5.1 Segmentation

Introduction to Segmentation. A blob is an object in an image whose gray levels distinguish it from the background and other objects. For example, it could be a cancer cell or bacterium, an airplane, or a part to be attached to a subassembly. It is a type of region, which is a connected set of pixels that have one or more similar properties that are different than the surrounding pixels. Segmentation of an image is a process of finding any blobs or determining regions of interest in a certain context. The context may be simple, such as gray level only, or more complex such as gray level, closeness and on the same side of an edge. Before any segmentation is done, processes of contrast stretching, histogram equalization, despeckling and light smoothing or other noise removal can be helpful because noise can affect the process of decomposing the image into segments.

Segmentation is used increasingly in computer vision, where machine systems use a video camera (or x-ray machine, laser, radar sonar or infrared sensor devices) and an algorithm running in a processor to find objects and perhaps their spatial orientation from captured images. Applications include inspection of metal objects for cracks, target detection, identification and orientation of parts, contamination in boxes of material, classification of crop fields from satellite images, determining tissue type (bone, fluid, gray and white matter) in brain scans, etc.

There are multiple approaches to segmentation. The most common ones are

Thresholding: one or more thresholds are determined and all pixels with values between two adjacent thresholds, or on one side of a threshold, are lumped together into a region or blob of one gray level.

Region growing: a pixel in a region is selected initially and adjacent pixels are adjoined if they have similar gray levels; small subregions are merged if their interiors have similar properties (such as gray level). These methods may be unstable and are computationally expensive.

Morphology: the boundaries of regions are expanded or eroded away by certain morphological operations. These are efficient and effective for black and white images but can also be done on grayscale images by treating the lighter pixels as white and the darker ones as black.

Edge approaches: edges are determined and then pieces of edges are linked to form boundaries for blobs or a line drawing image. These use a large volume of computation.

Clustering: grouping pixels into clusters if their gray levels or colors are similar to a prototype gray or color. These are very powerful.

Other: a variety of other methods and techniques for implementing the methods are used such as template matching, texture segmentation, neural networks, boundary tracking beetles and fuzzy methods.

Segmentation by Thresholding.  This is the easiest to implement and so we cover it here as a pedagogical tool before discussing the more specialized methods and techniques. The basic case uses a single threshold $T$. This is satisfactory for a global approach when the background is uniformly different from the blobs of interest and the contrast is rather uniform (else we can perform histogram equalization or a statistical adjustment). Let the blob of interest have higher intensity than the background (the opposite case is analogous). We can select $T$ automatically as the average gray level of the image that is computed without the pixels in small blocks in the four outer corners (substantial numbers of pixels are both below and above this $T$). Then each pixel is tested, and those with gray levels greater than $T$ are changed to a light shade of gray, while those whose values are less than or equal to $T$ are given a dark gray level.

It is useful to zero out the pixels below a threshold just above the background level and put everything above that at 1 and save as an image. This new binary image $\{f_{\text{bin}}(m,n)\}$ is a logic mask for multiplying the original image $\{f_{\text{orig}}(m,n)\}$ by this binary image to zero out the background in the original image while
keeping the blobs in their original form. Thus we put

\[ g(m,n) = f_{\text{orig}}(m,n) \cdot f_{\text{bin}}(m,n), \quad \text{if binary value is 1} \]  

\[ g(m,n) = f_{\text{orig}}(m,n) \cdot f_{\text{bin}}(m,n) = 0, \quad \text{if binary value is 0} \]  

(5.1a)  

(5.1b)

**Figure 5.1. Finding a threshold.**

A threshold can be determined by looking at the histogram and finding the point at a local minimum where the pixels in the blob become increasingly more numerous as the shade of gray increases. Figure 5.1 shows the situation for L gray levels. In the case where the background is lighter than the blobs of interest, the image inverse is used. *LView Pro* or *XView* can be used handily for this by manipulating the line in the intensity level adjustment box.

**An Example: Segmentation by Thresholding.** Figure 5.2 shows an original image of pollens and Figure 5.3 presents an enlarged cropped section. Figure 5.4 shows the twice despeckled image of Fig. 5.3 with a 3x3 median filter, which also lightly smoothed it. Figure 5.5 shows the cropped decomposed section that has been thresholded into four segments via the use of three thresholds \( T_1, T_2 \) and \( T_3 \).
Here we used Matlab to look at the histogram of the cropped and despeckled image of Figure 5.4. We used the following commands to obtain the histogram shown in Figure 5.6. Upon examining the histogram, we selected the thresholds $T_1 = 120$ and $T_2 = 170$ and $T_3 = 225$. However, the histogram shows that we could perhaps have picked 4 thresholds to good advantage.

```matlab
>> Im = imread('pol-crop-despeck.tif'); //read in cropped and despeckled pollen image
>> imshow(Im); //show image on screen
>> figure, imhist(Im); //show histogram as new figure
```

Figure 5.6. Histogram of cropped despeckled image of pollens.

