Chapter 10

Segmentation

The division of an image into meaningful structures, *image segmentation*, is often an essential step in image analysis, object representation, visualization, and many other image processing tasks. In chapter 8, we focussed on how to analyze and represent an object, but we assumed the group of pixels that identified that object was known beforehand. In this chapter, we will focus on methods that find the particular pixels that make up an object.

A great variety of segmentation methods has been proposed in the past decades, and some categorization is necessary to present the methods properly here. A disjunct categorization does not seem to be possible though, because even two very different segmentation approaches may share properties that defy singular categorization[1]. The categorization presented in this chapter is therefore rather a categorization regarding the *emphasis* of an approach than a strict division.

The following categories are used:

- **Threshold based segmentation.** Histogram thresholding and slicing techniques are used to segment the image. They may be applied directly to an image, but can also be combined with pre- and post-processing techniques.
- **Edge based segmentation.** With this technique, detected edges in an image are assumed to represent object boundaries, and used to identify these objects.
- **Region based segmentation.** Where an edge based technique may attempt to find the object boundaries and then locate the object itself by filling them in, a region based technique takes the opposite approach, by (*e.g.*) starting in the middle of an object and then "growing" outward until it meets the object boundaries.

[1] In much the same way as the platypus does not seem to fit in any of normal zoological categories. It seems that any segmentation method categorized as a mammal, upon closer inspection, appears to have some aspect that shows it to be laying eggs.

- **Clustering techniques.** Although clustering is sometimes used as a synonym for (agglomerative) segmentation techniques, we use it here to denote techniques that are primarily used in exploratory data analysis of high-dimensional measurement patterns. In this context, clustering methods attempt to group together patterns that are similar in some sense. This goal is very similar to what we are attempting to do when we segment an image, and indeed some clustering techniques can readily be applied for image segmentation.
- **Matching.** When we know what an object we wish to identify in an image (approximately) looks like, we can use this knowledge to locate the object in an image. This approach to segmentation is called matching.

Perfect image segmentation –*i.e.*, each pixel is assigned to the correct object segment– is a goal that cannot usually be achieved. Indeed, because of the way a digital image is acquired, this may be impossible, since a pixel may straddle the "real" boundary of objects such that it partially belongs to two (or even more) objects. Most methods presented here –indeed most current segmentation methods– only attempt to assign a pixel to a single segment, which is an approach that is more than adequate for most applications. Methods that assign a segment probability distribution to each pixel are called *probabilistic*. This class of methods is theoretically more accurate, and applications where a probabilistic approach is the only approach accurate enough for specific object measurements can easily be named. However, probabilistic techniques add considerable complexity to segmentation –both in the sense of concept and implementation– and as such are still little used.

Perfect image segmentation is also often not reached because of the occurrence of *oversegmentation* or *undersegmentation*. In the first case, pixels belonging to the same object are classified as belonging to different segments. A single object may be represented by two or more segments. In the latter case, the opposite happens: pixels belonging to different objects are classified as belonging to the same object. A single segment may contain several objects. Figure 10.1 shows a simple example of over- and undersegmentation.

10.1 Threshold based segmentation

Thresholding is probably the most frequently used technique to segment an image. The thresholding operation is a grey value remapping operation g defined by:

$$g(v) = \begin{cases} 0 & \text{if} \quad v < t \\ 1 & \text{if} \quad v \geqslant t, \end{cases}$$

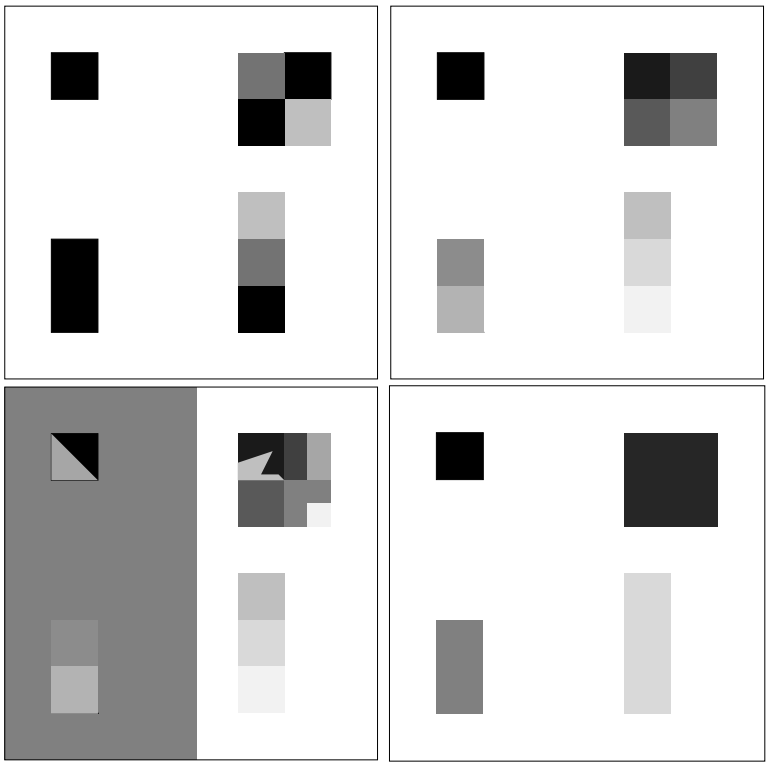


Figure 10.1 An original image is shown at the top left. If it is known that this image contains only uniformly sized squares, then the image on the top right shows the correct segmentation. Each segment has been indicated by a unique grey value here. The bottom left and right images show examples of oversegmentation and undersegmentation respectively.

where v represents a grey value, and t is the threshold value. Thresholding maps a grey-valued image to a binary image. After the thresholding operation, the image has been segmented into two segments, identified by the pixel values 0 and 1 respectively.

If we have an image which contains bright objects on a dark background, thresholding can be used to segment the image. See figure 10.2 for an example. Since in many types of images the grey values of objects are very different from the background value, thresholding is often a well-suited method to segment an image into objects and background. If the objects are not overlapping, then we can create a separate segment from each object by running a labelling algorithm (see chapter 8) on the thresholded binary image, thus assigning a unique pixel value to each object.

Many methods exist to select a suitable threshold value for a segmentation task. Perhaps the most common method is to set the threshold value interactively; the user manipulating the value and reviewing the thresholding result until a satisfying segmentation has been obtained. The histogram is often a valuable tool in establishing a suitable threshold value. In figure 10.3 we show the image from the previous figure together with its

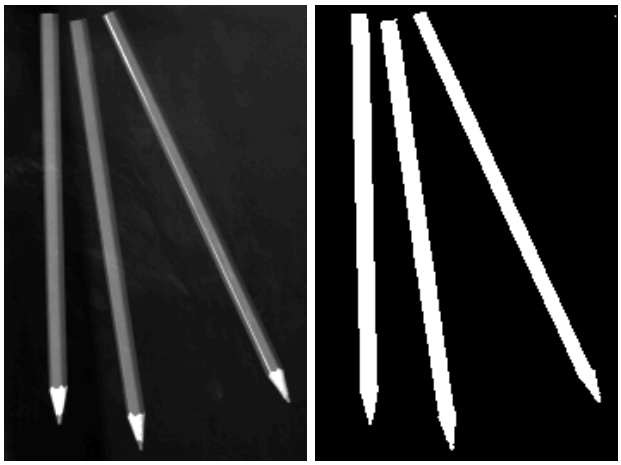


Figure 10.2 Example of segmentation by thresholding. On the left, an original image with bright objects (the pencils) on a dark background. Thresholding using an appropriate threshold segments the image into objects (segment with value 1) and background (segment with value 0).

histogram, and the thresholding results using four different threshold values obtained from the histogram. In the next section, threshold selection methods are discussed in more detail.

When several desired segments in an image can be distinguished by their grey values, threshold segmentation can be extended to use multiple thresholds to segment an image into more than two segments: all pixels with a value smaller than the first threshold are assigned to segment 0, all pixels with values between the first and second threshold are assigned to segment 1, all pixels with values between the second and third threshold are assigned to segment 2, *etc.* If n thresholds $(t_1, t_2, \ldots, t_n)$ are used:

$$g(v) = \begin{cases} 0 & \text{if} \quad v < t_1 \\ 1 & \text{if} \quad t_1 \leqslant v < t_2 \\ 2 & \text{if} \quad t_2 \leqslant v < t_3 \\ \vdots & \vdots \quad \vdots \\ n & \text{if} \quad t_n \leqslant v. \end{cases}$$

After thresholding, the image has been segmented into $n+1$ segments identified by the grey values 0 to n respectively. Figure 10.4 shows an example using two thresholds.

10.1.1 Threshold selection

Many methods exist to find a suitable threshold for segmentation. The simplest method is the interactive selection of a threshold by the user –possibly with the aid of the image

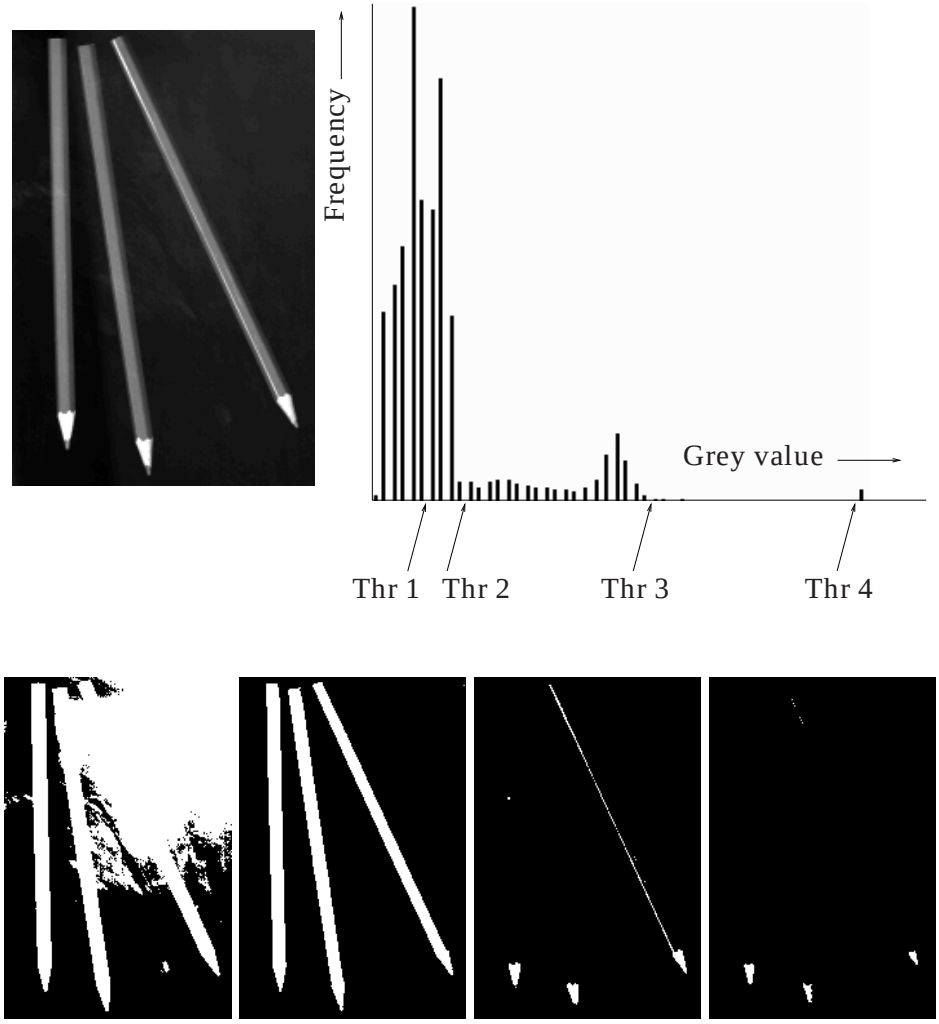


Figure 10.3 Example of threshold selection from the histogram. Top row: original image and histogram. Four special grey values (indicated by Thr 1,2,3,4) have been chosen. The bottom row shows the respective thresholding results at each of the values. At a first glance, the original image appears to have only three grey values. But the histogram shows that the grey value distribution is more diffuse; that the three basic values are in fact spread out over a certain range. Because the background grey values occur most frequently, we expect all of the large values in the left part of the histogram to correspond to the background. This is indeed the case. The result of threshold 1 shows that the peak between thresholds 1 and 2 also corresponds to background values. The result of threshold 2 shows the desired segmentation; every grey value to the right of threshold 2 corresponds to the pencils. Threshold 4 shows that the right-most little peak corresponds to the bright grey value in the tips of the pencils.

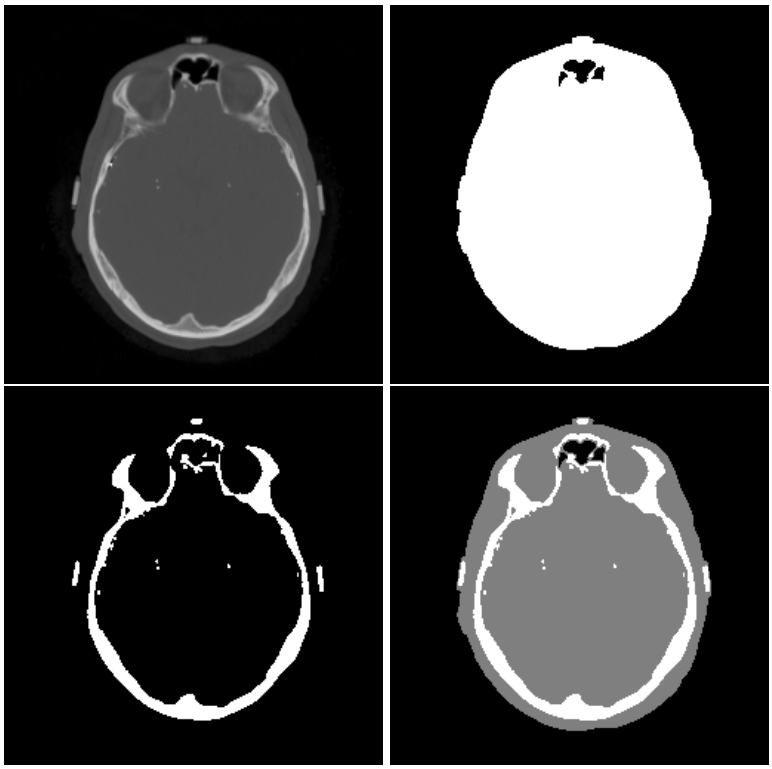


Figure 10.4 Example of using multiple thresholds for segmentation. Top left: original image. Top right: thresholding result after using a low threshold value to segment the image into head and background pixels. Bottom left: result after using a higher value to segment the bone pixels. Bottom right: result after using both thresholds at once.

histogram– a method that is usually accompanied by a graphical tool which lets the user immediately assess the result of a certain choice of threshold. Automatic methods often make use of the image histogram to find a suitable threshold. Some of these methods are detailed below.

Using histogram extrema. A typical objects-on-a-background image as shown in figure 10.5 will have a bimodal[2] histogram. The peaks of the histogram will not generally be as sharp as we would like them to be, because of the influence of image degrading factors such as noise, illumination artifacts and partial volume effects.[3]

In practice, the curves in the histogram corresponding to certain objects may overlap. When this happens, an errorless segmentation based on global thresholding is no longer

[2]Two-peaked.

[3]The partial volume effect occurs when a pixel is on or near the boundary of two or more image objects; when the area that influences the grey value of the pixel contains parts of more than one object. With many types of image acquisition, the grey value of the pixel will then not correspond to the expected grey value of one of the objects, but will be a weighted average of the grey values of the objects involved.

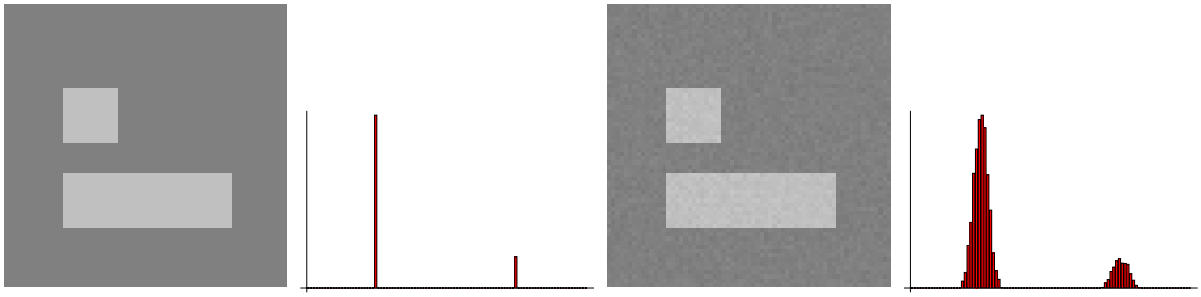


Figure 10.5 At the top left, an image containing only two grey values is shown. The histogram (next to it) shows only two peaks. In practice, image artifacts will cause the image and its histogram to degrade from this ideal situation. In particular, the histogram blurs as is shown in the image and histogram on the right. In practice, it is quite possible that the two curves in the histogram start to overlap.

possible.

Regardless of whether or not there is overlap in the histogram, we can use the maxima (peaks) of the histogram to establish a segmentation threshold. This threshold t may be midway between two peaks with grey values p_1 and p_2:

$$t = \frac{p_1 + p_2}{2},$$

or better still, it may be the grey value at the minimum between the two peaks:

$$t = \arg \min_{v \in [p_1, p_2]} H(v),$$

where $H(v)$ gives the histogram value at grey value v, and we assume that p_1 is smaller than p_2. Figure 10.6 shows why the minimum choice for t is usually better than the midway choice.

The histogram is often not as smooth as we would like, as the figure shows. For many histogram analysis tasks (such as finding the global extrema) it is therefore useful to smooth the histogram beforehand. This can, *e.g.*, be done by convolution with a Gaussian or a discrete averaging kernel.

Minimum variance within segments. If we assume that a segment should have relatively homogeneous grey values, it makes sense to select a threshold that minimizes the variance of the original grey values within segments. Alternatively, we may select a threshold that maximizes the variance *between* objects and background, or one that attempts to optimize both these "within" and "between" variances.

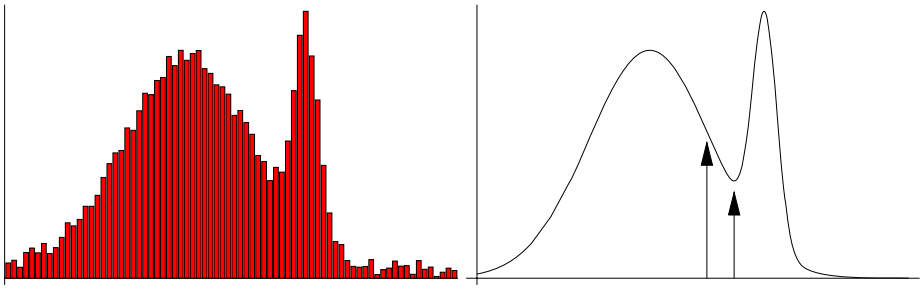


Figure 10.6 Example comparing a threshold midway between peaks and at the minimum between peaks. On the left, a bimodal histogram is shown. The right image shows a smoothed version of this histogram, together with an arrow showing the threshold midway between the peaks (left arrow) and showing the threshold at the minimum (right arrow). The threshold at the minimum will misclassify fewer pixels.

We will give an example using one threshold, *i.e.*, two segments. For simplicity, we normalize the histogram $H(v)$ of the image to a proper distribution $h(v)$, *i.e.*, $\sum h(v) = 1$. This can be achieved by setting $h(v) = H(v)/n$ for all v, where n is the total number of pixels in the image. The variance of the grey values σ^2 in the image then (by definition) equals

$$\sigma^2 = \sum_v (v - \mu)^2 h(v),$$

where $\mu = \sum_v vh(v)$ is the mean grey value of the image. If we segment the image with threshold value t into a background segment 0 and an object segment 1, then the variance of grey values within each segment (respectively σ_0 and σ_1) is[4]

$$\begin{aligned}\sigma_0^2 &= \sum_{v<t} (v - \mu_0)^2 h(v) \\ \sigma_1^2 &= \sum_{v \geqslant t} (v - \mu_1)^2 h(v),\end{aligned}$$

where

$$\mu_0 = \tfrac{1}{h_0} \sum_{v<t} vh(v)$$

and

$$\mu_1 = \tfrac{1}{h_1} \sum_{v \geqslant t} vh(v)$$

[4]Note that these definitions are different –by a factor– from the 'real' statistical definition. This factor is corrected for later in the definition of σ_w^2. This formulation requires less operations to compute σ_w^2.

are the mean grey values of the respective segments 0 and 1. The probabilities h_0 and h_1 that a randomly selected pixel belongs to segment 0 or 1 are

$$h_0 = \sum_{v<t} h(v)$$
$$h_1 = \sum_{v\geqslant t} h(v).$$

Note that $h_0 + h_1 = 1$. The total variance within segments σ_w^2 is

$$\sigma_w^2 = h_0\sigma_0^2 + h_1\sigma_1^2.$$

This variance only depends on the threshold value t: $\sigma_w^2 = \sigma_w^2(t)$. This means that we can find the value of t that minimizes the variance within segments by minimizing $\sigma_w^2(t)$.

