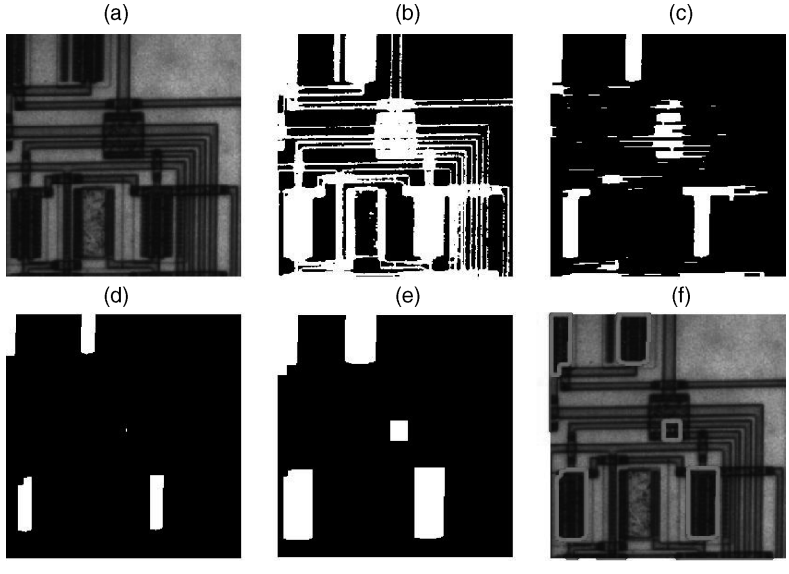

Figure 8.6 Using dilation and erosion to identify features based on shape: (a) original; (b) result after thresholding; (c) After erosion with horizontal line. (d) after erosion with vertical line; (e) after dilation with same vertical and horizontal lines; (f) boundary of remaining objects superimposed on original

Example 8.4

Matlab code

```
>>length=18; tlevel=0.2;

>>A=imread('circuit.tif'); subplot(2,3,1), imshow(A)
>>B=im2bw(A,tlevel); subplot(2,3,2), imshow(~B);

>>SE=ones(3,length); bw1=imerode(~B,SE);
>>subplot(2,3,3), imshow(bw1);
>>bw2=imerode(bw1,SE'); subplot(2,3,4), imshow(bw2);
>>bw3=imdilate(bw2,SE');bw4=imdilate(bw3,SE);
>>subplot(2,3,5), imshow(bw4);
>>boundary=bwperim(bw4);[i,j]=find(boundary);
>> subplot(2,3,6), imshow(A); hold on; plot(j,i,'r.');
```

What is happening?

```
%Define SE and percent
   threshold level
%Read image and display
%Threshold image and
   display
%Erode vertical lines
%Display result
%Erode horizontal lines
%Dilate back
%Display
%Superimpose boundaries
```

Comments

Matlab functions: ***bwperim***.

This function produces a binary image consisting of the boundary pixels in the input binary image.

circuit in which there are two types of object: the rectangular chips and the horizontal and vertical conduction tracks. The aim is to identify (i.e. segment) the chips automatically. The microprocessor chips and the tracks are darker than the background and can be identified reasonably well by simple intensity thresholding (see Figure 8.6b). The thin *vertical* tracks can first be removed by eroding with a suitable long *horizontal* structuring element. As is evident (see Figure 8.6c), the horizontal structuring element used here (a 3×18 array of 1s) tends to preserve horizontal lines but remove thin vertical ones. Analogously, we can remove horizontal lines (see Figure 8.6d) using an appropriate vertical structuring element (an 18×3 array of 1s). These two erosions tend to remove most of the thin horizontal and vertical lines. They leave intact the rectangular chips but have significantly reduced their size. We can remedy this by now dilating twice with the same structuring elements (Figure 8.6e). The boundaries of the structures identified in this way are finally superimposed on the original image for comparison (Figure 8.6f). The result is quite good considering the simplicity of the approach (note the chip in the centre was not located properly due to poor segmentation from thresholding).

8.7.1 Application of erosion to particle sizing

Another simple but effective use of erosion is in granulometry - the counting and sizing of granules or small particles. This finds use in a number of automated inspection applications. The typical scenario consists in having a large number of particles of different size but approximately similar shape and the requirement is to rapidly estimate the probability distribution of particle sizes. In automated inspection applications, the segmentation step in which the individual objects are identified is often relatively straightforward as the image background, camera view and illumination can all be controlled.

We begin the procedure by first counting the total number of objects present. The image is then repeatedly eroded using the same structuring element until no objects remain. At each step in this procedure, we record the total number of objects F which have been removed from the image as a function of the number of erosions n. Now it is clear that the number of erosions required to make an object vanish is *directly proportional* to its size; thus, if the object disappears at the kth erosion we can say that its size $X \approx \alpha k$, where α is some proportionality constant determined by the structuring element.[2] This process thus allows us to form an estimate of $F(X)$ which is essentially a cumulative distribution function (CDF) of particle size, albeit unnormalized.

Elementary probability theory tells us that the CDF and the probability density function (PDF) are related as:

$$F(X) = \int_{-\infty}^{X} p(x)\,\mathrm{d}x \qquad p(x) = \left.\frac{\mathrm{d}F}{\mathrm{d}X}\right|_{x}$$

[2] This is not an exact relationship as we have not properly defined 'size'. The astute reader will recognize that its accuracy depends on the shape of the objects and of the structuring element. However, provided the structuring element is of the same approximate shape as the particles (a normal condition in such applications), it is a good approximation.

Thus, by differentiating $F(X)$, we may obtain an estimate of the particle size distribution. As the structuring element is necessarily of finite size in a digital implementation, X is a discrete variable. Therefore, it can be useful in certain instances to form a smooth approximation to $F(X)$ by a fitting technique before differentiating to estimate the PDF.

Morphological operations can be implemented very efficiently on specialist hardware, and so one of the primary attractions of this simple procedure is its speed. Figure 8.7 illustrates the essential approach using a test image. The Matlab code which generates Figure 8.7 is given in Example 8.5 (note that the estimated distribution and density functions remain *unnormalized* in this code)

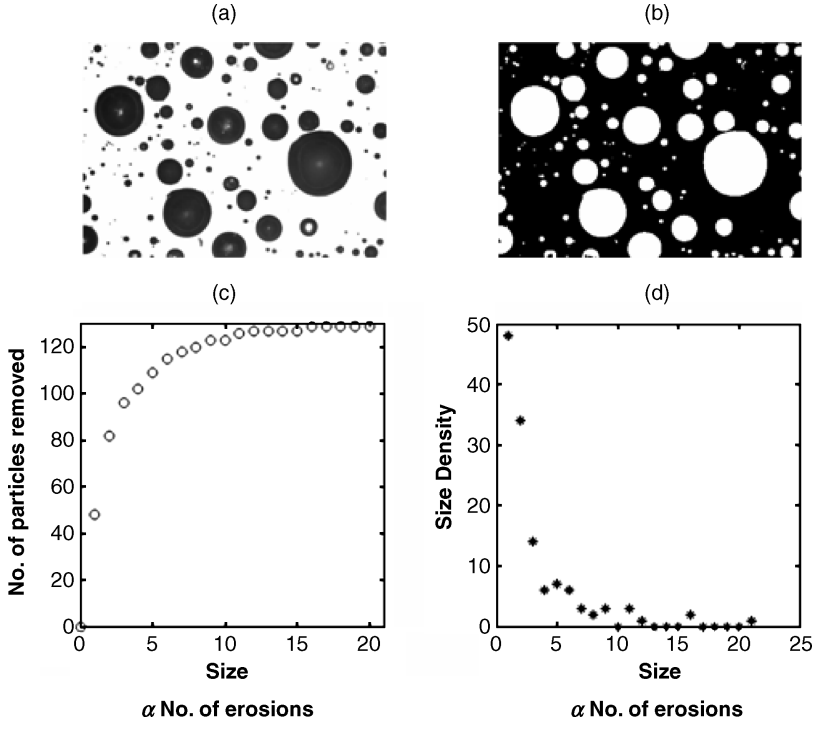

Figure 8.7 Using repeated erosion and object counting to estimate the distribution of particle sizes: (a) original image; (b) binary image resulting from intensity thresholding; (c) estimated cumulative distribution (unnormalized); (d) estimated size density function (unnormalized)

Example 8.5

Matlab code	**What is happening?**
A=imread('enamel.tif'); subplot(1,3,1), imshow(A);	%Read in image and display
bw=~im2bw(A,0.5); bw = imfill(bw,'holes');	%Threshold and fill in holes
subplot(1,3,2), imshow(bw);	%Display resulting binary image
[L,num_0]=bwlabel(bw);	%Label and count number of objects %in binary image

```
se=strel('disk',2);                          %Define structuring element,
                                               radius=2
count =0;                                    %Set number of erosions = 0
num=num_0;                                   %Initialise number of objects in
                                               image

while num>0                                  %Begin iterative erosion
count=count + 1
   bw=imerode(bw,se);                        %Erode
   [L,num]=bwlabel(bw);                      %Count and label objects
   P(count)=num_0-num;                       %Build discrete distribution
   figure(2); imshow(bw); drawnow;           %Display eroded binary image
end

   figure(2); subplot(1,2,1), plot(0:count,[0 P],'ro');   %Plot Cumulative distribution
   axis square;axis([0 count 0 max(P)]);     %Force square axis
   xlabel('Size'); ylabel('Particles removed')   %Label axes
   subplot(1,2,2), plot(diff([0 P]),'k*'); axis square;   %Plot estimated size density
                                                            function
```

Comments

Matlab functions: ***bwlabel***, ***imfill***.

