CHAPTER

Basic image processing operations

3

CHAPTER OUTLINE HEAD

3.1 Overview

We shall now start to process digital images. First, we shall describe the brightness variation in an image using its histogram. We shall then look at operations that manipulate the image so as to change the histogram, processes that shift and

Feature Extraction & Image Processing for Computer Vision.

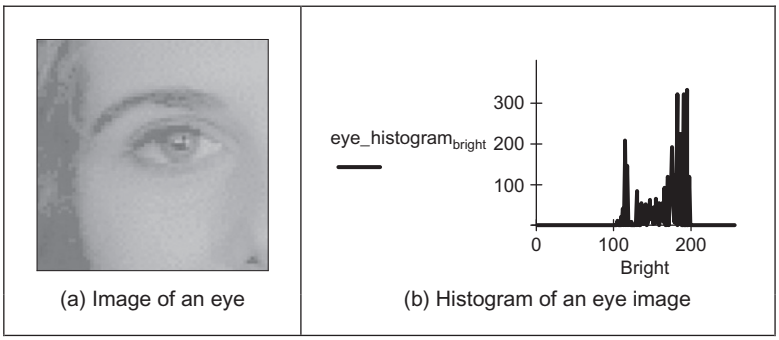
Table 3.1 Overview of Chapter 3

Main Topic	Subtopics	Main Points
Image description	Portray **variation** in image brightness content as a graph/ **histogram**	*Histograms*, image *contrast*
Point operations	Calculate **new** image points as a **function** of the point at the same place in the original image. The functions can be **mathematical** or can be computed from the image itself and will change the image's histogram. Finally, **thresholding** turns an image from **gray** level to a **binary** (black and white) representation	*Histogram* manipulation; *intensity mapping*: addition, inversion, scaling, logarithm, exponent. *Intensity normalization*; *histogram equalization*. *Thresholding* and *optimal thresholding*
Group operations	Calculate new image points as a function of **neighborhood** of the point at the same place in the original image. The functions can be **statistical** including mean (average), median, and mode. **Advanced** filtering techniques including feature preservation. **Morphological** operators process an image according to **shape**, starting with binary and moving to gray-level operations	*Template convolution* (including frequency domain implementation). *Statistical operators*: *direct averaging*, *median* filter, and *mode* filter. *Anisotropic diffusion* for image smoothing. Other operators: *force field* transform. *Mathematical morphology*: *hit or miss transform*, *erosion*, *dilation* (including gray-level operators), and *Minkowski* operators

scale the result (making the image brighter or dimmer, in different ways). We shall also consider thresholding techniques that turn an image from gray level to binary. These are called single-point operations. After, we shall move to group operations where the group is those points found inside a template. Some of the most common operations on the groups of points are statistical, providing images where each point is the result of, say, averaging the neighborhood of each point in the original image. We shall see how the statistical operations can reduce noise in the image, which is of benefit to the feature extraction techniques to be considered later. As such, these basic operations are usually for preprocessing for later feature extraction or to improve display quality as summarized in Table 3.1.

3.2 Histograms

The intensity *histogram* shows how individual brightness levels are occupied in an image; the *image contrast* is measured by the range of brightness levels. The



(a) Image of an eye (b) Histogram of an eye image

FIGURE 3.1

An image and its histogram.

histogram plots the number of pixels with a particular brightness level against the brightness level. For 8-bit pixels, the brightness ranges from 0 (black) to 255 (white). Figure 3.1 shows an image of an eye and its histogram. The histogram (Figure 3.1(b)) shows that not all the gray levels are used and the lowest and highest intensity levels are close together, reflecting moderate **contrast**. The histogram has a region between 100 and 120 brightness values, which contains the dark portions of the image, such as the hair (including the eyebrow) and the eye's iris. The brighter points relate mainly to the skin. If the image was darker, overall, the histogram would be concentrated toward black. If the image was brighter, but with lower contrast, then the histogram would be thinner and concentrated near the whiter brightness levels.

This histogram shows us that we have not used all available gray levels. Accordingly, we can stretch the image to use them all, and the image would become clearer. This is essentially cosmetic attention to make the image's appearance better. Making the appearance better, especially in view of later processing, is the focus of many basic image processing operations, as will be covered in this chapter. The histogram can also reveal if there is much noise in the image, if the ideal histogram is known. We might want to remove this noise not only to improve the appearance of the image but also to ease the task of (and to present the target better for) later feature extraction techniques. This chapter concerns these basic operations that can improve the appearance and quality of images.

The histogram can be evaluated by the operator histogram as given in Code 3.1. The operator first initializes the histogram to zero. Then, the operator works by counting up the number of image points that have an intensity at a particular value. These counts for the different values form the overall histogram. The counts are then returned as the 2D histogram (a vector of the count values) which can be plotted as a graph (Figure 3.1(b)).

```
histogram(pic):= | for bright∈0..255
                 |   pixels_at_level_bright←0
                 | for x∈0..cols(pic)-1
                 |   for y∈0..rows(pic)-1
                 |     level←pic_y,x
                 |     pixels_at_level_level←pixels_at_level_level+1
                 | pixels_at_level
```

CODE 3.1

Evaluating the histogram.

3.3 Point operators

3.3.1 Basic point operations

The most basic operations in image processing are point operations where each pixel value is replaced with a new value obtained from the old one. If we want to increase the brightness to stretch the contrast, we can simply multiply all pixel values by a scalar, say by 2, to double the range. Conversely, to reduce the contrast (though this is not usual), we can divide all point values by a scalar. If the overall brightness is controlled by a *level*, l, (e.g., the brightness of global light) and the range is controlled by a *gain*, k, the brightness of the points in a new picture, **N**, can be related to the brightness in old picture, **O**, by

$$\mathbf{N}_{x,y} = k \times \mathbf{O}_{x,y} + l \quad \forall x, y \in 1, N \tag{3.1}$$

This is a point operator that replaces the brightness at points in the picture according to a linear brightness relation. The level controls overall brightness and is the minimum value of the output picture. The gain controls the contrast, or range, and if the gain is greater than unity, the output range will be increased; this process is illustrated in Figure 3.2. So the image of the eye, processed by $k = 1.2$ and $l = 10$ will become brighter (Figure 3.2(a)), and with better contrast, though in this case the brighter points are mostly set near to white (255). These factors can be seen in its histogram (Figure 3.2(b)).

The basis of the implementation of point operators was given earlier, for addition in Code 1.3. The stretching process can be displayed as a mapping between the input and output ranges, according to the specified relationship, as in Figure 3.3. Figure 3.3(a) is a mapping where the output is a direct copy of the input (this relationship is the dotted line in Figure 3.3(c) and (d)); Figure 3.3(b) is the mapping for brightness **inversion** where dark parts in an image become bright and vice versa. Figure 3.3(c) is the mapping for **addition**, and Figure 3.3(d) is the mapping for **multiplication** (or **division**, if the slope was less than that of the input).

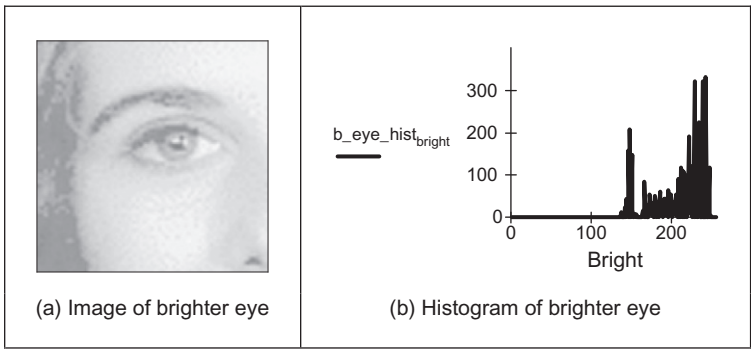


FIGURE 3.2

Brightening an image.

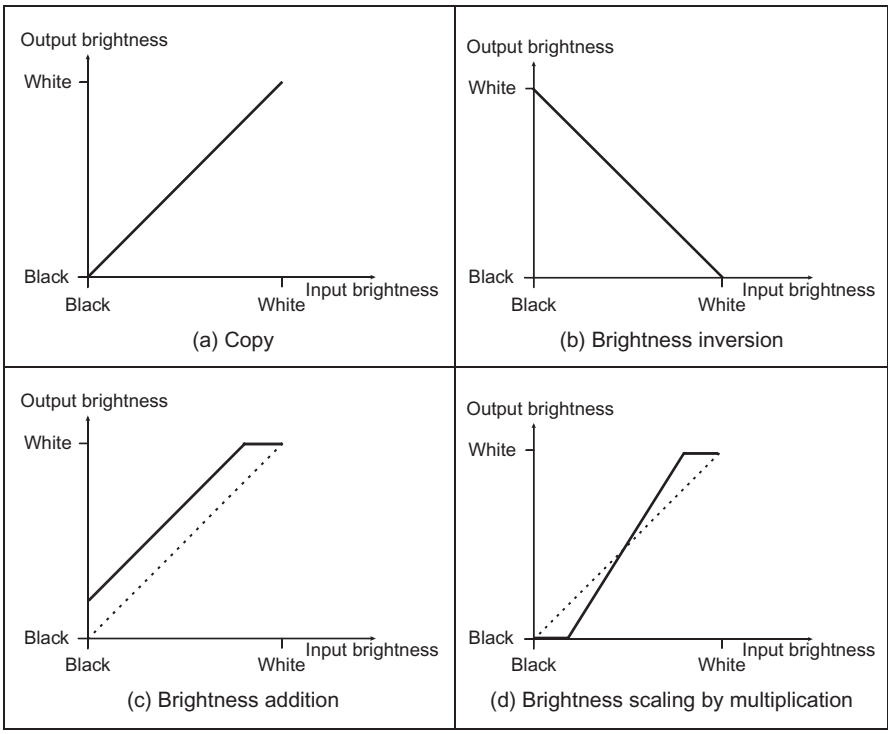


FIGURE 3.3

Intensity mappings.

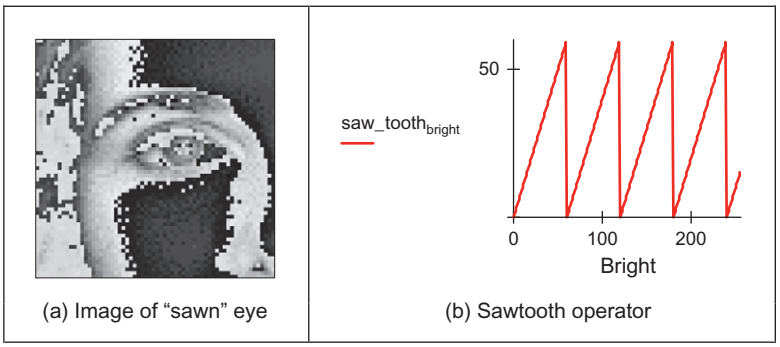


FIGURE 3.4

Applying the sawtooth operator.

In these mappings, if the mapping produces values that are smaller than the expected minimum (say negative when zero represents black) or larger than a specified maximum, then a *clipping* process can be used to set the output values to a chosen level. For example, if the relationship between input and output aims to produce output points with intensity value greater than 255, as used for white, the output value can be set to white for these points, as given in Figure 3.3(c).

The *sawtooth* operator is an alternative form of the linear operator and uses a repeated form of the linear operator for chosen intervals in the brightness range. The sawtooth operator is actually used to emphasize local contrast change (as in images where regions of interest can be light or dark). This is illustrated in Figure 3.4 where the range of brightness levels is mapped into four linear regions by the sawtooth operator (Figure 3.4(b)). This remaps the intensity in the eye image to highlight local intensity variation, as opposed to global variation, as given in Figure 3.4(a). The image is now presented in regions, where the region selection is controlled by its pixel's intensities.

Finally, rather than simple multiplication, we can use arithmetic functions such as logarithm to reduce the range or exponent to increase it. This can be used, say, to equalize the response of a camera or to compress the range of displayed brightness levels. If the camera has a known exponential performance and outputs a value for brightness that is proportional to the exponential of the brightness of the corresponding point in the scene of view, the application of a *logarithmic point operator* will restore the original range of brightness levels. The effect of replacing brightness by a scaled version of its natural logarithm (implemented as $\mathbf{N}_{x,y} = 20 \ln(100\mathbf{O}_{x,y})$) is shown in Figure 3.5(a); the effect of a scaled version of the exponent (implemented as $\mathbf{N}_{x,y} = 20 \exp(\mathbf{O}_{x,y}/100)$) is shown in Figure 3.5(b). The scaling factors were chosen to ensure that the resulting image can be displayed since the logarithm or exponent greatly reduces or magnifies the pixel values, respectively. This can be seen in the results: Figure 3.5(a) is dark with a small range of brightness levels, whereas

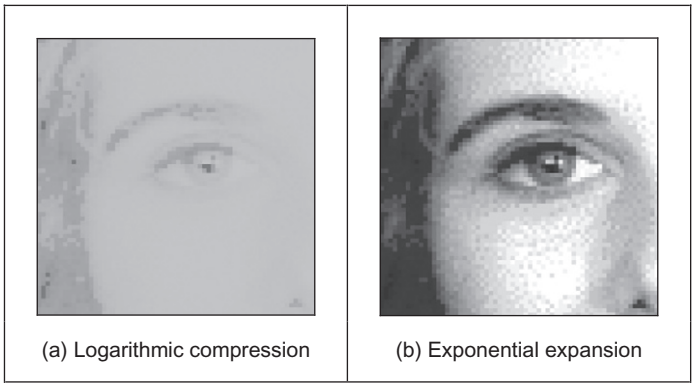


FIGURE 3.5

Applying exponential and logarithmic point operators.

Figure 3.5(b) is much brighter, with greater contrast. Naturally, application of the logarithmic point operator will change any **multiplicative** changes in brightness to become **additive**. As such, the logarithmic operator can find application in reducing the effects of multiplicative intensity change. The logarithm operator is often used to compress Fourier transforms, for display purposes. This is because the d.c. component can be very large with contrast too large to allow the other points to be seen.

In hardware, point operators can be implemented using LUTs that exist in some framegrabber units. LUTs give an output that is programmed, and stored, in a table entry that corresponds to a particular input value. If the brightness response of the camera is known, it is possible to preprogram a LUT to make the camera response equivalent to a uniform or flat response across the range of brightness levels (in software, this can be implemented as a CASE function).