The variance *between* the segments 0 and 1, σ_b^2, is

$$\sigma_b^2 = h_0(\mu_0 - \mu)^2 + h_1(\mu_1 - \mu)^2.$$

Again, this variance is only dependent on the threshold value t. Finding the t that maximizes σ_b^2 maximizes the variance between segments. A hybrid approach that attempts to maximize σ_b^2 while minimizing σ_w^2 is to find the threshold t that maximizes the ratio σ_b^2/σ_w^2.

If more than two segments are required, the method described above can be extended to use multiple thresholds. The variances σ_w^2 and σ_b^2 will then be functions of more than one threshold, so we need multi-dimensional optimization to find the set of optimal thresholds. Since this is especially cumbersome if the number of segments is large, a more practical algorithm that minimizes the variances within segments is often used, an iterative algorithm known as *K-means clustering*.

Algorithm: K-means clustering

The objective of the K-means clustering algorithm is to divide an image into K segments (using $K-1$ thresholds), minimizing the total within-segment variance. The variable K must be set before running the algorithm.

The within-segment variance σ_w^2 is defined by

$$\sigma_w^2 = \sum_{i=0}^{K-1} h_i\sigma_i^2,$$

where $h_i = \sum_{v\in S_i} h(v)$ is the probability that a random pixel belongs to segment i (containing the grey values in the range S_i), $\sigma_i^2 = \sum_{v\in S_i}(v-\mu_i)^2 h(v)$ is the variance of grey values of segment i, and $\mu_i = \sum_{v\in S_i} vh(v)$ is the mean grey value in segment i. All definitions are as before in the case with a single threshold.

1. Initialization: distribute the $K-1$ thresholds over the histogram. (For example in such a way that the grey value range is divided into K pieces of equal length.) Segment the image according to the thresholds set. For each segment, compute the 'cluster center',*i.e.,* the value midway between the two thresholds that make up the segment.
2. For each segment, compute the mean pixel value μ_i.
3. Reset the cluster centers to the computed values μ_i.
4. Reset the thresholds to be midway between the cluster centers, and segment the image.
5. Go to step 2. Iterate until the cluster centers do not move anymore (or do not move significantly).

Note that this algorithm –although it minimizes a variance– does not require any variance to be computed explicitly!

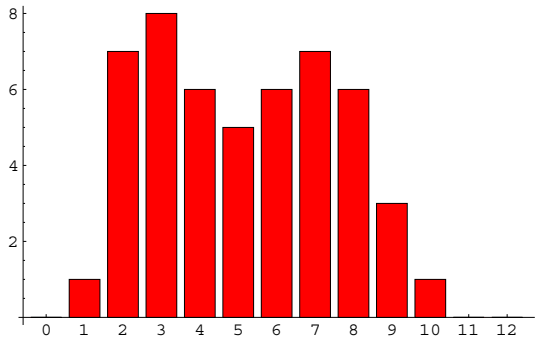
The version of this algorithm with only one threshold (two segments) can be extracted from the general form above. In practice, the initialization step in this case is often slightly different to try to get the initial threshold close to the optimal position, so the algorithm will converge faster. A heuristic rule is often used to achieve this. The algorithm –with a widely employed heuristic– then becomes:

Algorithm: Iterative thresholding

1. Assume that the four corner points of the image are background pixels (part of segment 0), and set μ_0 to the average grey value of these four pixels. Assume all of the other pixels are object pixels, and set μ_1 to their average grey value.
2. Set the threshold t to $t = \frac{1}{2}(\mu_0 + \mu_1)$, and segment the image.
3. Recompute μ_0 and μ_1, the mean of the original grey values of the two segments.
4. Go to step 2 and iterate until the threshold value no longer changes (or no longer changes significantly).

Example

If we have an image with the following histogram:



and if we know that the four corner pixels have grey values 2, 2, 2, and 1 respectively, then the initial value for μ_0 is $\mu_0^{(0)} = \frac{2+2+2+1}{4} = 1.75$. We can compute the average value of the remaining pixels from the image or the histogram: $\mu_1^{(0)} \approx 5.46$. So we set the initial threshold $t^{(0)} = \frac{1}{2}(\mu_0^{(0)} + \mu_1^{(0)}) \approx 3.61$.

Thresholding then results in a segment 0 containing all pixels with original grey values in the range $\{0, \ldots, 3\}$, and a segment 1 containing pixels with grey values in the range $\{4, \ldots, 12\}$.

Computing the mean values of these two segments, we find $\mu_0^{(1)} \approx 2.44$ and $\mu_1^{(1)} \approx 6.44$. The new threshold will therefore be $t^{(1)} = \frac{1}{2}(\mu_0^{(1)} + \mu_1^{(1)}) \approx 4.44$.

Thresholding then results in segment 0 containing the pixels with values in $\{0, \ldots, 4\}$, and segment 1 with the range $\{5, \ldots, 12\}$. Recomputing the mean segment values gives $\mu_0^{(2)} \approx 2.86$ and $\mu_1^{(2)} \approx 6.96$, so $t^{(2)} \approx 4.91$.

This last threshold will result in the same segmentation as the previous step, so we terminate the algorithm here. The last computations of the segments 0 and 1 are assumed to be the optimal ones.

Optimal thresholding. Figure 10.5 showed us that an object that theoretically has a uniform grey value and should cause a single peak in the histogram, in practice shows a distribution of grey values in the histogram. This distribution can often be approximated using a known statistical distribution function such as the Gaussian. *Optimal thresholding* is a technique that approximates the histogram using a weighted sum of distribution functions, and then sets a threshold in such a way that the number of incorrectly segmented pixels (as predicted from the approximation) is minimal.

For an example, we assume the histogram is bimodal (two segments; objects and background), and that the histogram can be approximated well using a sum of two Gaussians. In this case, this means we model the normalized histogram h as

$$\begin{aligned} h(v) &\approx P_0 g_0(v) + P_1 g_1(v) \\ &= (1 - P_1) g_0(v) + P_1 g_1(v), \end{aligned}$$

where g_0 and g_1 are Gaussian functions with unknown mean and variance, and P_0 and P_1 are the global probabilities –also unknown– that a pixel belongs to segment 0 or 1 respectively. Note that $P_0 + P_1 = 1$. This leaves us with five unknowns –two means, two variances, and P_1– to be estimated from the image histogram. Figure 10.7 shows an example bimodal histogram and its approximation using two Gaussians. The estimation of the unknowns is usually done using a non-linear curve fitting technique which minimizes the sum of squared distances (as a function of the unknowns) between the histogram data and the fitted curve, *i.e.*,

$$\text{minimize} \sum_v (h(v) - m(v))^2,$$

where $m(v)$ is the chosen model for the histogram, which has the unknowns for parameters (here $m(v) = P_0 g_0(v) + P_1 g_1(v)$). The estimation of the (here) five unknowns can therefore be formulated as the minimization of a function of five variables. How this minimization is achieved is beyond the scope of this book, but techniques for this are fairly standard and can easily be found in mathematics literature.

Assuming we can estimate all of the unknowns, the optimal threshold t is the grey value where the two Gaussian curves intersect:

$$(1 - P_1) g_0(t) = P_1 g_1(t).$$

Intermezzo*

This last equation can be deduced as follows:

The normalized histogram is approximated by

$$h(v) \approx (1 - P_1) g_0(v) + P_1 g_1(v),$$

with $g_0(v)$ the distribution of background pixels, and $g_1(v)$ the distribution of object pixels. P_1 and $(1 - P_1)$ are the fractions of object and background pixels respectively. If we threshold the image at a certain threshold t, then the fraction of background misclassified as object pixels is

$$(1 - P_1) \int_t^\infty g_0(v) dv.$$

The fraction of object pixels misclassified as background pixels is

$$P_1 \int_{-\infty}^t g_1(v) dv.$$

The total error fraction $E(t)$ is the sum of these partial errors:

$$E(t) = (1 - P_1) \int_t^\infty g_0(v)dv + P_1 \int_{-\infty}^t g_1(v)dv.$$

The error $E(t)$ is minimal when the derivative is zero:

$$\begin{aligned} &E'(t) = 0 \\ &(1 - P_1)(-)g_0(t) + P_1 g_1(t) = 0 \\ &(1 - P_1)g_0(t) = P_1 g_1(t). \end{aligned}$$

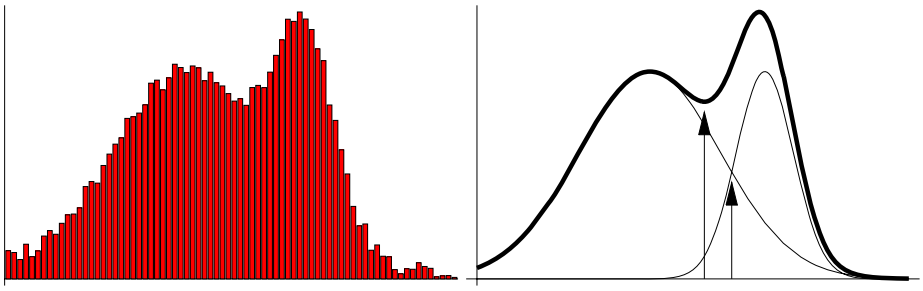


Figure 10.7 Example of optimal thresholding. Left: image histogram. Right: approximation of the histogram (thick curve) using two Gaussians (thin curves). The right arrow shows the optimal threshold. For comparison, the left arrow shows the threshold value if we had used the minimum-between-peaks criterion.

Optimal thresholding may be extended to use multiple thresholds, *i.e.*, the normalized histogram is approximated using a sum of more than two Gaussians:

$$h(v) \approx \sum_{i=0}^{k-1} P_i g_i(v),$$

where g_i is a Gaussian, P_i the global probability of a pixel belonging to segment i, and k the number of Gaussians used to model the histogram. Since the addition of each Gaussian to the model adds three parameters to be estimated, using this model becomes increasingly fickle for large k. Moreover, selecting an appropriate value for k can be a difficult problem by itself. The model can also be extended using other distributions than the Gaussian. The choice of the distribution function should be based on knowledge of the image acquisition process, from which the correct choice can often be deduced.

Histogram estimation. The optimal thresholding method described above uses an estimation of the histogram –actually, a smooth curve that best fits the measured image histogram– rather than the histogram itself. Some of the other threshold selection methods described previously also benefit from using a smooth estimate instead of the raw histogram data. For example, the methods that require finding the extrema of the histogram curve may be better off using a smooth estimate, since this way the chance of getting stuck in a local optimum is decreased, and the process is less susceptible to noise in the raw histogram data.

Good estimation of the histogram often does not require that all of the available pixel information is used. Using a small fraction of the grey values is usually sufficient for a good estimate. For example, instead of using all of the pixels (and their grey values) in the estimation process, we could randomly select 5% of the image pixels and use only those in the process. In practice –unless the noise is exceedingly large or the number of pixels in the image very small– adding the remaining 95% will not significantly improve the estimation.

The approach to estimation used in optimal thresholding is an example of a *parametric* approach. In this type of approach, the histogram is modeled by a function with a fixed number of parameters, and the function is fitted to the histogram by twiddling with the parameters. A *non-parametric* approach to histogram estimation is any approach that does not require the computation or approximation of parameters. The most frequently used method in this category is called *Parzen windowing*.

Parzen windowing estimates the histogram by a sum of simple distribution functions. In contrast to parametric techniques, the number of distribution functions is large rather than small. In fact, one function may be used for each pixel that is employed in the estimation process. Each distribution function used is identical to all others, and any parameters of the distribution function are fixed to identical values beforehand. For example, the estimate may be a sum of Gaussians, all with a mean of 0 and a variance of 1. In general the Parzen windowing estimate $p(v)$ of the normalized histogram can be written as

$$p(v) = \frac{1}{N} \sum_i d(v - v_i),$$

where N is the number of pixels from the original image used in the estimation, v_i is the grey value of a pixel, $i \in \{1, 2, \ldots, N\}$, and d is a distribution function (kernel). The N pixels are usually sampled randomly or equidistantly from the original image. The kernel d is usually a Gaussian or a block or triangular function centered around zero with unit area, but other functions may also be used. Figure 10.8 shows an example of a Parzen histogram estimate using a Gaussian function for the kernel d.

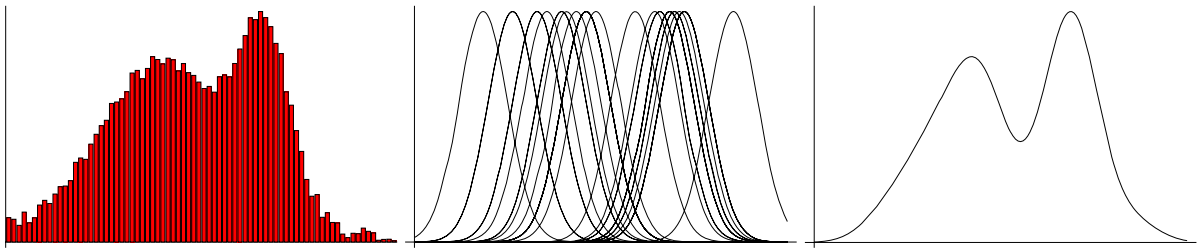


Figure 10.8 Example of Parzen windowing to estimate the histogram. In this example, the Gaussian function with zero mean and unit variance was chosen for the distribution kernel. A small fraction of the pixels was sampled randomly from the image, and a Gaussian placed a the corresponding grey value in the histogram. Left: original image histogram. Middle: Gaussian functions of the sampled pixels. Right: normalized sum of the Gaussians in the middle figure; the Parzen windowing estimate of the histogram.

In practice, non-parametric approaches such as Parzen windowing are often more flexible than parametric ones, as well as less computationally expensive. The better flexibility stems from the fact that no strong assumptions are made on the shape of the histogram, while parametric approaches effectively straitjacket the shape by assuming the histogram to be shaped like –for instance– the sum of two Gaussians.

10.1.2 Enhancing threshold segmentation

The segmentation of an image by thresholding is totally insensitive to the *spatial context* of a pixel. There are many cases conceivable where a human observer may decide on the basis of spatial context that a pixel is not part of a segment, even though it satisfies a threshold criterion. For example, noise in an image may cause many very small segments[5] when thresholding, while in many applications it is known that such small segments cannot physically exist (see figure 10.9). Or: segment boundaries may appear bumpy while we know them to be smooth. Or: thresholding shows two objects connected by a thin 'bridge', while we know them to be separate objects.

Many techniques exist that address such problems. These techniques can be divided into three categories: processing of the original image prior to segmentation, processing of the segmented result, or adaptation of the segmentation process. Often, a combination of these techniques is used in a single application. In this section we will give some examples of common approaches.

The next two examples show the use of non-linear filters such as a median filter and morphological operators for removing (often noise related) small scale artifacts.

[5] Or, depending on the implementation, a single segment including many small disjunct fragments.

Figure 10.9 Example of noise causing small erroneous segments and object holes. Left: original noisy image. Middle: after thresholding. Right: the histogram of the original image. Since the two curves corresponding to object and background grey values overlap for a small part, we can be certain there exists no ideal threshold that avoids the small errors.

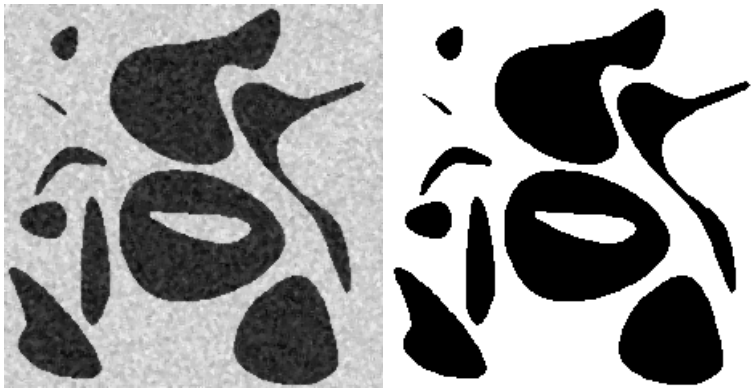
Example: removing small segments and object holes

Figure 10.9 shows that noise can cause small erroneous segments and object holes after thresholding. There are many possible approaches to remove these small artifacts. For example, we could use a closing[6] to remove the holes, and an opening to remove the small segments. We may also search for isolated pixels (or segments below a certain area) and set their grey value to the value of their neighborhood. Another approach is the application of a median filter to the segmented result, which also effectively removes all small artifacts. Below is the result of applying a 3×3 median filter to the thresholding result in figure 10.9. The original image has dimensions 256×256.

The median filter may also be applied *prior* to the thresholding. This effectively removes the outlier grey values that caused the original segmentation artifacts.

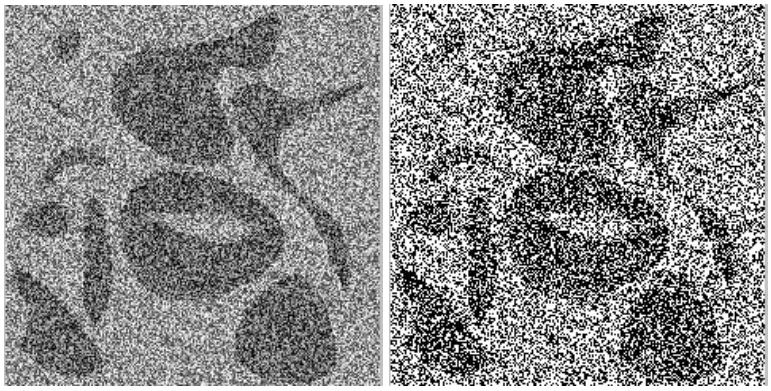
[6] Assuming the binary-image convention here: objects (black) have grey value 1 and the background (white) has grey value 0.

Below are the original image after application of a 3×3 median filter and thresholding:

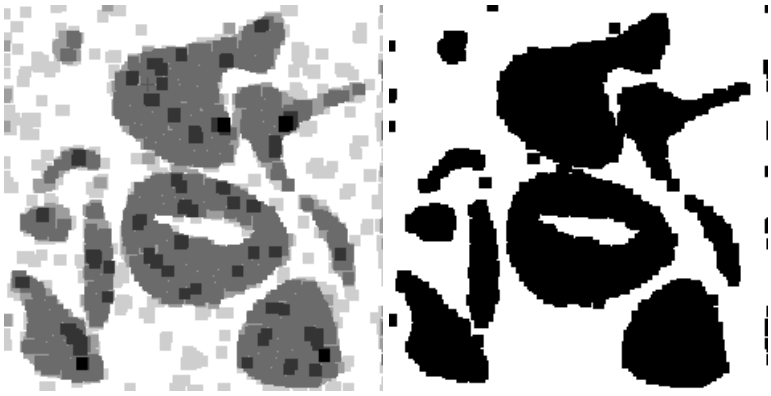


Example: pre- and postprocessing using morphology

Consider these images:

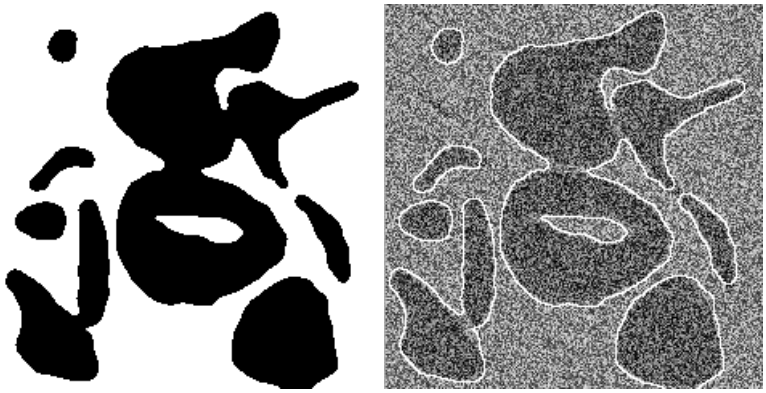


The left image is a noisy original (256×256) image, the right image a binary image after thresholding the original. Needless to say, direct thresholding performs poorly on this type of original. The influence of noise can be reduced, however, by making the object in the original more coherent using a morphological operation such as closing. The left image below shows the original image after closing using a 7×7 square, the right image the result after thresholding this image:



Although the result is much improved compared to the original thresholding, some small erroneous segments are still present. Post-processing of the result can remove these. The left image below shows the result of closing using a 9×9 square, and the right image shows a result where all segments below a certain area have been removed:

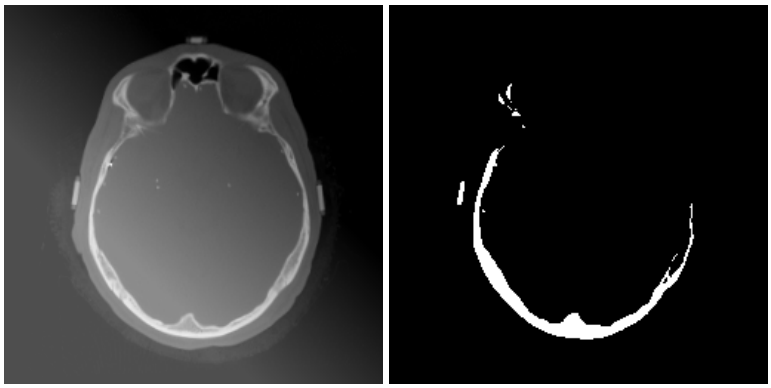
The 'blockiness' of the last result is an artifact of using square structuring elements. If a smoother result is preferred, a slight diffusion and thresholding of the last image gives the left image below. For verification of the final result, the right image below shows the original image again with overlaid contours obtained from the left image.