---

**Adaptive Thresholding.** In many images the background is not uniformly distributed and the blobs do not have uniform contrast. One approach is to break the image into small blocks of, say, 64x64 pixels. The next step consists of finding the minimum between the peaks of background and blob pixels and using this as the threshold $T_i$ for the $i$th block. Each block is processed according to its threshold to put the background at a darker (or lighter) level while putting the blob subregions at a lighter (or darker) level. The advantage of adaptive thresholding is that it usually produces more clearly separated blobs and fewer errors due to the lumping together of different blobs than does a single threshold. Statistical methods can also be used to obtain a uniform contrast throughout the entire image. We can use Matlab’s $T = \text{graythresh}(I1)$ function and then $I2 = \text{im2bw}(I1,T)$.

---

**Segmentation by Region Growing.** The first method of growing a region for a segment is the technique known as *decomposition into block subregions*. The image is divided into a large number of very small subregions of pixels with similar gray levels where the initial blocks are single pixels. A new method that we have used to obtain the initial blocks is the fuzzy averaging over neighborhoods of pixels (see below). A simpler method is to use the average or median on a block of pixels if the pixels have small differences, or else the block is not changed. The image is partitioned into small blocks, say 4x4 or 8x8, and each pixel in any block that is written to the output image has the block’s average value. Region growing methods are usually computationally expensive and some can be unstable.

**Adaptive Region Growing.** Starting with the first pixel in the image, $f(0,0)$, this method checks its adjacent neighbors $f(0,1)$, $f(1,0)$, $f(1,1)$. If any neighbor $f(m,n)$ has a difference $d = |f(0,0) - f(m,n)|$ less than a small threshold, then we write them to an output file \{g(m,n)\} that we initialized at $g(m,n) = 0$ for all m and n.

Now let $f(m,n)$ be the current pixel examined: moving sequentially along each row and along consecutive rows in order, we apply the same test to the entire 3x3 neighborhood of pixels $f(m\pm1,n\pm1)$, $0 \leq i,j \leq 1$, adjacent to the current pixel. If $d < T$ for any neighbors, then we write them to the output file in the same locations and same gray levels. If no neighbors satisfy this criterion, none are written to the output file (which remains at 0 by default) and the next pixel in order is examined. If a pixel is within $T$ of the zero gray level, it is skipped over (given that the blobs are light and the background is to be dark).
Region Separation via Flooding. The flooding method, also called the watershed method, is a region growing method. Let there be light blobs on a dark background. We take a 1-dimensional slice across the mth image row \{f(m,n): 0 \leq n \leq N-1\} and graph the inverted grayscale values 255 - f(m,n). We would not invert if the blob were darker than the background. A threshold \( T_n \) on the gray level (vertical axis) is taken to be very low and then raised consecutively as the 1-dimensional regions expand until they meet. The point where they meet with threshold \( T_n \) is a boundary point, as shown in Figure 5.7. By taking slices across all rows, the boundaries between the blobs, and between the blobs and the background, can be mapped. Horizontal slices can be taken also. The image should have good contrast for better results.

An Example: Watershed Segmentation with Matlab. We load in the image that is the cropped part of the image pollens.tif of Figure 5.3 and implement the following steps: i) read in image; ii) enhance image by stretching the contrast; iii) take complement so the former low flood areas (valleys) will now be high intensity regions and the boundaries between them will be low intensity valleys; iv) modify the image with a parameter to provided the intensity threshold below which valleys will become boundaries; and v) do the watershed segmentation. The following commands are used to obtain the segmentation of the figure shown in Figure 5.8.

```
>> Im1 = imread('polcrop.tif');  //read in cropped and despeckled pollen image
>> figure, imshow(Im1);         //show image on screen
>> Im2 = imadjust(Im1,[0.24 0.76],[0 1]);  //stretch the contrast
>> figure, imshow(Im2);  //show the contrast stretched image as new figure
>> Iec = imcomplement(Im2);     // taking the inverse of the contrast stretched image
>> Iemin = imextendedmin(Iec, 18);  //detect all the intensity valleys deeper than 18
>> Iimpose = imimposemin(Iec, Iemin);  //modify the image to contain only the valleys
>> wat = watershed(Iimpose);  //watershed on the valleys.
>> rgb = label2rgb(wat); //color in segments for visualization
>> figure, imshow(rgb); //display the watershed segmentation
```

Figure 5.7. Flooding (watershed).

Figure 5.8. Cropped original.
Figure 5.9. Contrasted image.
Figure 5.10. Complement image.
Clustering into Regions. The number $K_0$ of regions must be decided first and it is safer to use a reasonably large number such as $K_0 = 32$ to 64 to start. Examination of the histogram can provide a good value of the number of small clusters that make a good start. If we put $K_0 = 32$, for example, then we choose 32 gray levels $g_k$ to be: 4, 12, 20, 28, ..., $4 + 8k$, ..., 252 according to

$$g_k = 4 + 8k, k = 0, \ldots, 31$$

(5.7)

Each pixel is tested to determine the prototype to which it is closest and the output pixel is set to that prototype gray level. After that assignment, any prototype gray levels that have not been assigned to output pixels are eliminated. The remainder $K$ clusters form regions, although they are not necessarily connected. At this point a merging process can be applied for the remaining clusters that are adjacent to each other in prototype gray level (for this reason we need more clusters with prototypes close to each other in gray level).