3.3.2 Histogram normalization

Popular techniques to stretch the range of intensities include *histogram* (*intensity*) *normalization*. Here, the original histogram is stretched, and shifted, to cover all the 256 available levels. If the original histogram of old picture $\mathbf{O}$ starts at $\mathbf{O}_{\min}$ and extends up to $\mathbf{O}_{\max}$ brightness levels, then we can scale up the image so that the pixels in the new picture $\mathbf{N}$ lie between a minimum output level $\mathbf{N}_{\min}$ and a maximum level $\mathbf{N}_{\max}$, simply by scaling up the input intensity levels according to

$$\mathbf{N}_{x,y} = \frac{\mathbf{N}_{\max} - \mathbf{N}_{\min}}{\mathbf{O}_{\max} - \mathbf{O}_{\min}} \times (\mathbf{O}_{x,y} - \mathbf{O}_{\min}) + \mathbf{N}_{\min} \quad \forall x, y \in 1, N \tag{3.2}$$

A Matlab implementation of intensity normalization, appearing to mimic Matlab's `imagesc` function, the `normalise` function in Code 3.2, uses an output

ranging from $\mathbf{N}_{min} = 0$ to $\mathbf{N}_{max} = 255$. This is scaled by the input range that is determined by applying the `max` and `min` operators to the input picture. Note that in Matlab, a 2D array needs double application of the `max` and `min` operators, whereas in Mathcad `max(image)` delivers the maximum. Each point in the picture is then scaled as in Eq. (3.2) and the `floor` function is used to ensure an integer output.

```
function normalised=normalise(image)
%Histogram normalisation to stretch from black to white

%Usage: [new image]=normalise(image)
%Parameters: image-array of integers
%Author: Mark S. Nixon

%get dimensions
[rows,cols]=size(image);

%set minimum
minim=min(min(image));

%work out range of input levels
range=max(max(image))-minim;

%normalise the image
for x=1:cols %address all columns
  for y=1:rows %address all rows
    normalised(y,x)=floor((image(y,x)-minim)*255/range);
  end
end
```

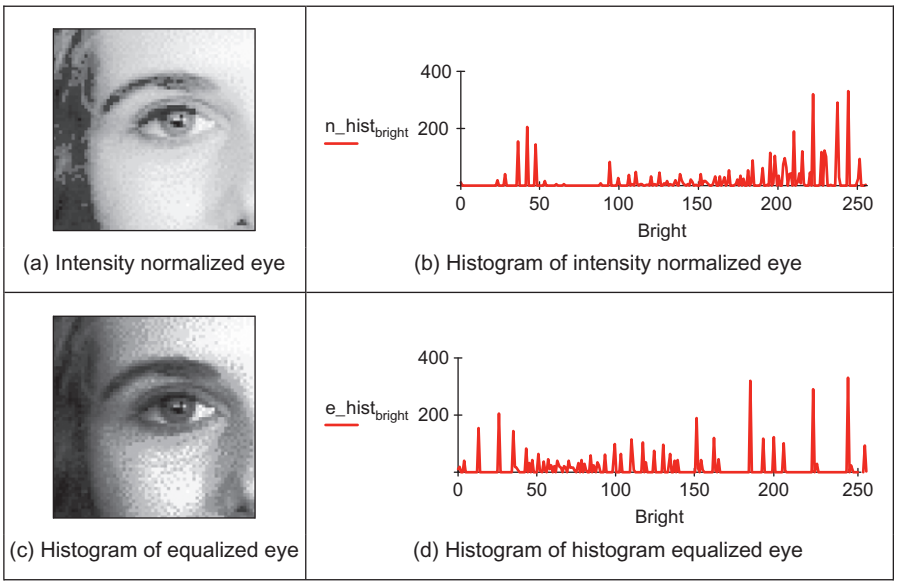
CODE 3.2

Intensity normalization.

The process is illustrated in Figure 3.6 and can be compared with the original image and histogram in Figure 3.1. An intensity normalized version of the eye image is shown in Figure 3.6(a) which now has better contrast and appears better to the human eye. Its histogram (Figure 3.6(b)) shows that the intensity now ranges across all available levels (there is actually one black pixel!).

3.3.3 Histogram equalization

Histogram equalization is a **nonlinear** process aimed to highlight image brightness in a way particularly suited to human visual analysis. Histogram equalization aims to change a picture in such a way as to produce a picture with a **flatter** histogram, where all levels are equiprobable. In order to develop the operator, we can first inspect the histograms. For a range of M levels, the histogram plots the points per level against level. For the input (old) and output (new) images, the



(a) Intensity normalized eye

(b) Histogram of intensity normalized eye

(c) Histogram of equalized eye

(d) Histogram of histogram equalized eye

FIGURE 3.6

Illustrating intensity normalization and histogram equalization.

number of points per level is denoted as $\mathbf{O}(l)$ and $\mathbf{N}(l)$ (for $0 < l < M$), respectively. For square images, there are N^2 points in the input and output images, so the sum of points per level in each should be equal:

$$\sum_{l=0}^{M} \mathbf{O}(l) = \sum_{l=0}^{M} \mathbf{N}(l) \tag{3.3}$$

Also, this should be the same for an arbitrarily chosen level p since we are aiming for an output picture with a uniformly flat histogram. So the cumulative histogram up to level p should be transformed to cover up to the level q in the new histogram:

$$\sum_{l=0}^{p} \mathbf{O}(l) = \sum_{l=0}^{q} \mathbf{N}(l) \tag{3.4}$$

Since the output histogram is uniformly flat, the cumulative histogram up to level p should be a fraction of the overall sum. So the number of points per level in the output picture is the ratio of the number of points to the range of levels in the output image:

$$\mathbf{N}(l) = \frac{N^2}{\mathbf{N}_{\max} - \mathbf{N}_{\min}} \tag{3.5}$$

So the cumulative histogram of the output picture is

$$\sum_{l=0}^{q} \mathbf{N}(l) = q \times \frac{N^2}{\mathbf{N}_{\max} - \mathbf{N}_{\min}} \tag{3.6}$$

By Eq. (3.4), this is equal to the cumulative histogram of the input image, so

$$q \times \frac{N^2}{\mathbf{N}_{\max} - \mathbf{N}_{\min}} = \sum_{l=0}^{p} \mathbf{O}(l) \tag{3.7}$$

This gives a mapping for the output pixels at level q, from the input pixels at level p as,

$$q = \frac{\mathbf{N}_{\max} - \mathbf{N}_{\min}}{N^2} \times \sum_{l=0}^{p} \mathbf{O}(l) \tag{3.8}$$

This gives a mapping function that provides an output image that has an approximately flat histogram. The mapping function is given by phrasing Eq. (3.8) as an equalizing function (E) of the level (q) and the image ($\mathbf{O}$) as

$$E(q, \mathbf{O}) = \frac{\mathbf{N}_{\max} - \mathbf{N}_{\min}}{N^2} \times \sum_{l=0}^{p} \mathbf{O}(l) \tag{3.9}$$

The output image is

$$\mathbf{N}_{x,y} = E(\mathbf{O}_{x,y}, \mathbf{O}) \tag{3.10}$$

The result of equalizing the eye image is shown in Figure 3.6. The intensity equalized image (Figure 3.6(c)) has much better-defined features (especially around the eyes) than in the original version (Figure 3.1). The histogram (Figure 3.6(d)) reveals the nonlinear mapping process whereby white and black are not assigned equal weight, as they were in intensity normalization. Accordingly, more pixels are mapped into the darker region and the brighter intensities become better spread, consistent with the aims of histogram equalization.

Its performance can be very convincing since it is well mapped to the properties of human vision. If a linear brightness transformation is applied to the original image, then the equalized histogram will be the same. If we replace pixel values with ones computed according to Eq. (3.1), the result of histogram equalization will not change. An alternative interpretation is that if we equalize images (prior to further processing), then we need not worry about any brightness transformation in the original image. This is to be expected, since the linear operation of the brightness change in Eq. (3.2) does not change the overall shape of the histogram but only its size and position. However, noise in the image acquisition process will affect the shape of the original histogram, and hence the equalized version. So the equalized histogram of a picture will not be the same as the equalized histogram of a picture with some noise added to it. You cannot avoid noise in electrical systems, however well you design a system to reduce its effect.

Accordingly, histogram equalization finds little use in generic image processing systems though it can be potent in **specialized** applications. For these reasons, intensity normalization is often preferred when a picture's histogram requires manipulation.

In implementation, the function `equalise` in Code 3.3, we shall use an output range where $\mathbf{N}_{min} = 0$ and $\mathbf{N}_{max} = 255$. The implementation first determines the cumulative histogram for each level of the brightness histogram. This is then used as a LUT for the new output brightness at that level. The LUT is used to speed implementation of Eq. (3.9) since it can be precomputed from the image to be equalized.

```
equalise(pic):=  range ← 255
                 number ← rows(pic).cols(pic)
                 for bright ∈ 0..255
                   pixels_at_level_bright ← 0
                 for x ∈ 0..cols(pic)-1
                   for y ∈ 0..rows(pic)-1
                     pixels_at_level_pic_y,x ← pixels_at_level_pic_y,x+1

                 sum ← 0
                 for level ∈ 0..255
                     sum ← sum+pixels_at_level_level
                     hist_level ← floor[(range/number)·sum+0.00001]

                 for x ∈ 0..cols(pic)-1
                   for y ∈ 0..rows(pic)-1
                     newpic_y,x ← hist_pic_y,x

                 newpic
```

CODE 3.3

Histogram equalization.

An alternative argument against use of histogram equalization is that it is a nonlinear process and is irreversible. We cannot return to the original picture after equalization, and we cannot separate the histogram of an unwanted picture. On the other hand, intensity normalization is a linear process and we can return to the original image, should we need to, or separate pictures, if required.

3.3.4 Thresholding

The last point operator of major interest is called *thresholding*. This operator selects pixels that have a particular value or are within a specified range. It can

FIGURE 3.7

Thresholding the eye image.

be used to find objects within a picture if their brightness level (or range) is known. This implies that the object's brightness must be known as well. There are two main forms: uniform and adaptive thresholding. In *uniform thresholding*, pixels above a specified level are set to white, those below the specified level are set to black. Given the original eye image, Figure 3.7 shows a thresholded image where all pixels **above** 160 brightness levels are set to white and those below 160 brightness levels are set to black. By this process, the parts pertaining to the facial skin are separated from the background; the cheeks, forehead, and other bright areas are separated from the hair and eyes. This can therefore provide a way of isolating points of interest.

Uniform thresholding clearly requires knowledge of the gray level, or the target features might not be selected in the thresholding process. If the level is not known, histogram equalization or intensity normalization can be used, but with the restrictions on performance stated earlier. This is, of course, a problem of image interpretation. These problems can only be solved by simple approaches, such as thresholding, for very special cases. In general, it is often prudent to investigate the more sophisticated techniques of feature selection and extraction, to be covered later. Prior to that, we shall investigate group operators that are a natural counterpart to point operators.

There are more advanced techniques, known as *optimal thresholding*. These usually seek to select a value for the threshold that separates an object from its background. This suggests that the object has a different range of intensities to the background, in order that an appropriate threshold can be chosen, as illustrated in Figure 3.8. Otsu's method (Otsu and Threshold, 1979) is one of the most popular techniques of optimal thresholding; there have been surveys (Sahoo et al., 1988; Lee et al., 1990; Glasbey, 1993) that compare the performance different methods can achieve. Essentially, Otsu's technique maximizes the likelihood that the threshold is chosen so as to split the image between an object and its

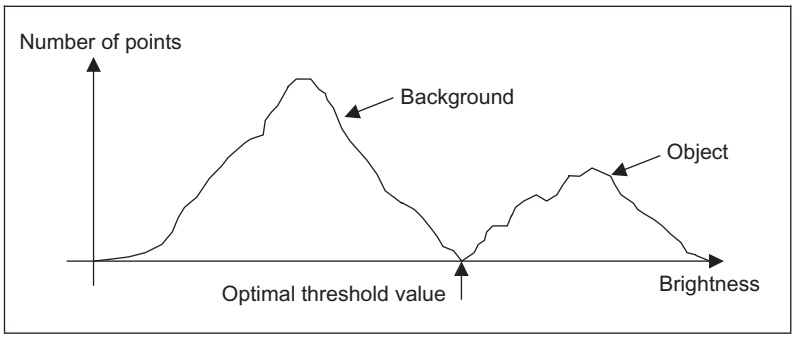


FIGURE 3.8

Optimal thresholding.

background. This is achieved by selecting a threshold that gives the best separation of classes, for all pixels in an image. The theory is beyond the scope of this section, and we shall merely survey its results and give their implementation. The basis is use of the normalized histogram where the number of points at each level is divided by the total number of points in the image. As such, this represents a probability distribution for the intensity levels as

$$p(l) = \frac{\mathbf{N}(l)}{N^2} \tag{3.11}$$

This can be used to compute the zero- and first-order cumulative moments of the normalized histogram up to the kth level as

$$\omega(k) = \sum_{l=1}^{k} p(l) \tag{3.12}$$

and

$$\mu(k) = \sum_{l=1}^{k} l \cdot p(l) \tag{3.13}$$

The total mean level of the image is given by

$$\mu\mathrm{T} = \sum_{l=1}^{N_{\max}} l \cdot p(l) \tag{3.14}$$

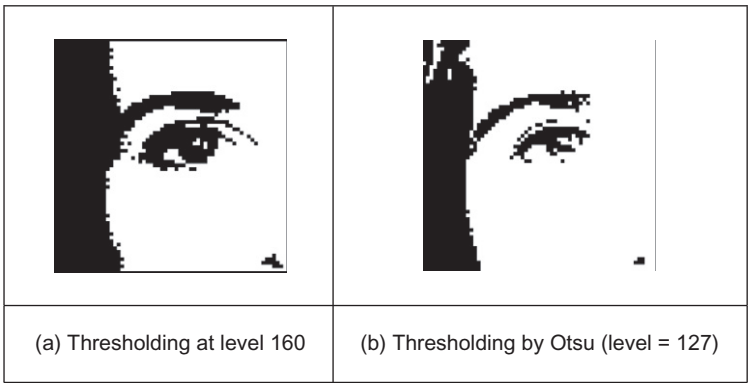
The variance of the class separability is then the ratio

$$\sigma_B^2(k) = \frac{(\mu\mathrm{T} \cdot \omega(k) - \mu(k))^2}{\omega(k)(1 - \omega(k))} \quad \forall k \in 1, N_{\max} \tag{3.15}$$

The optimal threshold is the level for which the variance of class separability is at its maximum, namely, the optimal threshold T_{opt} is that for which the variance

$$\sigma_B^2(T_{\text{opt}}) = \max_{1 \le k < N_{\max}} (\sigma_B^2(k)) \tag{3.16}$$

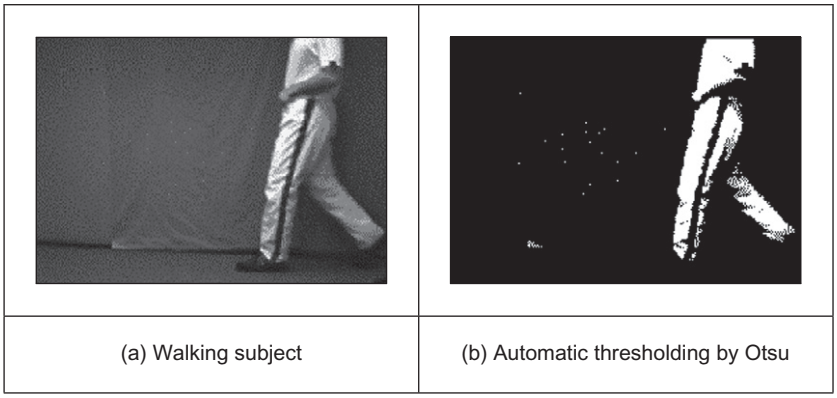
A comparison of uniform thresholding with optimal thresholding is given in Figure 3.9 for the eye image. The threshold selected by Otsu's operator is actually slightly lower than the value selected manually, and so the thresholded image does omit some detail around the eye, especially in the eyelids. However, the selection by Otsu is **automatic**, as opposed to **manual** and this can be to application advantage in automated vision. Consider, for example, the need to isolate the human figure in Figure 3.10(a). This can be performed automatically by Otsu as shown in Figure 3.10(b). Note, however, that there are some extra points,



(a) Thresholding at level 160 (b) Thresholding by Otsu (level = 127)

FIGURE 3.9

Thresholding the eye image: manual and automatic.



(a) Walking subject (b) Automatic thresholding by Otsu

FIGURE 3.10

Thresholding an image of a walking subject.

due to illumination, that have appeared in the resulting image together with the human subject. It is easy to remove the isolated points, as we will see later, but more difficult to remove the connected ones. In this instance, the size of the human shape could be used as information to remove the extra points though you might like to suggest other factors that could lead to their removal.