Large scale grey value artifacts, such as the occurrence of an illumination gradient across an image, can be handled in various ways. In chapter 6, we have already shown that the morphological white top hat transform (see figure 6.18) can be effective to remove such a gradient. Fourier techniques are also often effective, because the gradient image is a distinct low-frequency feature. A popular technique is the subtraction of a low-pass filtered image from an original image. Below are two examples showing yet other approaches. The first is an adaptation of the thresholding algorithm that bases a threshold on the *local* image grey values, *i.e.*, based only on the neighborhood of a pixel rather than all of the grey values in an image. The second is a pre-processing technique that estimates and removes the large grey value artifact from the image.

Example: local thresholding

All of the thresholding techniques presented before applied a *global* threshold to an image, *i.e.*, the threshold was the same for the entire image. This can not give good segmentation result for images where there is an 'illumination' gradient such as this one (shown with a global thresholding result on the right):



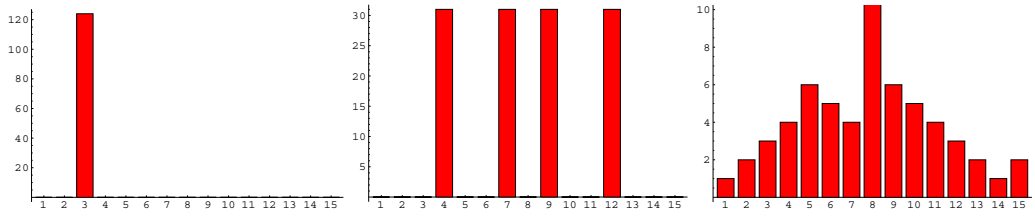
In this image, a bone pixel on the lower-left of the image has a grey value of approximately 230, while a bone pixel on the top-right has a grey value of only 100. This large gradient of grey values causes the global histogram to be diffused, and no good global threshold to extract bone may be found.

If we tile the image into, say, 50 subimages, then the grey value of bone pixels in each subimage will be relatively constant, because the gradient effect is comparatively small in each subimage. For each subimage, we can now find an appropriate threshold to extract the bone pixels. Merging the segmentation results of each subimage into a complete image again will give us a good segmentation of the whole image. Thresholding with a threshold that is not global, but based on image structure and the histogram of subimages is called local thresholding.

Local thresholding can be used effectively when the gradient effect is small with respect to the chosen subimage size. If the gradient is too large, similar errors as in the global example will now occur within a subimage, and the segments found within subimages will no longer match up adequately at the subimage seams. This is a telltale that the subimage size should be reduced. However, we cannot make the subimage too small, because then a reliable histogram can no longer be made.

Example: using histogram entropy

Consider these three histograms of images A, B, and C respectively, each containing 124 entries:



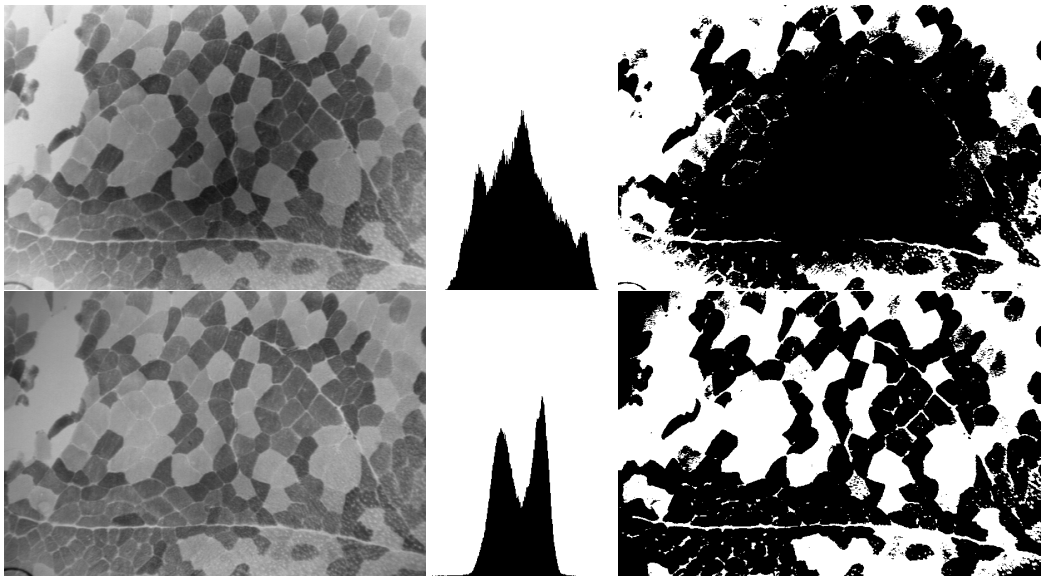
If we randomly pick a pixel from image A, we can be certain beforehand that its grey value will be three. If we randomly pick a pixel from B, we are sure it's 4,7,9, or 12, but each is equally likely. If we pick a pixel from C, we are even less certain; it may be anything from 1 to 15. Eight seems most likely, but even then we have about a 90% probability of being wrong.

These three histograms are examples of histograms where the *uncertainty* of grey values increases from A to C. Another word for this uncertainty is the *entropy* of a histogram. An often used numerical measure for entropy is the *Shannon* entropy H:

$$H = -\sum_{h(v)\neq 0} h(v) \log h(v),$$

where $h(v)$ is the normalized histogram of an image. Images with a histogram that has only few crisp peaks usually have a low entropy (low uncertainty), and images with a diffuse, flat histogram have a large entropy.

When an image is distorted by a large scale gradient artifact, its histogram will diffuse, and its entropy will rise. To correct the artifact, we subtract possible artifact images from the corrupted image until we find a result that has minimal entropy. We will not detail the search process of finding the correct artifact image here, but an example of a light microscope image where the gradient image is removed by minimizing the entropy is given below: the top row shows a corrupted image (with gradient artifact), its histogram and the result of thresholding. The bottom row shows the same images after correction using histogram entropy minimization.



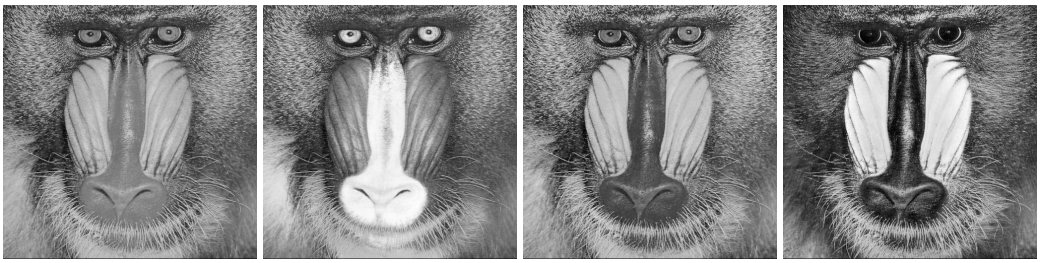
10.1.3 Multispectral thresholding

Many types of images have more than one value attached to each pixel. For example, a color image will usually have three values (*e.g.*, RGB values) attached to each pixel. The separate value images, in this case the R, G, and B images, are often called *channels*. For another example, we may have n photographs of the same scene, so we can attach n grey values to each pixel location. Yet another example is an MR image, which may have several physical measurement values attached to each pixel.[7] We may also attach computed values to a pixel location, for instance, next to the grey value we attach the result of locally applying a Sobel operator (or any other image processing routine) to a pixel location. Such multi-valued images allow an extension of thresholding called *multispectral* thresholding to be applied. The most basic form of multispectral thresholding entails a separate thresholding applied to each channel, and the results are combined using a boolean combination.

Example

A color image has three channels associated with it. Below is a grey valued representation of a color image of a baboon, and the red, green and blue channel of the same color image:

[7] The most common measurements are known as T_1, T_2 and *PD* values.



Suppose we wish to find the image parts that are yellow. We know that yellow is a composite of red and green, so we can find the yellow pixels by thresholding the red and green channels: we take all pixels where both the red and green values are above a threshold. Unfortunately, white or bright grey pixels also have a high red and green value, as well as a high blue value. To find only yellow pixels, we demand the blue value to be *below* a threshold. Below on the left is the result after thresholding and combining (by a logical 'and') the red and green channel, and on the right the result after excluding the pixels with a high blue value: the yellow pixels.

By thresholding each channel separately and combining the results using boolean operations, we effectively find segments in rectangular regions of the 'channel space', as shown in figure 10.10. Channel space is the n-dimensional space (where n is the number of channels), where each image pixel is plotted using its channel values for coordinates. For example, a pure red pixel has coordinates $(1, 0, 0)$ in the RGB channel space. The figure also gives an example of how other regions than rectangles can be segmented by thresholding some function of all channel values.

10.2 Edge based segmentation

Since a (binary) object is fully represented by its edges, the segmentation of an image into separate objects can be achieved by finding the edges of those objects. A typical approach to segmentation using edges is (1) compute an edge image, containing all

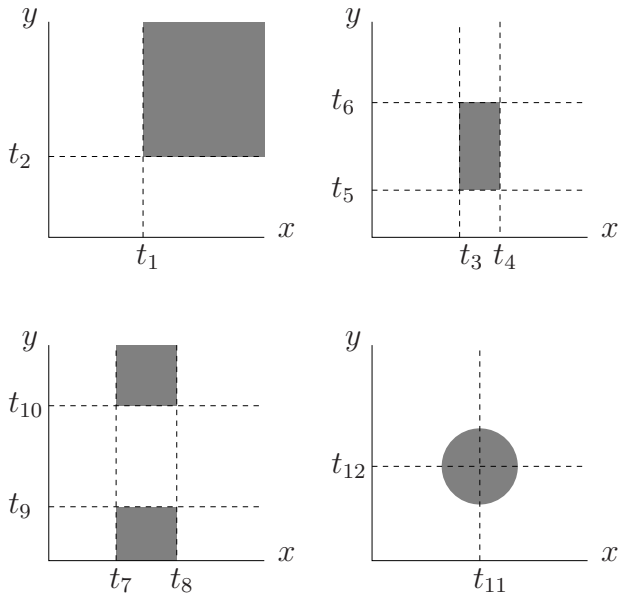


Figure 10.10 Examples of multispectral thresholding, using two channels X and Y. Top left: $(x \geqslant t_1$ AND $y \geqslant t_2)$ Top Right: $(t_3 \leqslant x < t_4$ AND $t_5 \leqslant y < t_6)$ Bottom left: $(t_7 \leqslant x < t_8$ AND $(y < t_9$ OR $y \geqslant t_{10}))$ Bottom right: $((x - t_{11})^2 + (y - t_{12})^2 < r^2)$, where r is the circle radius.

(plausible) edges of an original image, (2) process the edge image so that only closed object boundaries remain, and (3) transform the result to an ordinary segmented image by filling in the object boundaries. The edge detection step (1) has been discussed in chapter 9. The third step, the filling of boundaries, is not a difficult step, and an example of how this can be achieved is given for the technique below. The difficulty often lies in the middle step: transforming an edge (or edgeness) image to closed boundaries often requires the removal of edges that are caused by noise or other artifacts, the bridging of gaps at locations where no edge was detected (but there should logically be one) and intelligent decisions to connect those edge parts that make up a single object. The section on edge linking below addresses some of these difficulties. The next sections on watershed segmentation and active contours deal with methods that avoid having to link edge parts by manipulating –like a rubber band– contours that are always closed[8] until they best fit an edge image.

In those rare cases where an edge image already shows perfect closed object boundaries, edge-based segmentation can be achieved by the following algorithm:

Algorithm: edge-based segmentation
Given an image f,

[8] Or, at least, already linked.

1. Compute an edgeness image ∇f from f. Any preferred gradient operator can be used for this.
2. Threshold ∇f to an image $(\nabla f)_t$, so we have a binary image showing edge pixels.
3. Compute a Laplacian image Δf from f. Any preferred discrete or continuous Laplacian operator may be used.
4. Compute the image $g = (\nabla f)_t \cdot \mathrm{sgn}(\Delta f)$.

The sgn operator returns the sign of its argument. The result image g will therefore contain only three values: 0 at non-edge pixels of f, 1 at edge pixels on the bright side of an edge, and -1 at edge pixels on the dark side of an edge.

The image g contains the boundaries of the objects to be segmented. The Laplacian is used to facilitate the final steps of the algorithm: turning the boundary image into a segmented image h containing solid objects. If we traverse the image g from left to right, two adjacent pixels with values -1 and 1 means we move into an object, and the values 1 and -1 means we move out of one. The image h can therefore be created by setting all pixel values to zero, except for those pixels that are between the transitions $1 \to -1$ and $-1 \to 1$ in each line of g, which are set to 1. If unique values are desired for each separate segment, a labelling algorithm can be run on h.

Figure 10.11 shows an example of applying this algorithm.

10.2.1 Edge linking

Edge detection very seldom gives you the perfect unambiguous and closed boundaries you need for a direct segmentation. There will frequently be spurious edges detected where they shouldn't be, and gaps occur where there should be edges. The latter problem of partial edges needs some form of *edge linking* to tie the available partial edges into an object boundary.

The Hough transform. If we have some idea of what the edges should look like (*e.g.*, lines, circles, *etc.*) we can use the Hough transform (see chapter 8) to find the ideal edges that best fit the partial edges. This approach works best when the objects in the image have fairly simple (*i.e.*, few parameters) parametric shapes. Finding complex shapes with the Hough transform is often a time-consuming task. Another disadvantage is that the Hough transform is inflexible in the sense that unexpected variations in object shapes cannot usually be coped with. When lines are used, some post-processing of the result is necessary to connect the correct line segments, as figure 10.12 shows.

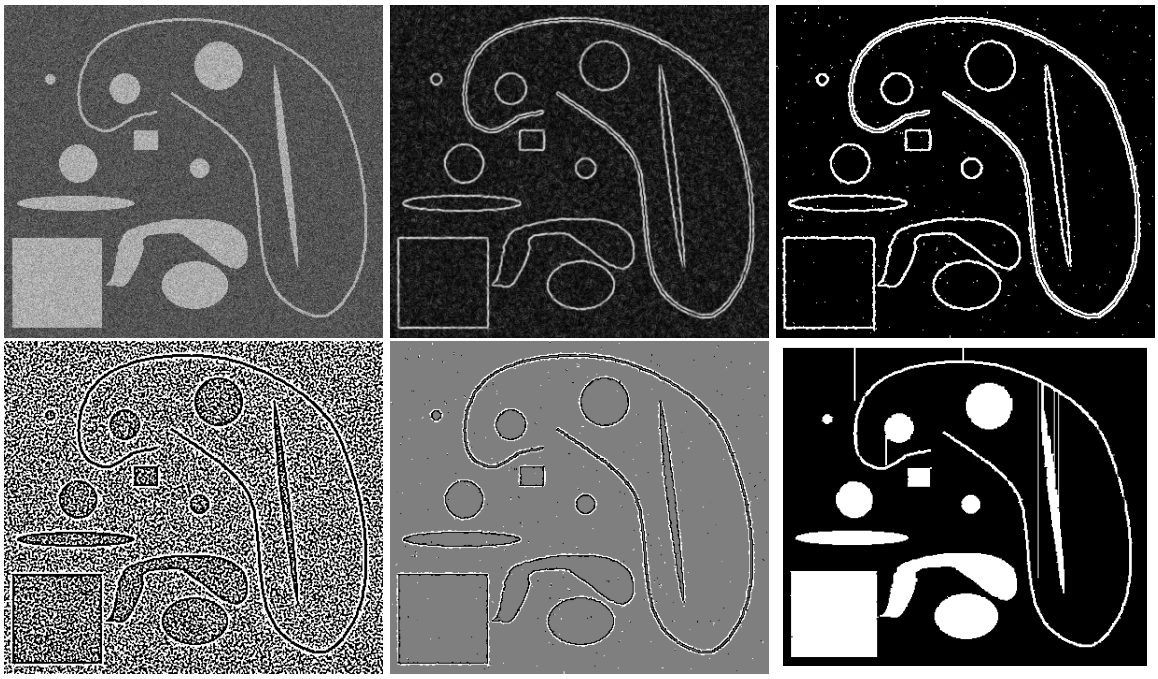


Figure 10.11 Example of edge-based segmentation. Top left: original 400×350 artificial image with added noise. Top middle: edgeness image; computed using a scale space operator (f_w) with $\sigma = 1$ pixel. Top right: same image after thresholding. Bottom left: sign of Laplacian image. Laplacian image computed using a scale space operator (f_{ii}) with $\sigma = 1$ pixel. Bottom middle: product of Laplacian sign and thresholded edge image. Bottom right: result after filling in of the boundaries as in the algorithm above. The noise was added to the original image to show some of the artifacts that they cause: notches in the edges of segmented objects, and lines where the filling algorithm encounters erroneous sign transitions. If no or less noise is added to the original, the segmentation result is perfect.

Neighborhood search. A local approach to edge linking is searching in a small neighborhood around an edge pixel for possible candidates to link the edge pixel to. A typical procedure is to select a starting edge pixel, find the best suitable link candidate in a small neighborhood, link this pixel to the first, then start a new search around the newly linked pixel, and so on. The process is stopped when one of the following occurs: (1) no suitable linking candidate is found, (2) the image boundary is reached, or (3) the pixel to be linked is already part of the current linked boundary.

Many criteria can be used to select a pixel B that is suitable to be linked to the current pixel A. For example:

- B must have a high edgeness value, and / or the edgeness value should be close to the edgeness value in A.

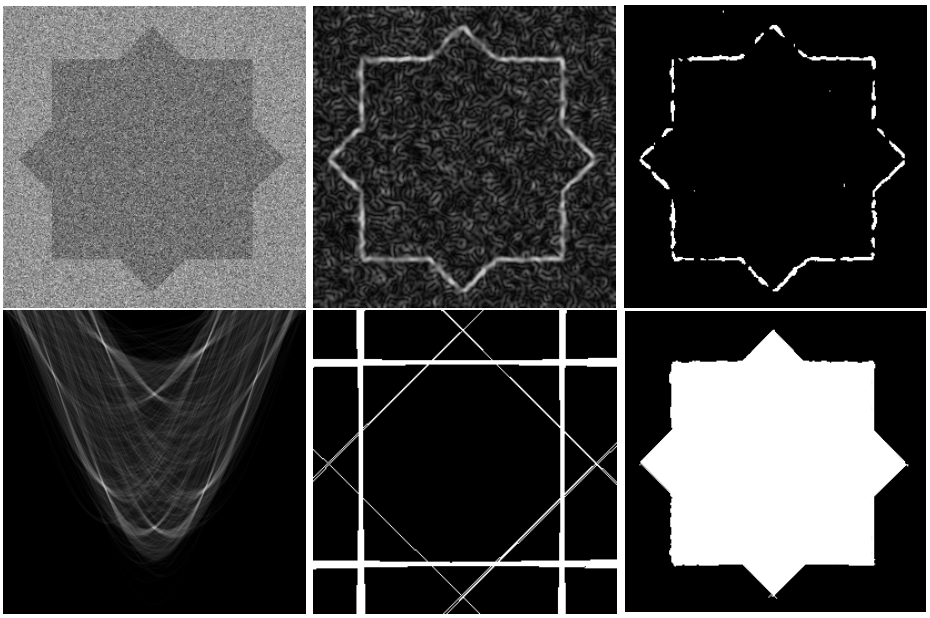


Figure 10.12 Example of edge linking using the Hough transform. Top left: original noisy 256×256 image. Top middle: edge image, computed using the f_w operator at a scale of $\sigma = 2$ pixels. Top right: threshold of this image, showing gaps in the object boundary. Bottom left: the Hough transform (lines) of the thresholded edge image, showing eight maxima. Bottom middle: the corresponding line image of the maxima in the Hough transform. Note that intelligent processing of this image is necessary to obtain only those line segments that make up the original object. (In this case we dilated the binary edge image, took the logical 'and' with the Hough lines image, and performed an opening by reconstruction of erosion.) Bottom right: result after processing and filling; we now have a closed boundary.