Algorithm 5.1. Clustering into Regions

input $K_0$; //initial number of clusters to be tried
\[ g = \frac{256}{K_0}; \] //get increment gray value
for $k = 0$ to $K_0 - 1$ do //for each cluster center index
\[ g[k] = g/2 + g^*k; \] // compute the center (prototype) gray level
for $m = 0$ to $M-1$ do //for all rows in image and
for $n = 0$ to $N-1$ do // for all columns
\[ d_{\text{min}} = 0.0; \] // get minimum distance to a prototype
for $k = 0$ to $K_0$ do // over all $K_0$ prototypes
\[ d[k] = |f[m,n] - g[k]| \] // by computing distance
if $d_{\text{min}} > d[k]$ then // and comparing min. distance with each distance
\[ d_{\text{min}} = d[k]; \]
\[ k_{\text{min}} = k; \] //update current minimum distance
\[ g[m,n] = g[k_{\text{min}}]; \] // and record its index
//finally, output the pixel at the winning gray level

Segmentation with Weighted Fuzzy Expected Values. The weighted fuzzy expected value (WFEV), denoted by $\mu_p$, is computed over a pxq neighborhood of pixels to achieve small subregions such that the pixels of each subregion are similar in gray level. Similar small regions are merged afterwards and this is repeated until no more merging occurs. We show the method here for 3x3 neighborhoods.

The WFEV $\mu_p$ on a 3x3 neighborhood of $p_i$

$$p_1 \ p_2 \ p_3 \ p_4 \ p_5 \ p_6 \ p_7 \ p_8 \ p_9$$

is defined by Equation (5.2). It replaces the pixel $p_i$ in the new image. Each pixel in the image is processed this way. We compute the preliminary weights $\{w_i: 1 \leq i \leq 9\}$ on a particular neighborhood via
\[ w_i = \exp[(p_i - \mu_i)^2/(2s^2)] \]  

(5.2)

where \( s^2 \) is the variance of the pixel values. Next, we compute the (positive) fuzzy weights

\[ \alpha_i = w_i / \sum_{i=1}^{9} w_i \quad (\sum_{i=1}^{9} \alpha_i = 1) \]  

(5.3)

The weighted fuzzy expected value (WFEV) for the 9 pixels is the weighted average

\[ \mu_F = \sum_{i=1}^{9} \alpha_i p_i = \frac{\sum_{i=1}^{9} w_i p_i}{\sum_{i=1}^{9} w_i} \]  

(5.4)

The weighting is greatest for the pixels \( p_i \) that are the closest to \( \mu_F \) (see Equation (5.2)). However, \( \mu_F \) is a function of \( \mu_{F_3} \), and so we use Picard iteration starting from the initial value \( \mu_F^{(0)} \) that is the arithmetic mean. Convergence is extremely quick and requires only a few (4 to 6) iterations to be sufficiently accurate in most cases. Thus we compute

\[ \mu_F^{(0)} = (1/9) \sum_{i=1}^{9} p_i \]  

(5.5)

\[ \vdots \]

\[ \mu_F^{(r+1)} = \sum_{i=1}^{9} \{\exp[(p_i - \mu_F^{(r)})^2/(2s^2)]p_i\} / \sum_{i=1}^{9} w_i \]  

(5.6)

Another technique is to decompose an image into small subregions first and then consider their boundaries. A boundary between two adjacent subregions is weak if some property of the pixels inside the two subregions is not significantly different, or else the boundary is strong. A commonly used property is the gray level average of the pixels in the interior of the subregion (not on the boundary). Region growing starts with the first subregion in an image. If there is a weak boundary with any adjacent subregion, then that subregion is merged (the weak boundary is dissolved by putting the boundary pixels at the same gray levels as the regions being connected). The process continues until there are no remaining adjacent subregions with weak boundaries, in which case we jump to the next subregion, and so forth.

Interactive processing is where the user selects a point in a blob and an algorithm then starts at that point and grows a region by including all adjacent pixels whose gray levels are close to those of the selected pixels. Interactive methods can be very powerful because they rely on the extra knowledge and intuition of the user.

5.2 Boundary Construction

Edge Detection for Linking. The first step here is to perform an edge detection process. The Laplacian is a method to establish strong edges and it can be implemented with a 3x3 or 5x5 mask as was done in Unit 3. However, it strongly enhances noise which could be detected as boundary points, so another edge detection method is desirable. This is the gradient magnitude, defined to be

\[ |Df(m,n)| = \left( \left( \frac{\partial f}{\partial x} \right)^2 + \left( \frac{\partial f}{\partial y} \right)^2 \right)^{1/2} \]  

(5.8)

The approximations of the partial derivatives can be done as given in Unit 3. In practice, this method requires extra computation to perform the squaring and the square root (an iterative algorithm). The gradient magnitude can be approximated with significantly less computation via

\[ |Df(m,n)| = \max \{|f(m,n) - f(m, n+1)|, |f(m, n+1) - f(m+1, n)|\} \]  