The code implementing Otsu's technique is given in Code 3.4 that follows Eqs (3.11)−(3.16) to directly provide the results in Figures 3.9 and 3.10. Here, the histogram function of Code 3.1 is used to give the normalized histogram. The remaining code refers directly to the earlier description of Otsu's technique.

```
ω(k,histogram) := Σ_{l=1}^{k} histogram_{l-1}

μ(k,histogram) := Σ_{l=1}^{k} l·histogram_{l-1}

μT(histogram) := Σ_{l=1}^{256} l·histogram_{l-1}

Otsu(image) := | image_hist ← histogram(image) / (rows(image)·cols(image))
               | for k∈1..255
               |     values_k ← (μT(image_hist)·ω(k,image_hist) − μ(k,image_hist))^2 / (ω(k,image_hist)·(1 − ω(k,image_hist)))
               | find_value(max(values),values)
```

CODE 3.4

Optimal thresholding by Otsu's technique.

Also, we have so far considered **global** techniques, methods that operate on the entire image. There are also **locally adaptive** techniques that are often used to binarize document images prior to character recognition. As mentioned before, surveys of thresholding are available, and one approach (Rosin, 2001) targets thresholding of images whose histogram is unimodal (has a single peak). One survey (Trier and Jain, 1995) compares global and local techniques with reference to document image analysis. These techniques are often used in statistical pattern recognition: the thresholded object is classified according to its statistical properties. However, these techniques find less use in image interpretation, where a common paradigm is that there is more than one object in the scene, such as Figure 3.7, where the thresholding operator has selected many objects of potential interest. As such, only uniform thresholding is used in many vision applications since objects are often occluded (hidden), and many objects have similar ranges of pixel intensity. Accordingly, more sophisticated metrics are required to separate them, by using the uniformly thresholded image, as discussed in later chapters. Further, the operation to process the thresholded image, say to fill in the holes in the silhouette or to remove the noise on its boundary or outside, is *morphology* which is covered later in Section 3.6.

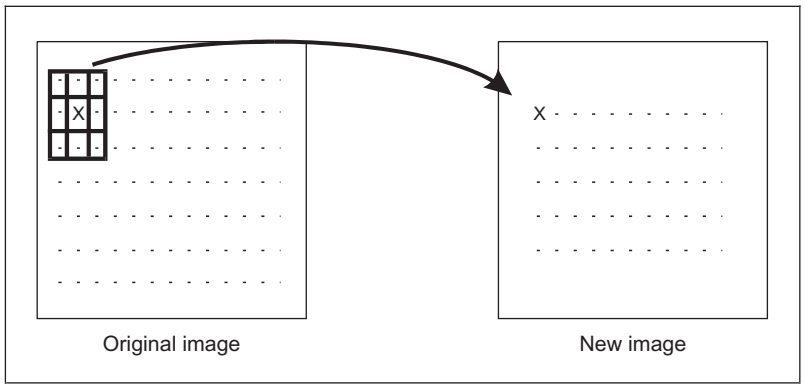


FIGURE 3.11

Template convolution process.

3.4 Group operations

3.4.1 Template convolution

Group operations calculate new pixel values from a pixel's neighborhood by using a "grouping" process. The group operation is usually expressed in terms of *template convolution* where the template is a set of weighting coefficients. The template is usually square, and its size is usually odd to ensure that it can be positioned appropriately. The size is usually used to describe the template; a 3×3 template is three pixels wide by three pixels long. New pixel values are calculated by placing the template at the point of interest. Pixel values are multiplied by the corresponding weighting coefficient and added to an overall sum. The sum (usually) evaluates a new value for the center pixel (where the template is centered) and this becomes the pixel in a, new, output image. If the template's position has not yet reached the end of a line, the template is then moved horizontally by one pixel and the process repeats.

This is illustrated in Figure 3.11 where a new image is calculated from an original one by template convolution. The calculation obtained by template convolution for the center pixel of the template in the original image becomes the point in the output image. Since the template cannot extend beyond the image, the new image is smaller than the original image since a new value cannot be computed for points in the border of the new image. When the template reaches the end of a line, it is repositioned at the start of the next line. For a 3×3 neighborhood, nine weighting coefficients w_t are applied to points in the original image to calculate a point in the new image. The position of the new point (at the center) is shaded in the template.

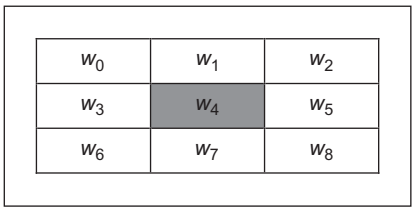


FIGURE 3.12

3 × 3 Template and weighting coefficients.

To calculate the value in new image, **N**, at point with coor dinates (x,y), the template in Figure 3.12 operates on an original image **O** according to

$$\mathbf{N}_{x,y} = \begin{matrix} w_0 \times \mathbf{O}_{x-1,y-1} & + & w_1 \times \mathbf{O}_{x,y-1} & + & w_2 \times \mathbf{O}_{x+1,y-1} & + \\ w_3 \times \mathbf{O}_{x-1,y} & + & w_4 \times \mathbf{O}_{x,y} & + & w_5 \times \mathbf{O}_{x+1,y} & + \\ w_6 \times \mathbf{O}_{x-1,y+1} & + & w_7 \times \mathbf{O}_{x,y+1} & + & w_8 \times \mathbf{O}_{x+1,y+1} & \end{matrix} \quad \forall x,\ y \in 2,\ N-1 \tag{3.17}$$

Note that we cannot ascribe values to the picture's borders. This is because when we place the template at the border, parts of the template fall outside the image and have no information from which to calculate the new pixel value. The width of the border equals half the size of the template. To calculate values for the border pixels, we now have three choices:

1. set the border to black (or deliver a smaller picture);
2. assume (as in Fourier) that the image replicates to infinity along both dimensions and calculate new values by cyclic shift from the far border; or
3. calculate the pixel value from a smaller area.

None of these approaches is optimal. The results here use the first option and set border pixels to black. Note that in many applications, the object of interest is imaged centrally or, at least, imaged within the picture. As such, the border information is of little consequence to the remainder of the process. Here, the border points are set to black, by starting functions with a zero function which sets all the points in the picture initially to black (0).

An alternative representation for this process is given by using the convolution notation as

$$\mathbf{N} = \mathbf{W} * \mathbf{O} \tag{3.18}$$

where **N** is the new image that results from convolving the template **W** (of weighting coefficients) with the image **O**.

The Matlab implementation of a general template convolution operator `convolve` is given in Code 3.5. This function accepts, as arguments, the picture image and the template to be convolved with it, i.e., template. The result of template convolution is

```
function convolved=convolve(image,template)
%New image point brightness convolution of template with image
%Usage:[new image]=convolve(image,template of point values)
%Parameters:image-array of points
% template-array of weighting coefficients
%Author: Mark S. Nixon

%get image dimensions
[irows,icols]=size(image);

%get template dimensions
[trows,tcols]=size(template);

%set a temporary image to black
temp(1:irows,1:icols)=0;

%half of template rows is
trhalf=floor(trows/2);
%half of template cols is
tchalf=floor(tcols/2);

%then convolve the template
for x=trhalf+1:icols-trhalf %address all columns except border
 for y=tchalf+1:irows-tchalf %address all rows except border
   sum=0;
   for iwin=1:trows %address template columns
     for jwin=1:tcols %address template rows
      sum=sum+image(y+jwin-tchalf-1,x+iwin-trhalf-1)*
                        template(jwin,iwin);
     end
   end
   temp(y,x)=sum;
 end
end

%finally, normalise the image
convolved=normalise(temp);
```

CODE 3.5

Template convolution operator.

a picture convolved. The operator first initializes the temporary image temp to black (zero brightness levels). Then the size of the template is evaluated. These give the range of picture points to be processed in the outer for loops that give the coordinates of all points resulting from template convolution. The template is convolved at each picture point by generating a running summation of the pixel values within the template's window multiplied by the respective template weighting coefficient. Finally, the resulting image is normalized to ensure that the brightness levels are occupied appropriately.

Template convolution is usually implemented in software. It can, of course, be implemented in hardware and requires a two-line store, together with some further

latches, for the (input) video data. The output is the result of template convolution, summing the result of multiplying weighting coefficients by pixel values. This is called pipelining since the pixels, essentially, move along a pipeline of information. Note that two-line stores can be used if only the video fields are processed. To process a full frame, one of the fields must be stored if it is presented in interlaced format.

Processing can be analog, using operational amplifier circuits and CCD for storage along bucket brigade delay lines. Finally, an alternative implementation is to use a parallel architecture: for Multiple Instruction Multiple Data (MIMD) architectures, the picture can be split into blocks (spatial partitioning); Single Instruction Multiple Data (SIMD) architectures can implement template convolution as a combination of shift and add instructions.

3.4.2 Averaging operator

For an *averaging operator*, the template weighting functions are unity (or 1/9 to ensure that the result of averaging nine white pixels is white, not more than white!). The template for a 3×3 averaging operator, implementing Eq. (3.17), is given by the template in Figure 3.13, where the location of the point of interest is again shaded. The result of averaging the eye image with a 3×3 operator is shown in Figure 3.14. This shows that much of the detail has now disappeared

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

FIGURE 3.13

3×3 Averaging operator template coefficients.

FIGURE 3.14

Applying direct averaging.

revealing the broad image structure. The eyes and eyebrows are now much clearer from the background, but the fine detail in their structure has been removed.

For a general implementation (Code 3.6), we can define the width of the operator as `winsize`, the template size is winsize × winsize. We then form the average of all points within the area covered by the template. This is normalized (divided by) the number of points in the template's window. This is a direct implementation of a general averaging operator (i.e., without using the template convolution operator in Code 3.5).

$$\begin{array}{l}\texttt{ave(pic,winsize):=} \\ \quad \texttt{new} \leftarrow \texttt{zero(pic)} \\ \quad \texttt{half} \leftarrow \texttt{floor}\left(\frac{\texttt{winsize}}{2}\right) \\ \quad \texttt{for } x \in \texttt{half..cols(pic)-half-1} \\ \quad\quad \texttt{for } y \in \texttt{half..rows(pic)-half-1} \\ \quad\quad\quad \texttt{new}_{y,x} \leftarrow \texttt{floor}\left[\frac{\sum_{\texttt{iwin=0}}^{\texttt{winsize-1}} \sum_{\texttt{jwin=0}}^{\texttt{winsize-1}} \texttt{pic}_{y+\texttt{iwin-half},x+\texttt{jwin-half}}}{(\texttt{winsize}\cdot\texttt{winsize})}\right] \\ \quad \texttt{new}\end{array}$$

CODE 3.6

Direct averaging.

In order to implement averaging by using the template convolution operator, we need to define a template. This is illustrated for direct averaging in Code 3.7, even though the simplicity of the direct averaging template usually precludes such implementation. The application of this template is also shown in Code 3.7. (Also note that there are averaging operators in Mathcad and Matlab, which can also be used for this purpose.)

$$\begin{array}{l}\texttt{averaging\_template(winsize):=} \\ \quad \texttt{sum} \leftarrow \texttt{winsize.winsize} \\ \quad \texttt{for } y \in \texttt{0..winsize-1} \\ \quad\quad \texttt{for } x \in \texttt{0..winsize-1} \\ \quad\quad\quad \texttt{template}_{y,x} \leftarrow 1 \\ \quad \frac{\texttt{template}}{\texttt{sum}} \\ \texttt{smoothed:=tm\_conv(p,averaging\_template(3))}\end{array}$$

CODE 3.7

Direct averaging by template convolution.

The effect of averaging is to reduce noise, which is its advantage. An associated disadvantage is that averaging causes blurring that reduces detail in an image. It is also a **low-pass** filter since its effect is to allow low spatial frequencies to be retained and to suppress high frequency components. A larger template,

say 3×3 or 5×5, will remove more noise (high frequencies) but reduce the level of detail. The size of an averaging operator is then equivalent to the reciprocal of the bandwidth of a low-pass filter it implements.

Smoothing was earlier achieved by low-pass filtering via the Fourier transform (Section 2.8). In fact, the Fourier transform actually gives an alternative method to implement template convolution and to speed it up, for larger templates. In Fourier transforms, the process that is dual to **convolution** is **multiplication** (as in Section 2.3). So template convolution (denoted *) can be implemented by multiplying the Fourier transform of the template $\Im(\mathbf{T})$ with the Fourier transform of the picture, $\Im(\mathbf{P})$, to which the template is to be applied. It is perhaps a bit confusing that we appear to be multiplying matrices, but the multiplication is point-by-point in that the result at each point is that of multiplying the (single) points at the same positions in the two matrices. The result needs to be inverse transformed to return to the picture domain.

$$\mathbf{P} * \mathbf{T} = \Im^{-1}(\Im(\mathbf{P}) \times \Im(\mathbf{T})) \tag{3.19}$$

The transform of the template and the picture need to be the same size before we can perform the point-by-point multiplication. Accordingly, the image containing the template is *zero-padded* prior to its transform which simply means that zeroes are added to the template in positions which lead to a template of the same size as the image. The process is illustrated in Code 3.8 and starts by calculation of the transform of the zero-padded template. The convolution routine then multiplies the transform of the template by the transform of the picture point-by-point (using the vectorize operator, symbolized by the arrow above the operation). When the routine is invoked, it is supplied with a transformed picture. The resulting transform is reordered prior to inverse transformation to ensure that the image is presented correctly. (Theoretical study of this process is presented in Section 5.3.2 where we show how the same process can be used to find shapes in images.)

```
conv(pic,temp):= | pic_spectrum ←Fourier(pic)
                 | temp_spectrum ←Fourier(temp)
                 |
                 | convolved_spectrum ←(pic_spectrum.temp_spectrum)
                 | result←inv_Fourier(rearrange(convolved_spectrum))
                 | result

new_smooth:=conv(p,square)
```

CODE 3.8

Template convolution by the Fourier transform.

3.4.3 On different template size

Templates can be larger than 3×3. Since they are usually centered on a point of interest, to produce a new output value at that point, they are usually of odd

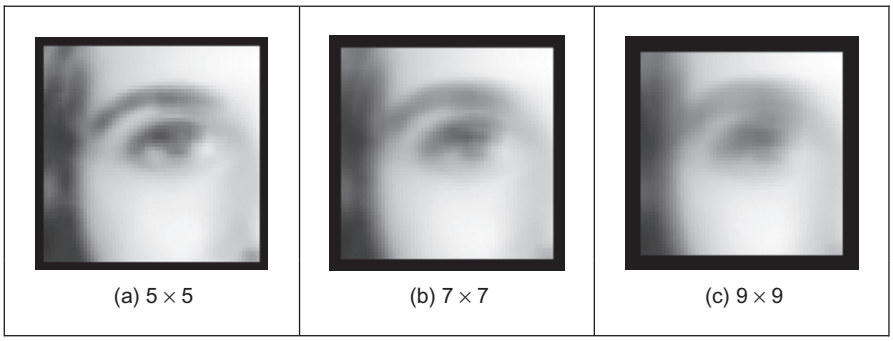


FIGURE 3.15

Illustrating the effect of window size.

dimension. For reasons of speed, the most common sizes are 3×3, 5×5, and 7×7. Beyond this, say 9×9, many template points are used to calculate a single value for a new point, and this imposes high computational cost, especially for large images. (For example, a 9×9 operator covers 9 times more points than a 3×3 operator.) Square templates have the same properties along both image axes. Some implementations use vector templates (a line), either because their properties are desirable in a particular application or for reasons of speed.

The effect of larger averaging operators is to smooth the image more and to remove more detail while giving greater emphasis to the large structures. This is illustrated in Figure 3.15. A 5×5 operator (Figure 3.15(a)) retains more detail than a 7×7 operator (Figure 3.15(b)) and much more than a 9×9 operator (Figure 3.15(c)). Conversely, the 9×9 operator retains only the largest structures such as the eye region (and virtually removing the iris), whereas this is retained more by the operators of smaller size. Note that the larger operators leave a larger border (since new values cannot be computed in that region), and this can be seen in the increase in border size for the larger operators, in Figure 3.15(b) and (c).