- The direction of the gradient in B should not differ too much from the gradient direction in A, and / or the direction from A to B should not differ too much from the direction of previous links.

The latter criterion ensures that edges cannot bend too much, which ensures that the search is kept on the right path when we reach a point where edges cross. It has also proven to be a necessary heuristic for noisy images. A disadvantage, however, is that edges at places where objects really have sharp angles are not linked. In these cases, a post-processing of all found linked boundary segments is necessary, where the boundary segments themselves are linked (using proximity and other criteria) to form closed boundaries. In addition, some form of post-processing is usually necessary to remove those linked edge segments that are unlikely to belong to realistic boundaries. The removal criteria are often application dependent, but commonly edge segments below a certain length are removed.

A problem by itself is the proper choice of starting points –the *seeds*– for the search pro-

cesses. These starting points may be indicated by a user or determined in an application-dependent way. It is also possible to use the (say) hundred points that have the highest edgeness value for starting points. This works well when the contrast of objects and background is about the same for all objects. If this is not the case, it may be that no starting points are picked at low-contrast objects. An approach that remedies this problem is to pick a hundred starting points with highest edgeness values, do edge linking, and then pick a new set of starting points (with highest edgeness), *disregarding all pixels tracked in the first run* (and possibly their neighborhoods too).

Many enhancements to the local search process have been proposed, and we give two of them here that address some common problems. A problem that often occurs is that tracked edge segments are more convoluted than they should realistically be, because of noise or the occurrence of thick edges. To ensure the edge tracking remains central to a thick edge, an additional criterion to use in the edge tracking is the demand that the *ridgeness of edgeness* is high. This is illustrated in figure 10.13.

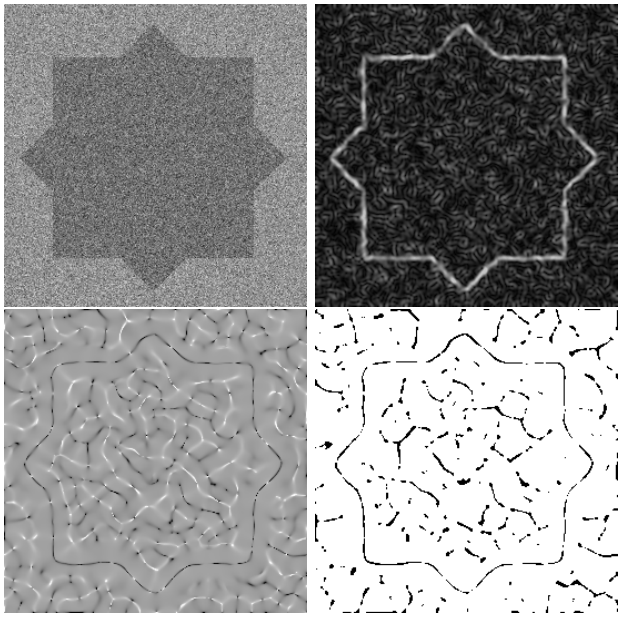


Figure 10.13 Noise and thick edges may cause edge tracking to meander. The use of a ridgeness in addition to (or instead of) the edgeness image in the tracking process may be beneficial, since the ridge is by definition a narrow structure. Top left: Original 256×256 noisy image. Top right: edgeness image (f_w, $\sigma = 2$ pixels). Note the thickness of the edges. Bottom left: ridgeness image *of the edgeness image* (f_{vv}/f_w, $\sigma = 4$ pixels). Bottom right: thresholded ridgeness image. Note that the ridges are much thinner than the edges.

Another problem is that noise may locally disrupt an edge, causing the edge tracking to locally follow a spurious edge branch –that usually ends quickly– instead of following the main edge. This can be addressed by the following adaptation of the search process:

instead of linking an edge pixel to the best linking candidate and then moving on to this candidate, we make a small list of suitable linking candidates. If linking of the best candidate results in a short dead end, we backtrack and see if the next best candidate results in a longer branch.

Network analysis: mathematical programming. If we not only have a good starting point A for a boundary segment, but also a likely end point B (or a set of potential end points), then the edge linking process is effectively the same as finding the best path from A to B. To find this best path, we first need to construct some network from which all possible paths from A to B can be gleaned. Figure 10.14 shows an example of how such a network can be constructed from a gradient image: each edge pixel is transformed to a network node –only pixels with an edgeness value above a certain threshold are part of the network– and arrows are drawn to all neighboring nodes (pixels) that can reached from a certain node. An arrow is drawn if certain criteria are met, *e.g.*, the gradient direction at the two nodes does not differ too much.

Our goal now is to find the best path, *i.e.*, the best sequence of arrows, from A to B. 'Best' can be defined in several ways here, depending on the application. The best path may, *e.g.*, be the shortest path, the path with highest average edgeness value, the path with least curvature, or a combination of these and yet other criteria. These criteria are included in our network by assigning costs to each arrow. For instance, if we are interested in the shortest path, we assign distances to each arrow: 1 to each horizontal or vertical arrow, $\sqrt{2}$ to each diagonal arrow. If we are interested in a path that is both short and runs along nodes with high edgeness values, we may assign an arrow cost that is a weighed sum of a distance term and a term inversely proportional to edgeness. The cost of each *path* from A to B is defined as the sum of costs of the arrows that make the path. The best path is the path with lowest cost.

The problem of finding the best path has been –and still is– an area of considerable research. It is known as the *shortest-route problem* in the field of operations research, and many algorithms for (approximately) solving the problem can be found. The problem solving strategy is known as mathematical programming.[9] Other areas of research may present the problems under different headings, such as graph searching or network analysis. We present here a simple but effective algorithm for finding the best path.

Algorithm: Solving the shortest-route problem

Problem: find the shortest route from A to B given a network of nodes and branches, with given distances (costs) for each branch.

The idea of this iterative algorithm is to find the n-th nearest node to A, where n is the iteration number. For example, if $n = 1$, we find the nearest node to A. If $n = 2$, we find the second closest node to A, *etc.* The algorithm is terminated if the n-th closest node is the desired end node B.

[9] Mathematical programming also contains the techniques indicated as *dynamic programming* in other books in reference to the problem at hand.

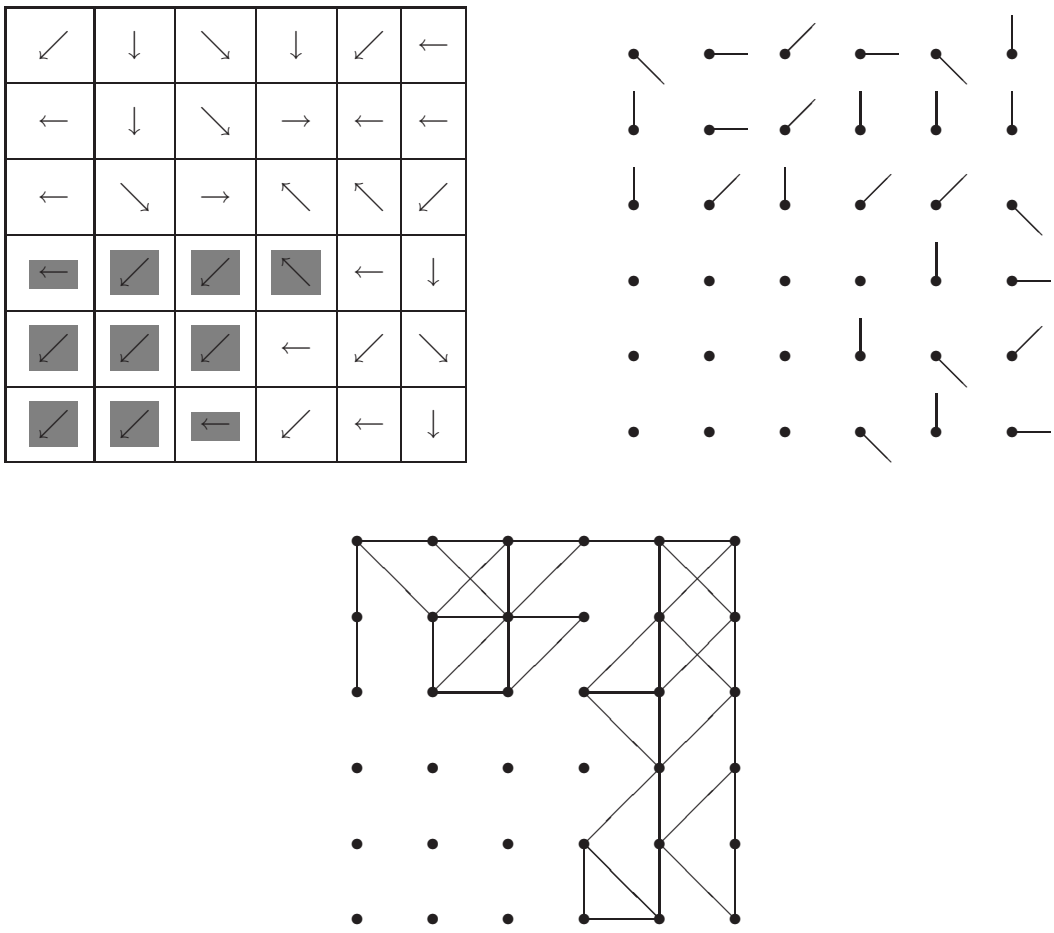
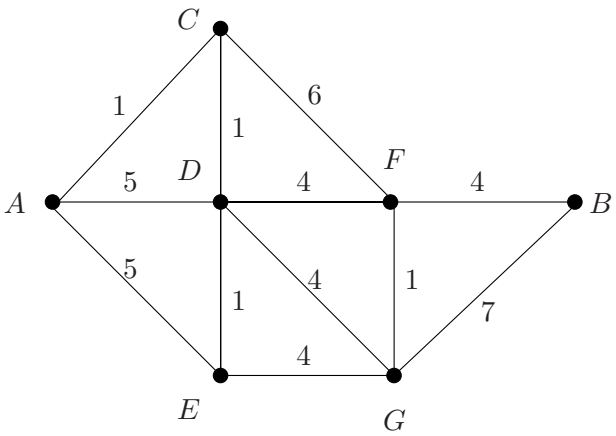


Figure 10.14 Example of creating a network from a gradient image. Top left: image with arrows indicating the gradient direction. (Actually, the nearest direction in the set of $\{0, 45, 90, 135, 180, 225, 270, 315\}$ degrees.) Pixels where the gradient (edgeness) value falls below a threshold are marked in grey. Top right: network created from the gradient image. Each pixel has been turned into a node. Lines show the significant gradients, turned 90 degrees so they are oriented along the edge at a node. Only the directions north, northeast, east, and southeast are used; they are unique modulo 180 degrees. Bottom: final network. For each node with an arrow in the top right network, we examine the nodes in the approximately along the edge direction of a node (not more than 45 degrees deviation). If the gradient direction of the new node differs by 45 degrees (modulo 180) or less from the current direction, a connection is drawn.

The input for iteration n is all $n-1$ nearest nodes found in previous iterations (called solved nodes, all other nodes are called unsolved) that are connected to unsolved nodes. For each solved node, the shortest path to A and its length is known from previous iterations. Each of these input nodes provides a candidate for the n-th nearest node to be found in this iteration. This candidate is the unsolved node connected to the solved node by the shortest branch. The n-th nearest node is then the candidate with the shortest route to A.

Example

Given this network:



we apply the algorithm described above. At the start of the algorithm, the only solved node is A.

First iteration: we find candidate nodes by finding –for each solved node connected to unsolved nodes– the unsolved node connected by the shortest path. We have only one solved node: A. The closest unsolved connected node is C, with path length 1. C is added to the set of solved nodes.

Second iteration: for the solved node C, the candidate node now is D (closest connected unsolved node), with path length 2. For the solved node A, we have two candidates: D and E, because both can be reached by a branch of length 5. From all candidates, we select D as second-nearest node, because path length 2 is the shortest.

Third iteration: solved node A provides candidate E, solved node C provides candidate F, solved node D provides candidate E. Path lengths are 5, 7, and 3 respectively, so E becomes the third-nearest node.

Fourth iteration: solved node A can no longer provide candidates: there are no connected unsolved nodes. Node C provides F, D provides F and G, and E provides G, with path lengths of 7,6,6, and 7 respectively. There is a tie between nodes F and G; both are fourth-nearest node.

Fifth iteration: only F and G of the solved nodes have a connected unsolved node, B in both cases. The path lengths are 10 and 13 respectively. B is fifth-nearest node, and the algorithm terminates.

Backtracking provides us with the optimal path: from B to F to D to C to A; or the right way round: $ACDFB$.

The table below summarizes the course of the algorithm:

n	solved nodes connected to unsolved nodes	closest connected unsolved node	path length	n-th nearest node
1	A	C	1	C
2	A	D	5	
		E	5	
	C	D	$1+1=2$	D
3	A	E	5	
	C	F	$1+6=7$	
	D	E	$2+1=3$	E
4	C	F	$1+6=7$	
	D	F	$2+4=6$	F
		G	$2+4=6$	G
	E	G	$3+4=7$	
5	F	B	$6+4=10$	B
	G	B	$6+7=13$	

10.2.2 Watershed segmentation

In chapter 8 we introduced the watershed transformation as a means to separating overlapping objects. A watershed is formed by 'flooding' an image from its local minima, and forming 'dams' where waterfronts meet. When the image is fully flooded, all dams together form the watershed of an image. The watershed of an edgeness image (or, in fact, the watershed of the original image) can be used for segmentation. Figure 10.15 shows how this works. The idea is that when visualizing the edgeness image as a three-dimensional landscape, the catchment basins of the watershed correspond to objects, *i.e.*, the watershed of the edgeness image will show the object boundaries. As the landscape in the figure shows, the object boundaries mark the catchment basins, but there are small defects because of image artifacts. Because of the way the watershed is constructed –it forms 'dams' where waterfronts meet– these small defects do not disturb the watershed segmentation much.

Typically, some pre- and post-processing of the watershed image is needed to obtain a good segmentation. Usual post-processing is the filling in of the boundaries to obtain

solid segments. In the case of watershed segmentation, we do not need to worry about small 'leaks' in a boundary (a small leak between two segments will fill in the two segments as being a single segment). Because of the way the watershed is constructed, leaks cannot occur.

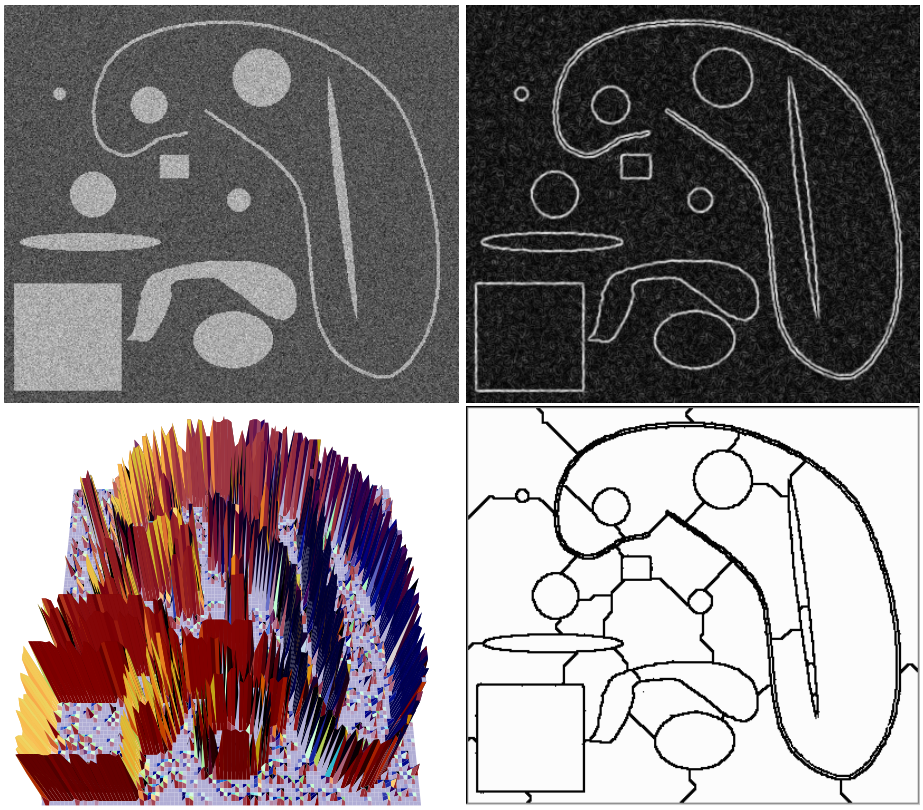


Figure 10.15 Example of edgeness watershed segmentation. Top left: original noisy image. Top right: edgeness image. Bottom left: landscape version of the edgeness image. Note that objects are surrounded by a 'palisade', that will hold the water in the flooding process that forms the watershed. Dams will be formed when the water level reaches the tops of the palisades. An uneven palisade height will not disturb the formation of the correct dam marking an object boundary. Bottom right: all dams formed; the watershed of the edgeness image.

Some pre- and post-processing is usually necessary to avoid an oversegmentation. In the example in our figure, we set all grey values below a threshold to zero. This ensures no dams are formed early in the flooding process. Since such dams are caused by very weak edges, they are unlikely to correspond to object boundaries, and should not be part of the segmentation watershed. Our example still shows some oversegmentation: besides the correct object boundaries, several erroneous branches are present. Many approaches have been proposed to remove these erroneous branches. One effective method is to merge those segments that are likely to belong together. How this likelihood is measured is discussed in the section on region based segmentation. An-

other method that is effective in our example is to compute the average edgeness value of each branch, and remove those branches where this average falls below a threshold. Figure 10.16 shows that this effectively removes all spurious branches from our example.

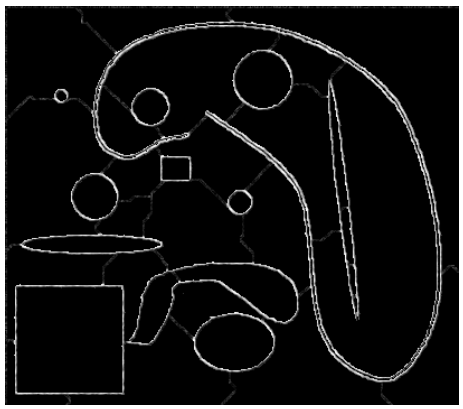


Figure 10.16 Example of removing the spurious branches of the edgeness watershed to reduce oversegmentation of the original image. The image shows a multiplication of the edgeness image and its watershed from the previous figure, clearly showing that the erroneous branches can be easily removed by thresholding the average edgeness of branches.

10.2.3 Snakes

An *active contour* or *snake* is a curve defined in an image that is allowed to change its location and shape until it best satisfies predefined conditions. It can be used to segment an object by letting it settle –much like a constricting snake– around the object boundary.