The function ***bwlabel*** analyses an input binary image so as to identify all the connected components (i.e. objects) in the image. It returns a so-called labelled image in which all the pixels belonging to the first connected component are assigned a value 1, all the pixels belonging to the second connected component are assigned a value 2 and so on. The function ***imfill*** identifies the pixels constituting holes within connected components and fills them (ie. sets their value to 1).

8.8 Morphological opening and closing

Opening is the name given to the morphological operation of *erosion followed by dilation with the same structuring element*. We denote the opening of A by structuring element B as $A \circ B = (A \ominus B) \oplus B$.

The general effect of opening is to remove small, isolated objects from the foreground of an image, placing them in the background. It tends to smooth the contour of a binary object and breaks narrow joining regions in an object.

Closing is the name given to the morphological operation of *dilation followed by erosion with the same structuring element*. We denote the closing of A by structuring element B as $A \bullet B = (A \oplus B) \ominus B$.

Closing tends to remove small holes in the foreground, changing small regions of background into foreground. It tends to join narrow isthmuses between objects.

On initial consideration of these operators, it is not easy to see how they can be useful or indeed why they differ from one another in their effect on an image. After all, erosion and dilation are logical opposites and superficial consideration would tempt us to conclude that it will make little practical difference which one is used first? However, their different effects stems from two simple facts.

(1) If erosion eliminates an object, *dilation cannot recover it* – dilation needs at least one foreground pixel to operate.

(2) Structuring elements can be selected to be both *large* and have arbitrary shape if this suits our purposes.

8.8.1 The rolling-ball analogy

The best way to illustrate the difference between morphological opening and closing is to use the 'rolling-ball' analogy. Let us suppose for a moment that the structuring element B is a flat (i.e. 2-D) rolling ball and the object A being opened corresponds to a simple binary object in the shape of a hexagon, as indicated in Figure 8.8. (These choices are quite arbitrary but illustrate the process well.) Imagine the ball rolling around freely within A but constrained so as to always stay inside its boundary. *The set of all points within object* A *which can be reached by* B *as it rolls around in this way belongs in the opening of* A *with* B. For a 'solid' object (with no holes), such as that depicted in Figure 8.8, the opening can be visualized as the region enclosed by the contour which is generated by rolling the ball all the way round the inner surface such that the ball always maintains contact with the boundary.

We can also use the rolling-ball analogy to define the closing of A by B. The analogy is the same, except that this time we roll structuring element B all the way around the *outer* boundary of object A. The resulting contour defines the boundary of the closed object $A \bullet B$. This concept is illustrated in Figure 8.9.

The effect of closing and opening on binary images is illustrated in Figure 8.10 (which is produced using the Matlab code in Example 8.6). Note the different results of opening and

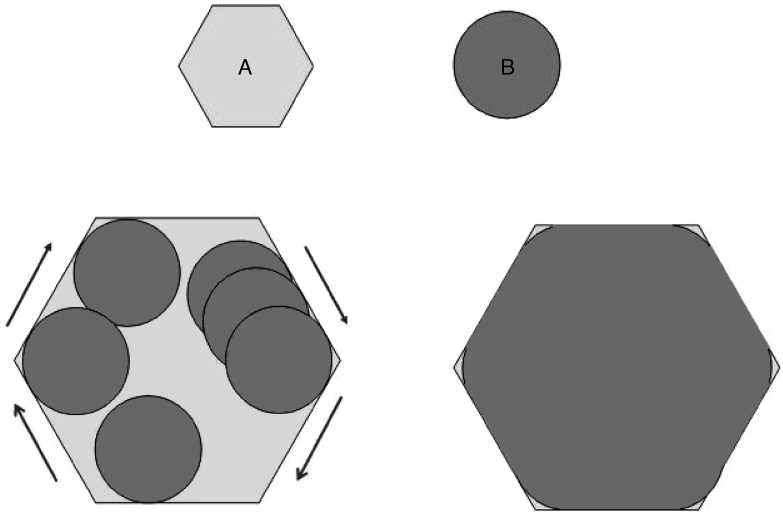

Figure 8.8 The *opening* of object A by structuring element B, $A \circ B$. This can be visualized as all possible points within object A which can be reached by moving the ball within object A without breaching the boundary. For a solid object A (no holes), the boundary of $A \circ B$ is simply given by 'rolling' B within A so as to never lose contact with the boundary. This is the circumference of the area shaded dark grey

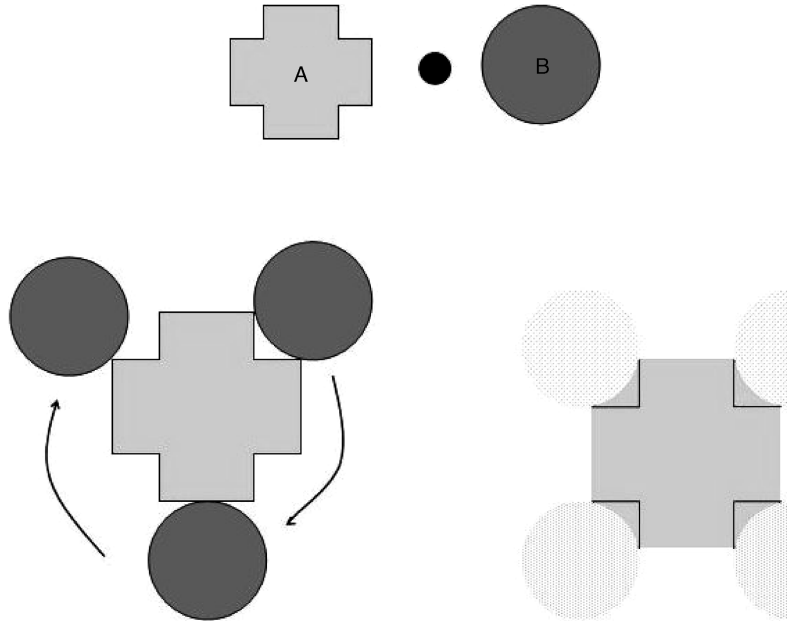

Figure 8.9 The *closing* of object *A* by structuring element *B*, $A \bullet B$. This can be visualized as all possible points contained within the boundary defined by the contour as *B* is rolled around the outer boundary of object *A*. Strictly, this analogy holds only for a 'solid' object *A* (one containing no holes)

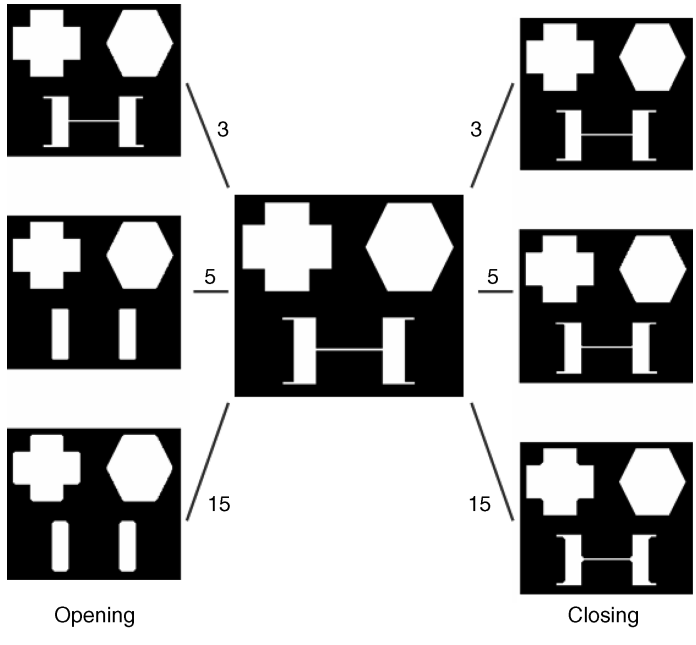

Figure 8.10 Illustrating the effects of opening and closing upon some example shapes. The original image shown in the centre was *opened* using disk-shaped structuring elements of radii 3, 5 and 15 pixels, producing the images to the left. The images to the right were similarly produced by *closing* the image using the same structuring elements. The differences become more pronounced the larger the structuring element employed

closing on the test objects and how this becomes more pronounced as the structuring element increases in size.