3.4.4 Gaussian averaging operator

The *Gaussian averaging operator* has been considered to be optimal for image smoothing. The template for the Gaussian operator has values set by the Gaussian relationship. The Gaussian **function** g at coordinates (x,y) is controlled by the *variance* σ^2 according to

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \tag{3.20}$$

Equation (3.20) gives a way to calculate coefficients for a Gaussian template that is then convolved with an image. The effects of selection of Gaussian templates of differing size are shown in Figure 3.16. The Gaussian function essentially removes the influence of points greater than 3σ in (radial) distance from the

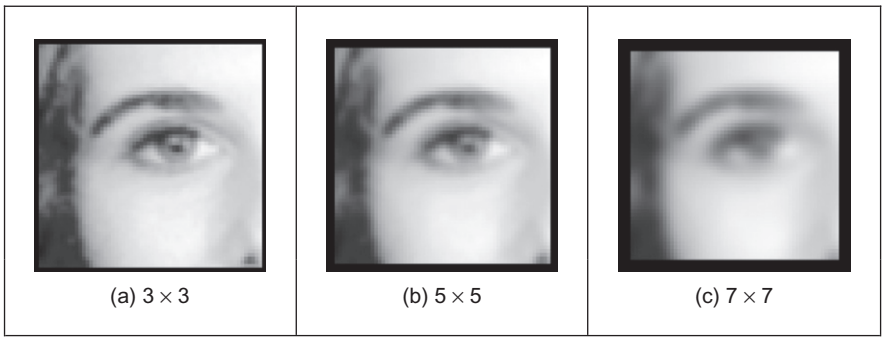


FIGURE 3.16

Applying Gaussian averaging.

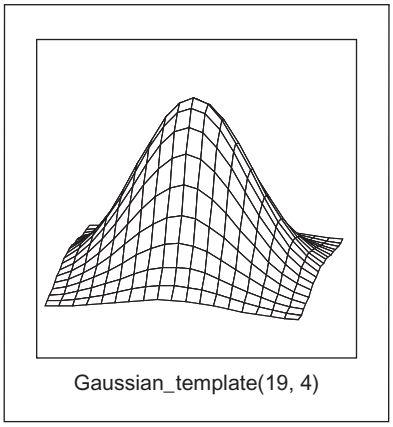


FIGURE 3.17

Gaussian function.

center of the template. The 3×3 operator (Figure 3.16(a)) retains many more of the features than those retained by direct averaging (Figure 3.14). The effect of larger size is to remove more detail (and noise) at the expense of losing features. This is reflected in the loss of internal eye component by the 5×5 and the 7×7 operators in Figure 3.16(b) and (c), respectively.

A surface plot of the 2D Gaussian function of Eq. (3.20) has the famous bell shape, as shown in Figure 3.17. The values of the function at discrete points are the values of a Gaussian template. Convolving this template with an image gives Gaussian averaging: the point in the averaged picture is calculated from the sum of a region where the central parts of the picture are weighted to contribute more than the peripheral points. The size of the template essentially dictates appropriate choice of the variance. The variance is chosen to ensure that template coefficients

0.002	0.013	0.022	0.013	0.002
0.013	0.060	0.098	0.060	0.013
0.022	0.098	0.162	0.098	0.022
0.013	0.060	0.098	0.060	0.013
0.002	0.013	0.022	0.013	0.002

FIGURE 3.18

Template for the **5** × **5** Gaussian averaging operator ($\sigma = 1.0$).

drop to near zero at the template's edge. A common choice for the template size is 5×5 with variance unity, giving the template shown in Figure 3.18.

This template is then convolved with the image to give the Gaussian blurring function. It is actually possible to give the Gaussian blurring function antisymmetric properties by scaling the x and y coordinates. This can find application when an object's shape, and orientation, is known prior to image analysis.

By reference to Figure 3.16, it is clear that the Gaussian filter can offer improved performance compared with direct averaging: more features are retained while the noise is removed. This can be understood by Fourier transform theory. In Section 2.5.2 (Chapter 2), we found that the Fourier transform of a **square** is a 2D **sinc** function. This has a frequency response where the magnitude of the transform does not reduce in a smooth manner and has regions where it becomes negative, called sidelobes. These can have undesirable effects since there are high frequencies that contribute **more** than some lower ones, a bit paradoxical in low-pass filtering to remove noise. In contrast, the Fourier transform of a Gaussian function is another Gaussian function, which decreases smoothly without these sidelobes. This can lead to better performance since the contributions of the frequency components reduce in a controlled manner.

In a software implementation of the Gaussian operator, we need a function implementing Eq. (3.20), the `Gaussian_template` function in Code 3.9. This is used to calculate the coefficients of a template to be centered on an image point. The two arguments are `winsize`, the (square) operator's size, and the standard deviation σ that controls its width, as discussed earlier. The operator coefficients are normalized by the sum of template values, as before. This summation is stored in `sum`, which is initialized to zero. The center of the square template is then evaluated as half the size of the operator. Then, all template coefficients are calculated by a version of Eq. (3.20) that specifies a weight relative to the center coordinates. Finally, the normalized template coefficients are returned as the Gaussian template. The operator is used in template convolution, via `convolve`, as in direct averaging (Code 3.5).

```
function template=gaussian_template(winsize,sigma)
%Template for Gaussian averaging

%Usage:[template]=gaussian_template(number, number)

%Parameters: winsize-size of template (odd, integer)
% sigma-variance of Gaussian function
%Author: Mark S. Nixon

%centre is half of window size
centre=floor(winsize/2)+1;

%we'll normalise by the total sum
sum=0;

%so work out the coefficients and the running total
for i=1:winsize
  for j=1:winsize
    template(j,i)=exp(-(((j-centre)*(j-centre))+((i-centre)
                  *(i-centre)))/(2*sigma*sigma))
    sum=sum+template(j,i);
  end
end

%and then normalise
template=template/sum;
```

CODE 3.9

Gaussian template specification.

3.4.5 More on averaging

Code 3.8 is simply a different implementation of direct averaging. It achieves the same result, but by transform domain calculus. It can be faster to use the transform rather than the direct implementation. The computational cost of a 2D FFT is of the order of $2N^2 \log(N)$. If the transform of the template is precomputed, there are two transforms required and there is one multiplication for each of the N^2 transformed points. The total cost of the Fourier implementation of template convolution is then of the order of

$$C_{\text{FFT}} = 4N^2 \log(N) + N^2 \tag{3.21}$$

The cost of the direct implementation for an $m \times m$ template is then m^2 multiplications for each image point, so the cost of the direct implementation is of the order of

$$C_{\text{dir}} = N^2 m^2 \tag{3.22}$$

For $C_{\text{dir}} < C_{\text{FFT}}$, we require

$$N^2 m^2 < 4N^2 \log(N) + N^2 \tag{3.23}$$

If the direct implementation of template matching is faster than its Fourier implementation, we need to choose m so that

$$m^2 < 4 \log(N) + 1 \tag{3.24}$$

This implies that for a 256×256 image a direct implementation is fastest for 3×3 and 5×5 templates, whereas a transform calculation is faster for larger ones. An alternative analysis (Campbell, 1969) has suggested that (Gonzalez and Wintz, 1987) "if the number of non-zero terms in (the template) is less than 132 then a direct implementation ... is more efficient than using the FFT approach". This implies a considerably larger template than our analysis suggests. This is in part due to higher considerations of complexity than our analysis has included. There are, naturally, further considerations in the use of transform calculus, the most important being the use of windowing (such as Hamming or Hanning) operators to reduce variance in high-order spectral estimates. This implies that template convolution by transform calculus should perhaps be used when large templates are involved, and only when speed is critical. If speed is indeed critical, it might be better to implement the operator in dedicated hardware, as described earlier.

The averaging process is actually a statistical operator since it aims to estimate the mean of a local neighborhood. The error in the process is naturally high, for a population of N samples, the statistical error is of the order of

$$\text{Error} = \frac{\text{Mean}}{\sqrt{N}} \tag{3.25}$$

Increasing the averaging operator's size improves the error in the estimate of the mean but at the expense of fine detail in the image. The average is of course an estimate optimal for a signal corrupted by additive *Gaussian noise* (see Appendix 2, Section 11.1). The estimate of the mean maximized the probability that the noise has its mean value, namely zero. According to the *central limit theorem*, the result of adding many noise sources together is a Gaussian-distributed noise source. In images, noise arises in sampling, in quantization, in transmission, and in processing. By the central limit theorem, the result of these (independent) noise sources is that image noise can be assumed to be Gaussian. In fact, image noise is **not** necessarily Gaussian distributed, giving rise to more statistical operators. One of these is the *median* operator that has demonstrated capability to reduce noise while retaining feature boundaries (in contrast to smoothing which blurs both noise and the boundaries) and the *mode* operator that can be viewed as optimal for a number of noise sources, including *Rayleigh noise*, but is very difficult to determine for small, discrete, populations.

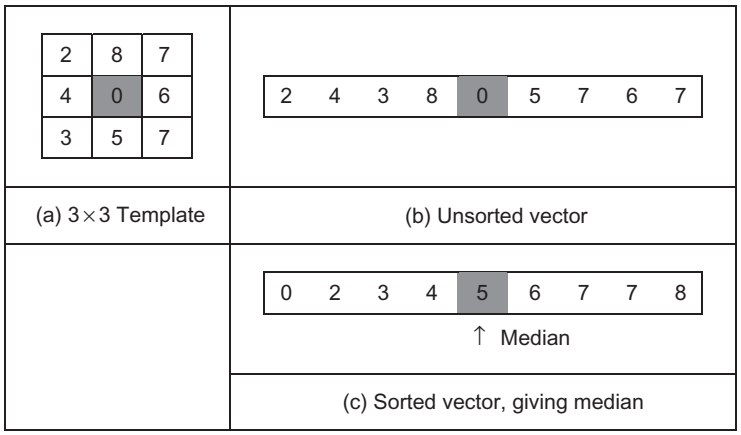


FIGURE 3.19

Finding the median from a **3** × **3** template.

3.5 Other statistical operators

3.5.1 Median filter

The *median* is another frequently used statistic; the median is the center of a rank-ordered distribution. The median is usually taken from a template centered on the point of interest. Given the arrangement of pixels in Figure 3.19(a), the pixel values are arranged into a vector format (Figure 3.19(b)). The vector is then sorted into ascending order (Figure 3.19(c)). The median is the central component of the sorted vector; this is the fifth component since we have nine values.

The median operator is usually implemented using a template, here we shall consider a 3 × 3 template. Accordingly, we need to process the nine pixels in a template centered on a point with coordinates (*x*,*y*). In a Mathcad implementation, these nine points can be extracted into vector format using the operator `unsorted` in Code 3.10. This requires an integer pointer to nine values, `x1`. The modulus operator is then used to ensure that the correct nine values are extracted.

```
x1:=0..8
unsorted_x1:=p_x+mod(x1,3)-1,x+floor(x1/3)-1
```

CODE 3.10

Reformatting a neighborhood into a vector.

We then arrange the nine pixels, within the template, in ascending order using the Mathcad `sort` function (Code 3.11).

```
sorted:=sort(unsorted)
```

CODE 3.11

Using the Mathcad sort function.

This gives the rank-ordered list, and the median is the central component of the sorted vector, in this case the fifth component (Code 3.12).

```
our_median:=sorted4
```

CODE 3.12

Evaluating the median.

These functions can then be grouped to give the full median operator as given in Code 3.13.

```
med(pic):= | newpic←zero(pic)
           |  for x∈1..cols(pic)-2
           |    for y∈1..rows(pic)-2
           |       for x1∈ 0..8
           |
           |         unsorted_x1 ← pic_(y+mod(x1,3)-1, x+floor(x1/3)-1)
           |
           |       sorted← sort(unsorted)
           |       newpic_(y,x) ← sorted4
           |
           | newpic
```

CODE 3.13

Determining the median.

The median can of course be taken from **larger** template sizes. The development here has aimed not only to demonstrate how the median operator works but also to provide a basis for further development. The rank ordering process is computationally demanding (**slow**) and motivates study into the deployment of fast algorithms, such as Quicksort (e.g., Huang et al. (1979) is an early approach), though other approaches abound (Weiss, 2006). The computational demand also has motivated use of template shapes other than a square. A selection of alternative shapes is shown in Figure 3.20. Common alternative shapes include a cross or a line (horizontal or vertical), centered on the point of interest, which can

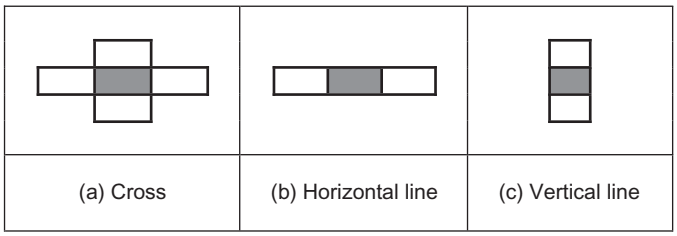


FIGURE 3.20

Alternative template shapes for median operator.

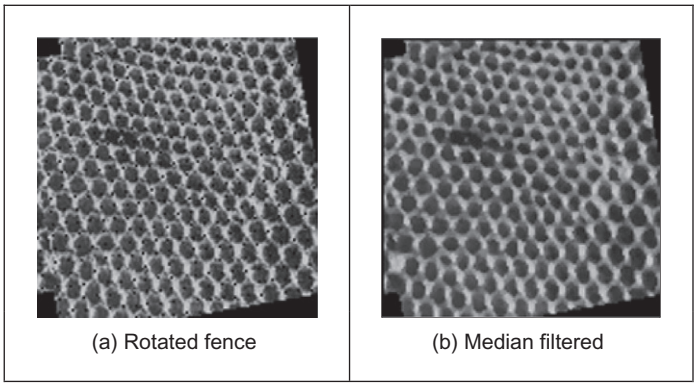


FIGURE 3.21

Illustrating median filtering.

afford much faster operation since they cover fewer pixels. The basis of the arrangement presented here could be used for these alternative shapes, if required.

The median has a well-known ability to remove *salt and pepper noise*. This form of noise, arising from, say, decoding errors in picture transmission systems, can cause isolated white and black points to appear within an image. It can also arise when rotating an image, when points remain unspecified by a standard rotation operator (Chapter 10, Appendix 1), as in a texture image, rotated by 10° in Figure 3.21(a). When a median operator is applied, the salt and pepper noise points will appear at either end of the rank-ordered list and are removed by the median process, as shown in Figure 3.21(b). The median operator has practical advantage due to its ability to retain edges (the boundaries of shapes in images) while suppressing the noise contamination. As such, like direct averaging, it remains a worthwhile member of the stock of standard image processing tools.

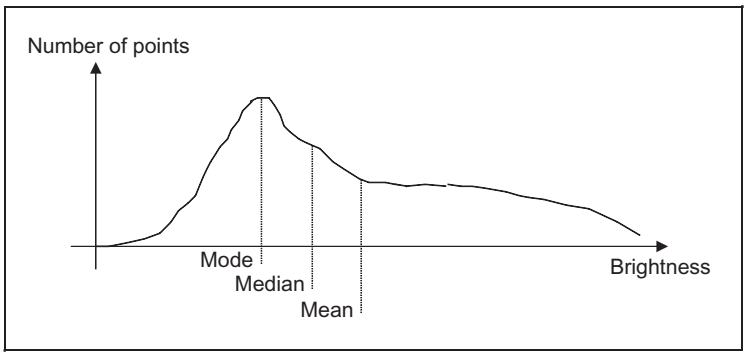


FIGURE 3.22

Arrangement of mode, median, and mean.