A snake C is usually modeled as a parametrized curve $C(s) = (x(s), y(s))$, where the parameter s varies from 0 to 1. So, $C(0)$ gives the coordinate pair $(x(0), y(0))$ of the starting point, $C(1)$ gives the end coordinates, and $C(s)$ with $0 < s < 1$ gives all intermediate point coordinates. The movement of the snake is modeled as an energy minimization process, where the total energy E to be minimized consists of three terms:

$$E = \int_0^1 E(C(s))ds = \int_0^1 \Big(E_i(C(s)) + E_e(C(s)) + E_c(C(s))\Big)ds.$$

The term E_i is based on internal forces of the snake; it increases if the snake is stretched or bent. The term E_e is based on external forces; it decreases if the snake moves closer to a part of the image we wish it to move to. For example, if we wish the snake to move to

edges, we may base this energy term on edgeness values. The last term E_c can be used to impose additional constraints, such as penalizing the creation of loops in the snake, penalizing moving too far away from the initial position, or penalizing moving into an undesired image region. For many applications, E_c is not used, *i.e.*, simply set to zero. Common definitions for the internal and external terms are

$$\begin{aligned} E_i &= c_1 \left\| \frac{dC(s)}{ds} \right\|^2 + c_2 \left\| \frac{d^2C(s)}{ds^2} \right\|^2 \\ E_e &= -c_3 \left\| \nabla f \right\|^2, \end{aligned}$$

where the external term is based on the assumption that the snake should be attracted to edges of the original image f. By using other external terms, we can make use of different image features, making the snake follow ridges, find corner points, *etc.* The constants c_1, c_2, and c_3 determine the relative influence of each term on the movement of the snake. The elasticity of the snake is controlled by c_1; the norm of the derivative $\frac{dC(s)}{ds}$ measures how much the snake is stretched locally, so setting c_1 to zero will result in high stretch values having no impact on the energy, hence the snake may stretch infinitely. Alternatively, a high (relative to the other c's) value of c_1 makes that the snake can stretch very little. In the same manner, c_2 controls how much the snake can bend (*i.e.*, its stiffness or rigidity), since the second derivative $\frac{d^2C(s)}{ds^2}$ is a measure for the snake's curvature, and c_3 sets the relative influence of the edge attraction force.

When implementing a snake, it is commonly modeled by a connected series of splines. This model allows the snake –on the one hand– to be easily connected to the mathematical model: the spline representation is easily converted to the parametrized curve $C(s)$, and its derivatives can also be computed in a straightforward manner. On the other hand, the discrete set of control points of the spline give a handle to optimizing the snake; only the control points need to be moved, and the set is easily related to discrete images. The optimization itself is not usually carried out by direct minimization of the energy functional E, but by numerically solving its Euler-Lagrange form:

$$-\frac{d}{ds}\left(c_1 \left\| \frac{dC(s)}{ds} \right\|^2 \right) + \frac{d^2}{ds^2}\left(c_2 \left\| \frac{d^2C(s)}{ds^2} \right\|^2 \right) + \nabla(E_e + E_c) = 0.$$

Figure 10.17 shows an example of a snake used to find the boundary of an object.

For a snake to converge to the desired location and shape, we need a good starting estimate in most practical applications. This can be provided by the user, or by processing and linking of extracted edges as described in the previous section. To improve the chances of the snake converging to the desired shape, many current algorithms have added features to the original snake algorithm, such as:

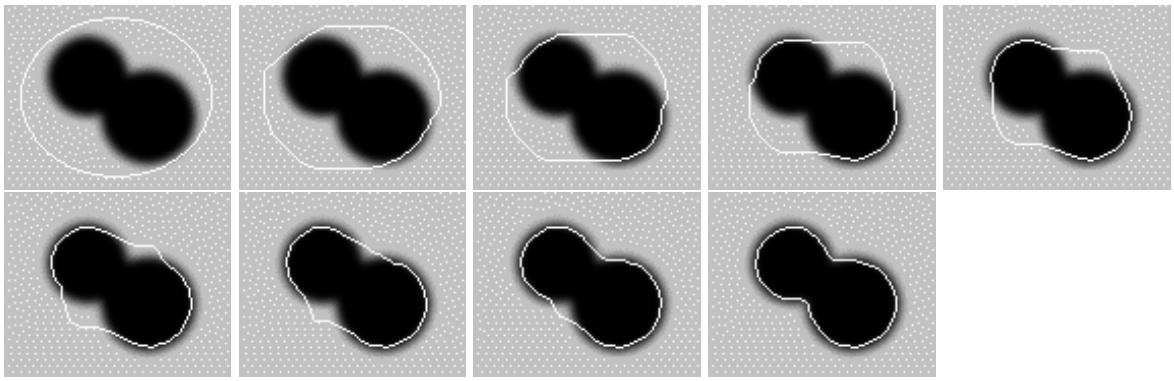


Figure 10.17 Example of a snake. The sequence of images shows the evolution of the snake from a user-defined original position to its final state.

- An inflation force: if the initial snake is too far to the inside of a boundary, the inflation force (like in a balloon) causes it to move closer and avoid getting stuck on local spurious edges. For an external snake, a deflating force can be used.
- The possibility of a snake to split itself into separate snakes.
- A 'growing' ability: if a snake stretches too much locally, extra control points can be inserted there.
- Added region criteria to closed snakes: this entails that besides the usual energy terms, energy terms that measure aspects of the *region* enclosed by a snake are added. For example, we may demand that the region has minimal grey value variance (which ensures that the snake encloses a homogeneous area), or that the entropy or a higher central moment of the histogram of the enclosed region is minimal.

10.2.3.1 Implicit snakes*

We usually regard a discrete image as a collection of grey values arranged on a grid. But if we are especially interested in image contours, it may be helpful to think of the image as a set of contours. The most natural set of contours that describe an image are its isophotes; the iso-height lines if the image is viewed as a landscape. The complete set of isophotes fully describes an image. Figure 10.18 shows some isophotes of an image.

In the classical snake approach described in the previous section, the initial snake is defined explicitly, and is then evolved subject to internal and external forces. An alternate approach to snakes is to evolve a natural image curve –say, an isophote– subject to internal and external forces. This is called an *implicit* snake, since it is implicitly defined by the geometrical image structure. The evolution of an implicit snake is also different from the classical explicit snake; where the classical snake 'crawls' over the image to

Figure 10.18 An image and some of its isophotes. The full collection of isophotes gives a complete description of the image.

its final destination and shape, in the case of the implicit snake, we modify the image itself. For example, suppose our implicit snake is the isophote defined by[10] grey value 100. Then the evolved snake is obtained by processing the image and then extracting isophote 100.

An implicit snake is not necessarily an isophote. It may be any curve defined in terms of the geometry of the image. A common mathematical way of expressing the implicit snake is to define it as the zero-level curve of a function g on the image f: $g(f) = 0$, where g is modeled so it extracts the isophote, flowline, or whatever curve you want.[11] For the sake of simplicity, we will show here only the evolution of implicit snakes defined by isophotes.

Controlling the evolution of the implicit snake is the tricky bit in the process here. We have simply said that evolving the snake is done by processing the image, but what processing is necessary to make the snake evolve the way we want? The next paragraphs deal with this problem.

If we have a curve C parametrized by s, then $\frac{\partial C}{\partial t}$ denotes the evolution (change, movement) of the curve over time t. This movement can be locally decomposed into movement *along* the curve, and movement *perpendicular* (normal) to the curve:

$$\frac{\partial C}{\partial t} = \alpha(s,t)\mathbf{T} + \beta(s,t)\mathbf{N},$$

[10] The isophote at value x can be extracted from a grey-valued image f by thresholding it at value x and then taking the contours of the binary result.

[11] An additional constraint is that $\nabla g(f) \neq 0$.

where $\mathbf{T}$ and $\mathbf{N}$ are the local tangential and normal unit vectors to the curve, and α and β are arbitrary functions. Tangential movement, however, does not alter the curve,[12] so we can effectively write the curve evolution as an evolution in the normal direction only:

$$\frac{\partial C}{\partial t} = \beta'(s,t)\mathbf{N},$$

where β' depends only on α and β. Without proof, we note her that if the curve C is an isophote of the image f, then the curve evolution can be written in terms of *image* evolution by the equivalence:

$$\frac{\partial C}{\partial t} = \beta'(\cdot)\mathbf{N} \quad \Longleftrightarrow \quad \frac{\partial f}{\partial t} = -f_w \beta'(\cdot),$$

where we use the local gradient-based coordinate system (v, w) defined in chapter 9. The arguments of β' on the left side must be 'translated' to arguments in image form. This is not a trivial matter in general. A frequently occurring argument is the isophote curvature κ, which (as we have seen before) can be expressed as $-f_{vv}/f_w$. Table 10.1 lists some frequently used curve evolutions. Note that we have seen the image evolution equations before as the equations driving non-linear diffusion in chapter 9.

Curve evolution	Image evolution
$\frac{\partial C}{\partial t} = c\mathbf{N}$	$\frac{\partial f}{\partial t} = cf_w$
$\frac{\partial C}{\partial t} = \kappa\mathbf{N}$	$\frac{\partial f}{\partial t} = f_{vv}$
$\frac{\partial C}{\partial t} = (\alpha + \beta\kappa)\mathbf{N}$	$\frac{\partial f}{\partial t} = \alpha f_w + \beta f_{vv}$
$\frac{\partial C}{\partial t} = \kappa^{1/3}\mathbf{N}$	$\frac{\partial f}{\partial t} = f_{vv}^{1/3} f_w^{2/3}$

Table 10.1 Some examples of implicit snake curve evolution equations and the corresponding image evolution equations.

A type of implicit snake that can be used well for many segmentation tasks is controlled by the equation $\frac{\partial C}{\partial t} = g\kappa\mathbf{N} - \nabla g$, where g is a function that decreases with edgeness (such as $g = \exp\left(\frac{-f_w^2}{k^2}\right)$), and κ is the isophote curvature. The image evolution form is $\frac{\partial f}{\partial t} = gf_{vv} + \nabla g \cdot \nabla f$. Without going into the mathematical explanation, we note that this snake will be attracted to edges in such a way that evolving an image will result in an image where segmentation can optimally be done using simple thresholding. Figure 10.19 shows an example of this.

[12] Only its parameterization.

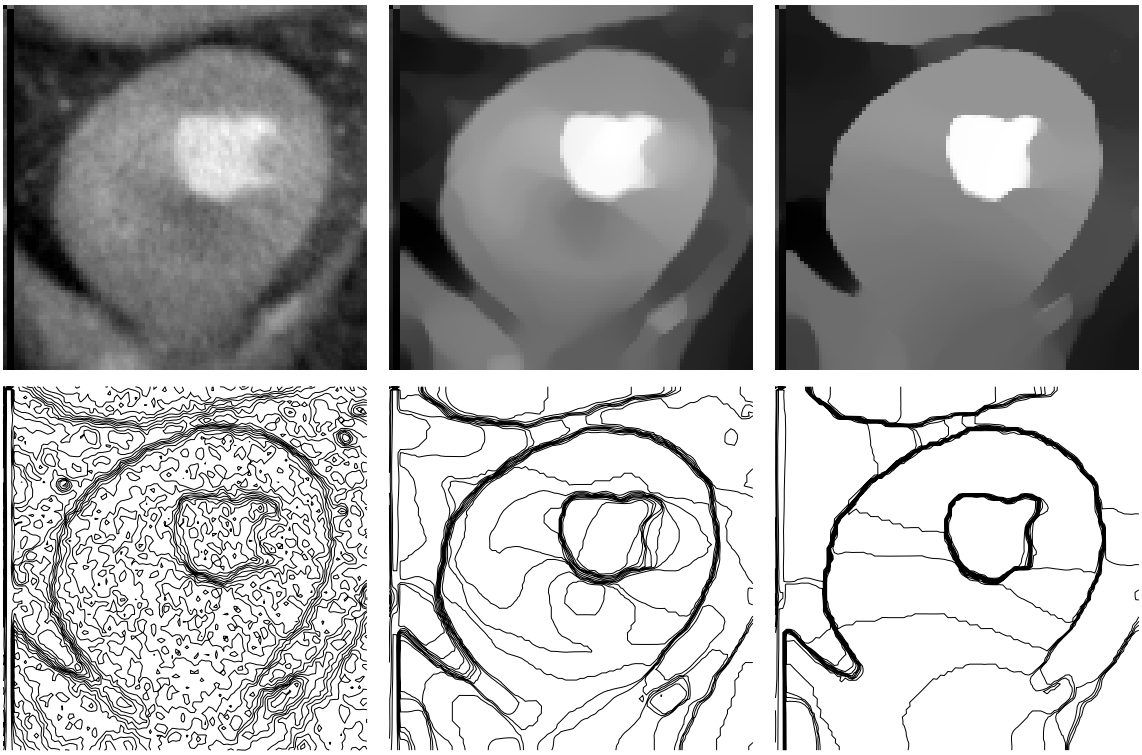


Figure 10.19 Example of implicit snakes. Top left: original image. Top middle: intermediate stage of the implicit snake algorithm mentioned in the text. Right: final stage. Bottom row: corresponding snakes; the isophotes at twenty separate grey levels.

There are several advantages of implicit over explicit snakes. First, there are less topological restrictions; the topology of the snake may even change (*e.g.*, the snake of a level set may start out as two separate contours that merge in the evolution). Second, most evolution schemes allow the simultaneous evolution of more than one snake, possibly *all* level sets (say, all isophotes) of an image. Last, the image evolution schemes can directly be applied to three-dimensional (or higher dimensional) images, while the extension of classical snakes to three dimensions is a far-from-easy task.

10.3 Region based segmentation

In the previous section, objects were found by locating their boundaries. In this section, we discuss the dual approach of finding the object *region* instead of its edges. In theory, finding an object by locating its boundary and finding it by establishing the region it covers will give you exactly the same object; the boundary and the region are just different representations of the same object. In practice, however, taking an edge based ap-

proach to segmentation may give radically different results than taking a region based approach. The reason for this is that we are bound to using imperfect images and imperfect methods, hence the practical result of locating an object boundary may be different from locating its region.

Region based segmentation methods have only two basic operations: *splitting* and *merging*, and many methods even feature only one of these. The basic approach to image segmentation using merging is:

1. obtain an initial (over)segmentation of the image,
2. merge those adjacent segments that are similar –in some respect– to form single segments,
3. go to step 2 until no segments that should be merged remain.

The initial segmentation may simply be all pixels, *i.e.*, each pixel is a segment by itself. The heart of the merging approach is the similarity criterion used to decide whether or not two segments should be merged. This criterion may be based on grey value similarity (such as the difference in average grey value, or the maximum or minimum grey value diference between segments), the edge strength of the boundary between the segments, the texture of the segments, or one of many other possibilities.

The basic form of image segmentation using splitting is:

1. obtain an initial (under)segmentation of the image,
2. split each segment that is inhomogeneous in some respect (*i.e.*, each segment that is unlikely to *really* be a single segment).
3. go to step 2 until all segments are homogeneous.

The initial segmentation may be no segmentation at all, *i.e.*, there is only a single segment, which is the entire image. The criterion for inhomogeneity of a segment may be the variance of its grey values, the variance of its texture, the occurrence of strong internal edges, or various other criteria. The basic merging and splitting methods seem to be the top-down and bottom-up approach to the same method of segmentation, but there is an intrinsic difference: the merging of two segments is straightforward, but the splitting of a segment requires we establish suitable sub-segments the segments can be split into. In essence, we still have the segmentation problem we started with, except it is now defined on a more local level. To avoid this problem, the basic splitting approach is often enhanced to a combined *split and merge* approach, where inhomogeneous segments are split into simple geometric forms (usually into four squares) recursively. This of course creates arbitrary segment boundaries (that may not be correlated to realistic boundaries), and merge steps are included into the process to remove incorrect boundaries.

10.3.1 Merging methods

Region growing. Many merging methods of segmentation use a method called *region growing* to merge adjacent single pixel segments into one segment. Region growing needs a set of starting pixels[13] called *seeds*. The region growing process consists of picking a seed from the set, investigating all 4-connected neighbors of this seed, and merging suitable neighbors to the seed. The seed is then removed from the seed set, and all merged neighbors are added to the seed set. The region growing process continues until the seed set is empty. The algorithm below implements an example with a single seed, where all connected pixels with the same grey value as the seed are merged.

Algorithm: Region growing

The data structure used to keep track of the set of seeds is usually a *stack*. Two operations are defined on a stack: *push*, which puts a pixel (or rather, its coordinates) on the top of the stack, and *pop*, which takes a pixel from the top of the stack.

In the algorithm, the image is called f, the seed has coordinates (x, y) and grey value $g = f(x, y)$. The region growing is done by setting each merged pixel's grey value to a value h (which must not equal g). The pixel under investigation has coordinates (a, b). The algorithm runs:

1. push (x, y)
2. as long as the stack is not empty do
   (a) pop (a, b)
   (b) if $f(a, b) = g$ then
      i. set $f(a, b)$ to h
      ii. push $(a - 1, b)$
      iii. push $(a + 1, b)$
      iv. push $(a, b - 1)$
      v. push $(a, b + 1)$

The final region can be extracted from the image by selecting all pixels with grey value h. To ensure the correctness of the result, we must select h to be a value that is not present in the original image prior to running the algorithm. The statement that decides if a pixel should be merged ('if $f(a, b) = g$') can be modified to use a different merging criterion. A simple modification would be to allow merging of pixels that are in a certain range of grey values ('if $l < f(a, b) < h$'). An example of this is shown in figure 10.20.

[13] Which usually contains only a single pixel.

Another modification is to allow merging only if the gradient value of the candidate pixel is low (a high gradient could signify we are at the edge of an object). This last criterion is often combined with a grey value range criterion to ensure the region growing does not grow right through a weak spot in the boundary where the gradient value is low. More elaborate merging criteria can be used, but they do not usually result in as good a segmentation as may be hoped for, since the heart of region growing is the evaluation of the criterion at a single pixel –the merging candidate– which may be prone to noise. Elaborate criteria are usually more effectively used when evaluating whether to merge two segments of a larger size.

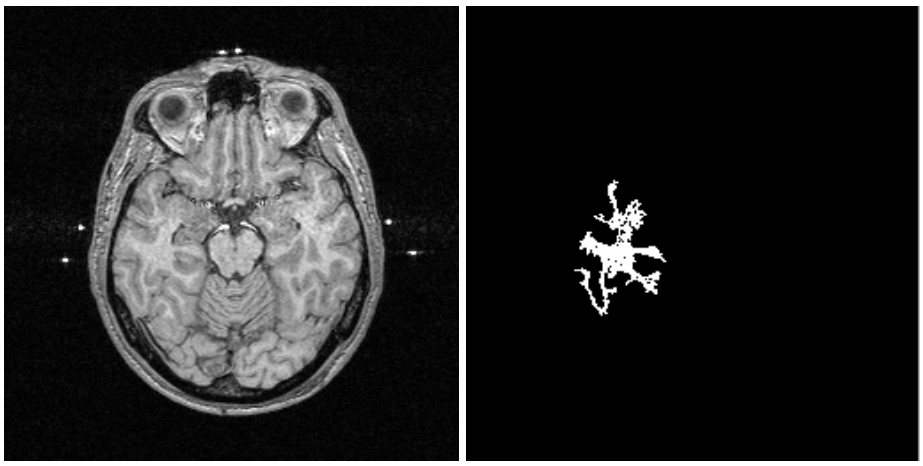


Figure 10.20 Example of region growing based on a grey level range. On the left, an original 256×256 image is shown, with a grey level range of 256. On the right, the result of region growing with the seed roughly at the center of the structure, allowing a grey value range of ± 30 around the grey value of the seed.

Region merging. In this section, we will assume we have the 'larger size' segments as meant above available, but we still have an oversegmentation of the image, so we still need to do region merging to obtain a proper segmentation. Such an oversegmentation can, *e.g.*, be obtained by

- a watershed segmentation such as in figure 10.21, or
- a multiple thresholding as in figure 10.4 –but with many more thresholds– followed by a labeling step,
- using implicit snakes as in figure 10.19 (with segments defined by the contours on the bottom right), or
- a series of region growings –with, say, using the grey value range criterion– with $(0, 0)$ as the first seed, and subsequent seeds picked from the unsegmented image parts until no pixels remain unsegmented, or

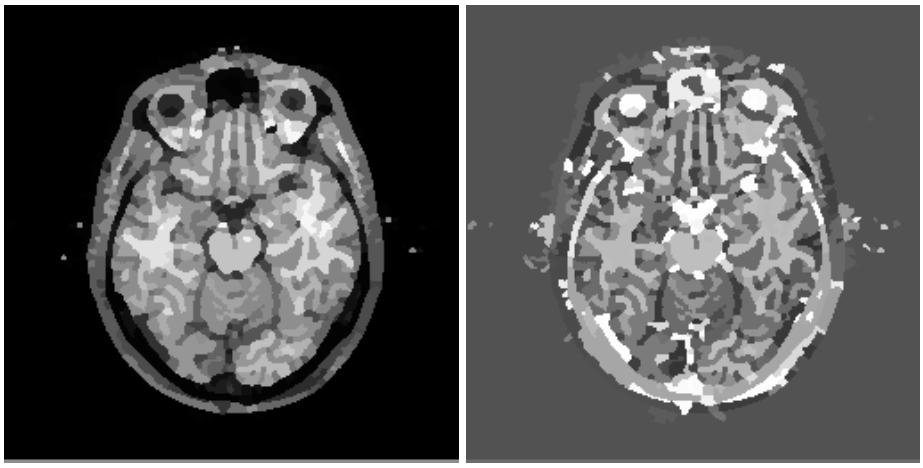


Figure 10.21 Example of an oversegmentation (left) obtained by the watershed of a gradient image of the original. Each segment is visualized here by a grey value equal to the average of the original values of the segment. The right image shows a random grey value remapping of the left image to better display the fact that it is an oversegmentation of the original.