(5.9a)
Another popular gradient edge detector is the Roberts edge operator given by
\[
g(m,n) = \left\{ \left( f(m,n) - f(m+1,n+1) \right)^{1/2} + \left( f(m+1,n) - f(m,n+1) \right)^{1/2} \right\}^{1/2}
\] (5.9b)

The innermost square roots make the operation analogous to the processing of the human visual system. The trade-off is that this is somewhat computationally expensive because of the square roots, although the outermost square root can be omitted. Gradient magnitudes are large over steep rises in the gray level in any direction, although the directional information is lost with Equations (5.8) or (5.9a,b). The edge direction (slope) is
\[
\Theta(m,n) = \arctan \left( \frac{\| \partial f/\partial x \|}{\| \partial f/\partial y \|} \right)
\] (5.10)

The following algorithm uses any of the gradient magnitudes. It shows the use of 3 gray levels in the output image \{g(m,n)\}. A different number of levels may be used, and after an examination of the output image is made, these intermediate levels may be pushed up to the lightest level or pushed down to zero, if desired.

Boundary construction requires: i) preprocessing to eliminate noise, especially speckle; ii) the connection of points with higher gradients (ridge points); and iii) cleaning up the boundaries (we will use morphological methods for this, to be covered in a later section).

**Algorithm 5.2. Edge Determination by Gradients**

for m = 1 to M do
  for n = 1 to N do
    D(m,n) = gradient_magnitude(); //compute the gradient magnitude
    if D(m,n) > T_2 then g(m,n) = 255; //put output pixel to white
    else if D(m,n) < T_1 then g(m,n) = 0; // or else to black
    else g(m,n) = 150; // or else to gray
  
After the edges have been determined, the problem remains to link up the pieces that occur due to noise and the effects of shading (images should be despeckled before any edge detection/enhancement, boundary linking or segmentation). Such linking is dependent upon the situation and is described in the following subsections.

**Simple Edge Linking.** For low noise and closely spaced pieces and points, the simplest technique of linking is to start with a 5x5 or 7x7 neighborhood of an initial boundary point which can be taken to be the pixel with the greatest gradient magnitude. Other pieces and edges within that neighborhood are checked and the closest one is connected. To avoid too many connections in complex scenes, we can use other properties such as edge direction, strength and gray level to make decisions as to which points to connect. There are numerous properties to use in the design of a linking algorithm.

If no second boundary point exists in the neighborhood of the current boundary point, then we expand to a larger neighborhood of it. For example, if we are using a 7x7 neighborhood of the current boundary point and a search of it fails to locate another boundary point, then we use a 9x9, an 11x11 if need be, and so forth. The connections made in such a manner are straight lines. While this is sometimes useful, in certain cases the actual boundary is curved and we need to fit a curve to the boundary points. These and similar algorithms can become rather complex.

**Directional Edge Linking.** This uses the simple edge linking method, but also uses the directions computed by Equation (5.10) above. The direction \(\Theta(m,n)\) at the current boundary pixel \(f(m,n)\) can be used in addition to the gradient \(|D(m,n)|\) to make the decision of where the next boundary point lies. When two edge points are linked, the average of their directions can be used to interpolate boundary points in between them.
**Polynomial Boundary Fitting.** The above method uses straight line segments to connect boundary points. This can be modified by keeping track of the points and fitting an \( k \)th degree polynomial through \( k+1 \) consecutive edge points. For example, quadratic polynomials can be fit through 3 consecutive points. The quadratic polynomial for the next 3 consecutive points would not fit smoothly with the previous quadratic, however, because their derivatives would be different where they meet. For this reason, splines are useful for fitting boundaries with curves. Cubic splines can be found in any standard textbook on numerical analysis. Consecutive cubics through 4 points that overlap between two points are adjusted so that the slopes of the curves are the same on the overlapping port.

**Simple Boundary Tracking.** Boundary tracking is a process of moving along the boundary and picking the next point on the boundary from its maximum gray level. Starting at a point with a high gradient magnitude, which certainly must be on a boundary, we take a 3x3 neighborhood of that boundary point. We take the pixel in the 3x3 deleted neighborhood (that is, not the center) that has the highest gray level (or the greatest gradient magnitude). If two pixels have a tie then we choose one arbitrarily (at random).

We now proceed iteratively using the previous and current boundary points. The current boundary point is at the center of its 3x3 neighborhood and the previous boundary point is next to it. Moving in the same direction, we check the pixel on the other side of the center from the previous boundary point. If it is a maximum gray level pixel, then it is the next boundary point. If it is not a maximum gray level, but one of its adjacent (noncentral) pixels is, then that pixel is chosen as the next boundary point. If both adjacent pixels are maximal, then one is chosen arbitrarily. Figure 5.7 shows the situation. Noise can send the path off on a nonboundary trajectory.

An alternative method is to use the directional information. Assuming that the actual boundary is changing in a continuous manner, we can smooth the consecutive boundary directions with the more recent one more heavily weighted, i.e., we can use a moving average of the directions to select the next boundary point.