For further details concerning properties and implementation, see Hodgson et al. (1985). (Note that practical implementation of image rotation is a Computer Graphics issue and is usually by **texture mapping**; further details can be found in Hearn and Baker (1997).)

3.5.2 Mode filter

The *mode* is the final statistic of interest, though there are more advanced filtering operators to come. The mode is of course very difficult to determine for small populations and theoretically does not even exist for a continuous distribution. Consider, for example, determining the mode of the pixels within a square 5×5 template. Naturally, it is possible for all 25 pixels to be different, so each could be considered to be the mode. As such we are forced to estimate the mode: the truncated median filter, as introduced by Davies (1988), aims to achieve this. The *truncated median filter* is based on the premise that for many non-Gaussian distributions, the order of the mean, the median, and the mode is the same for many images, as illustrated in Figure 3.22. Accordingly, if we truncate the distribution (i.e., remove part of it, where the part selected to be removed in Figure 3.22 is from the region beyond the mean), then the median of the truncated distribution will approach the mode of the original distribution.

The implementation of the truncated median, `trun_med`, operator is given in Code 3.14. The operator first finds the mean and the median of the current window. The distribution of intensity of points within the current window is truncated on the side of the mean so that the median now bisects the distribution of the remaining points (as such not affecting symmetrical distributions); if the median is less than the mean, the point at which the distribution is truncated, is

```
trun_med(p,wsze):= | newpic←zero(p)
                   | ha←floor(wsze/2)
                   | for x∈ha..cols(p)−ha−1
                   |   for y∈ha..rows(p)−ha−1
                   |     win←submatrix(p,y−ha,y+ha,x−ha,x+ha)
                   |     med←median(win)
                   |     ave←mean(win)
                   |     upper←2·med−min(win)
                   |     lower←2·med−max(win)
                   |     cc←0
                   |     for i∈0..wsze−1
                   |       for j∈0..wsze−1
                   |         if (win_j,i<upper)·(med<ave)
                   |           | trun_cc←win_j,i
                   |           | cc←cc+1
                   |         if (win_j,i>lower)·(med>ave)
                   |           | trun_cc←win_j,i
                   |           | cc←cc+1
                   |     newpic_y,x←median(trun) if cc>0
                   |     newpic_y,x←med otherwise
                   | newpic
```

CODE 3.14

The truncated median operator.

$$\begin{aligned} \text{upper} &= \text{median} + (\text{median} - \min(\text{distribution})) \\ &= 2 \cdot \text{median} - \min(\text{distribution}) \end{aligned} \tag{3.26}$$

If the median is greater than the mean, then we need to truncate at a lower point (before the mean), given by

$$\text{lower} = 2 \cdot \text{median} - \max(\text{distribution}) \tag{3.27}$$

The median of the remaining distribution then approaches the mode. The truncation is performed by storing pixels' values in a vector `trun`. A pointer, `cc`, is incremented each time a new point is stored. The median of the truncated vector is then the output of the truncated median filter at that point. Naturally, the window is placed at each possible image point, as in template convolution. However, there can be several iterations at each position to ensure that the mode is approached. In practice, only few iterations are usually required for the median to converge to the mode. The window size is usually large, say 7×7 or 9×9 or even more.

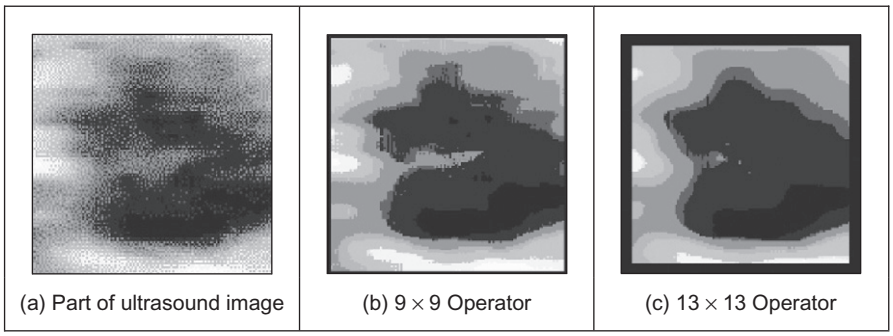


FIGURE 3.23

Applying truncated median filtering.

The action of the operator is illustrated in Figure 3.23 when applied to a 128×128 part of the ultrasound image (Figure 1.1(c)), from the center of the image and containing a cross-sectional view of an artery. Ultrasound results in particularly noisy images, in part, because the scanner is usually external to the body. The noise is actually multiplicative Rayleigh noise for which the mode is the optimal estimate. This noise obscures the artery that appears in cross section in Figure 3.23(a); the artery is basically elliptical in shape. The action of the 9×9 truncated median operator (Figure 3.23(b)) is to remove noise while retaining feature boundaries, while a larger operator shows better effect (Figure 3.23(c)).

Close examination of the result of the truncated median filter is that a selection of boundaries is preserved, which are not readily apparent in the original ultrasound image. This is one of the known properties of median filtering: an ability to reduce noise while retaining feature boundaries. Indeed, there have actually been many other approaches to speckle filtering; the most popular include direct averaging (Shankar, 1986), median filtering, adaptive (weighted) median filtering (Loupas and McDicken, 1987), and unsharp masking (Bamber and Daft, 1986).

3.5.3 Anisotropic diffusion

The most advanced form of smoothing is achieved by preserving the **boundaries** of the image features in the smoothing process (Perona and Malik, 1990). This is one of the advantages of the median operator and a disadvantage of the Gaussian smoothing operator. The process is called *anisotropic diffusion* by virtue of its basis. Its result is illustrated in Figure 3.24(b) where the feature boundaries (such as those of the eyebrows or the eyes) in the smoothed image are crisp and the skin is more matte in appearance. This implies that we are filtering within the

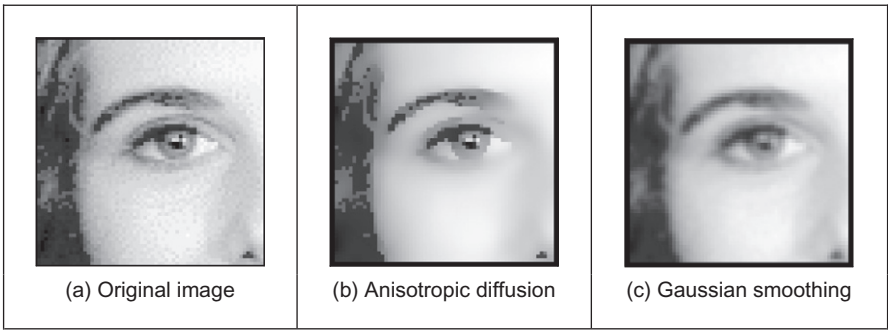


FIGURE 3.24

Filtering by anisotropic diffusion and the Gaussian operator.

features and not at their edges. By way of contrast, the Gaussian operator result in Figure 3.24(c) smoothes not just the skin but also the boundaries (the eyebrows in particular seem quite blurred) giving a less pleasing and less useful result. Since we shall later use the boundary information to interpret the image, its preservation is of much interest.

As ever, there are some parameters to select to control the operation, so we shall consider the technique's basis so as to guide their selection. Further, it is computationally more complex than Gaussian filtering. The basis of anisotropic diffusion is, however, rather complex, especially here, and invokes concepts of low-level feature extraction, which are covered in Chapter 4. One strategy you might use is to mark this page, then go ahead and read Sections 4.1 and 4.2 and return here. Alternatively, you could just read on since that is exactly what we shall do. The complexity is because the process not only invokes low-level feature extraction (to preserve feature boundaries) but also its basis actually invokes concepts of *heat flow*, as well as introducing the concept of *scale space*. So it will certainly be a hard read for many, but comparison of Figure 3.24(b) with Figure 3.24(c) shows that it is well worth the effort.

The essential idea of scale space is that there is a *multiscale* representation of images, from low resolution (a coarsely sampled image) to high resolution (a finely sampled image). This is inherent in the sampling process where the coarse image is the structure and the higher resolution increases the level of detail. As such, we can derive a *scale space* set of images by convolving an original image with a Gaussian function, by Eq. (3.24)

$$\mathbf{P}_{x,y}(\sigma) = \mathbf{P}_{x,y}(0) * g(x, y, \sigma) \tag{3.28}$$

where $\mathbf{P}_{x,y}(0)$ is the original image, $g(x,y,\sigma)$ is the Gaussian template derived from Eq. (3.20), and $\mathbf{P}_{x,y}(\sigma)$ is the image at level σ. The coarser level corresponds to

larger values of the standard deviation σ; conversely the finer detail is given by smaller values. (Scale space will be considered again in Section 4.4.2 as it pervades the more modern operators). We have already seen that the larger values of σ reduce the detail and are then equivalent to an image at a coarser scale, so this is a different view of the same process. The difficult bit is that the family of images derived this way can equivalently be viewed as the solution of the heat equation:

$$\partial \mathbf{P}/\partial t = \nabla \mathbf{P}_{x,y}(t) \tag{3.29}$$

where ∇ denotes del, the (directional) gradient operator from vector algebra, and with the initial condition that $\mathbf{P}_0 = \mathbf{P}_{x,y}(0)$. The heat equation itself describes the temperature T changing with time t as a function of the thermal diffusivity (related to conduction) κ as

$$\partial T/\partial t = \kappa \nabla^2 T \tag{3.30}$$

and in 1D form, this is

$$\partial T/\partial t = \kappa \frac{\partial^2 T}{\partial x^2} \tag{3.31}$$

So the temperature measured along a line is a function of time, distance, the initial and boundary conditions, and the properties of a material. The direct relation of this with image processing is clearly an enormous ouch! There are clear similarities between Eqs (3.31) and (3.29). This is the same functional form and allows for insight, analysis, and parameter selection. The heat equation, Eq. (3.29), is the anisotropic diffusion equation:

$$\partial \mathbf{P}/\partial t = \nabla \cdot (c_{x,y}(t) \nabla \mathbf{P}_{x,y}(t)) \tag{3.32}$$

where $\nabla \cdot$ is the divergence operator (which essentially measures how the density within a region changes), with diffusion coefficient $c_{x,y}$. The diffusion coefficient applies to the local change in the image $\nabla \mathbf{P}_{x,y}(t)$ in different directions. If we have a lot of local change, we seek to retain it since the amount of change is the amount of boundary information. The diffusion coefficient indicates how much importance we give to local change: how much of it is retained. (The equation reduces to isotropic diffusion—Gaussian filtering—if the diffusivity is constant since $\nabla c = 0$.) There is no explicit solution to this equation. By approximating differentiation by differencing (this is explored more in Section 4.2) the rate of change of the image between time step t and time step $t + 1$, we have

$$\partial \mathbf{P}/\partial t = \mathbf{P}(t+1) - \mathbf{P}(t) \tag{3.33}$$

This implies we have an iterative solution, and for later consistency, we shall denote the image $\mathbf{P}$ at time step $t + 1$ as $\mathbf{P}^{<t+1>} = \mathbf{P}(t+1)$, so we then have

$$\mathbf{P}^{<t+1>} - \mathbf{P}^{<t>} = \nabla \cdot (c_{x,y}(t) \nabla \mathbf{P}_{x,y}^{<t>}) \tag{3.34}$$

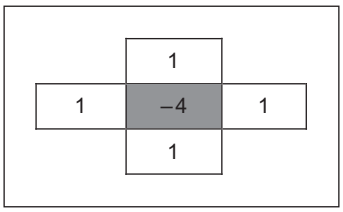


FIGURE 3.25

Approximations by spatial difference in anisotropic diffusion.

And again by approximation, using differences evaluated this time over the four compass directions North, South, East, and West, we have

$$\nabla_{\mathrm{N}}(\mathbf{P}_{x,y}) = \mathbf{P}_{x,y-1} - \mathbf{P}_{x,y} \tag{3.35}$$

$$\nabla_{\mathrm{S}}(\mathbf{P}_{x,y}) = \mathbf{P}_{x,y+1} - \mathbf{P}_{x,y} \tag{3.36}$$

$$\nabla_{\mathrm{E}}(\mathbf{P}_{x,y}) = \mathbf{P}_{x-1,y} - \mathbf{P}_{x,y} \tag{3.37}$$

$$\nabla_{\mathrm{W}}(\mathbf{P}_{x,y}) = \mathbf{P}_{x+1,y} - \mathbf{P}_{x,y} \tag{3.38}$$

The template and weighting coefficients for these are shown in Figure 3.25.

When we use these as an approximation to the right-hand side in Eq. (3.34), we then have $\nabla\cdot(c_{x,y}(t)\nabla\mathbf{P}_{x,y}^{<t>}) = \lambda(c\mathrm{N}_{x,y}\nabla_{\mathrm{N}}(\mathbf{P}) + c\mathrm{S}_{x,y}\nabla_{\mathrm{S}}(\mathbf{P}) + c\mathrm{E}_{x,y}\nabla_{\mathrm{E}}(\mathbf{P}) + c\mathrm{W}_{x,y}\nabla_{\mathrm{W}}(\mathbf{P}))$ that gives

$$\mathbf{P}^{<t+1>} - \mathbf{P}^{<t>} = \lambda(c\mathrm{N}_{x,y}\nabla_{\mathrm{N}}(\mathbf{P}) + c\mathrm{S}_{x,y}\nabla_{\mathrm{S}}(\mathbf{P}) + c\mathrm{E}_{x,y}\nabla_{\mathrm{E}}(\mathbf{P}) + c\mathrm{W}_{x,y}\nabla_{\mathrm{W}}(\mathbf{P}))\ |\mathbf{P}=\mathbf{P}_{x,y}^{<t>} \tag{3.39}$$

where $0 \leq \lambda \leq 1/4$ and $c\mathrm{N}_{x,y}$, $c\mathrm{S}_{x,y}$, $c\mathrm{E}_{x,y}$, and $c\mathrm{W}_{x,y}$ denote the conduction coefficients in the four compass directions. By rearrangement of this, we obtain the equation we shall use for the anisotropic diffusion operator

$$\mathbf{P}^{<t+1>} = \mathbf{P}^{<t>} + \lambda(c\mathrm{N}_{x,y}\nabla_{\mathrm{N}}(\mathbf{P}) + c\mathrm{S}_{x,y}\nabla_{\mathrm{S}}(\mathbf{P}) + c\mathrm{E}_{x,y}\nabla_{\mathrm{E}}(\mathbf{P}) + c\mathrm{W}_{x,y}\nabla_{\mathrm{W}}(\mathbf{P}))\ |\mathbf{P}=\mathbf{P}_{x,y}^{<t>} \tag{3.40}$$

This shows that the solution is **iterative**: images at **one** time step (denoted by $<t+1>$) are computed from images at the **previous** time step (denoted $<t>$), given the initial condition that the first image is the original (noisy) image. Change (in time and in space) has been approximated as the difference between two adjacent points, which gives the iterative equation and shows that the new image is formed by adding a controlled amount of the local change consistent with the main idea: that the smoothing process retains some of the boundary information.

We are not finished yet though, since we need to find values for $c\mathrm{N}_{x,y}$, $c\mathrm{S}_{x,y}$, $c\mathrm{E}_{x,y}$, and $c\mathrm{W}_{x,y}$. These are chosen to be a function of the difference along the compass directions, so that the boundary (edge) information is preserved. In this way, we seek a function that tends to zero with increase in the difference

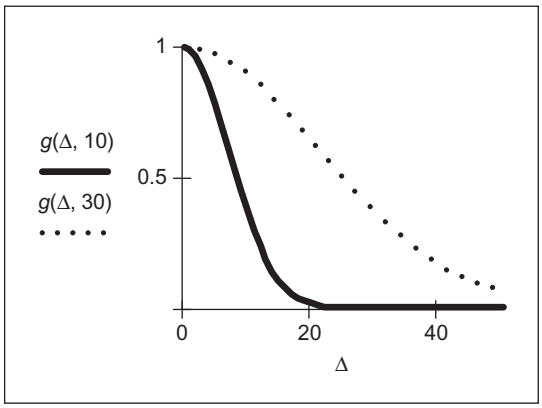


FIGURE 3.26

Controlling the conduction coefficient in anisotropic diffusion.