Example 8.6

Matlab code	What is happening?
A=imread('openclose_shapes.png'); A=~logical(A);	%Read in image and convert to binary
se=strel('disk',3); bw1=imopen(A,se); bw2=imclose(A,se);	%Define SEs then open and close
subplot(3,2,1), imshow(bw1); subplot(3,2,2), imshow(bw2);	%Display results
se=strel('disk',6); bw1=imopen(A,se); bw2=imclose(A,se);	%Define SEs then open and close
subplot(3,2,3), imshow(bw1); subplot(3,2,4), imshow(bw2);	%Display results
se=strel('disk',15); bw1=imopen(A,se); bw2=imclose(A,se);	%Define SEs then open and close
subplot(3,2,5), imshow(bw1); subplot(3,2,6), imshow(bw2);	%Display results

Comments

Matlab functions: ***imopen***, ***imclose***.

The Image Processing Toolbox provides functions to carry out morphological opening and closing on both binary and grey-scale images.

We have seen how the basic morphological operators of dilation and erosion can be combined to define opening and closing. They can also be used together with simple logical operations to define a number of practical and useful morphological transforms. In the following sections we discuss some of the most important examples.

8.9 Boundary extraction

We can define the boundary of an object by first eroding the object with a suitable small structuring element and then subtracting the result from the original image. Thus, for a binary image A and a structuring element B, the boundary A_P is defined as

$$A_P = A - A \ominus B \tag{8.1}$$

The example in Figure 8.11 (the Matlab code for which is given in Example 8.7) illustrates the process. Note that the thickness of the boundary can be controlled through the specific choice of the structuring element B in Equation (8.1).

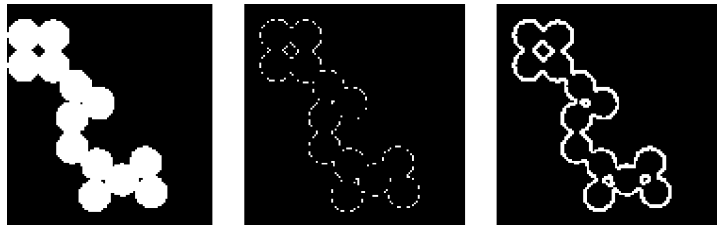

Figure 8.11 Boundary extraction. Left: original.; centre: single-pixel boundary; right: thick boundary extracted through use of larger structuring element

Example 8.7

Matlab code	**What is happening?**
A=imread('circles.png');	%Read in binary image
bw=bwperim(A);	%Calculate perimeter
se=strel('disk',5); bw1=A-imerode(A,se);	%se allows thick perimeter extraction
subplot(1,3,1), imshow(A);	
subplot(1,3,2), imshow(bw);	
subplot(1,3,3), imshow(bw)1;	%Display results

Comments

Matlab functions: ***bwperim***, ***imerode***.

Boundary extraction which is a single-pixel thick is implemented in the Matlab Image Processing Toolbox through the function ***bwperim***. Arbitrary thickness can be obtained by specifying an appropriate structuring element.

8.10 Extracting connected components

Earlier in this chapter we discussed the 4-connection and 8-connection of pixels. The set of all pixels which are connected to a given pixel is called the *connected component* of that pixel. A group of pixels which are all connected to each other in this way is differentiated from others by giving it a unique label. Typically, the process of extracting connected components leads to a new image in which the connected groups of pixels (the objects) are given sequential integer values: the background has value 0, the pixels in the first object have value 1, the pixels in the second object have value 2 and so on. To identify and label 8-connected components in a binary image, we proceed as follows:

- Scan the entire image by moving sequentially along the rows from top to bottom.

- Whenever we arrive at a foreground pixel p we examine the four neighbours of p which have already been encountered thus far in the scan. These are the neighbours (i) to the left of p, (ii) above p, (iii) the upper left diagonal and (iv) the upper right diagonal.

The labelling of p occurs as follows:

- if all four neighbours are background (i.e. have value 0), assign a new label to p; else
- if only one neighbour is foreground (i.e. has value 1), assign its label to p; else
- if more than one of the neighbours is foreground, assign one of the labels to p and resolve the equivalences.

An *alternative* method exists for extracting connected components which is based on a constrained region growing. The iterative procedure is as follows.

Let A denote our image, x_0 be an arbitary foreground pixel identified at the beginning of the procedure, x_k the set of connected pixels existing at the kth iteration and B be a structuring element. Apply the following iterative expression:

$$\begin{array}{ll} \text{Do} & x_k = (x_{k-1} \oplus B) \cap A \quad k = 1, 2, 3, \ldots \\ \text{until} & x_k = x_{k-1} \end{array} \tag{8.2}$$

The algorithm thus starts by dilating from a single pixel within the connected component using the structuring element B. At each iteration, the intersection with A ensures that no pixels are included which do not belong to the connected component. When we reach the point at which a given iteration produces the same connected component as the previous iteration (i.e. $x_k = x_{k-1}$), all pixels belonging to the connected component have been found.

This is repeated for another foreground pixel which is again arbitrary except that it must not belong to the connected components found this far. This procedure continues until all foreground pixels have been assigned to a connected component.

Extracting connected components is a very common operation, particularly in automated inspection applications. Extracting connected components can be achieved with the Matlab Image Processing Toolbox function ***bwlabel***. A simple example is given in Example 8.8 and Figure 8.12.

Example 8.8

Matlab code

```
bw=imread('basic_shapes.png');
[L,num]=bwlabel(bw);

subplot(1,2,1), imagesc(bw); axis image; axis off;
colorbar('North'); subplot(1,2,2), imagesc(L);
axis image; axis off; colormap(jet); colorbar('North')
```

What is happening?

```
%Read in image
%Get labelled image and number
%of objects
%Plot binary input image
%Display labelled image
```

Comments

The function ***bwlabel*** returns a so-called labelled image in which all the pixels belonging to the first connected component are assigned a value 1, the pixels belonging to the second connected component are assigned a value 2 and so on.

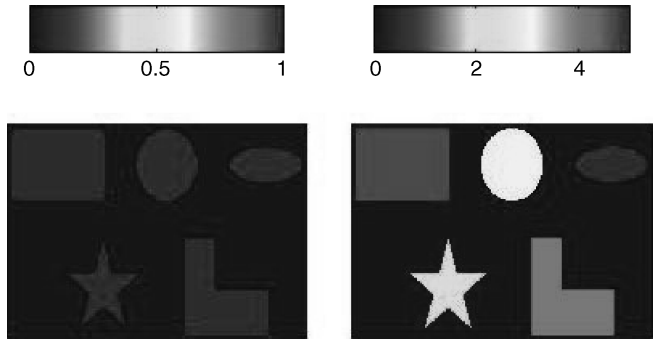

Figure 8.12 Connected components labelling. The image displayed on the left is a binary image containing five connected components or objects. The image displayed on the right is a 'label matrix' in which the first group of connected pixels is assigned a value of 1, the second group a value of 2 and so on. The false-colour bar indicates the numerical values used

8.11 Region filling

Binary images usually arise in image processing applications as the result of thresholding or some other segmentation procedure on an input grey-scale or colour image. These procedures are rarely perfect (often due to uncontrollable variations in illumination intensity) and may leave 'holes', i.e. 'background' pixels within the binary objects that emerge from the segmentation. Filling the holes within these objects is often desirable in order that subsequent morphological processing can be carried out effectively.