**Boundary Tracking Beetles.** Because simple boundary tracking can lose the boundary, we may want to use a tracking beetle (or tracking bug) to average the gray levels in the directions considered. This smooths off some noise. A thought experiment reveals this method. Suppose that a beetle is walking along the ridge of a noisy boundary and arrives at the current boundary point. The goal is to find the next boundary point (to stay on the ridge) as was done in the simple boundary tracking method. Moving in the same direction that brought it to the current point, the beetle proceeds a few steps and examines the terrain under itself. The beetle has a length and a width in numbers of pixels, for example, of 5 pixels long and 3 pixels wide. The beetle straddles the boundary ridge but has an average tilt to one side or the other due to the averaging of the ridge pixel values underneath it.

The beetle can go in one of three directions from the current boundary point without drastically changing direction (straight, 45° or -45°). The average gradient magnitude of the pixels underneath the beetle are computed for each of these 3 directions as designated in Figure 5.9. For each such determination, the beetle leaves its rear end centered over the current boundary point. The direction that has the highest average
gradient magnitude is selected as the direction of the next boundary point and a new boundary point is selected. This process continues with the new boundary becoming the current boundary point.

The size of the beetle provides the extent of the smoothing of the boundary gradient. A larger beetle gives more smoothing while a smaller one gives a possibly noisier value. However, we need to keep the beetle small enough so the gradient magnitude will have meaning.

**Gradient Boundary Tracing.** An effective method for finding the boundaries of blobs tests the derivative magnitudes $|\partial I/\partial x|$ and $|\partial I/\partial y|$ over a neighborhood centered on the current boundary point. If the maximum of these is sufficiently high, then we make the tests

\[
\begin{align*}
|\partial I/\partial x| &> \alpha|\partial I/\partial y| \\
|\partial I/\partial y| &> \alpha|\partial I/\partial x|
\end{align*}
\]

(5.11a)

(5.11b)

where $\alpha$ is close to 2.0. In the first case we deduce that the next boundary point is in the x-direction, while in the second case we surmise that it is in the y direction. If neither case holds, then we may use a diagonal direction: if the maximum magnitude is high in the diagonal direction then the boundary is in that direction. We can also re-examine the situation with a larger neighborhood of the current boundary point.

### 5.3 Binary Morphological Methods

**Blocks and Morphological Operations.** An operator block of pixels in this section will denote a connected set of pixels and be denoted by $B$. One pixel in each block is designated as the origin. An operation on region $R$ (also a connected set of pixels) of an image is performed with the block $B$ by putting the origin of $B$ over each pixel, in turn, and using the block as a mask in set theoretical operations. This process is called a morphological operation. Figure 5.9 shows some blocks and their origins. Some morphological operations are given in the subsections that follow. We assume here that the image has been reduced to black and white (0 or 1), i.e., the image is binary.

**Figure 5.9. Origins of Blocks.**

![Figure 5.9. Origins of Blocks.](image)

**Figure 5.10. Erosion of R by B.**

![Figure 5.10. Erosion of R by B.](image)

**Erosion.** A region-reducing morphological operation called erosion can be performed on a region $R$ by a block $B$. The operation is denoted by

\[ E = B \circ R \]

(5.12)

The process moves the origin of block $B$ over each pixel $p$ of the image and so of the region $R$, where the translated block is denoted by $B_p$ and assigns the pixel $p$ to the new eroded region $E$ only if $B_p \subseteq R$. Figure 5.10 shows the process. The eroded region $E$ in Figure 5.10 clearly consists of the interior points of $R$ (points inside the boundary of $R$). $E$ is clearly smaller than $R$ in that $E \subseteq R$. 

9
**Dilation.** Dilation is another morphological operation on a region R by a block B. In this case, for each pixel \( p \) in the image the origin of the block \( B_p \) is translated to \( p \) (the origin is placed over \( p \)). The corresponding pixel \( p \) in the output dilated region \( D \) is determined by the rule for each image pixel: \textit{the pixel } \( p \text{ is included in } D \text{ if and only if the block } B_p \text{ intersects } R \), i.e., whenever \( B_p \cap R \neq \emptyset \). Figure 5.11 displays the operation of dilation, which is denoted via

\[
D = B \circ R
\]  

(5.13)

The dilated region \( D \) is seen to be larger than the original region \( R \) in that \( D \supseteq R \). \( D \) includes all of the boundary points of \( R \) in the topological sense that a neighborhood of a boundary point intersects both the region \( R \) and its complement \( \sim R \) in the image (\( \sim R \) is the set of all pixels in the image that do not belong to \( R \)) as described below.

**Figure 5.11. Dilation.**

**Figure 5.12. Boundarizing.**

**Boundarizing.** This is a process of establishing the boundary of a region \( R \) in an image. Figure 5.12 shows the boundarizing process. The origin of the block \( B \) is placed over a pixel \( p \) in the region \( R \) and the pixel is kept in the boundary \( b(R) \) of \( R \) if and only if \( B \) contains pixels both in \( R \) and in the complement \( \sim R \) of \( R \) in the image, i.e., whenever

\[
B \cap R \neq \emptyset \text{ and } B \cap (\sim R) \neq \emptyset
\]  

(5.14)

**Binary Shrinking.** A region \( R \) can be operated on by a block \( B \) to shrink \( R \) by repetitively applying the erosion operation, that is, by

\[
(B \circ (B \circ (B \circ ... (B \circ R)...) )
\]  

(5.15)

A region can be shrunk down to the empty set with sufficiently many erosions. While this is seldom useful, it is often desired to reduce the body of a blob to some underlying thin set of lines. The next subsections cover this.