- anything else you can think of.

The oversegmentation can be reduced to a better segmentation by merging adjacent segments. A merging of two adjacent segments can be achieved by removing their common boundary. Two types of criteria exist to judge whether two segments should be merged: those evaluating the edge strength of the common boundary, and those comparing region characteristics of the two segments. Some example heuristics considering edge strength are: remove a common boundary if

- the average edgeness is too low: $\frac{e}{l} < t_1$, where e is the sum of edgeness values of pixels in the common boundary with length l, and t_1 is a threshold,
- the common boundary contains too many weak edge pixels: $\frac{w}{l} > t_2$, with w the number of weak edge pixels (all pixels in the common boundary with edgeness value below a threshold), and t_2 a threshold,
- the common boundary contains too many weak edge pixels compared to the smallest segment size: $\frac{w}{\min\{l_1,l_2\}} > t_3$, where l_1 and l_2 are the respective adjacent segment perimeters, and t_3 is a threshold.

Heuristics considering region characteristics are commonly used combined with edge heuristics, since it is possible that two very similar regions are separated by a strong edge, in which case merging is usually not desired. Example of region heuristics are: remove a common boundary if

- the grey values of the adjacent regions are similar. Various measures can be used to measure this, *e.g.*,
  - the average grey value of the two segments differs by less than a threshold: $|\overline{g}_1 - \overline{g}_2| < t_1$,
  - the maximum (or minimum) grey value difference found between the two segments is below a threshold: $\max_{(x_1,y_1)\in S_1,(x_2,y_2)\in S_2} |g(x_1,y_1) - g(x_2,y_2)| < t_2$,
  - in addition to one of the above measures: the variance of grey values in each segment must be similar: $|\sigma^2(S_1) - \sigma^2(S_2)| < t_3$, or $|\Delta(S_1) - \Delta(S_2)| < t_4$,
- the texture of the regions is similar. Example of texture measures are treated in a separate section later in this chapter.
- the histogram of the adjacent segments is similar. The similarity can be measured by comparing scalar histogram features such as (central) moments or entropy of the histogram. Note that comparing first and second (central) histogram moments boils down to the same average and variance grey value criteria as described above.

In the above, S_1 and S_2 denote the respective segment regions, g is a grey value, $\overline{g_i}$ is the average grey value of segment i, t_j is a threshold, $\sigma^2(S_i)$ denotes the grey value variance, and $\Delta(S_i)$ is the maximum grey value difference occurring in segment i. Combinations of the criteria above are also possible. The removal of common boundaries is usually done recursively: all common boundaries in an image are considered for removal, those that do not meet the set criteria are removed, and the process is repeated until no more common boundaries are removed. Note that the *order* in which the common boundaries are removed may have an impact on the merging process: suppose we have three segments S_1, S_2, and S_3, with the pairs (S_1, S_2) and (S_2, S_3) adjacent. Then merging S_1 and S_2 to a new segment S_{12} may cause that S_{12} and S_3 are unsuitable for merging, while previously S_2 and S_3 *were* to be merged according to the criteria set. In practice, it is often a good idea to start by removing the weakest boundary, re-evaluating the boundary strengths, again remove the weakest, *etc.* .

10.3.2 Splitting and split & merge methods

Where region merging is an agglomerative approach, region splitting is divise. We mentioned before that this difference makes that the two approaches are not opposites, but fundamentally different problems; the merging of two segments is straightforward, but the splitting of a segment requires that suitable sub-segments are established to split the original segment into. The problem of *how* to split a segment is of course itself a segmentation problem, and we can treat it as such: any segmentation method (such as the ones treated in this chapter) can be applied to the segment to establish sub-segments. Besides the hierarchical level, there is no intrinsic difference The problem of how to decide if a

segment need splitting can be solved using the same measures of region homogeneity mentioned in the section on region merging, *e.g.*, a segment needs to be split if

- the grey value variance exceeds a threshold, or
- the variance of a texture measure exceeds a threshold, or
- the histogram entropy (or another histogram measure) exceeds a threshold, or
- high edgeness pixels are present.

When a segment needs splitting, a faster approach than starting a segmentation on the segment is to simply split it into four quadrants. This of course has a high probability of placing segment boundaries at unrealistic positions. This problem is reduced by adding merging operations to the segmentation process; recursively splitting and merging the image segments in a way such as the one used in figure 10.22. Especially if different

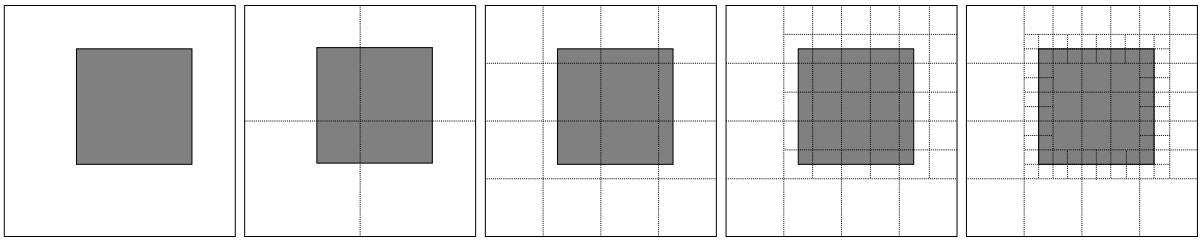


Figure 10.22 Example of split and merge segmentation. Left: original image. From left to right: recursively splitting each segment into four segments when the grey value variance of a segment is non-zero. If we subsequently recursively merge all adjacent segments with the same (average) grey value, we end up with two segments; exactly the desired segments of object and background.

criteria are used for splitting and merging –for example, edge heuristics for the merging and region homogeneity for splitting– the split and merge segmentation may not converge to a stable result, but instead keep alternating between various solutions. This problem can sometimes be avoided by modifying the criteria in the course of the process, but is usually simply ignored by terminating the process after a fixed number of iterations. Another problem of the 'split into quadrants and merge' approach is that the resulting segmentation may be very blocky. Figure 10.23 shows examples of using this approach.

10.3.3 Pyramid, tree, and scale space methods

Trees are often used as data structures that represent segmentation by splitting and merging. For example, figure 10.24 shows there is an easy one-to-one relationship be-

tween splitting an image recursively into quadrants and a quadtree, as we have seen previously in section 8.3.7. The quadtree is often a more convenient and efficient data structure to use in a segmentation by splitting than the image itself. However, using a tree of this type is limited, because we cannot –for instance– model a merging of two segments on different levels of the pyramid. A more common use of a tree structure is to use it in combination with a multiresolution pyramid (or a scale space): the pyramid level defines the tree level, each pixel in the pyramid defines a tree node, and branches can be formed between nodes in adjacent levels. In this context, a branch is usually called a *link*, the high-level end a *parent*, and the other end a *child*. An example of such a pyramid-tree can be seen in figure 10.25. Ways to construct a pyramid have been discussed in section9.3.

There are many ways to construct and use a pyramid-tree for segmentation, but all approaches share three steps:

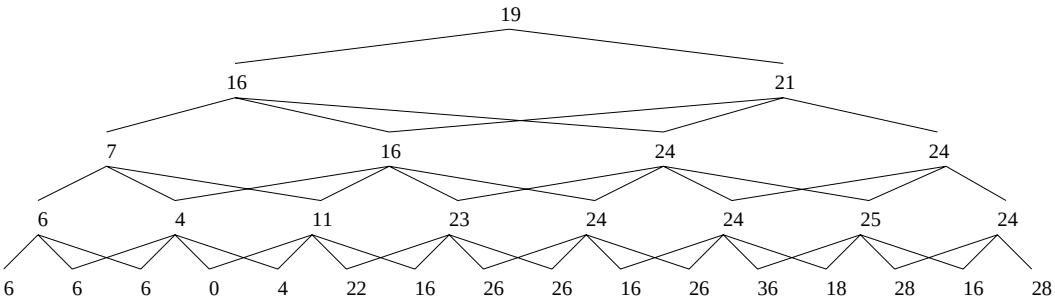
- creating the multiresolution pyramid,
- creating links,
- finding nodes whose leaves (*i.e.*, connected nodes on the base pyramid level) identify a segment. Such nodes are called *roots*.

Many methods iterate between the various steps, *e.g.*, the pyramid may be re-created using the results of the first linking step. In this section, we will show two examples of pyramid-tree segmentation; a classical approach (Burt-pyramid) and an advanced one using scale space techniques.

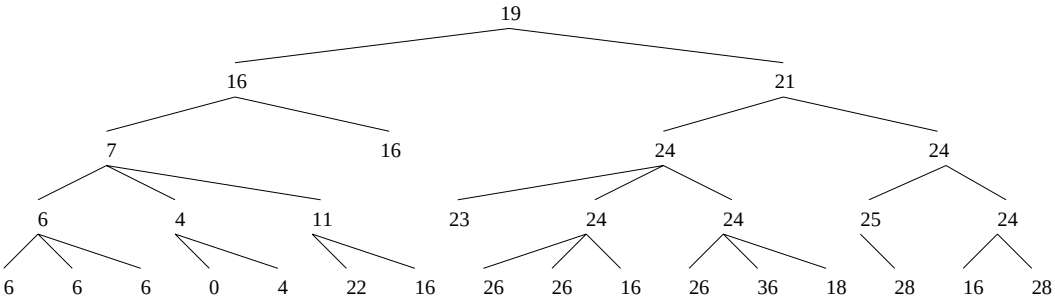
A classical example. The initial pyramid is created by averaging of the grey values of each 4×4 neighborhood. Links are created upwards from the base level of the pyramid. Each child is given exactly one link upwards, to the candidate parent in the 4×4 neighborhood that is closest in grey value. The pyramid is then re-created by assigning to each pixel the average grey value of its children–the base level remains unmodified. The linking and re-creation is iterated to stability. A segmentation is obtained by going down the final pyramid-tree until the number of nodes on the current level equals the pre-defined number of desired segments. These nodes are then assigned to be the segment roots, and the leaves of their sub-trees constitute the various segments.

Example

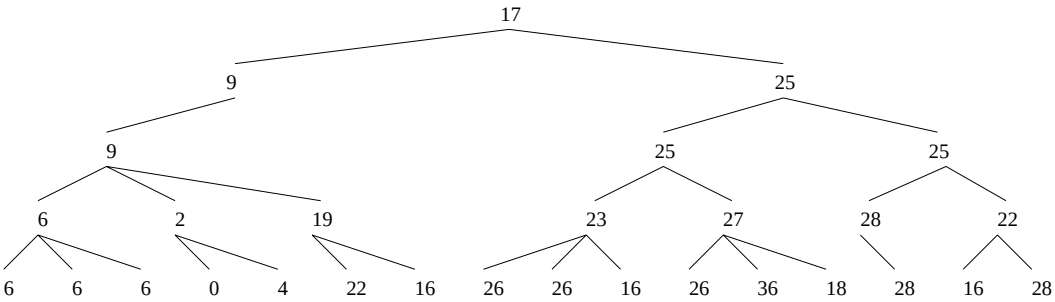
For reasons of clarity, we show here a one-dimensional example of the algorithm described above. We have image data of a noisy edge; the subsequent 16 grey values are $\{6, 6, 6, 0, 4, 22, 16, 26, 26, 16, 26, 38, 18, 28, 16, 28\}$. From this data, we can construct the following pyramid by averaging each 4 grey values. Each pyramid level reduces resolution by a factor of two. Lines show which children where used in the computation of each parent.



Now we link each child to the parent in the 4×4 neighborhood on the current level that is closest in grey value. In case of a tie between candidate parents, we link to the spatially closest parent:



Actually, an additional heuristic was used in the linking process that avoids the crossing of links ('22' and '36' grey values on the base level). When this happens, one of the crossing links is destroyed, and the child is linked to the next closest parent. Now we recompute the pyramid values: each parent is given the average value of its children. Parents without children are deleted from the tree.



Since a repetition of the linking process does not alter the tree, we terminate the process. The segmentation of the image can now be done by bunching all the leaves of the sub-trees at an appropriate pyramid level. For example, if we desire two segments, we take the second level from the top: the sub-tree with root '9' defines one segment, and the sub-tree with root '25' defines the other. Segmentation at lower pyramid levels gives us respectively three, seven, and 16 segments.

A pyramid-tree segmentation using scale space. In this example, the pyramid levels are taken from a scale space built of the original image. The advantage of using scale space is that it is easier to control the reduction of resolution between two pyramid levels. In the pyramid in the previous example, the resolution in each spatial dimension was reduced by a factor of two if we move up a level in the pyramid. In a formula:[14] $N_n = \frac{1}{2}N_{n-1}$, where N_i is the number of pixels in one dimension at level i of the pyramid, the base level being level 0. In terms of the original number of pixels, this becomes $N_n = \left(\frac{1}{2}\right)^n N_0$. In scale space, the story is a bit more complex: although scaling an image lowers its intrinsic resolution, it is common practice to represent the scaled image using the same number of pixels as in the original image, instead of the more natural number of pixels that is inversely proportional to the scale of the image.[15] This is not very practical in our case; we need a decreasing number of pixels as we rise in the pyramid / scale space. To achieve this, we subsample scaled images. The correct subsampling rate and the correct scale σ_n of each pyramid level are related by the equations:

$$\begin{aligned} \sigma_n &= \varepsilon e^{\tau_0 + n\delta} \\ N_n &= N_0 e^{-n\delta}, \end{aligned}$$

where δ is a scale sampling parameter and ε is the pixel distance in the original image. The constant τ_0 is used to relate ε to the inner scale of the original image. Often τ_0 is assumed to be zero, which gives us $\sigma_0 = \varepsilon$ for the original image, *i.e.*, the inner scale of the original image equals the pixel size. The example below shows how we can build a pyramid for segmentation based on scale space.

Example

Suppose we wish to construct a pyramid using scale space where the resolution in each spatial dimension is reduced by a factor of two if we go up a level. Then the following holds (see text above for an explanation): $N_n = \left(\frac{1}{2}\right)^n N_0$. We also know that $N_n = N_0 e^{-n\delta}$. Some calculation then leads to $\delta = \ln 2$. Substituting this in $\sigma_n = \varepsilon e^{\tau_0 + n\delta}$ –where we set ε to 1 (*i.e.*, distances are measured in original pixel lengths), and take τ_0 to be zero– we arrive at $\sigma_n = 2^n$.

We now have almost everything to compute pyramid based on scale space of, say, a 256×256 image: the base of the pyramid is the original image. The first level is created by computing the image at scale $\sigma = 2^1 = 2$, and subsampling it using 128×128 pixels. Level two is created by taking the image at scale $\sigma = 2^2 = 4$, and subsampling it at 64×64. And so on, until the desired number of pyramid levels is reached.

There is only one snag: scaled images must be computed based on the original image, which we assumed to have scale $\sigma = 1$. We must compensate for this 'offset'

[14] For simplicity, we assume that the original resolution is a power of 2.

[15] More accurately, the number of pixels along each spatial axis (*i.e.*, the x-dimension and y-dimension of the image) is inversely proportional to the scale.

scale when computing the pyramid levels. We can do this by using the formula $\sigma_c = \sqrt{\sigma_a^2 + \sigma_b^2}$, where σ_c is the scale of an image with original scale σ_a when convolved with a Gaussian of width σ_b. For example, pyramid level 1, with scale $\sigma = 2$, is computed by applying convolution with a Gaussian with scale $\sigma = \sqrt{3}$ to the original image, because then the final scale σ_1 will be $\sigma_1 = \sqrt{(\sqrt{3})^2 + 1^2} = 2$.

The example above shows how a pyramid can be constructed with a resolution reduction of a factor of two when rising a level in the pyramid. We chose this example because it creates a pyramid similar to the 'ordinary' multiresolution pyramid. For many practical applications, however, the reduction by a factor of two per level is too fast to give satisfactory segmentation results. Using the scale space approach, we now have an easy mechanism to set the resolution reduction per level to a lower value: simply set the variable δ to a lower value ($\frac{1}{2}\ln 2$ instead of $\ln 2$ is a much used heuristic value) and evaluate the pyramid as in the example above.

After creating the pyramid, the next steps of creating links and finding roots are basically the same as in the Burt-pyramid example. The only intrinsic difference in the linking process is that we need to establish a set of candidate parents a child may link to. A common procedure is to examine all parents in a circular region directly above the child, with a radius that is proportional to the scale of the parent level. A heuristic value for the proportionality constant is 2.5.

Linking criteria. In our first pyramid example, a child was linked to the parent that was closest in grey value. For a more effective segmentation, it is for many applications beneficial to use other linkage criteria in addition to the grey value difference. Examples include many of the criteria we have seen before:

- gradient magnitude; a large gradient magnitude (*i.e.*, indication of an edge) of a parent should reduce the probability of a child linking to that parent,
- average grey value of the leaves of the sub-tree of a child; this should be close to the grey value of the parent
- spatial distance of child to a potential parent; a large distance should reduce the link probability.

Probabilistic segmentation. Up until now, we linked children to a single parent only. An interesting addition to the segmentation method is to allow children to link to several parents, assigning a *link strength* to each link in such a way that the combined strength of all links leaving a child is one; so-called *probabilistic linking*. Although this addition makes the process and implementation more complex, the amount of information gained over conventional single-linking is considerable: by forcing a child to have

only a single parent, you essentially throw away most of the available local information, reducing it to a single binary link. The cumulative loss of information considering all children and pyramid levels is substantial. This information loss is not present when probabilistic linking is applied.

When a pyramid-tree is probabilistically linked, and we have identified roots that indicate segments in the tree, a leaf of the pyramid (*i.e.*, a base level pixel) may no longer be uniquely connected to a single root. Instead, we can compute the probability of a leaf belonging to a particular root. This probability equals the sum of the edge strength of all paths leading from the root to the leaf via links. The strength of each path equals the product of the link strengths of the links in the path.

Probabilistic segmentation allows for a more realistic image segmentation in many cases. For example, suppose we have three pixels in a row: the left pixel is inside an object, the right on the outside, and the middle one straddling the object boundary. Conventional segmentation techniques will assign the middle pixel either as object or background pixel[16], which is clearly wrong. Probabilistic segmentation will assign the pixel partially to the object, and partially to the background, which is much more realistic. Examples of probabilistic segmentation are shown in figure 10.26 and figure 10.27. *Using* a probabilistic segmentation is often little more complex than using a conventional one. For example: in non-probabilistic methods, the volume of a segment is determined by counting all pixels contained in the segment. The probabilistic volume is determined by counting each pixel for the fraction that it belongs to the segment. Another example: visualization of the surface of segments obtained by non-probabilistic segmentation often gives a 'blocky' result. By making pixels on the surface that have a lower segment probability partially transparent, the visualization is greatly enhanced. An example of this is shown in figure 10.28.

In our example on scale space based pyramid-tree segmentation, we used linear scale space, *i.e.*, the 'ordinary' scale space generated by convolving an image with Gaussians. As a final note, we mention that non-linear scale spaces, such as the ones mentioned in section 9.2 (with generating equations shown in table 10.1) can often be used effectively, especially those that preserve strong edges.

10.3.4 Texture measures

In the previous sections, we have come across several measures that are used to decide whether regions should be merged or split. In this section, we treat a useful class of measures based on the *texture* of regions. Texture measures are used for segmentation in exactly the same way as the other measures: regions having a similar measure are

[16] or possibly as part of some unrealistic boundary-straddling segment.

candidates for merging, and a region that has high variance of a measure is a candidate for splitting.