Assume that we have a binary object A within which lies one or more background pixels (a hole). Let B be a structuring element and let x_0 be a 'seed' background pixel lying within the hole (for the moment, we gloss over the question of how we find x_0 in the first place). Setting x_0 to be foreground (value 1) to initialize the procedure, the hole may be filled by applying the following iterative expression:

$$\begin{array}{ll} \text{Do} & x_k = (x_{k-1} \oplus B) \cap \bar{A} \quad \text{where } k = 1, 2, 3, \ldots \\ \text{until} & x_k = x_{k-1} \end{array} \tag{8.3}$$

The filled region is then obtained as $A \cup x_k$.

The algorithm thus works by identifying a point (x_0) within the hole and then growing this region by successive dilations. After each dilation, we take the intersection (logical AND) with the logical complement of the object A. Without this latter step, the filled region would simply grow and eventually occupy the entire image. Taking the intersection with the complement of A only allows the object to grow within the confines of the internal boundary. When the region has grown to the extent that it touches the boundary at all points, the next iteration will grow the region into the boundary itself. However, the intersection with $\bar{A}$ will produce the *same pixel set* as the previous iteration, at which point the algorithm stops.[3] In the final step, the union of A with the grown region gives the entire filled object.

[3] The astute reader will note that this is true provided the dilation only extends the growing region into and *not beyond* the boundary. For this reason, the structuring element must not be larger than the boundary thickness.

Now, a hole is defined as a set of connected, background pixels that cannot be reached by filling in the background from the edge of the image. So far, we have glossed over how we *formally* identify the holes (and thus starting pixels x_0 for Equation (8.3)) in the first place. Starting from a background pixel x_0 at the edge of the image, we apply exactly the same iterative expression Equation (8.3):

$$\begin{array}{ll} \text{Do} & x_k = (x_{k-1} \oplus B) \cap \bar{A} \quad \text{where } k = 1, 2, 3, \ldots \\ \text{until} & x_k = x_{k-1} \end{array}$$

The complete set of pixels belonging to all holes is then obtained as $H = (\overline{A \cup x_k})$. The seed pixels x_0 for the hole-filling procedure described by Equation (8.3) can thus be obtained by: (i) sampling arbitrarily from H; (ii) applying Equation (8.3) to fill in the hole; (iii) removing the set of filled pixels resulting from step (ii) from H; (iv) repeating (i)–(iii) until H is empty.

Region filling is achieved in the Matlab Image Processing Toolbox through the function ***imfill*** (see Example 8.2).

8.12 The hit-or-miss transformation

The hit-or-miss transform indicates the positions where a certain pattern (characterized by a structuring element B) occurs in the input image. As such, it operates as a basic tool for shape detection. This technique is best understood by an illustrative example. Consider Figure 8.13, in which we depict a binary image A alongside a designated 'target' pixel configuration B (foreground pixels are shaded). The aim is to identify all the locations within the binary image A at which the target pixel configuration defined by B can be found. It is important to stress that we are seeking the correct combination of *both* foreground (shaded) and background (white) pixels, not just the foreground, and will refer to this combination of foreground and background as the *target shape*. The reader should note that by this definition the target shape occurs at just one location within image A in Figure 8.13.

Recalling our earlier definitions, it is readily apparent that the erosion of image A by the target shape B_1 will preserve *all those pixels in image A at which the foreground pixels of the target B_1 can be entirely contained within foreground pixels* in the image. These points are indicated in Figure 8.14 as asterisks and are designated as 'hits'. In terms of our goal of finding the precise target shape B_1, the hits thus identify all locations at which the correct

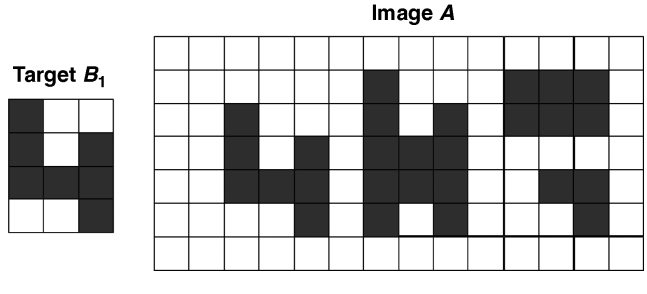

Figure 8.13 The hit-or-miss transformation. The aim is to identify those locations within an image (right) at which the specified target configuration of pixels or *target shape* (left) occurs

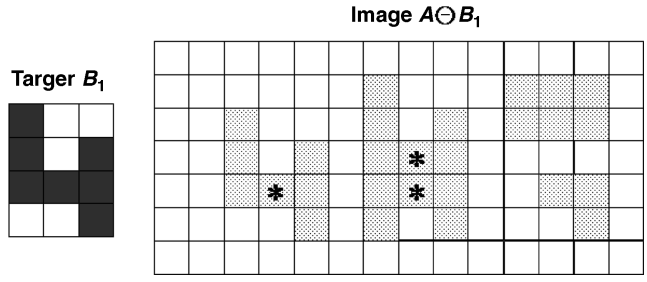

Figure 8.14 The hit-or-miss transformation. The first step is erosion of the image by the target configuration. This produces the *hits* denoted by asterisks

configuration of foreground pixels are found. However, this step does not test for the required configuration of background pixels in B_1.

Consider now Figure 8.15. This depicts the logical complement of image A together with a target pixel configuration B_2 which is the logical complement of B_1 (i.e. $B_2 = \bar{B}_1$).

Consider now applying $\bar{A} \ominus B_2$, namely erosion of the image *complement* $\bar{A}$ by the *complement* of the target shape B_2. By definition, this erosion will preserve all those pixels at which the foreground pixels of the target complement B_2 can be entirely contained within the foreground pixels of the image complement $\bar{A}$. These points are indicated in Figure 8.16 as crosses and are designated as 'misses'. Note, however, that because we have carried out the erosion using the complements of the target and image, this second step logically translates to the *identification of all those locations at which the* background *pixel configuration of the target is entirely contained within the* background *of the image*. The misses thus identify all locations at which the correct configuration of background pixels are found but does not test for the required configuration of foreground pixels.

The first step has identified all those points in the image foreground at which the required configuration of foreground pixels in the target may be found (but has ignored the required background configuration). By contrast, the second step has identified all those points in the image background at which the required configuration of background pixels in the target may be found (but has ignored the required foreground configuration). It follows, therefore, that any point identified by *both* of these steps gives a location for the target configuration. Accordingly, the third and final step in the process is to take a logical intersection (AND) of the two images. Thus, the hit-or-miss transformation of A by $\bar{B}_1$ is

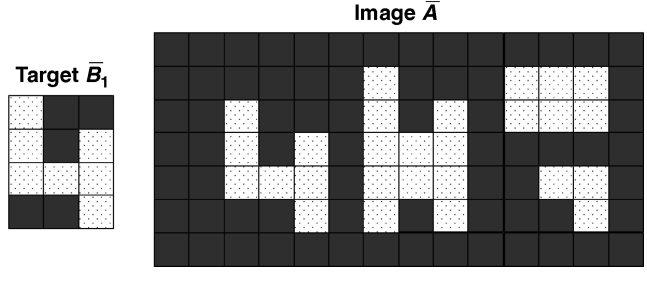

Figure 8.15 The hit-or-miss transformation. For the second step where we locate the misses, we consider the *complement* of the image $\bar{A}$ and the *complement* of the target shape $B_2 = \bar{B}_1$

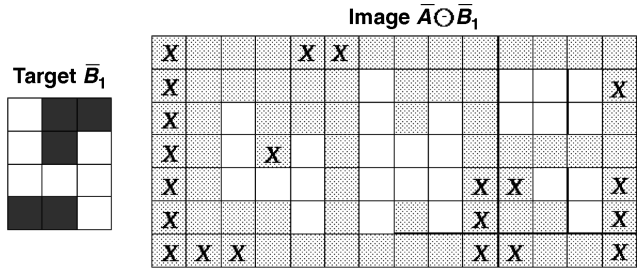

Figure 8.16 The hit-or-miss transformation. The second step consists of eroding the *complement* of the image $\bar{A}$ by the *complement* of the target shape ($B_2 = \bar{B}_1$). This produces the *misses* denoted here by crosses

formally defined as:

$$A \otimes B = (A \ominus B) \cap (\bar{A} \ominus \bar{B}) \tag{8.4}$$

The result obtained for our example is indicated in Figure 8.17. The sole surviving pixel (shaded) gives the only location of the target shape to be found in the image.