**Binary Skeletonizing.** The concept of skeletonizing is to construct a 1-pixel thin skeleton of a blob. The concept can be grasped by imagining a large connected section of dry grass. Suppose that the outer perimeter is set afire at the same instant and burns inward at the same rate. The fire lines meet at the skeleton lines. Figure 5.13 shows the situation as we go through the skeletonizing algorithm given below. In the original region \( R \), all of the pixels have value 1 and the background has value of 0. The algorithm follows which uses a function called difference that returns a value of false if any difference is found between the current image and the previous image (or true otherwise). The algorithm consists of two passes: i) construct an image of integer values that represents the distance from the boundary; and ii) in this integer image select the pixels for the skeleton to be those in the boundary distance image for which no adjacent values horizontally or vertically are larger.
Algorithm 5.3. Skeletonizing

\[ k = 0; \]
\[ \text{write original as binary image } \{ f_0(m,n) \}; \]
\[ \text{//first pass: construction} \]
\[ \text{repeat} \]
\[ \text{//repeat the entire outer loop} \]
\[ \text{for } m = 1 \text{ to } M \text{ do} \]
\[ \text{//for every row } m \text{ and column } n \]
\[ \text{for } n = 1 \text{ to } N \text{ do} \]
\[ f_{k+1}(m,n) = f_0(m,n) + \min_{(i,j)} \{ f_0(i,j) : (i-m)^2 + (j-n)^2 \leq 1 \}; \]
\[ k = k+1; \]
\[ \text{//compute new skeleton pixel} \]
\[ \text{compare = difference}(\{ f_k(m,n) \}, \{ f_{k-1}(m,n) \}); \]
\[ \text{//false if difference exists} \]
\[ \text{until compare; \text{//stop if compare = TRUE}} \]
\[ \text{for } m = 1 \text{ to } M \text{ do} \]
\[ \text{//second pass: select skeleton} \]
\[ \text{for } n = 1 \text{ to } N \text{ do} \]
\[ \text{//for each row } m \text{ and column } n \]
\[ \text{for } i = 0 \text{ to } 1 \text{ do} \]
\[ \text{//for each } i \text{ and } j \text{ offsets} \]
\[ \text{for } j = 0 \text{ to } 1 \text{ do} \]
\[ \text{if } (i-m)^2 + (j-n)^2 \leq 1 \text{ and } f_k(m,n) > f_k(i,j) \text{ select = true; //set select to true or false} \]
\[ \text{else} \]
\[ \text{if select = true then } s(m,n) = 1; \]
\[ \text{else} \]
\[ s(m,n) = 0; \]
\[ \text{//s(m,n) is the skeleton \text{//skeleton is binary 0 or 1}} \]

Binary Thinning. A white blob \( R \) in a black and white image is thinned by deleting all boundary points of \( R \) that have more than one 3x3 neighbor in \( R \) unless such a deletion would disconnect \( R \). The basic idea is to erode \( R \) except at end points and repeat until no more deletions can be made without disconnecting \( R \). Figure 5.14 shows a special numbering of pixels in a neighborhood of \( p_0 \), so that the algorithm can be applied when \( p_0 = 1 \) (else skip to the next pixel). The pixels are ordered via \( p_0, p_1, p_2, ..., p_7, p_8 \). We use Zhang* and Suen’s algorithm.

Let \( C(p_0) \) be the number of times there is a change from 0 to 1 (not from 1 to 0) as a cycle is made from \( p_1 \) to \( p_8 \). \( N_s(p_0) \) designates the number of pixels in the neighborhood that are nonzero. Deleting a pixel changes it from \( p_0 = 1 \) to \( p_0 = 0 \). The "NO" or "YES" by each neighborhood indicates the decision of whether or not to delete the pixel \( p_0 \) (convert it from 1 to 0) according to the rules in the algorithm below.


Algorithm 5.4. Binary Thinning

\[ \text{repeat} \]
\[ \text{//repeat this loop until nothing is deleted} \]
\[ \text{deleted = 0;} \]
\[ \text{//boolean, no deleted points yet} \]
\[ \text{for } m = 1 \text{ to } M \text{ do} \]
\[ \text{//for each row } m \text{ and column } n \]
\[ \text{for } n = 1 \text{ to } N \text{ do} \]
\[ p_0 = f(m,n); \]
\[
\text{if } p_0 = 1 \text{ then}
\]
\[
\text{order nbhd;}
\]
\[
\text{get } C(p_0);
\]
\[
\text{get } N_z(p_0);
\]
\[
\text{if } \left( \begin{array}{c}
2 \leq N_z(p_0) \leq 6 \\
\text{and } (C(p_0) = 1) \\
\text{and } (p_1p_3p_5 = 0) \\
\text{and } (p_1p_3p_7 = 0)
\end{array} \right) \]
\[
\text{then } p_1 = 0; \]
\[
\text{change = difference();}
\]
\[
\text{until (change = false);}
\]

Thinning can also be done with blocks and their rotations (see pruning below) by putting the origin over each binary pixel and deleting it if a match is made. Blocks need not be 3x3 in size and need not be rectangles, but one pixel must be denoted as the origin.