Defining 'texture' clearly in words is not an easy task. My dictionary tells me texture is *the mode of union or disposition of elementary constituent parts, as in a photograph, or surface of paper, etc.; minute structure or make; structural order.* In the context of image processing, we might think up a definition featuring keywords such as *local pattern*, *repetition*, *orientation*, *graininess*, but perhaps the easiest path to understanding what we mean by a texture is to simply show a number of specific ones; see figure 10.29. Most texture measures are limited in the sense that they only measure a certain aspect of texture, such as graininess or repetitive behavior. Some care is therefore needed when selecting an appropriate texture measure for a certain task; selecting the wrong measure may very well imply that the textures involved cannot be discriminated sufficiently.

Many image processing papers devoted to the quantification and measuring of texture can be found. For the most part, the approaches presented are based on local image statistics, a (Fourier) image transform, or the syntaxis of image patterns. In this section, we present the texture measures most frequently used in practice. Most of these are based on local image statistics.

We have already encountered many measures that can be used to characterize textures in different contexts. For example, the *range* of grey values in a certain neighborhood of a pixel can be used as a measure for discriminating textures. But the range is nothing but the difference of the maximum and minimum grey value in the neighborhood, *i.e.*, the residue of dilation and erosion, *i.e.*, the morphological gradient. In fact, a large number of the measures presented before can be used for discriminating certain types of texture to some degree. Some that can be used are

- range, morphological gradient,
- homogeneity, (see the region heuristics in section 10.3.1),
- histogram features, such as moments or entropy,
- edge density (f_{ww}), or number of edge pixels in a neighborhood,
- edge (gradient) orientation.

Many texture measures can be embedded in a scale space framework, either by using scale space directly in the computation (*e.g.*, by using scaled derivatives) or by applying the texture measures to scaled images. Since many textures exhibit repetitive patterns of a certain size (*i.e.*, a certain scale), there is often a 'natural' scale for optimal texture measuring. Note that this scale itself can be used as a measure to discriminate textures.

Although a multitude of texture measures can be defined, it is very hard to find a single measure that performs well in all practical situations. In practice, several scalar texture measures are often used simultaneously to discriminate between different textures. This

is often much more effective than using a single measure. For example, measures like average grey value of a region, variance of grey values, and size of the texture pattern can each separately be used to discriminate textures, but it is obvious that very different textures may have a similar average grey value or variance or pattern size, decreasing the discriminative power of the separate measures. If we use all three measures simultaneously, we can discriminate textures even if one or two of the measures have similar values. A simple way to combine several texture measures $x_1, x_2, \ldots, x_n$ is to combine them into a vector $(x_1, x_2, \ldots, x_n)^T$, and define the difference in texture of two regions to be the distance of the respective texture measure vector heads of the regions. To make sure the separate texture measures x_i all have a similar influence on the final result, it is common to put $\frac{x_i - \overline{x_i}}{\sigma^2_{x_i}}$ in the vector instead of x_i. Here, $\overline{x_i}$ denotes the average value of x_i, and $\sigma^2_{x_i}$ denotes the variance of x_i. This ensures no single texture measure has a disproportional influence on the texture vector disctance of two regions.

Autocorrelation. The correlation of an image with translated versions of itself can give us –amongst others– information on the pattern size of a texture. This *autocorrelation* is computed by multiplication of each grey value $f(x, y)$ in a certain region with the translated grey value $f(x + k, y + l)$, and then taking the sum of these multiplications over all the pixels in the region. In practice, it is often convenient to use a slightly more complex formula, which computes a value that is normalized between 0 and 1: the *autocorrelation coefficient* $r(k, l)$ (with translation variables k and l) of a region R –which may be the whole image– is defined by

$$r(k,l) = \frac{\sum_{(x,y)\in R} f(x,y) f(x+k, y+l)}{\sum_{(x,y)\in R} f^2(x,y)}.$$

Example autocorrelation functions can be seen in figure 10.30. A repetitive pattern will cause peaks in the autocorrelation function, with the inter-peak distance equal to the pattern size. If peaks cannot be distinguished with any precision, the slope of the function at $(0, 0)$ may still give information: a relatively small slope, *i.e.,* the function decreases slowly as we move away from $(0, 0)$ indicates a large pattern size, while a large slope indicates a small pattern. The shape of the autocorrelation function, characterized by a vector of suitable scalar shape measures obtained from it, can itself be used for a texture measure. Typically, moments (especially second order) computed from the autocorrelation function are included in the vector, but other information, such as derivatives or even samples of the autocorrelation function at fixed locations, may be included as well. Shape vectors can be used for texture discrimination as described earlier in this section.

The Hurst coefficient. The idea of grey value *range* of a neighborhood as a texture measure can be extended to using neighborhoods of various sizes by using the *Hurst*

coefficient.[17] For a certain pixel (x, y), the Hurst coefficient is computed as follows: for each distance d, find the range $r_{(x,y)}(d)$, *i.e.*, the range of grey values of all pixels at a distance d from (x, y). Now, make a plot of $r_{(x,y)}(d)$ on double logarithmic paper[18], and fit a line through the result. The slope of this line equals the Hurst coefficient. Figure 10.31 shows examples of Hurst coefficient images computed from fabric images. The practical computation of the Hurst coefficient uses only practical values for the distance d. For example, suitable values for d are 1 (neighbors to the left or right, above or below), $\sqrt{2}$ (diagonal neighbors), 2 (neighbors once removed to the left or right, above or below), $\sqrt{5}$ (neighbors a horse jump removed), *etc.* A maximum value for d is defined, which is effectively the neighborhood size around a pixel in which the Hurst coefficient is computed.

Co-occurrence matrices. The entries of a co-occurrence matrix $P_{\alpha,d}(g, h)$ equal how many times the grey value pair (g, h) occurs at a distance d in the orientation α in an image.

Example

Consider this image

0	1	2
2	1	0
0	0	2

Suppose we set distance to $d = 1$ and orientation to $\alpha = 0$, *i.e.*, we consider only directly neighboring pixel pairs in the direction of the x-axis. We see that the pair $(0, 1)$ occurs twice ($0 - 1$ on the top row and $1 - 0$ on the second row), So the entry of $P_{0,1}(0, 1)$ equals two. The pair $(0, 0)$ also occurs twice, so $P_{0,1}(0, 0) = 2$. Note that we count this pattern once for *each* pixel with value 0 that has a neighbor in direction 0 at distance 1, so the occurrence of $0 - 0$ on the third row counts for two occurrences. The full co-occurrence matrix is:[19]

$$P_{0,1} = \begin{pmatrix} 2 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix}$$

The matrix constructed as in the example above is always symmetrical, because both patterns aligned with the orientation α and those 'reversely' aligned ($\alpha + 180^o$) to it are counted as the same pattern ($0 - 1$ is the same pattern as $1 - 0$). It is also possible to

[17] The Hurst coefficient is closely related to the perhaps better known *fractal dimension* D, which is related to the Hurst coefficient H by $D = C - H$, where C is a constant.

[18] In other words, make a plot of $\log r_{(x,y)}(\log d)$.

[19] where we index the matrix image-fashion for convenience instead of mathematical matrix-fashion.

construct a matrix where patterns aligned with and opposite to the orientation are considered to be different. This results in an asymmetric matrix. The asymmetric variant of the co-occurrence matrix is less common than the symmetric version in the context of texture measurement. In practice, co-occurrence matrices are often only computed for choices of d and α that are convenient for digital images. For example, α is usually chosen at 90^o ot 45^o increments, and d is usually an integer or a multiple of $\sqrt{2}$ for diagonal orientations α. For textures where the orientation α is not critical, it is common to use co-occurrence matrices $P_d(g,h)$ that are independent of orientation, *i.e.,* they have entries listing the number of times a grey value pair (g,h) occurs at a distance d, regardless of the orientation α of the pair. Such a matrix can be computed by computing the co-occurence matrices $P_{\alpha_i,d}$ for a number of orientations α_i, and then taking their sum: $P_d(g,h) = \sum_i P_{\alpha_i,d}(g,h)$.

Co-occurrence matrices capture the characteristics of local patterning well, especially if the texture patterns are relatively small. Several scalar measurements obtained from co-occurrence matrices have been shown to be effective for texture discrimination. These measures are usually computed from normalized versions $P^*_{\alpha,d}$ (*i.e.,* the sum of the entries is one) of the co-occurrence matrices $P_{\alpha,d}$. Some useful ones are

- energy (or uniformity): $\sum_g \sum_h (P^*_{\alpha,d}(g,h))^2$.
  A uniform region will cause $P^*_{\alpha,d}$ to have relatively large entries. Regions with random grey values will show the opposite: no large entries, but many small ones. The square in this formula attaches a large weight to large entries, *i.e.,* this measure is large for uniform regions.
- contrast (or inertia): $\sum_g \sum_h (g-h)^2 P^*_{\alpha,d}(g,h)$.
  This measure attaches a large weight to frequently occurring pairs of high contrast (large $|g-h|$).
- maximum probability: $\max_{(g,h)} P^*_{\alpha,d}(g,h)$.
  The probability range is also sometimes used: $\max_{(g,h)} P^*_{\alpha,d}(g,h) - \min_{(g,h)} P^*_{\alpha,d}(g,h)$.
- entropy: $-\sum_g \sum_h P^*_{\alpha,d}(g,h) \log P^*_{\alpha,d}(g,h)$.

Typically, the texture of a region is characterized by computing *several* of the above measures of *several* co-occurrence matrices of the region obtained for different distances d and orientations α. The obtained measurement vector has been shown to have a good texture discriminating power for many practical applications.

Fourier techniques. Since many texture patterns exhibit distinct periodicity and orientation, Fourier techniques can offer a suitable framework to characterize such patterns. Similar to the techniques above, a texture is commonly characterized by a vector of scalar texture measurements. In this context, each measurement is usually obtained from a masked version of the Fourier magnitude image of a region. Common mask shapes are rings (for pattern size related measurements) and wedges (for orientation related measurements), such as the ones in figure 10.32. The measurement itself is usually

energy, *i.e.*, the sum of squared values in the masked Fourier image, or the maximum value occurring in the masked Fourier image.

10.4 Clustering techniques

Clustering techniques is the collective name for methods that attempt to group together measurements points ('patterns'). For an off-topic example, let's say we measure the weight and length of a population of rabbits. We can then plot the measurements of each rabbit together as shown in figure 10.33. When looking at the figure, it will be clear that three clusters can be identified –there are very likely three types of rabbit in the population. The object of clustering techniques is to identify such clusters in data. The relation to segmentation will be clear; when viewing the rabbit plot as an image, the most intuitive segmentation would be to divide the image into segments equal to the three clusters and the background.

Clustering techniques are often formulated for data of arbitrary dimension (not just two as in our rabbit example), but many clustering methods can readily be applied to two- or three-dimensional images. The images best suited for applying clustering techniques to are those similar in appearance to the rabbit plot, *i.e.*, of a very sparse nature, the dark pixels forming quasi-coherent clouds.

The rabbit data consists of a list of pairs of measurements. This data representation is the most natural for clustering techniques. If we wish to apply clustering to a binary image, we therefore use a similar representation: each pixel with value one is included in a data list of coordinate pairs. For grey-valued images there may not be a natural conversion to a list of data points. For some types of grey-valued images, it is possible to consider the grey value as the number of data points measured at a certain location.

Example

This binary image:

0	1	0
0	1	1
0	0	1

can be represented as a set of data points by listing the coordinates of the 'one' pixels:

$$\{(1,0),(1,1),(2,1),(2,2)\}.$$

The following grey valued image can similarly be represented by listing all coordinates a number of times equal to the grey value:

0	1	2
0	1	1
0	0	3

$$\{(1,0),(1,1),(2,0),(2,0),(2,1),(2,2),(2,2),(2,2)\}.$$

The heart of most clustering techniques is the *distance matrix* D. An entry $D(i,j)$ gives the distance between two data points i and j. If there are N data points, then D is an $N \times N$ symmetric matrix.

Example

Given this list of data points:

$$\{(1,1),(3,1),(2,2),(2,4),(2,5),(3,5)\},$$

then the 6×6 symmetric distance matrix D is

$$\begin{pmatrix} 0 & 2 & \sqrt{2} & \sqrt{10} & \sqrt{17} & 2\sqrt{5} \\ 2 & 0 & \sqrt{2} & \sqrt{10} & \sqrt{17} & 4 \\ \sqrt{2} & \sqrt{2} & 0 & 2 & 3 & \sqrt{10} \\ \sqrt{10} & \sqrt{10} & 2 & 0 & 1 & \sqrt{2} \\ \sqrt{17} & \sqrt{17} & 3 & 1 & 0 & 1 \\ 2\sqrt{5} & 4 & \sqrt{10} & \sqrt{2} & 1 & 0 \end{pmatrix}$$

where an entry $D(i,j)$ gives the distance between the i-th and j-th data point in the list.

In the example above, we used the Euclidean distance between data points. Other types of distance may also be employed. In a more general vein, the matrix may also be a *dissimilarity* matrix, *i.e.,* its entries give the dissimilarity between two data points, which may not equal the distance between them.

Although many different clustering techniques exist, we will discuss in this section only two basic approaches; an agglomerative one and a partitional[20] one. We will see that these general techniques bear a great resemblance to techniques we have seen in different contexts.

[20] 'Partitional' could be replaced 'divise' as used in reference to other segmentation methods here, but the clustering literature usually reserves 'divise' for a different type of clustering method.

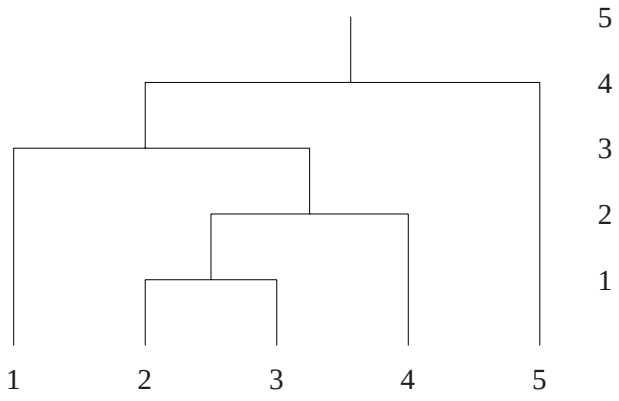
Agglomerative clustering. This approach starts out by calling each data point a separate cluster, and then proceeds to merge appropriate clusters into single clusters. The dissimilarity matrix is used to decide which clusters are to be merged; the smallest entry gives the data points that are least dissimilar and hence are the most likely candidates to be merged. After merging, the next smallest entry is selected, supplying two new merging candidates, *etc.*

Example

Given five data points with the following dissimilarity matrix:

$$D(i,j) = \begin{pmatrix} \times & 7 & 9 & 3 & 10 \\ & \times & 1 & 8 & 4 \\ & & \times & 2 & 5 \\ & & & \times & 6 \\ & & & & \times \end{pmatrix},$$

(where we have omitted the lower half because there is no additional information because of the matrix symmetry, and left out the diagonal which has only zeros). The distance between data points 2 and 3 is smallest (1), so the data points 2 and 3 will be merged first. The next smallest entries are 2, 3, and 4, so the next merging candidates are respectively the data point pairs 3 and 4; 1 and 4; and 2 and 5. The respective merges are visualized easiest in a tree structure:



The numbers at the bottom of the tree signify the five data points, the numbers to the right indicate the level of the tree. The tree shows that data points 2 and 3 are merged first (tree level 1), then their cluster is merged with data point 4 (level 2), then with 1 (level 3), and finally with 5 (level 4).

We can use the tree level to obtain the desired number of clusters in an image. For example, if we want two clusters, we select level 4, defining the two clusters: $(1,2,3,4)$ and (5). Level three defines three clusters: (1), $(2,3,4)$ and (5).

The type of tree used in the above example is usually called a *dendrogram*. An algorithm to perform the above described clustering is:

Algorithm:
Given n data points and a dissimilarity matrix D:

1. Initialize by defining each data point to be a separate cluster. Set an iteration counter $c = 0$.
2. Select the smallest entry in D: $D(a, b) = \min_{(i,j)} D(i, j)$.
3. Merge the clusters a and b.
4. Update D:
   (a) Update row b and column b by setting $D(k, b) = \min\{D(k, a), D(k, b)\}$ and $D(b, k) = D(k, b)$, both for all possible k, $k \neq a$, $k \neq b$.
   (b) Delete row a and column a
5. Increase c by one, and go to step 2 if $c < n$.

This algorithm will merge clusters if there are member data points that are closest considering *all* data point pairs. A variation on this algorithm is to merge two clusters if the *largest* distance between its member data points is the smallest when considering all clusters. This can be achieved by a simple modification of the above algorithm: the 'min' in the update step 4 must be replaced by 'max'.

Partitional clustering. The most frequently used form of partitional clustering is to divide the data points into clusters in such a way that the total distance of data points to their respective cluster centers is minimal. An algorithm to achieve this is called K-*means clustering*, and we have already seen this algorithm in a one-dimensional guise earlier in this chapter.

Algorithm: K-means clustering
This algorithm minimizes the total distance of data points to the cluster center of the cluster they are assigned to. Note that it does not require the actual computation of distances.

1. Select the number of desired clusters k. Arbitrarily (or better: intelligently) place the k cluster centers at different initial locations in the image.
2. Assign each data point to the cluster whose center is closest.
3. Recompute the cluster centers; the cluster center should be at the average coordinates (center of gravity) of the data points that make up the cluster.

4. Go to step 2 until no more changes occur or a maximum number of iterations is reached.

A drawback of this algorithm is that the number of desired clusters needs to be set beforehand. An addition to the above algorithm is therefore to embed it into an outer loop that modifies the number of clusters, or, more precisely, removes irrelevant clusters, and splits existing clusters where appropriate. After this modification, the entire new algorithm is iterated until stable. A criterion for removal of a cluster is when the number of contained data points falls below a set threshold. When this happens, the data points are probably outliers and can usually be safely ignored in the rest of the process. A criterion for splitting a cluster can be when a distance can be found of a contained data point to the cluster center that is relatively large compared to the average distance of the data point to *all* cluster centers. The splitting can then be implemented by setting this data point to be the center of a new cluster.

In a manner similar to pyramid-tree segmentation, the algorithm can be extended to be probabilistic, *i.e.,* data points can belong to several clusters simultaneously, or –more precisely– are assigned a probability distribution giving the probabilities of the data point belonging to particular clusters. This probabilistic variant is usually called *fuzzy clustering.*

10.5 Matching

If we want to locate an object in an image, and we have available an example of what it should look like (a *template*), we can find this object by matching the template to various image locations until we have found the object. For an example, see figure 10.34. The most straightforward way of determining whether a template 'fits' would be to place the template at a certain image location, and see whether the grey values of the template and the underlying image image grey values all match. However, because there will generally be some differences between the image and template values because of noise and other artifacts, this is not a very practical method. More useful is a quantitative measure of fit such as

$$M_1(p,q) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left(g(x,y) - f(x+p, y+q)\right)^2,$$

where f is the image, g the $M \times N$ template, and the variables p and q determine the location of the template in the image, see figure 10.35. This measure will be small if

the template is similar to the part of the image under investigation; then all grey value differences $g(x,y) - f(x+p, y+q)$ are small and the sum M_1 will be small. The location of optimal template fit is found by minimizing M_1 to p and q.

Besides the squared form M_1, another often used measure is M_2, which uses the actual grey value differences instead of their squares:

$$M_2(p,q) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} |g(x,y) - f(x+p, y+q)|.$$

M_2 puts less weight on relatively large grey value differences than M_1. But perhaps the most commonly used measure is the *cross correlation* M_3, which is defined by

$$M_3(p,q) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} g(x,y) f(x+p, y+q).$$

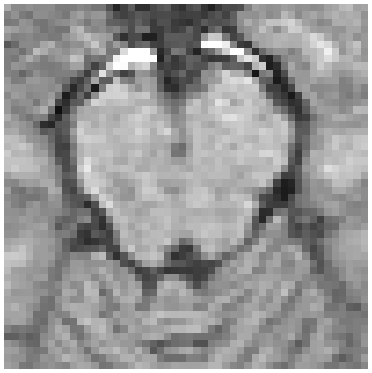
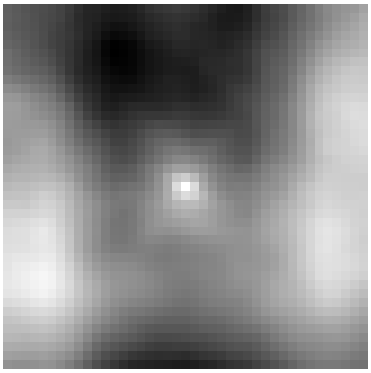
The maximum of M_3 with respect to p and q is assumed to indicate the location in the image where the template fits best. There is a strong relation between the measures M_1 and M_3, which is most obvious if we extend the sums in the measure M_1 to the entire image f (and pad the template g with zeros); then the measure M_3 appears literally between constants if we expand the square:

$$\begin{aligned} M_1^*(p,q) &= \sum_{x=0}^{d_x-1} \sum_{y=0}^{d_y-1} (f(x,y) - g(x-p, y-q))^2 \\ &= \underbrace{\sum\sum f^2(x,y)}_{\text{constant}} - 2 \underbrace{\sum\sum f(x,y) g(x-p, y-q)}_{=M_3} + \underbrace{\sum\sum g^2(x-p, y-q)}_{\text{constant}} \end{aligned}$$

where d_x and d_y are the dimensions of the image f.