In summary then, the hit-or-miss transformation identifies those locations at which both the defining foreground and background configurations for the target shape are to be simultaneously found – in this sense, the method might more properly be called the hit-*and*-miss, but the former title is the most common. Example 8.9 and Figure 8.18 illustrate the use of the hit-and-miss transform to identify the occurrences of a target letter 'e' in a string of text.

Note that the two structuring elements B_1 and B_2 described in our account so far of the hit-or-miss transform are logical complements. This means that only *exact* examples of the target shape (including background and foreground pixel configurations) are identified. If even one pixel of the configuration probed in the image differs from the target shape, then the hit-or-miss transform we have used will not identify it (see Figure 8.19).

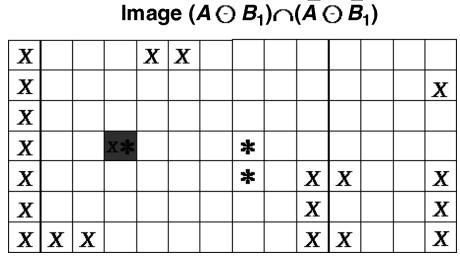

Figure 8.17 The hit-or-miss transformation. The final step takes the intersection (logical AND) of the two eroded images $(A \odot B) \cap (\bar{A} \odot \bar{B})$. Any surviving pixels give the location of the target shape in the original image A

Example 8.9

Matlab code	What is happening?
imread('text.png');	%Read in text
B=imcrop(A);	%Read in target shape interactively
se1=B; se2=~B;	%Define hit and miss structure elements
bw=bwhitmiss(A,se1,se2);	%Perform hit-miss transformation
[i,j]=find(bw==1);	%Get explicit coordinates of locations
subplot(1,3,1), imshow(A);	%Display image
subplot(1,3,2), imagesc(B); axis image; axis off;	%Display target shape
subplot(1,3,3), imshow(A); hold on; plot(j,i,'r*');	%Superimpose locations on image

Comments

Matlab functions: ***bwhitmiss***.

The Matlab Image Processing Toolbox has a purpose-made function for carrying out the hit-or-miss transformation: ***bwhitmiss***. This example illustrates its use to identify the occurrences of a target letter 'e' in a string of text.

8.12.1 Generalization of hit-or-miss

The hit-or-miss transformation given by Equation (8.4) and demonstrated in Example 8.9 was chosen to be relatively straightforward to understand. However, it is important to stress that it is actually a *special* case. The most general form of the hit-or-miss transformation is given by

$$A \otimes B = (A \ominus B_1) \cap (\bar{A} \ominus B_2) \qquad \text{where } B = (B_1, B_2) \tag{8.5}$$

$B = (B_1, B_2)$ signifies a pair of structuring elements and B_2 is a general structuring element that is *not necessarily the complement* of B_1 but which is mutually exclusive (i.e. $B_1 \cap B_2 = \varnothing$, the empty set). By allowing the second structuring element B_2 to take on forms other than the complement of B_1, we can effectively relax the constraint that the hit-or-miss operation only identifies the *exact* target shape and widen the scope of the method. This relaxation of the

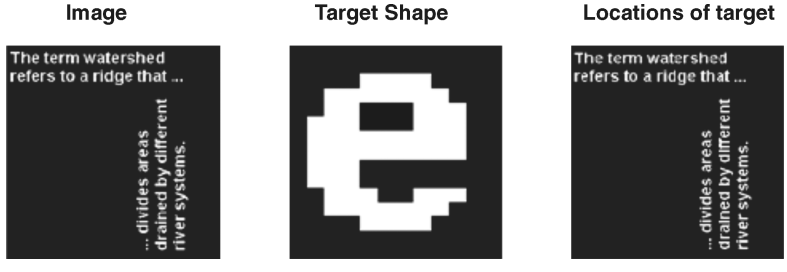

Figure 8.18 Application of the hit-or-miss transformation to detect a target shape in a string of text. Note that the target includes the background and hit-or-miss is strictly sensitive to both the scale and the orientation of the target shape (*See colour plate section for colour version*)

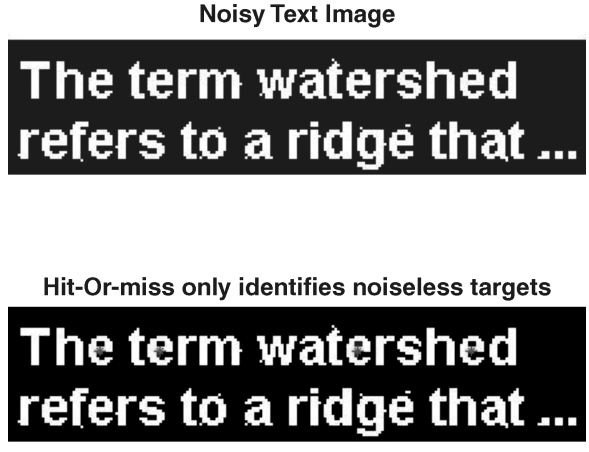

Figure 8.19 The fully constrained hit-or-miss transform is sensitive to noise and uncontrolled variations in the target feature. In the example above, three instances of the target letter "e" on the second line were missed due to the presence of a small amount of random noise in the image

method can make it less sensitive to noise and small but insignificant variations in the shape. Note that, in the limiting case where B_2 is an empty structuring element, hit-or-miss transformation reduces to simple erosion by B_1. We discuss this generalization of hit-or-miss next.

8.13 Relaxing constraints in hit-or-miss: 'don't care' pixels

It is apparent that the application of the hit-or-miss transformation as carried out in Figure 8.19 is 'unforgiving', since a discrepancy of even one pixel between the target shape and the probed region in the image will result in the target shape being missed. In many cases, it is desirable to be less constrained and allow a combination of foreground and background pixels that is 'sufficiently close' to the target shape to be considered as a match. We can achieve this relaxation of the exact match criterion through the use of 'don't care' pixels (also known as wild-card pixels). In other words, the target shape now conceptually comprises three types of pixel: strict foreground, strict background and 'don't care'. In the simple example shown in Figure 8.20 (produced using the Matlab code in Example 8.10), we consider the task of automatically locating the 'upper right corners' in a sample of shapes.

Example 8.10

Matlab code	**What is happening?**
A=imread	%CASE 1
Noisy_Two_Ls.png');	
se1=[0 0 0; 1 1 0; 0 1 0];	%SE1 defines the hits
se2=[1 1 1; 0 0 1; 0 0 1];	%SE2 defines the misses
bw=bwhitmiss(A,se1,se2);	%Apply hit-or-miss transform
subplot(2,2,1), imshow(A,[0 1]);	%Display Image
subplot(2,2,2), imshow(bw,[0 1]);	%Display located pixels
	%NOTE ALTERNATIVE SYNTAX

```
interval=[-1 -1 -1; 1 1 -1; 0 1 -1];    %1s for hits, -1 for misses; 0s for don't
                                            care
bw=bwhitmiss(A,interval);               %Apply hit-or-miss transform
subplot(2,2,3), imshow(bw,[0 1]);       %Display located pixels

                                        %CASE 2
interval=[0 -1 -1; 0 1 -1; 0 0 0];      %1s for hits, -1 for misses; 0s for don't
                                            care
bw=bwhitmiss(A,interval);               %Apply hit-or-miss transform
subplot(2,2,4), imshow(bw,[0 1]);       %Display located pixels
```

Comments

Note that bw2 = ***bwhitmiss***(bw,interval) performs the hit–miss operation defined in terms of a single array, called an 'interval'. An interval is an array whose elements can be 1, 0 or −1. The 1-valued elements specify the domain of the first structuring element SE1 (hits); the −1-valued elements specify the domain of the second structuring element SE2 (misses); the 0-valued elements, which act as 'don't care' pixels, are ignored.