**Pruning.** Pruning is a process of removing short lines from the edges of a region. Such short lines are a form of spurious noise. It is often done after thinning to obtain lines, in which case there are often small spikes along the lines. We use the 4 rotations of the two blocks shown in Figure 5.15. This removes the endpoints of short spurs. The origin of a block B is put over each pixel p of R (the translation B_p of B to p at its origin) and if the block B_p matches the pixels of R (the "don't cares" are denoted by "x" and are used as either 0 or a small square to make a match), then the pixel is eliminated. Figure 5.16 presents an example of pruning by applying the various rotations of the masks of Figure 5.15. The 8 rotations of the blocks are applied successively.

**Opening and Closing.** The process of opening a region R of an image consists of two steps in order

a. eroding R

b. dilating R

This has the effect of eliminating small and thin objects, breaking objects at thin points and smoothing boundaries of large regions without changing their area significantly.

Closing a region R consists of the two steps in the following order

a. dilating R

b. eroding R
This process fills in small thin holes in the region, connects nearby blobs and also smooths the boundaries without significantly changing them. Segmented boundaries are often jagged and contain small holes and thin parts, so closing a region enhances it in the sense that it is "cleaned up." The 8 rotations of the two masks are applied successively in a pruning algorithm, but Figure 5.16 shows only the ones that prune a pixel. A pruned pixel is shown as an "o" that represents 0 rather than the small squares that represents 1.

Figure 5.17. Original image.

Figure 5.18. Eroded original.

Figure 5.19. Dilated original.

Figure 5.20. Opened original.

Figure 5.21. Closed original.

Figure 5.22. 4-times closed original.
The original pollens image is shown in Figure 5.17 above. Figures 5.18 and 5.19 show the respective erosion and dilation of the original image. The erosion eats away at the relatively light regions and so yields less light and more dark pixels. On the other hand, dilation enlarges (dilates) the relatively light regions to yield more light and less dark area.

Figures 5.20 and 5.21 show the respective opened and closed images that were obtained by operating on the original image. Figure 5.22 displays the results of closing the original 4 times consecutively. In Figure 5.23 is the result of performing double dilations followed by double erosions and repeating this for a total of 4 times, which is a form of segmentation. Following up on the segmentation, we then blanked out the background with a threshold sufficient to blank out all of the background as shown in the enlarged image of Figure 5.24.

The tool for the grayscale erosion and dilation was *LView Pro*. Figure 5.25 shows the window where we have have loaded in a PGM version of the pollens image (we have a TIF version, but we loaded it in and did a File | Save As to save it as a PGM file and then loaded that in for processing - *LView Pro* requires a PGM file to process). Under the *Color* item on the menu bar is a pop-up menu on which we have selected *Filters*. This brings up another window with a list of filters from which we can select *Erode* or *Dilate* as shown in Figure 5.26.
Figure 5.25. The *LView Pro* menu bar, *Color* selection.

Figure 5.26. The *LView Pro* Pre-defined Filters selection.
**Opening and Closing with Matlab.** We present here an example of using Matlab for opening and closing. It is more complicated to use than LView Pro, but it has a lot more power. We show it here on black and white images, which is a common application.

```matlab
>> I1 = imread('palm.tif'); //read in palm image
>> Ibw1 = im2bw(I1, graythresh(I1)); //convert image to black and white w/threshold
>> imshow(Ibw1), title('Thresholded Image'); //show image with title
>> se = strel('disk',6); //form a structuring element for eroding, dilation
>> Ibw2 = imclose(Ibw1, se); //close the black and white image
>> figure, imshow(Ibw2), title('Closed Image'); //show the resulting closed image with title
>> Ibw3 = imopen(Ibw1, se); //open the black and white image
>> figure, imshow(Ibw3), title('Opened Image'); //show the opened image with title
```

Figure 5.27. The original *palm.tif* image.

Figure 5.28. Thresholded palm image.

Figure 5.29. Opened thresholded palm image.

Figure 5.30. Closed thresholded palm image.

Figure 5.27 shows the original palm image and Figure 5.28 displays its thresholded result. The result was opened and closed with Matlab with the results shown in Figures 5.29 and 5.30, respectively.

**Matlab Examples of Boundaryizing and Skeletonizing.** First we will load in the black and white image *circles.tif* and show it. We use the function `bwmorph()` on the black and white image to remove the interior of blobs by setting them to 0 to leave only the boundaries. Then we shown that image.

```matlab
>> Ibw = imread('circles.tif'); //read in binary image
>> Ibw = bwmorph(Ibw, 'boundary'); //boundaryize image
>> imshow(Ibw), title('Boundarized Image'); //show image with title
>> Ibw = bwmorph(Ibw, 'skel', 1); //skeletonize image
>> imshow(Ibw), title('Skeletonized Image'); //show image with title
```
As an example of skeletonizing, we use the same black and white image and the same function `bwmorph()` but this time we select the morphology string parameter `skel` rather than `remove`. The Matlab code is given below.

```matlab
>> lbw1 = imread('circles.tif'); //read in binary (black and white) image file
>> imshow(lbw1); //show binary file
>> lbw2 = bwmorph(lbw1,'remove'); //boundarize original by removing blob interiors
>> lbw3 = bwmorph(lbw1,'skel',Inf); //skeletonize original
>> figure, imshow(lbw2); //show boundaries
>> figure, imshow(lbw3); //show skeleton
```

Figure 5.31. Original black & white image. Figure 5.32. Boundaries of original image.