-normalized cc

-example

40x40 around optimal position, optimal pos obviously at image center

-other measures

-other transforms

-patt recogn

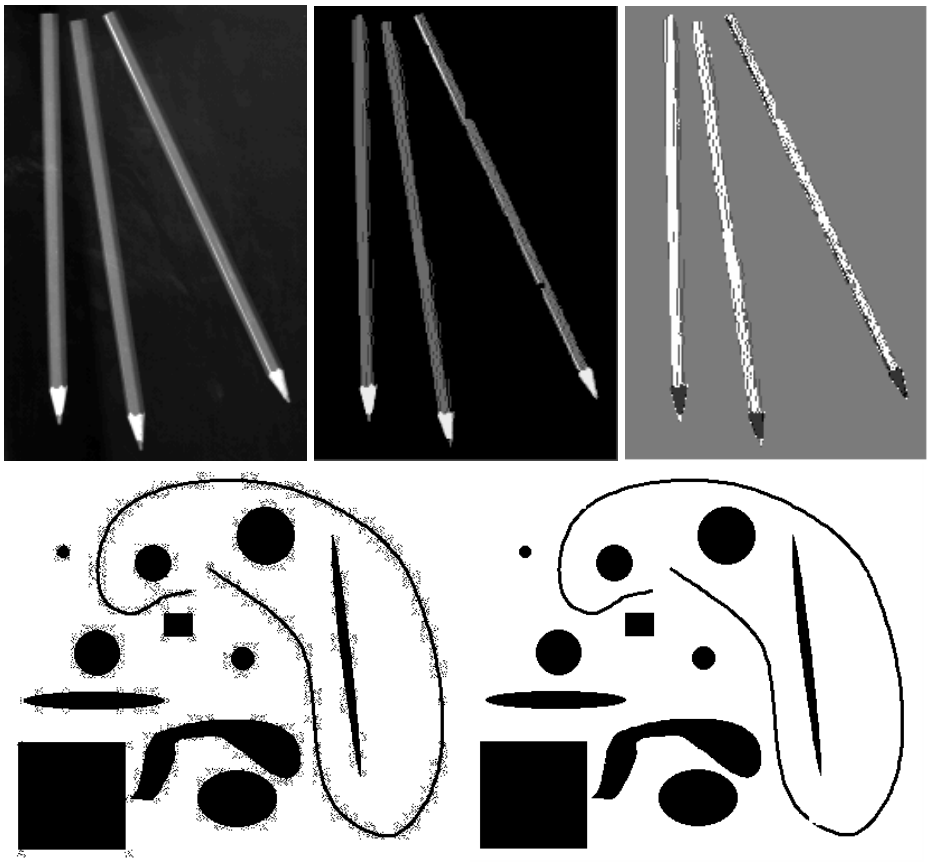


Figure 10.23 Examples of using a split and merge technique to segment images. The initial segmentation was the entire image, *i.e.,* one segment. Recursive splitting of segments into four quadrants and merging of neighboring segments was applied, both using grey value variance as a criterion. The process was iterated until stable. Top row, left: original image. Middle: after the segmentation process, with each segment assigned a grey value equal to the mean of the original grey values of the segment pixels. Right: after random remapping of the grey values to show individual segments. Bottom row, left: original binary image containing compression artifacts. Right: after segmentation, with each segment represented by the mean original grey value.

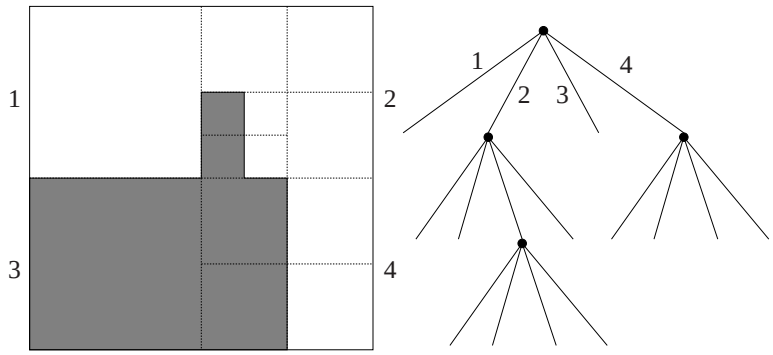


Figure 10.24 Example of modeling a recursive splitting of an image by a quadtree. The numbers show the correspondence between image quadrants and tree branches.

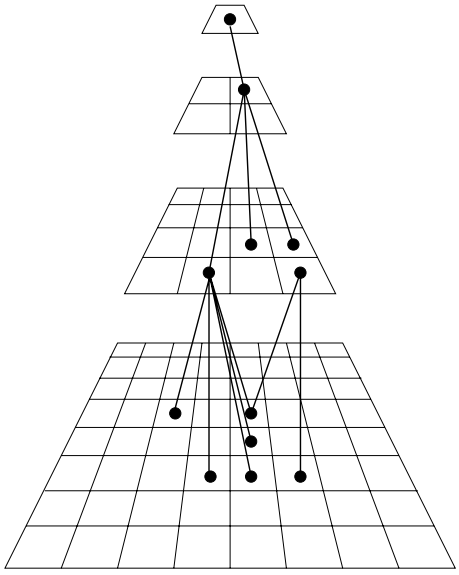


Figure 10.25 Example of a multiresolution pyramid-tree. The original 8×8 image rests at the bottom of the pyramid. Each level above is formed by reducing the resolution in each dimension by a factor of two. Each pixel represents a node in the tree, and links can be formed between nodes in adjacent levels. Some example links are drawn.

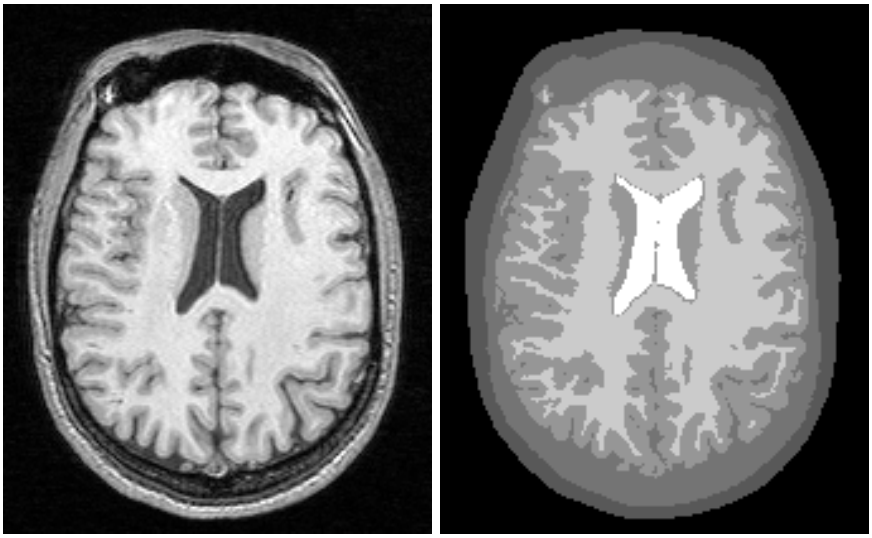


Figure 10.26 Example of probabilistic segmentation. Left: original image. Right: after probabilistic pyramid-tree segmentation (6 segments).

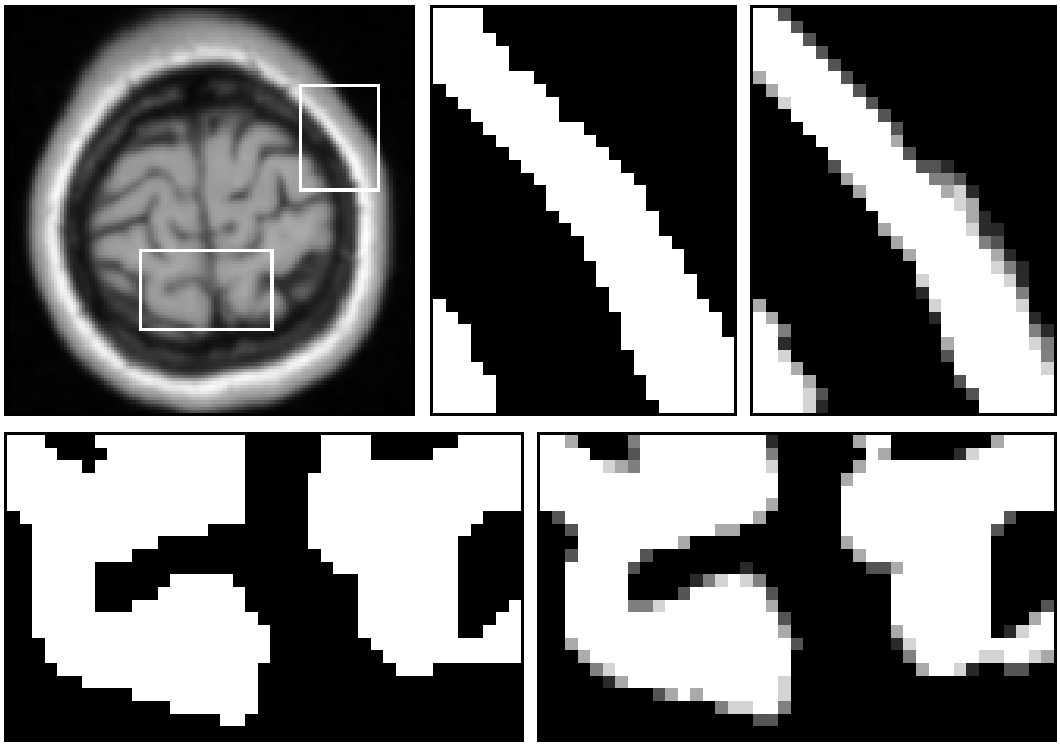


Figure 10.27 Example of the advantages of probabilistic pyramid-tree segmentation. Top left: original image. Middle: zoomed result of segmentation using single-parent linking, showing the problems when pixels are forced to belong exclusively to a single segment. Right: result of probabilistic segmentation, where the grey value of pixels indicates the probability they belong to the object segment (probability one corresponds to white, zero to black.) Bottom row: same images, zoomed in on a different image area. The zoom areas are indicated by the boxes in the original image. Note that the probabilistic segmentation is much more realistic. The conventional segmentation shows an unlikely jagged boundary.

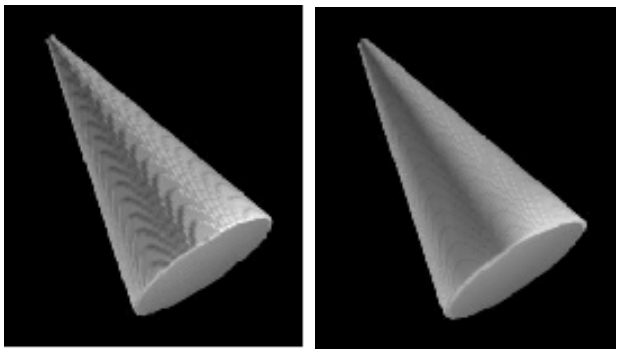


Figure 10.28 Example of the advantages of probabilistic segmentation. Left: visualization of a segment where each pixel belongs either fully to the object, or fully to the background. Right: probabilistic visualization. Pixels are made transparent proportional to the probability that they belong to the background. Note the much smoother looking surface.

Figure 10.29 Examples of various kinds of texture.

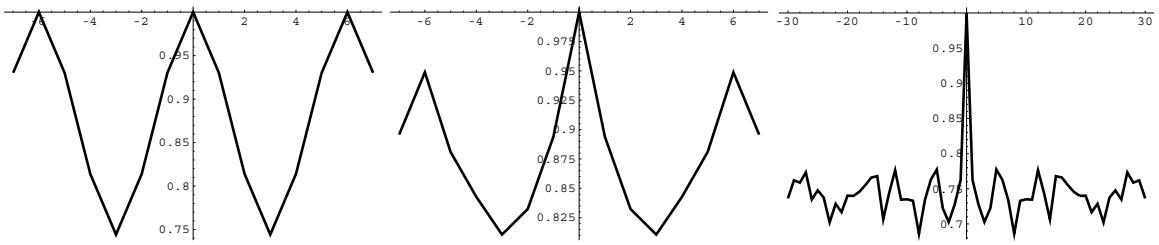


Figure 10.30 Plots of the autocorrelation coefficient, as a function of x-translation. The region used was a 42×1 rectangular window. Left: autocorrelation of the window containing the values $\{1,2,3,4,3,2,1,2,3,4,3,2,1,2,3,4,3,2,1,2,3,4,3,2,1,2,3,4,3,2,1,2,3,4,3,2,1,2,3,4,3,2,1,2,3,4,3,2\}$, *i.e.,* containing a pattern of length 6. Note that the peaks of the correlation function occur at a distance of 6 pixels. Middle: using the same window, but with added noise: $\{1,5,5,4,4,6,2,4,6,7,3,2,2,2,5,7,5,2,5,2,7,7,6,2,3,3,7,6,6,6,5,2,6,7,5,6,4,3,5,6,5,5,2,3,6,7,6,4\}$. Note that the peaks of the function still occur at a six pixel interval. Right: autocorrelation function of a window containing random values. The symmetry of all three plots is caused by using padding with the window itself to obtain unknown pixel values.

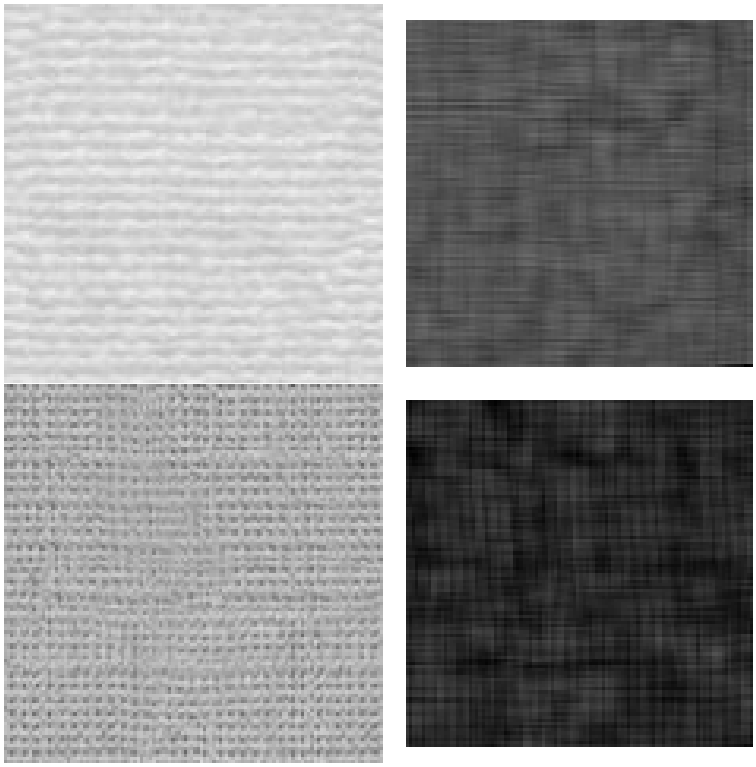


Figure 10.31 Hurst coefficients for two types of fabric. Left: original 120×120 images. Right: Hurst coefficient computed using a neighborhood of size 15. The white border stems from the fact that only neighborhoods that fit inside the image boundary were used.

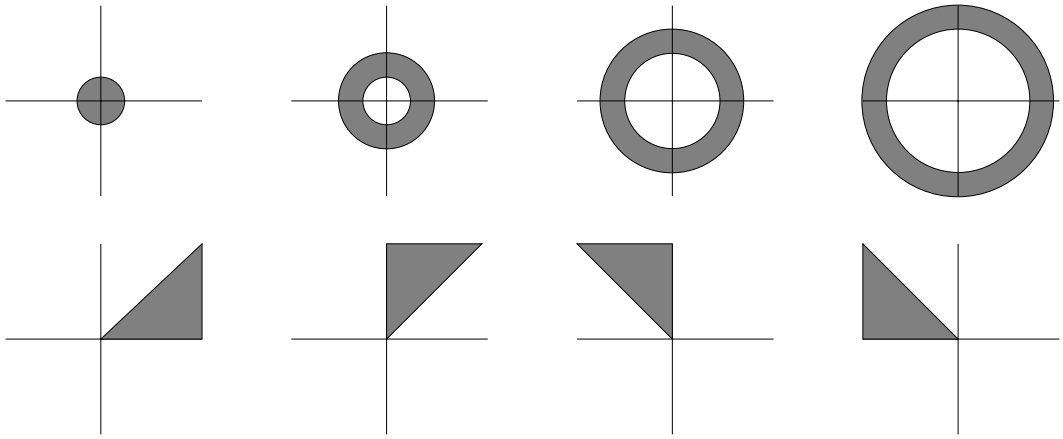


Figure 10.32 Example of masks used in Fourier texture description. Top row: ring-shaped masks, which let pass only a certain range of frequencies (band-pass). Bottom row: wedge shaped masks, which let pass only those frequencies directed along a limited range of angles. Typically, measurements obtained from all masked images are combined into a single texture-characterizing vector.

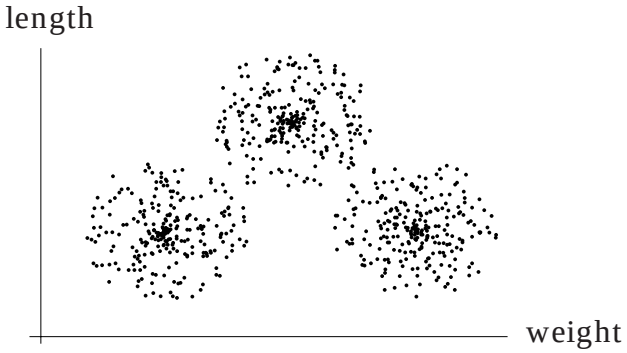


Figure 10.33 Finding clusters of measurement data points is often similar to image segmentation.

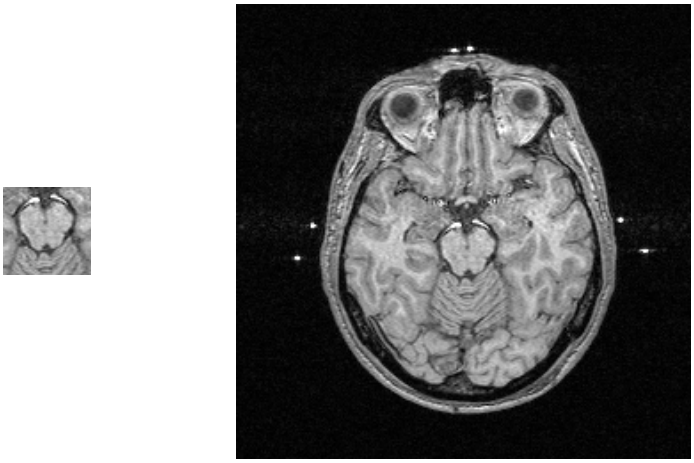


Figure 10.34 Image matching. The left image gives a template of the object we wish to locate in the right image. This is achieved by matching of the template to various locations in the right image.

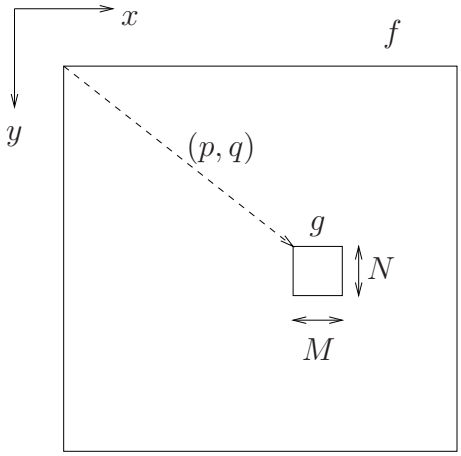


Figure 10.35 Definitions for image matching.