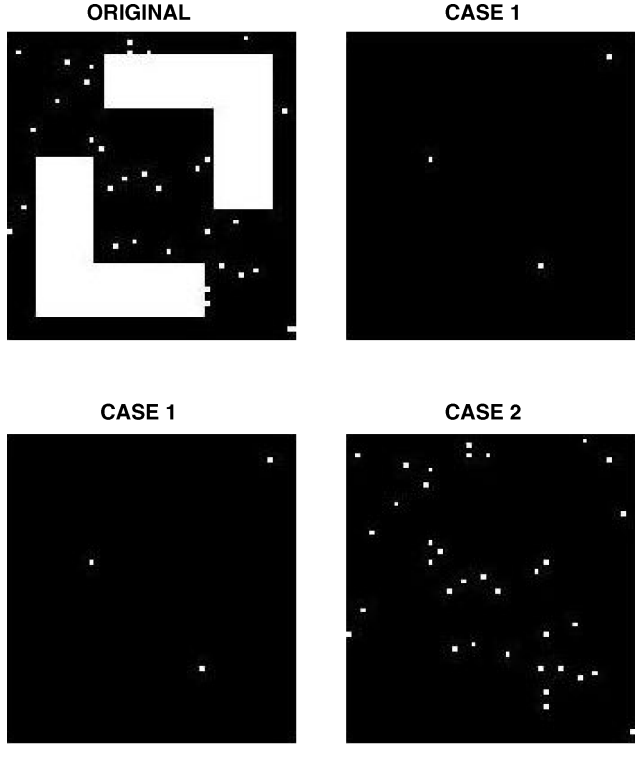

Figure 8.20 Generalizing the hit-or-miss transform. This illustrates the effect of relaxing constraints on the hit-or-miss transform. Top left: original image. Top right and bottom left (which uses an alternative computation): the result of hit-or-miss for strict definition of upper right corner pixels – only the upper right corner pixels of the solid L shapes are identified. Bottom right: in the second (relaxed) case, the noisy pixels are also identified

In the example shown in Figure 8.20, we consider two definitions for upper right corner pixels:

(1) A strict definition which requires that an 'upper right corner pixel' must have *no* neighbours to *north, north-west, north-east or east* (these are the misses) and must have neighbours to the south and west (these are the hits). The south-west neighbour is neither hit nor miss and, thus, a *don't care* pixel.

(2) A looser definition which only requires that an upper right corner pixel must have *no* neighbours to *north, north-west and north-east* (the misses). All other neighbours are 'don't care' pixels.

The differing effects of both definitions are shown in Figure 8.20 using the hit and miss transform.

8.13.1 Morphological thinning

Closely related to the hit-or-miss transformation, the thinning of an image A by a structuring element B is defined as

$$\text{thin}(A, B) = A \cap \overline{A \otimes B}$$

where $A \otimes B$ is the hit-or-miss transformation of A with B. The thinning of A with B is calculated by successively translating the origin of the structuring element to each pixel in the image and comparing it with the underlying image pixels. If both the foreground and background pixels in the structuring element exactly match those in the underlying image, then the image pixel underneath the origin of the structuring element is set to background (zero). If no match occurs, then the pixel is left unchanged. Thinning is a key operation in practical algorithms for calculating the skeleton of an image.

8.14 Skeletonization

Rather as its name implies, the *skeleton* of a binary object is a representation of the basic form of that object which has been reduced down to its minimal level (i.e. a 'bare bones' representation). A very useful conceptualization of the morphological skeleton is provided by the *prairie-fire analogy*: the boundary of an object is set on fire and spreads with uniform velocity in all directions inwards; the skeleton of the object will be defined by the points at which the fire fronts meet and quench (i.e. stop) each other.

Consider an arbitrary, binary object A. A point p within this binary object belongs to the skeleton of A if and only if the two following conditions hold:

(1) A disk D_z may be constructed, with p at its centre, that lies entirely within A and touches the boundary of A at two or more places.

(2) No other larger disk exists that lies entirely within A and yet contains D_z.

An equivalent geometric construction for producing the skeleton is to:

(1) Start at an arbitrary point on the boundary and consider the maximum possible size of disk which can touch the boundary at this point and at least one other point on the boundary and yet remain within the object.

(2) Mark the centre of the disk.

(3) Repeat this process at all points along the entire boundary (moving in infinitesimal steps) until you return to the original starting point. The trace or locus of all the disk centre points is the skeleton of the object.

Figure 8.21 shows a variety of different shapes and their calculated skeletons. Consideration of the calculated skeletons in Figure 8.21 should convince the reader that the circle is actually slightly elliptical and that the pentagon and star are not regular.

The skeleton is a useful representation of the object morphology, as it provides both topological information and numerical metrics which can be used for comparison and categorization. The topology is essentially encapsulated in the number of nodes (where branches meet) and number of end points and the metric information is provided by the lengths of the branches and the angles between them.

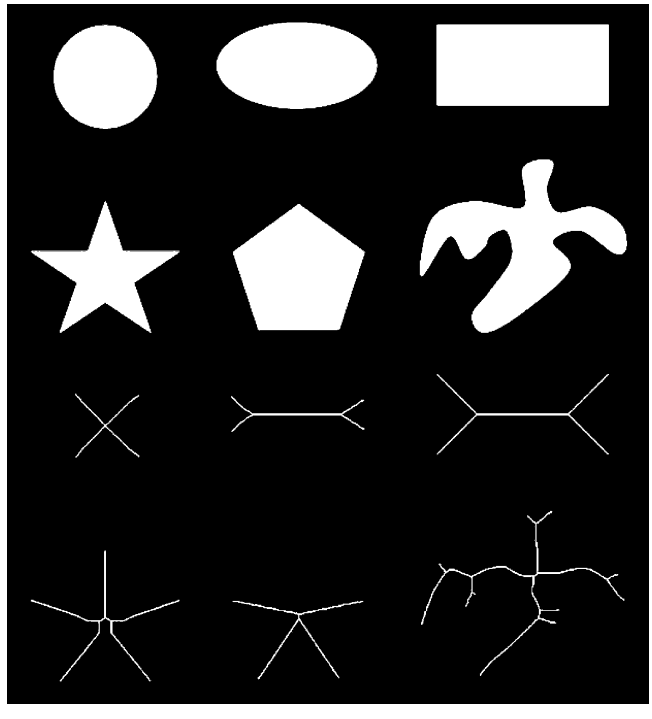

Figure 8.21 Some shapes and their corresponding morphological skeletons

A weakness of the skeleton as a representation of a shape is that it is sensitive (sometimes highly) to small changes in the morphology. Slight irregularities in a boundary can lead to spurious 'spurs' in the skeleton which can interfere with recognition processes based on the topological properties of the skeleton. So-called *pruning* can be carried out to remove spurs of less than a certain length, but this is not always effective as small perturbations in the boundary of an image can sometimes lead to large spurs in the skeleton.

Actual computation of a skeleton can proceed in a number of ways. The preferred method is based on an iterative thinning algorithm whose details we do not repeat here. As well its use in a variety of image segmentation tasks, skeletonization has been usefully applied in a number of medical applications, such as unravelling the colon and assessment of laryngotracheal stenosis.

8.15 Opening by reconstruction

One simple but useful effect of morphological opening (erosion followed by dilation) is the removal of small unwanted objects. By choosing a structuring element of a certain size, erosion with this element guarantees the removal of any objects within which that structuring element cannot be contained. The second step of dilation with the same structuring element acts to restore the surviving objects to their original dimensions. However, if we consider an arbitrary shape, opening will not *exactly* maintain the shape of the primary objects except in the simplest and most fortuitous of cases. In general, the larger the size of the structuring element and the greater the difference in shape between structuring element and object, the larger the error in the restored shape will be. The uneven effect on an object of erosion or dilation with a structuring element whose shape is different from the object itself is referred to as the *anisotropy* effect. The examples in Figure 8.22 (generated by Example 8.11) illustrate this effect.

In contrast to morphological opening, *opening by reconstruction* is a morphological transformation which enables the objects which survive an initial erosion to be *exactly* restored to their original shape. The method is conceptually simple and requires two images which are called the *marker* and the *mask*. The mask is the original binary image. The *marker image* is used as the starting point of the procedure and is, in many cases, the image obtained

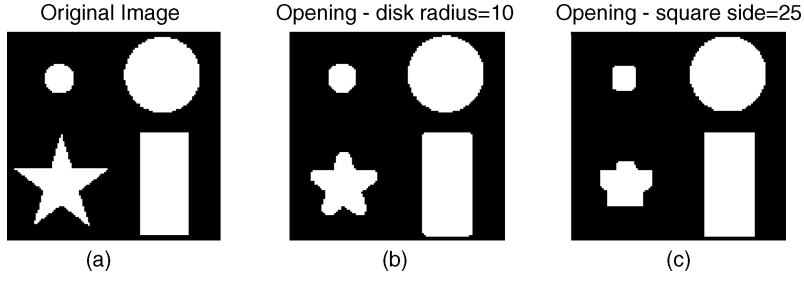

Figure 8.22 Effects of morphological opening. (a) Original binary image. (b) Result of opening using a circular structuring element of radius 10. Note the rounding on the points of the star and the corners of the rectangle. (c) Result of opening using a square structuring element of side length 25. Only the rectangle is properly restored, the other shapes are significantly distorted