Fig. 5.33. Skeleton of original image.

### 5.4 Data Types in Matlab Arrays

**Data Types and Images.** Matlab is an array oriented software package. The arrays usually contain elements of type `double`, but can contain type `complex` instead. Arrays may be indexed as well, which means that the array has entries of data type integer where each integer is a pointer (index, or count) into a file where real values are stored.

Images are arrays that can not contain complex values, but must contain

- `double` - double precision real values from 0.0 to 1.0
- `uint8` - unsigned integers of 8 bits (a byte) for grayscale intensity
**logical** - entry values of 0 or 1 for black and white

**indexed** - a pair of arrays where the first is an array of integer values (indices) with each index giving the offset (pointer, or index) to the entry in a second array.

Image are of type `uint8` for grayscale pixel values of 0 to 255, or type `double` for double precision real values, or type **logical** for black and white, or **rgb** for triples of bytes for pixel values for the respective red, green and blue color intensities at each pixel, or **indexed**. The latter is required for more than 256 colors that is the limit of type **rgb**. For grayscale (**uint8**), a pixel value of 117 at (m,n) means that the pixel at location (m,n) has grayscale intensity value of 117. For **double**, the value at (m,n) would be 117/255 = 0.458823529. For logical the value must at location (m,n) must be 0 or 1.

Now consider an image of type **index**. There is the first array of MxN pixels with an integer at each location (m,n), which we take to be 117 (but it could be much larger than 255). Associated with it is a color map array where at index 117 we find a triple of real values for red, green and blue. Thus we have

<table>
<thead>
<tr>
<th>Index</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td>0.2357</td>
<td>0.5689</td>
<td>0.06295</td>
</tr>
</tbody>
</table>

The index is followed a real value for red, a real value for green and one for blue. This allows for millions of colors because the integer valued index can be very large ($2^{12}$).

**Conversion of Data Types in Images**. **Matlab** contains several functions for converting images from one type to another. This is necessary at times because some functions work only on certain data types. For example, we can not subtract two black and white (logical) images - an error will occur.

```matlab
>> Iout = imsubtract(Ibw1, Ibw2); %this gives an error for black and white images
>> Ibwout = Ibw1 + Ibw2; %does this give an error?
>> Iout = double(Ibw1) - double(Ibw2); %this works for black and white images
```

We can also save the black and white images by exporting them as TIF files and then using the function `imsubtract()` on the TIF versions. Some conversion functions are:

- `im2bw()` converts `uint8` image to logical (black and white)
- `rgb2gray()` converts `rgb` image to `uint8`
- `rgb2ind()` converts `rgb` image to indexed image
- `ind2rgb()` converts indexed image to `rgb`
- `gray2ind()` converts `uint8` image to indexed image
- `ind2gray()` converts from indexed image to `uint8`

**Exercises 5**

5.1 Imagine the process of being a microbug located at a home base pixel. Develop an algorithm whereby the microbug searches in all directions from the home base for pixels that are similar to the home based one according to some property (gray level, gradient magnitude, etc.). Write the overall algorithm where this is applied to each pixel in the image and values written to an output image g(m,n) of particular shades of gray that is segmented.

5.2 Adapt a copy of the program mdip3.c to process an image by computing the weighted fuzzy expected value on a 3x3 neighborhood of each pixel. Use XV to crop a part of "shuttle.pgm" and then use this program to process it.
5.3 Write an algorithm that uses the Roberts edge operator to construct an edge image and then to link the edges. Put in each step so the algorithm can be programmed.

5.4 Write and run a program to implement the algorithm developed in Exercise 5.3 above.

5.5 Show that the boundary of the region R of Figure 5.13 can be extracted by the morphological process

\[ b(R) = R \sim (B \circ R) \]

Note that this is the complement with respect to R of the erosion of R.

5.6 Show that thinning can be done on the region R of Figure 5.13 by processing it with all 8 rotations of the two blocks

\[
\begin{array}{ccc}
0 & 0 & 0 \\
x & 1 & x \\
1 & 1 & 1
\end{array}
\quad \quad \quad
\begin{array}{ccc}
0 & 0 & x \\
x & 1 & 0 \\
1 & 1 & x
\end{array}
\]

where the origins are the center pixels and "x" designates a "don't care." Recall that a pixel \( p \) in the region R is eliminated if the block B placed with origin over that pixel to form \( B_p \) makes a match of 0's and 1's in the image.

5.7 Apply the pruning mask operations to the following region, assuming that it is surrounded by 0's.

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

5.8 Write an algorithm similar to the thinning algorithm that prunes regions of spurs on all sides.