(an edge or boundary with greater contrast) so that diffusion does not take place across the boundaries, keeping the edge information. As such, we seek

$$\begin{aligned} c\mathrm{N}_{x,y} &= g(||\nabla_{\mathrm{N}}(\mathbf{P})||) \\ c\mathrm{S}_{x,y} &= g(||\nabla_{\mathrm{S}}(\mathbf{P})||) \\ c\mathrm{E}_{x,y} &= g(||\nabla_{\mathrm{E}}(\mathbf{P})||) \\ c\mathrm{W}_{x,y} &= g(||\nabla_{\mathrm{W}}(\mathbf{P})||) \end{aligned} \tag{3.41}$$

and one function that can achieve this is

$$g(x,k) = \mathrm{e}^{-x^2/k^2} \tag{3.42}$$

(There is potential confusion with using the same symbol as for the Gaussian function, Eq. (3.20), but we have followed the original authors' presentation.) This function clearly has the desired properties since when the values of the differences ∇ are large, the function g is very small, conversely when ∇ is small, then g tends to unity. k is another parameter whose value we have to choose: it controls the rate at which the conduction coefficient decreases with increasing difference magnitude. The effect of this parameter is shown in Figure 3.26. Here, the solid line is for the smaller value of k, and the dotted one is for a larger value. Evidently, a larger value of k means that the contribution of the difference reduces less than for a smaller value of k. In both cases, the resulting function is near unity for small differences and near zero for large differences, as required. An alternative to this is to use the function

$$g2(x,k) = \frac{1}{(1 + (x^2/k^2))} \tag{3.43}$$

which has similar properties to the function in Eq. (3.42).

(a) One iteration | (b) Two iterations | (c) Five iterations | (d) Ten iterations | (e) Final result

FIGURE 3.27

Iterations of anisotropic diffusion.

This all looks rather complicated, so let's recap. First, we want to filter an image by retaining boundary points. These are retained according to the value of k chosen in Eq. (3.42). This function is operated in the four compass directions, to weight the brightness difference in each direction, Eq. (3.41). These contribute to an iterative equation which calculates a new value for an image point by considering the contribution from its four neighboring points, Eq. (3.40). This needs choice of one parameter λ. Further, we need to choose the number of iterations for which calculation proceeds. For information, Figure 3.24(b) was calculated over 20 iterations, and we need to use sufficient iterations to ensure that convergence has been achieved. Figure 3.27 shows how we approach this. Figure 3.27(a) is after a single iteration, Figure 3.27(b) after 2, Figure 3.27(c) after 5, Figure 3.27(d) after 10, and Figure 3.27(e) after 20. Manifestly, we could choose to reduce the number of iterations, accepting a different result—or even go further.

We also need to choose values for k and λ. By analogy, k is the conduction coefficient and low values preserve edges and high values allow diffusion (conduction) to occur—how much smoothing can take place. The two parameters are naturally interrelated though λ largely controls the amount of smoothing. Given that low values of both parameters mean that no filtering effect is observed, we can investigate their effect by setting one parameter to a high value and varying the other. In Figure 3.28(a)–(c), we use a high value of k which means that edges are not preserved, and we can observe that different values of λ control the amount of smoothing. (A discussion of how this Gaussian filtering process is achieved can be inferred from Section 4.2.4.) Conversely, we can see how different values for k control the level of edge preservation in Figure 3.28(d)–(f) where some structures around the eye are not preserved for larger values of k.

The original presentation of anisotropic diffusion (Perona and Malik, 1990) is extremely lucid and well worth a read if you consider selecting this technique. Naturally, it has greater detail on formulation and analysis of results than space here allows for (and is suitable at this stage). Among other papers on this topic, one (Black et al., 1998) studied the choice of conduction coefficient leading to a function that preserves sharper edges and improves automatic termination. As ever, with techniques that require much computation, there have been approaches that speed implementation or achieve similar performance faster (e.g., Fischl and Schwartz, 1999).

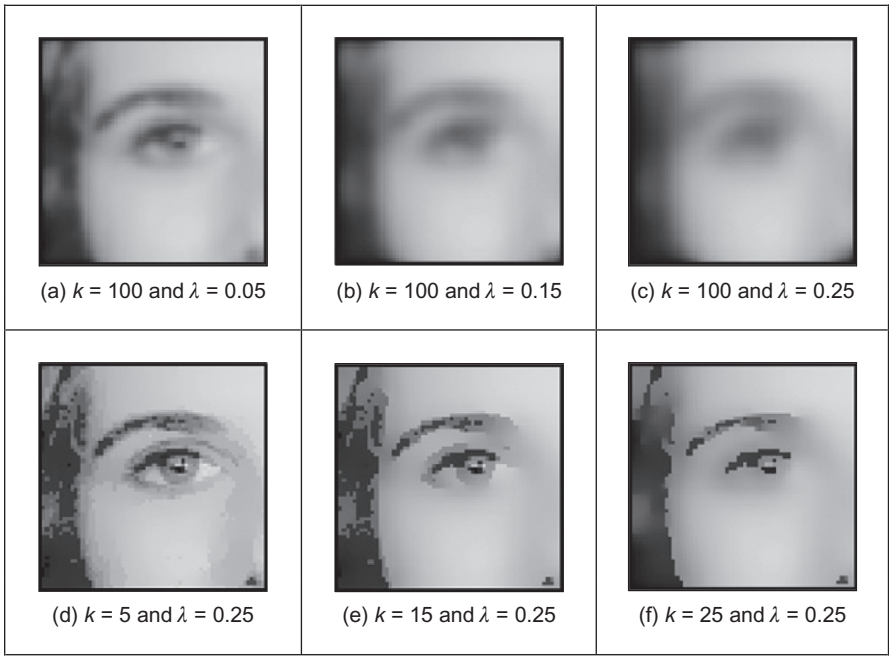


FIGURE 3.28

Applying anisotropic diffusion.

Bilateral filtering is a nonlinear filter introduced by Tomasi and Manduchi (1998). It derives from Gaussian blur, but it prevents blurring across feature boundaries by decreasing the filter weight when the intensity difference is too large. Essentially, the output combines smoothing with edge preservation, so the output $\mathbf{J}$ at point $\mathbf{s}$ is a function of

$$\mathbf{J}_\mathbf{s} = \frac{1}{k(\mathbf{s})} \sum_{\mathbf{c} \in \Omega} f(\mathbf{c} - \mathbf{s}) g(\mathbf{P}_\mathbf{c} - \mathbf{P}_\mathbf{s}) \mathbf{P}_\mathbf{c} \tag{3.44}$$

where Ω is the window of interest, $\mathbf{P}$ is image data, and $k(\mathbf{s})$ is for normalization. f is Gaussian smoothing in space, and g is Gaussian smoothing on the difference in intensity, thereby forming a result which is akin with that of bilateral filtering—indeed, the relationship has already been established (Barash, 2002). One of the major advantages is that the number of iterations is reduced and optimized versions are available, which do not need parameter selection (Weiss, 2006; Paris and Durand, 2008). Another method is based on the use of the frequency domain and uses linear filtering (Dabov et al., 2007). Naturally, this begs the question: when will denoising approaches be developed which are at the performance limit? One such study (Chatterjee and Milanfar, 2010) concluded that "despite the phenomenal recent progress in the quality of denoising algorithms, some room for improvement still remains for a wide class of general images." Denoising is not finished yet.

(a) Image of ear

(b) Magnitude of force field transform

FIGURE 3.29

Illustrating the force field transform.

3.5.4 Force field transform

There are of course many more image filtering operators; we have so far covered those that are among the most popular. There are others which offer alternative insight, sometimes developed in the context of a specific application. For example, Hurley developed a transform called the force field transform (Hurley et al., 2002, 2005) that uses an analogy to gravitational force. The transform pretends that each pixel exerts a force on its neighbors, which is inversely proportional to the square of the distance between them. This generates a force field where the net force at each point is the aggregate of the forces exerted by all the other pixels on a "unit test pixel" at that point. This very large-scale summation affords very powerful averaging that reduces the effect of noise. The approach was developed in the context of ear biometrics, recognizing people by their ears, that has unique advantage as a biometric in that the shape of people's ears does not change with age, and of course—unlike a face—ears do not smile! The force field transform of an ear (Figure 3.29(a)) is shown in Figure 3.29(b). Here, the averaging process is reflected in the reduction of the effects of hair. The transform itself has highlighted ear structures, especially the top of the ear and the lower "keyhole" (the notch).

The image shown is actually the magnitude of the force field. The transform itself is a vector operation and includes direction (Hurley et al., 2002). The transform is expressed as the calculation of the force $\mathbf{F}$ between two points at positions $\mathbf{r}_i$ and $\mathbf{r}_j$ which is dependent on the value of a pixel at point $\mathbf{r}_i$ as

$$\mathbf{F}_i(\mathbf{r}_j) = \mathbf{P}(\mathbf{r}_i)\frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} \tag{3.45}$$

which assumes that the point $\mathbf{r}_j$ is of unit "mass." This is a directional force (which is why the inverse square law is expressed as the ratio of the difference to its magnitude cubed), and the magnitude and directional information has been exploited to determine an ear "signature" by which people can be recognized. In application, Eq. (3.45) can be used to define the coefficients of a template that is convolved with an image (implemented by the FFT to improve speed), as with many of the techniques that have been covered in this chapter; a Mathcad implementation is also given (Hurley et al., 2002). Note that this transform actually exposes low-level features (the boundaries of the ears), which is the focus of the next chapter. How we can determine shapes is a higher level process, and how the processes by which we infer or recognize identity from the low- and the high-level features will be covered in Chapter 8.

3.5.5 Comparison of statistical operators

The different image filtering operators are shown by way of comparison in Figure 3.30. All operators are 5×5 and are applied to the earlier ultrasound image (Figure 3.23(a)). Figure 3.30(a), (b), (c), and (d) are the result of the mean (direct averaging), Gaussian averaging, median, and truncated median, respectively. We have just shown the advantages of anisotropic diffusion compared with Gaussian smoothing, so we will not repeat it here. Each operator shows a different performance: the mean operator removes much noise but blurs feature boundaries; Gaussian averaging retains more features but shows little advantage over direct averaging (it is not Gaussian-distributed noise anyway); the median operator retains some noise but with clear feature boundaries, whereas the truncated median removes more noise but along with picture detail. Clearly, the increased size of the truncated median template, by the results shown in Figure 3.23(b) and (c), can offer improved performance. This is to be expected since by increasing the size of the truncated median template, we are essentially increasing the size of the distribution from which the mode is found.

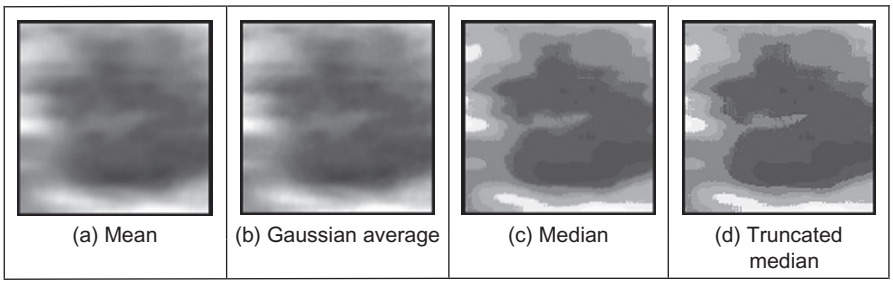


FIGURE 3.30

Comparison of filtering operators.

As yet, however, we have not yet studied any quantitative means to evaluate this comparison. We can only perform subjective appraisal of the images shown in Figure 3.30. This appraisal has been phrased in terms of the contrast boundaries perceived in the image and on the basic shape that the image presents. Accordingly, better appraisal is based on the use of feature extraction. Boundaries are the low-level features studied in Chapter 4; shape is a high-level feature studied in Chapter 5. Also, we shall later use the filtering operators as a basis for finding objects which move in sequences of images (see Section 9.2.1).

3.6 Mathematical morphology

Mathematical morphology analyzes images by using operators developed using set theory (Serra, 1986; Serra and Soille, 1994). It was originally developed for binary images and was extended to include gray-level data. The word morphology actually concerns shapes: in mathematical morphology we process images according to shape, by treating both as sets of points. In this way, morphological operators define **local transformations** that change pixel values that are represented as **sets**. The ways pixel values are changed are formalized by the definition of the *hit or miss transformation*.

In the hit and miss transformation, an object represented by a set X is examined through a structural element represented by a set B. Different structuring elements are used to change the operations on the set X. The hit or miss transformation is defined as the point operator

$$X \otimes B = \left\{ x \middle| B_x^1 \subset X \cap B_x^2 \subset X^c \right\} \tag{3.46}$$

In this equation, x represents one element of X, which is a pixel in an image. The symbol X^c denotes the complement of X (the set of image pixels which is not in the set X) and the *structuring element* B is represented by two parts, B^1 and B^2, that are applied to the set X or to its complement X^c. The structuring element is a shape and this is how mathematical morphology operations process images according to shape properties. The operation of B^1 on X is a "**hit**"; the operation of B^2 on X^c is a "**miss**." The subindex x in the structural element indicates that it is moved to the position of the element x. That is, in a manner similar to other group operators, B defines a window that is moved through the image.

Figure 3.31 illustrates a binary image and a structuring element. Image pixels are divided into those belonging to X and those belonging to its complement X^c. The figure shows a structural element and its decomposition into the two sets B^1 and B^2. Each subset is used to analyze the set X and its complement. Here, we use black for the elements of B^1 and white for B^2 to indicate that they are applied to X and X^c, respectively.

Equation (3.46) defines a process that moves the structural element B to be placed at each pixel in the image, and it performs a pixel by pixel comparison

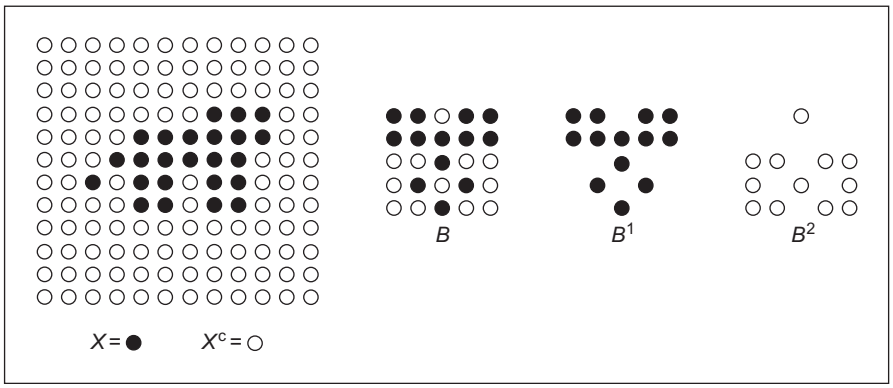


FIGURE 3.31

Image and structural element.

against the template B. If the value of the image is the same as that of the structuring element, then the image's pixel forms part of the resulting set $X \otimes B$. An important feature of this process is that it is not invertible. That is, information is removed in order to suppress or enhance geometrical features in an image.