Example 8.11

Matlab code	What is happening?
A=imread('open_shapes.png');	%Read in image
se=strel('disk',10); bw=imopen(A,se);	%Open with disk radius 10
subplot(1,3,1), imshow(A);	%Display original
title('Original Image');	
subplot(1,3,2), imshow(bw);	%Display opened image
title('Opening - disk radius=10');	
se=strel('square',25); bw=imopen	%Open with square side 25
(A,se);	
subplot(1,3,3), imshow(bw);	%Display opened image
title('Opening - square side=25');	

after an initial erosion of the original image. The marker image is then iteratively dilated using an elementary 3×3 structuring element with the condition that the output image at each iteration is given by the intersection (logical AND) of the marker image with the mask. In this way, *the mask constrains the marker*, never allowing foreground pixels to appear which were not present in the original image. When the output image which results from a given iteration is the same as the image resulting from the previous iteration, this indicates that the dilation is having no further effect and all objects which survived the initial erosion have been restored to their original state. The procedure then terminates. Formally, we can describe this procedure by the following simple algorithm:

- denote the marker image by A and the mask image by M;
- define a 3×3 structuring element of 1s $= B$;
- iteratively apply $A_{n+1} = (A_n \oplus B) \cap M$;
- when $A_{n+1} = A_n$, stop.

Example 8.12 and Figure 8.23 illustrate the use of morphological reconstruction. We preserve all alphabetic characters which have a long vertical stroke in a printed text sequence whilst completely removing all others.

Example 8.12

Matlab code	What is happening?
mask=~imread('shakespeare.pbm');	%Read in binary text
mask=imclose(mask,ones(5));	%Close to bridge breaks in letters
se=strel('line',40,90);	%Define vertical se length 40
marker=imerode(mask,se);	%Erode to eliminate characters
im=imreconstruct(marker,mask);	%Reconstruct image

```
subplot(3,1,1), imshow(~mask);
    title('Original mask Image');
subplot(3,1,2), imshow(~marker);
    title('marker image');
subplot(3,1,3), imshow(~im);
    title('Opening by reconstruction');
```

Comments

Opening by reconstruction is implemented in the Matlab Image Processing Toolbox through the function ***imreconstruct***.

Note that the "~" operator in this example is the logical NOT (inversion) operator being applied to binary images.

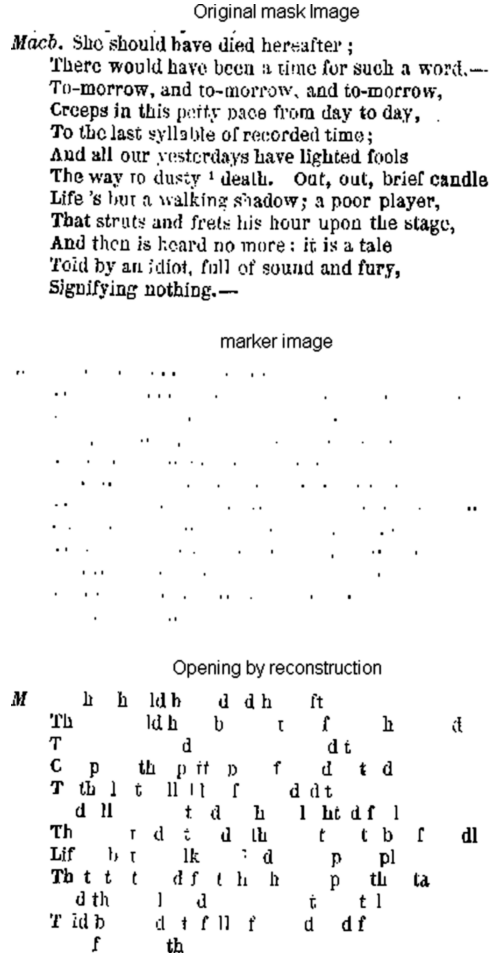

Figure 8.23 Opening by reconstruction. Top: the original text – the mask image. Middle: the text after erosion by a long, vertical structuring element. This acts as the marker image. Bottom: the image after opening by reconstruction. Only characters with long vertical strokes have been restored

8.16 Grey-scale erosion and dilation

To date, our discussion has only considered the application of morphology to binary images. The same basic principles of morphological processing can, however, be extended to intensity[4] images. Just as in the binary case, the building blocks of grey-scale morphology are the fundamental operations of erosion and dilation. For intensity images, the definitions of these operators are subtly different and can, in principle, be defined in a variety of ways. However, in practice, the following (informal) definitions are more or less universal.

Grey-scale erosion of image A by structuring element B is denoted $A \ominus B$ and the operation may be described as follows:

- successively place the structuring element B over each location in the image A;
- for each location, select the *minimum* value of $A - B$ occurring within the local neighbourhood defined by the structuring element B.

Grey-scale dilation of image A by structuring element B is denoted $A \oplus B$ and the operation may be described as follows:

- successively place the structuring element B over each location in the image;
- for each location, select the *maximum* value of $A + B$ occurring within the local neighbourhood defined by the structuring element B.

8.17 Grey-scale structuring elements: general case

Where grey-scale morphology is being considered, structuring elements have, in the most general case, two parts:

(1) An array of 0s and 1s, where the 1s indicate the domain or local neighbourhood defined by the structuring element. We denote this by b.

(2) An array of identical size containing the actual numerical values of the structuring element. We denote this by v_b.

To illustrate this general form of grey-scale erosion and dilation, consider the simple example in Example 8.13.

[4] Colour images can also be morphologically processed. Since these simply comprise three 2-D intensity planes (i.e. the three R,G and B colour channels), our discussion of grey-scale morphology will apply to these images too.

Example 8.13

A structuring element comprising a domain

$$b = \begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$$

and values

$$v_b = \begin{matrix} 0 & 3 & 0 \\ 3 & 1 & 3 \\ 0 & 3 & 0 \end{matrix}$$

is placed on a section of an image given by

$$A = \begin{matrix} 12 & 13 & 11 \\ 7 & 14 & 8 \\ 10 & 9 & 10 \end{matrix}$$

Consider positioning the central pixel of the structuring element on the centre pixel of the image segment (value = 14). What is the value after erosion?

Solution

The value of this pixel after grey-scale erosion is the minimum of the values $A - v_b$ over the domain defined by

$$b = \min\left\{\begin{matrix} - & 13-3 & - \\ 7-3 & 14-1 & 8-3 \\ - & 9-3 & - \end{matrix}\right\} = \min\left\{\begin{matrix} - & 10 & - \\ 4 & 13 & 5 \\ - & 6 & - \end{matrix}\right\} = 4$$

8.18 Grey-scale erosion and dilation with flat structuring elements

Although grey-scale morphology can use structuring elements with two parts defining the neighbourhood b and height values v_b respectively, it is very common to use *flat* structuring elements. Flat structuring elements have height values which are all zero and are thus specified entirely by their neighbourhood.

When a flat structuring element is assumed, grey-scale erosion and dilation are equivalent to *local minimum and maximum* filters respectively. In other words, erosion with a flat element results in each grey-scale value being replaced by the minimum value in the vicinity defined by the structuring element neighbourhood. Conversely, dilation with a flat element results in each grey-scale value being replaced by the maximum. Figure 8.24 shows how grey-scale erosion and dilation may be used to calculate a so-called

Figure 8.24 Calculating the morphological gradient: (a) grey-scale dilation with flat 3×3 structuring element; (b) grey-scale erosion with flat 3×3 structuring element; (c) difference of (a) and (b) = morphological gradient

morphological image gradient. The Matlab code in Example 8.14 was used to produce Figure 8.24.

Example 8.14

Matlab code	What is happening?
A=imread('cameraman.tif');	%Read in image
se=strel(ones(3));	%Define flat structuring element
Amax=imdilate(A,se);	%Grey-scale dilate image
Amin=imerode(A,se);	%Grey-scale erode image
Mgrad=Amax-Amin;	%subtract the two
subplot(1,3,1), imagesc(Amax); axis image; axis off;	%Display
subplot(1,3,2), imagesc(Amin); axis image; axis off;	
subplot(1,3,3), imagesc(Mgrad); axis image; axis off;	
colormap(gray);	

It is easy to see why the morphological gradient works: replacing a given pixel by the minimum or maximum value in the local neighbourhood defined by the structuring element will effect little or no change in smooth regions of the image. However, when the structuring element spans an edge, the response will be the difference between the maximum and minimum-valued pixels in the defined region and, hence, large. The thickness of the edges can be tailored by adjusting the size of the structuring elements if desired.