3.6.1 Morphological operators

The simplest form of morphological operators is defined when either B^1 or B^2 is empty. When B^1 is empty, Eq. (3.46) defines an *erosion* (reduction), and when B^2 is empty, it defines a *dilation* (increase). That is, an erosion operation is given by

$$X \ominus B = \{x | B_x^1 \subset X\} \tag{3.47}$$

and a dilation is given by

$$X \oplus B = \{x | B_x^2 \subset X^c\} \tag{3.48}$$

In the erosion operator, the hit or miss transformation establishes that a pixel x belongs to the eroded set if each point of the element B^1 translated to x is on X. Since all the points in B^1 need to be in X, this operator removes the pixels at the borders of objects in the set X. Thus, it actually erodes or shrinks the set. One of the most common applications of this is to remove noise in thresholded images. This is illustrated in Figure 3.32(a) where we have a noisy binary image, the image is eroded in Figure 3.32(b), removing noise but making the letters smaller, and this is corrected by opening in Figure 3.32(c). We shall show how we can use shape to improve this filtering process—put the morph into morphology.

Figure 3.33 illustrates the operation of the erosion operator. Figure 3.33(a) contains a 3×3 template that defines the structural element B^1. The center pixel is the origin of the set. Figure 3.33(b) shows an image containing a region of black pixels that defines the set X. Figure 3.33(c) shows the result of the erosion.

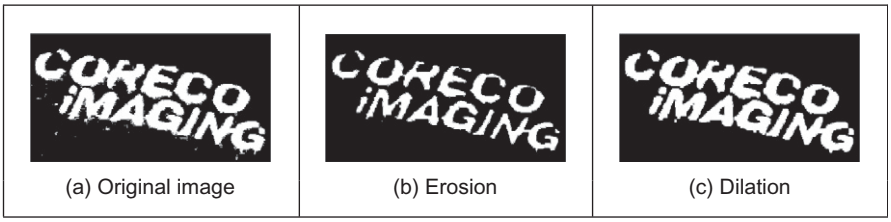


FIGURE 3.32

Filtering by morphology.

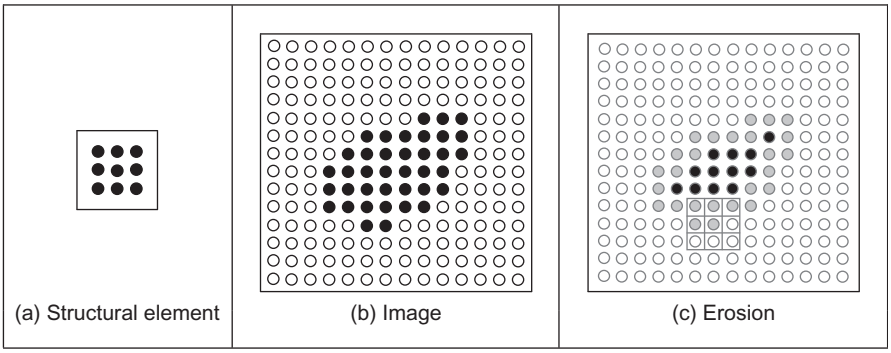


FIGURE 3.33

Example of the erosion operator.

The eroded set is formed only from black pixels, and we use gray to highlight the pixels that were removed from X by the erosion operator. For example, when the structural element is moved to the position shown as a grid in Figure 3.33(c), the central pixel is removed since only five pixels of the structural element are in X.

The dilation operator defined in Eq. (3.48) establishes that a point belongs to the dilated set when all the points in B^2 are in the complement. This operator erodes or shrinks the complement and when the complement is eroded, the set X is dilated.

Figure 3.34 illustrates a dilation process. The structural element shown in Figure 3.34(a) defines the set B^2. We indicate its elements in white since it should be applied to the complement of X. Figure 3.34(b) shows an image example and Figure 3.34(c) the result of the dilation. The black and gray pixels belong to the dilation of X. We use gray to highlight the pixels that are added to the set. During the dilation, we place the structural element on each pixel in the complement, i.e., the white pixels in Figure 3.34(b). When the structural element is not fully contained, it is removed from the complement, so it becomes part of X. For example, when the structural element is moved to the position shown as a grid in Figure 3.34(c), the central pixel is removed from the complement since one of the pixels in the template is in X.

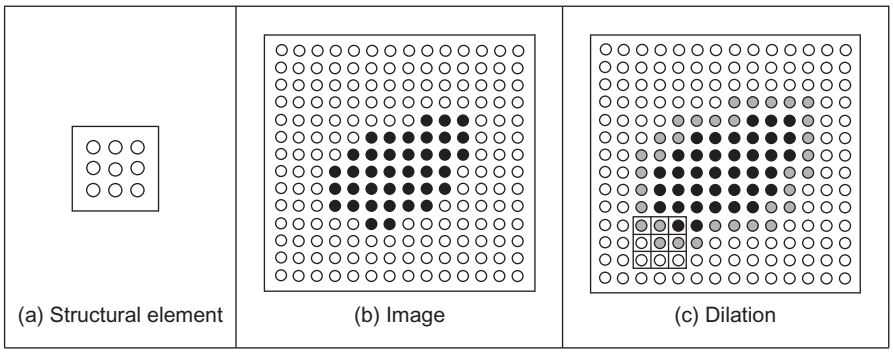


FIGURE 3.34

Example of the dilation operator.

There is an alternative formulation for the dilation operator that defines the transformation over the set X instead to its complement. This definition is obtained by observing that when all elements of B^2 are in X^c is equivalent to none of the elements in the negation of B^2 are in X. That is, dilation can also be written as intersection of translated sets as

$$X \oplus B = \{x | x \in \neg B_x^2\} \tag{3.49}$$

Here the symbol $\neg$ denotes negation, and it changes the structural element from being applied to the complement to the set. For example, the negation of the structural element in Figure 3.34(a) is the set in Figure 3.33(a). Thus, Eq. (3.49) defines a process where a point is added to the dilated set when at least one element of $\neg B^2$ is in X. For example, when the structural element is at the position shown in Figure 3.34(c), one element in X is in the template, thus the central point is added to the dilation.

Neither dilation nor erosion specify a required shape for the structuring element. Generally, it is defined to be square or circular, but other shapes like a cross or a triangle can be used. Changes in the shape will produce subtle changes in the results, but the main feature of the structural element is given by its size since this determines the "strength" of the transformation. In general, applications prefer to use small structural elements (for speed) and perform a succession of transformations until a desirable result is obtained. Other operators can be defined by sequences of erosions and dilations. For example, the *opening operator* is defined by an **erosion** followed by a **dilation**, i.e.,

$$X \circ B = (X \ominus B) \oplus B \tag{3.50}$$

Similarly, a *closing operator* is defined by a **dilation** followed of an **erosion**, i.e.,

$$X \bullet B = (X \oplus B) \ominus B \tag{3.51}$$

Closing and opening operators are generally used as filters that remove dots characteristic of pepper noise and to smooth the surface of shapes in images.

These operators are generally applied in succession and the number of times they are applied depends on the structural element size and image structure.

In addition to filtering, morphological operators can also be used to develop other image processing techniques. For example, edges can be detected by subtracting the original image and the one obtained by an erosion or dilation. Another example is the computation of skeletons that are thin representations of a shape. A skeleton can be computed as the union of subtracting images obtained by applying erosions and openings with structural elements of increasing sizes.

3.6.2 Gray-level morphology

In Eq. (3.46), pixels belong to either the set X or its complement. Thus, it applies only to binary images. Gray scale or *gray-level morphology* extends Eq. (3.46) to represent functions as sets, thus morphology operators can be applied to gray-level images. There are two alternative representations of functions as sets: the cross section (Serra, 1986; Serra and Soille, 1994) and the umbra (Sternberg, 1986). The cross-sectional representation uses multiple thresholds to obtain a pile of binary images. Thus, the definition of Eq. (3.46) can be applied to gray-level images by considering a collection of binary images as a stack of binary images formed at each threshold level. The formulation and implementation of this approach is cumbersome since it requires multiple structural elements and operators over the stack. The *umbra approach* is more intuitive and it defines sets as the points contained below functions. The umbra of a function $f(x)$ consists of all points that satisfy $f(x)$, i.e.,

$$U(X) = \{(x, z) | z < f(x)\} \tag{3.52}$$

Here, x represents a pixel and $f(x)$ its gray level. Thus, the space (x,z) is formed by the combination of all pixels and gray levels. For images, x is defined in 2D, thus all the points of the form (x,z) define a cube in 3D space. An *umbra* is a collection of points in this 3D space. Notice that morphological definitions are for discrete sets, thus the function is defined at discrete points and for discrete gray levels.

Figure 3.35 illustrates the concept of an umbra. For simplicity we show $f(x)$ as 1D function. In Figure 3.35(a), the umbra is drawn as a collection of points below the curve. The complement of the umbra is denoted as $U^c(X)$, and it is given by the points on and above the curve. The union of $U(X)$ and $U^c(X)$ defines all the image points and gray-level values (x,z). In gray-level morphology, images and structural elements are represented by umbrae. Figure 3.35(b) illustrates the definition of two structural elements. The first example defines a structural element for the umbra, i.e., B^1. Similar to an image function, the umbra of the structural elements is defined by the points under the curve. The second example in Figure 3.35(b) defines a structural element for the complement, i.e., B^2. Similar to the complement of the umbra, this operator defines the points on and over the curve.

The hit or miss transformation in Eq. (3.46) is extended to gray-level functions by considering the *inclusion operator* in the umbrae, i.e.,

$$U(X \otimes B) = \left\{ (x, z) \middle| U(B^1_{x,z}) \subset U(X) \cap U(B^2_{x,z}) \subset U^c(X) \right\} \tag{3.53}$$

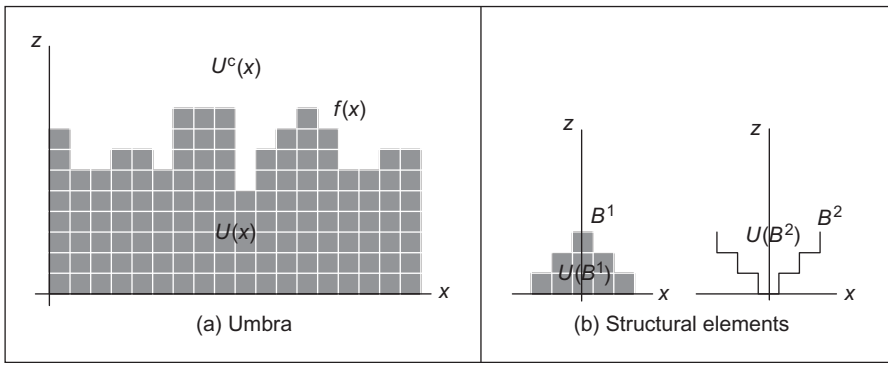


FIGURE 3.35

Gray-level morphology.

Similar to the binary case, this equation defines a process that evaluates the inclusion of the translated structural element B. At difference of the binary definition, the structural element is translated along the pixels and gray-level values, i.e., to the points (x,z). Thus, a point (x,z) belongs to the umbra of the hit or miss transformation, if the umbrae of the elements B^1 and B^2 translated to (x,z) are included in the umbra and its complement, respectively. The inclusion operator is defined for the umbra and its complement in different ways. An umbra is contained in other umbra if corresponding values of its function are equal or lower. For the complement, an umbra is contained if corresponding values of its function are equal or greater.

We can visualize the process in Eq. (3.53) by translating the structural element in the example given in Figure 3.35. To know if a point (x,z) is in the transformed set, we move the structural element B^1 to the point and see if its umbra fully intersects $U(X)$. If that is the case, the umbra of the structural element is contained in the umbra of the function and $U(B^1_{x,t}) \subset U(X)$ is true. Similarly, to test for $U(B^2_{x,t}) \subset U^c(X)$, we move the structural element B^2 and see if it is contained in the upper region of the curve. If both conditions are true, then the point where the operator is translated belongs to the umbra of the hit or miss transformation.

3.6.3 Gray-level erosion and dilation

Based on the generalization in Eq. (3.53), it is possible to reformulate operators developed for binary morphology, so they can be applied to gray-level data. The *erosion* and *dilation* defined in Eqs (3.47) and (3.48) are generalized to gray-level morphology as

$$U(X \ominus B) = \{(x,z) | U(B^1_{x,z}) \subset U(X)\} \tag{3.54}$$

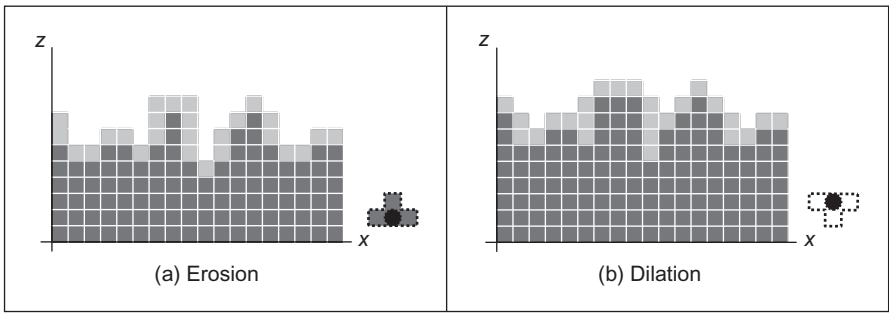


FIGURE 3.36

Gray-level operators.

and

$$U(X \oplus B) = \{(x, z) | U(B^2_{x,z}) \subset U^c(X)\} \tag{3.55}$$

The erosion operator establishes that the point (x,z) belongs to the umbra of the eroded set if each point of the umbra of the element B^1 translated to the point (x,z) is under the umbra of X. A common way to visualize this process is to think that we move the structural element upward in the gray-level axis. The erosion border is the highest point we can reach without going out of the umbra. Similar to the binary case, this operator removes the borders of the set X by increasing the separation in holes. Thus, it actually erodes or shrinks the structures in an image. Figure 3.36(a) illustrates the erosion operator for the image in Figure 3.35(a). Figure 3.36(a) shows the result of the erosion for the structural element shown in the right. For clarity, we have marked the origin of the structure element with a black spot. In the result, only the black pixels form the eroded set, and we use gray to highlight the pixels that were removed from the umbra of X. It is easy to see that when the structural element is translated to a point that is removed, its umbra intersects $U^c(X)$.

Analogous to binary morphology, the dilation operator can be seen as an erosion of the complement of the umbra of X. That is, a point belongs to the dilated set when all the points in the umbra of B^2 are in $U^c(X)$. This operator erodes or shrinks the set $U^c(X)$. When the complement is eroded, the umbra of X is dilated. The dilation operator fills holes decreasing the separation between prominent structures. This process is illustrated in Figure 3.36(b) for the example given in Figure 3.36(a). The structural element used is shown to the right in Figure 3.36 (b). In the results, the black and gray pixels belong to the dilation. We use gray to highlight points that are added to the set. Points are removed from the complement and added to $U(X)$ by translating the structural element looking for points where the structural element is not fully included in $U^c(X)$. It is easy to see that when the structural element is translated to a point that is added to the dilation, its umbra intersects $U(X)$.

Similar to Eq. (3.49), dilation can be written as intersection of translated sets, thus it can be defined as an operator on the umbra of an image, i.e.,

$$U(X \oplus B) = \{(x, z) | (x, z) \in U(\neg B^2_{x,z})\} \tag{3.56}$$

The negation changes the structural element from being applied to the complement of the umbra to the umbra. That is, it changes the sign of the umbra to be defined below the curve. For example in Figure 3.36(b), it easy to see that if the structural element $\neg B^2$ is translated to any point added during the dilation, it intersects at least in one point.

3.6.4 Minkowski operators

Equations (3.54)–(3.56) require the computation of intersections of the pixels of a structural element that is translated to all the points in the image and for each gray-level value. Thus, its computation involves significant processing. However, some simplifications can be made. For the erosion process in Eq. (3.54), the value of a pixel can be simply computed by comparing the gray-level values of the structural element and corresponding image pixels. The highest position that we can translate the structural element without intersecting the complement is given by the minimum value of the difference between the gray level of the image pixel and the corresponding pixel in the structural element, i.e.,

$$\ominus(x) = \min_i \{f(x - i) - B(i)\} \tag{3.57}$$

Here, $B(i)$ denotes the value of the ith pixel of the structural element. Figure 3.37(a) illustrates a numerical example for this equation. The structural element has three pixels with values 0, 1, and 0, respectively. The subtractions for the position shown in Figure 3.37(a) are $4 - 0 = 4$, $6 - 1 = 5$, and $7 - 0 = 7$. Thus, the minimum value is 4. As shown in Figure 3.37(a), this corresponds to the highest gray-level value that we can move up to the structural element, and it is still fully contained in the umbra of the function.

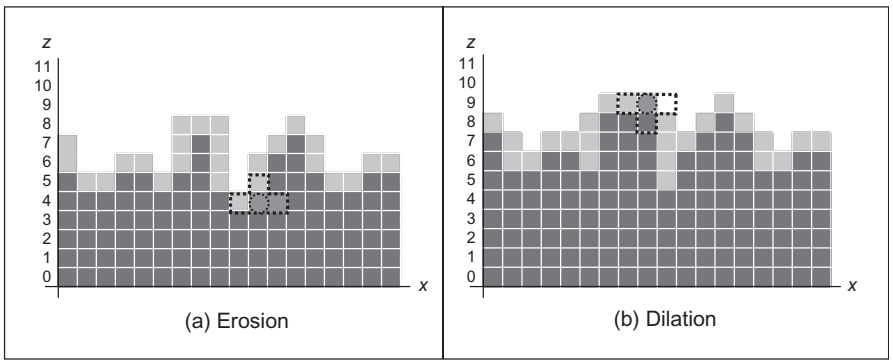


FIGURE 3.37

Example of Minkowski difference and addition.

Similar to Eq. (3.57), the dilation can be obtained by comparing the gray-level values of the image and the structural element. For the dilation we have that

$$\oplus(x) = \max_i\{f(x-i) + B(i)\} \tag{3.58}$$

Figure 3.37(b) illustrates a numerical example of this equation. For the position of the structural element in Figure 3.37(b), the summation gives the values $8+0=8$, $8+1=9$, and $4+0=4$. As shown in the figure, the maximum value of 9 corresponds to the point where the structural element still intersects the umbra; thus this point should be added to the dilation.

Equations (3.57) and (3.58) are known as the *Minkowski operators* and they formalize set operations as summations and differences. Thus, they provide definitions very useful for computer implementations. Code 3.15 shows the implement of the erosion operator based on Eq. (3.57). Similar to Code 3.5, the value pixels in the output image are obtained by translating the operator along the image pixels. The code subtracts the value of corresponding image and template pixels, and it sets the value of the pixel in the output image to the minima.

```
function eroded = Erosion(image,template)
%Implementation of erosion operator
%Parameters: Template and image array of points

%get the image and template dimensions
[irows,icols]=size(image);
[trows,tcols]=size(template);

%create result image
eroded(1:irows,1:icols)=uint8(0);

%half of template
trhalf=floor(trows/2);
tchalf=floor(tcols/2);

%Erosion
for x=trhalf+1:icols-trhalf %columns in the image except border
   for y=tchalf+1:irows-tchalf %rows in the image except border
      min=256;
      for iwin=1:tcols     %template columns
         for jwin=1:trows     %template rows
            xi=x-trhalf-1+iwin;
            yi=y-tchalf-1+jwin;
            sub=double(image(xi,yi))-double(template(iwin,jwin));
            if sub<min & sub>0
               min=sub;
            end
         end
      end
         eroded(x,y)=uint8(min);
   end
end
```

CODE 3.15

Erosion implementation.

Code 3.16 shows the implement of the dilation operator based on Eq. (3.58). This code is similar to Code 3.15, but corresponding values of the image and the structural element are added, and the maximum value is set as the result of the dilation.

```
function dilated = Dilation(image,template)
%Implementation of dilation operator
%Parameters: Template and image array of points

%get the image and template dimensions
[irows,icols]=size(image);
[trows,tcols]=size(template);

%create result image
dilated(1:irows,1:icols)=uint8(0);

%half of template
trhalf=floor(trows/2);
tchalf=floor(tcols/2);

%Dilation
for x=trhalf+1:icols-trhalf %columns in the image except border
   for y=tchalf+1:irows-tchalf %rows in the image except border
       max=0;
       for iwin=1:tcols    %template columns
         for jwin=1:trows    %template rows
             xi=x-trhalf-1+iwin;
             yi=y-tchalf-1+jwin;
             sub=double(image(xi,yi))+double(template(iwin,jwin));
             if sub>max & sub>0
                max=sub;
             end
         end
       end
       dilated(x,y)=uint8(max);
   end
end
```

CODE 3.16

Dilation implementation.

Figure 3.38 shows an example of the results obtained from the erosion and dilation using Codes 3.15 and 3.16. The original image shown in Figure 3.38(a) has 128×128 pixels, and we used a flat structural element defined by an image with 9×9 pixels set to zero. For its simplicity, flat structural elements are very common in applications, and they are generally set to zero to avoid creating offsets in the gray levels. In Figure 3.38, we can see that the erosion operation

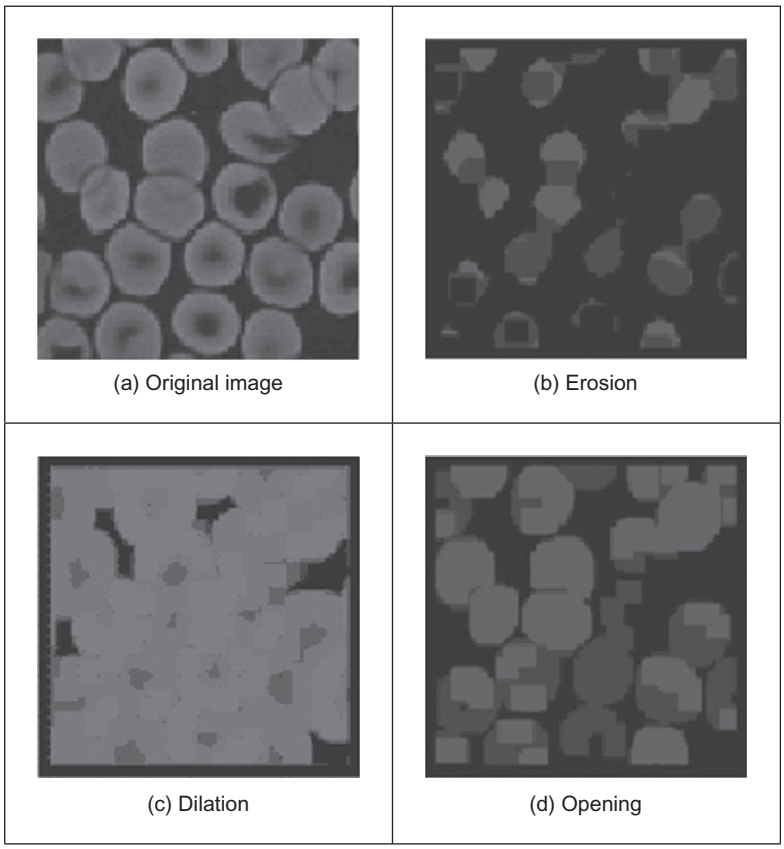


FIGURE 3.38

Examples of morphology operators.

reduces the objects in the image while dilation expands white regions. We also used the erosion and dilation in succession to perform the opening show in Figure 3.38(d). The opening operation has a tendency to form regular regions of similar size to the original image while removing peaks and small regions. The "strength" of the operators is defined by the size of the structural elements. In these examples, we use a fixed size and we can see that it strongly modifies regions during dilation and erosion. Elaborate techniques have combined multiresolution structures and morphological operators to analyze an image with operators of different sizes (Montiel et al., 1995). We shall see the deployment of morphology later, to improve the results when finding moving objects in sequences of images (see Section 9.2.1.2).

3.7 Further reading

Many texts cover basic point and group operators in much detail, in particular some texts give many more examples, such as Russ (2002) and Seul et al. (2000). Books with a C implementation often concentrate on more basic techniques including low-level image processing (Lindley, 1991; Parker, 1994). Some of the more advanced texts include more coverage of low-level operators, such as Rosenfeld and Kak (1982) and Castleman (1996). Parker (1994) includes C code for nearly all the low-level operations in this chapter and Seul et al. (2000) has code too, and there is MATLAB code in Gonzalez et al. (2003). For study of the effect of the median operator on image data, see Bovik et al. (1987). Some of the newer techniques receive little treatment in the established literature, except for Chan and Shen (2005) (with extensive coverage of noise filtering too). The Truncated Median Filter is covered again in Davies (2005). Notwithstanding the discussion on more recent denoising operators at the end of Section 3.5.3; for further study of the effects of different statistical operators on ultrasound images, see Evans and Nixon (1995) and Evans and Nixon (1996). The concept of scale space allows for considerably more-refined analysis than is given here and we shall revisit it later. It was originally introduced by Witkin (1983) and further developed by others including Koenderink (1984) (who also considers the heat equation). There is even a series of conferences devoted to scale space and morphology.

3.8 References

Bamber, J.C., Daft, C., 1986. Adaptive filtering for reduction of speckle in ultrasonic pulse-echo images. Ultrasonics 24 (3), 41–44.

Barash, D., 2002. A fundamental relationship between bilateral filtering, adaptive smoothing and the nonlinear diffusion equation. IEEE Trans. PAMI 24 (6), 844–849.

Black, M.J., Sapiro, G., Marimont, D.H., Meeger, D., 1998. Robust anisotropic diffusion. IEEE Trans. IP 7 (3), 421–432.

Bovik, A.C., Huang, T.S., Munson, D.C., 1987. The effect of median filtering on edge estimation and detection. IEEE Trans. PAMI 9 (2), 181–194.

Campbell, J.D., 1969. Edge Structure and the Representation of Pictures. PhD Thesis, University of Missouri, Columbia, SC.

Castleman, K.R., 1996. Digital Image Processing. Prentice Hall, Englewood Cliffs, NJ.

Chan, T., Shen, J., 2005. Image Processing and Analysis: Variational, PDE, Wavelet, and Stochastic Methods. Society for Industrial and Applied Mathematics.

Chatterjee, P., Milanfar, P., 2010. Is denoising dead? IEEE Trans. IP 19 (4), 895–911.

Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K., 2007. Image denoising by sparse 3-D transform-domain collaborative filtering. IEEE Trans. IP 16 (8), 2080–2095.

Davies, E.R., 1988. On the noise suppression characteristics of the median, truncated median and mode filters. Pattern Recog. Lett. 7 (2), 87–97.

Davies, E.R., 2005. Machine Vision: Theory, Algorithms and Practicalities, third ed. Morgan Kaufmann (Elsevier).

Evans, A.N., Nixon, M.S., 1995. Mode filtering to reduce ultrasound speckle for feature extraction. Proc. IEE Vision Image Signal Process. 142 (2), 87–94.

Evans, A.N., Nixon, M.S., 1996. Biased motion-adaptive temporal filtering for speckle reduction in echocardiography. IEEE Trans. Med. Imaging 15 (1), 39–50.

Fischl, B., Schwartz, E.L., 1999. Adaptive nonlocal filtering: a fast alternative to anisotropic diffusion for image enhancement. IEEE Trans. PAMI 21 (1), 42–48.

Glasbey, C.A., 1993. An analysis of histogram-based thresholding algorithms. CVGIP: Graph. Models Image Process. 55 (6), 532–537.

Gonzalez, R.C., Wintz, P., 1987. Digital Image Processing, second ed. Addison-Wesley, Reading, MA.

Gonzalez, R.C., Woods, R.E., Eddins, S., 2003. Digital Image Processing Using MATLAB, first ed. Prentice Hall.

Hearn, D., Baker, M.P., 1997. Computer Graphics C Version, second ed. Prentice Hall, Upper Saddle River, NJ.

Hodgson, R.M., Bailey, D.G., Naylor, M.J., Ng, A., McNeill, S.J., 1985. Properties, implementations and applications of rank filters. Image Vision Comput. 3 (1), 3–14.

Huang, T., Yang, G., Tang, G., 1979. A fast two-dimensional median filtering algorithm. IEEE Trans. Acoust. Speech Signal Process. 27 (1), 13–18.

Hurley, D.J., Nixon, M.S., Carter, J.N., 2002. Force field energy functionals for image feature extraction. Image Vision Comput. 20, 311–317.

Hurley, D.J., Nixon, M.S., Carter, J.N., 2005. Force field feature extraction for ear biometrics. Comput. Vision Image Understanding 98 (3), 491–512.

Koenderink, J., 1984. The structure of images. Biol. Cybern. 50, 363–370.

Lee, S.A., Chung, S.Y., Park, R.H., 1990. A comparative performance study of several global thresholding techniques for segmentation. CVGIP 52, 171–190.

Lindley, C.A., 1991. Practical Image Processing in C. Wiley, New York, NY.

Loupas, T., McDicken, W.N., 1987. Noise reduction in ultrasound images by digital filtering. Br. J. Radiol. 60, 389–392.

Montiel, M.E., Aguado, A.S., Garza, M., Alarcón, J., 1995. Image manipulation using M-filters in a pyramidal computer model. IEEE Trans. PAMI 17 (11), 1110–1115.

Otsu, N., Threshold, A, 1979. Selection method from gray-level histograms. IEEE Trans. SMC 9 (1), 62–66.

Paris, S., Durand, F., 2008. A fast approximation of the bilateral filter using a signal processing approach. Int. J. Comput. Vision 81 (1), 24–52.

Parker, J.R., 1994. Practical Computer Vision Using C. Wiley, New York, NY.

Perona, P., Malik, J., 1990. Scale-space and edge detection using anisotropic diffusion. IEEE Trans. PAMI 17 (7), 620–639.

Rosenfeld, A., Kak, A.C., 1982. second ed. Digital Picture Processing, vols. 1 and 2. Academic Press, Orlando, FL.

Rosin, P.L., 2001. Unimodal thresholding. Pattern Recog. 34 (11), 2083–2096.

Russ, J.C., 2002. The Image Processing Handbook, fourth ed. CRC Press (IEEE Press), Boca Raton, FL.

Sahoo, P.K., Soltani, S., Wong, A.K.C., Chen, Y.C., 1988. Survey of thresholding techniques. CVGIP 41 (2), 233–260.

Serra, J., 1986. Introduction to mathematical morphology. Comput. Vision Graph. Image Process. 35, 283–305.

Serra, J.P., Soille, P. (Eds.), 1994. Mathematical Morphology and its Applications to Image Processing. Kluwer Academic Publishers.

Seul, M., O'Gorman, L., Sammon, M.J., 2000. Practical Algorithms for Image Analysis: Descriptions, Examples, and Code. Cambridge University Press, Cambridge.

Shankar, P.M., 1986. Speckle reduction in ultrasound B scans using weighted averaging in spatial compounding. IEEE Trans. Ultrason. Ferroelectr. Freq. Control 33 (6), 754–758.

Sternberg, S.R., 1986. Gray scale morphology. Comput. Vision Graph. Image Process. 35, 333–355.

Tomasi, C., Manduchi, R., 1998. Bilateral filtering for gray and color images. Proceedings of the ICCV, Bombay, India, pp. 839–846.

Trier, O.D., Jain, A.K., 1995. Goal-directed evaluation of image binarisation methods. IEEE Trans. PAMI 17 (12), 1191–1201.

Weiss, B., 2006. Fast median and bilateral filtering. Proc. ACM SIGGRAPH 2006, 519–526.

Witkin, A., 1983. Scale-space filtering: a new approach to multi-scale description. Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1019–1021.