8.19 Grey-scale opening and closing

Grey-scale opening and closing are defined in exactly the same way as for binary images and their effect on images is also complementary. Grey-scale opening (erosion followed by dilation) tends to suppress small bright regions in the image whilst leaving the rest of the image relatively unchanged, whereas closing (dilation followed by erosion) tends to suppress small dark regions. We can exploit this property in the following example.

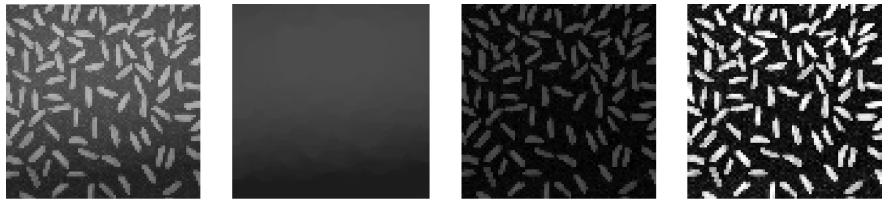

Figure 8.25 Correction of nonuniform illumination through morphological opening. Left to right: (a) original image; (b) estimate of illumination function by morphological opening of original; (c) original with illumination subtracted; (d) contrast-enhanced version of image (c)

Example 8.15

Matlab code	**What is happening?**
I = imread('rice.png');	%Read in image
background = imopen(I,strel('disk',15));	%Opening to estimate background
I2 = imsubtract(I,background);	%Subtract background
I3 = imadjust(I2);	%Improve contrast
subplot(1,4,1), imshow(I);subplot(1,4,2), imshow(background);	
subplot(1,4,3), imshow(I2);subplot(1,4,4), imshow(I3);	

Comments
imerode, ***imdilate***, ***imclose*** and ***imopen*** may be used for both binary and grey-scale images.

A relatively common problem in automated inspection and analysis applications is for the field to be unevenly illuminated. This makes segmentation (the process of separating objects of interest from their background) more difficult. In Figure 8.25 (produced using the Matlab code in Example 8.15), in which the objects/regions of interest are of similar size and separated from one another, opening can be used quite effectively to estimate the illumination function.

8.20 The top-hat transformation

The top-hat transformation is defined as the difference between the image and the image after opening with structuring element b, namely $I - I \ominus b$. Opening has the general effect of removing small light details in the image whilst leaving darker regions undisturbed. The difference of the original and the opened image thus tends to lift out the local details of the image *independently of the intensity* variation of the image as a whole. For this reason, the top-hat transformation is useful for uncovering detail which is rendered invisible by

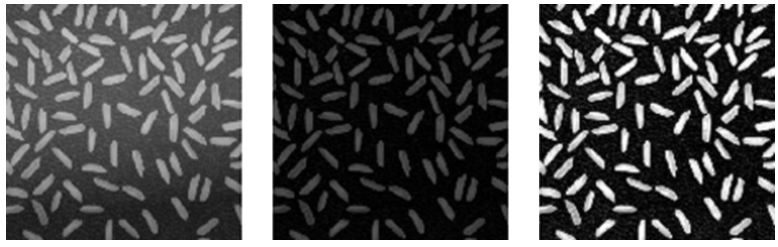

Figure 8.26 Morphological top-hat filtering to increase local image detail. Left to right: (a) original image; (b) after application of top-hat filter (circular structuring element of diameter approximately equal to dimension of grains); (c) after contrast enhancement

illumination or shading variation over the image as a whole. Figure 8.26 shows an example in which the individual elements are enhanced. The Matlab code corresponding to Figure 8.26 is given in Example 8.16.

Example 8.16

Matlab code	**What is happening?**
A = imread('rice.png');	%Read in unevenly illuminated image
se = strel('disk',12);	%Define structuring element
Atophat = imtophat(original,se);	%Apply tophat filter
subplot(1,3,1), imshow(A);	%Display original
subplot(1,3,2), imshow(Atophat);	%Display raw filtered image
B = imadjust(tophatFiltered);	%Contrast adjust filtered image
subplot(1,3,3), imshow(B);	%Display filtered and adjusted mage

Comments
imtophat is a general purpose function for top hat filtering and may be used for both binary and grey-scale images.

8.21 Summary

Notwithstanding the length of this chapter, we have presented here a relatively elementary account of morphological processing aimed at conveying some of the core ideas and methods. The interested reader can find a large specialist literature on the use of morphology in image processing. The key to successful application lies largely in the judicious combination of the standard morphological operators and transforms and intelligent choice of structuring elements; detailed knowledge of how specific transforms work is often not needed. To use an old analogy, it is possible to construct a variety of perfectly good houses

Table 8.1 A summary of some important morphological operations and corresponding Matlab functions

Operation	Matlab function	Description	Definition Image A Structuring element B
Erosion	***imerode***	If any foreground pixels in neighbourhood of structuring element are background, set pixel to background	$A \ominus B$
Dilation	***imdilate***	If any background pixels in neighbourhood of structuring element are foreground, set pixel to foreground	$A \oplus B$
Opening	***imopen***	Erode image and then dilate the eroded image using the same structuring element for both operations	$A \circ B = (A \ominus B) \oplus B$
Closing	***imclose***	Dilate image and then erode the dilated image using the same structuring element for both operations	$A \bullet B = (A \oplus B) \ominus B$
Hit-or-miss	***bwhitmiss***	Logical AND between (i) the image eroded with one structuring element and (ii) the image *complement* eroded with a second structuring element	$A \otimes B = (A \ominus B_1) \cap (A \ominus B_2)$
Top Hat	***imtophat***	Subtracts a morphologically opened image from the original image	$A - (A \ominus B) \oplus B$
Bottom Hat	***imbothat***	Subtracts the original image from a morphologically closed version of the image	$[(A \oplus B) \ominus B] - A$
Boundary extraction	***bwperim***	Subtracts eroded version of the image from the original	$A - (A \ominus B)$
Connected components labelling	***bwlabel***	Finds all connected components and labels them distinctly	See Section 8.10
Region filling	***imfill***	Fills in the holes in the image	See Section 8.11
Thinning	***bwmorph***	Subtracts the hit-or-miss transform from the original image.	$\text{thin}(A, B) = A \cap \overline{A \otimes B}$
Thickening	***bwmorph***	The original image plus additional foreground pixels switched on by the hit-and-miss transform	$\text{thicken}(A, B) = A \cup A \otimes B$
Skeletonization	***bwmorph***		See Section 8.14

without knowing how to manufacture bricks. With this in mind, we close this chapter with Table 8.1, which gives a summary of the standard morphological transforms and a reference to the appropriate function in the Matlab Image Processing Toolbox.

Exercises

Exercise 8.1 Consider the original image in Figure 8.4 and the following structuring element:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Sketch the result of erosion with this structuring element.
Sketch the result of dilation with this structuring element.

Exercise 8.2 Consider the original image in Figure 8.4 and the following structuring element:

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

What is the result of first eroding the image with this structuring element and then dilating the eroded image with the same structuring element (an operation called *closing*)?
What is the result of first dilating the image with this structuring element and then eroding the dilated image with the same structuring element (an operation called *opening*)?

Exercise 8.3 Consider the following structuring element:

$$\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{matrix}$$

- Read in and display the Matlab image 'blobs.png'.
- Carry out erosion using this structuring element directly on the image (do not use the ***strel*** function). Display the result.
- Find the decomposition of this structuring element using ***strel***. Explicitly demonstrate that successive erosions using the decomposition produce the same result as erosion with the original structuring element.

Exercise 8.4 Based on the geometric definitions of the skeleton given, what form do the skeletons of the following geometric figures take:

(1) A perfect circle?

(2) A perfect square?

(3) An equilateral triangle?

Exercise 8.5 Using the same structuring element and image segment as in Example 8.13, show that

$$A \ominus B = \begin{matrix} 4 & 8 & 5 \\ 6 & 4 & 7 \\ 4 & 7 & 5 \end{matrix} \quad \text{and that} \quad A \oplus B = \begin{matrix} 16 & 17 & 16 \\ 17 & 16 & 17 \\ 12 & 17 & 12 \end{matrix}$$

(Remember that, just as with binary images, if the structuring element goes outside the bounds of the image we consider only those image pixels which intersect with the structuring element.)

For further examples and exercises see http://www.fundipbook.com