Unit 5. Geometrical Processes I: Morphology

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 similary 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 graylevel only, or more complex such as graylevel, 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 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 are computationally expensive and some may become unstable during processing.

Morphology: the boundaries of regions are dilated 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. Special types of clustering can be very powerful.

Other: a variety of other methods and techniques for implementing the methods are used such as *template matching*, *texture segmentation, neural network training*, *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.

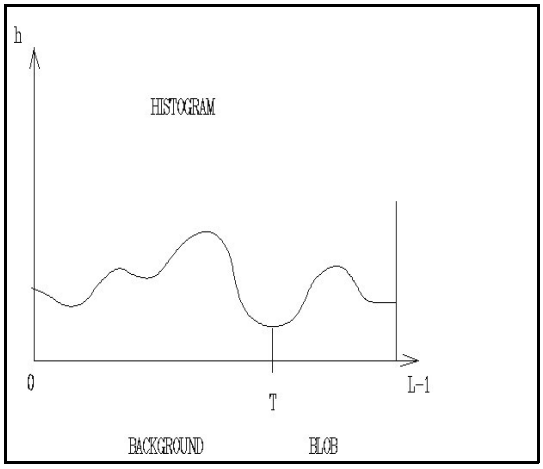
It is useful to zero out the pixels below a threshold just above the background level and put everything above that at 1 and save as an image. This new binary image $\{f_{bin}(m.n)\}$ is a logic mask for multiplying the

original image $\{f_{orig}(m.n)\}$ by this binary image to zero out the background in the original image while keeping the blobs in their original form. Thus we put

$$g(m,n) = f_{orig}(m,n) \cdot f_{bin}(m,n) = f_{orig}(m,n), \quad \text{if binary value is 1} \tag{5.1a}$$

$$g(m,n) = f_{orig}(m,n) \cdot f_{bin}(m,n) = 0, \quad \text{if binary value is 0} \tag{5.1b}$$

Figure 5.1. Finding a threshold.



One or more thresholds can be determined by looking at the histogram and finding the points at local minima where the pixels in the blob become increasingly more numerous as the shade of gray changes. 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.

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. It is not easy to better this even with more complex methods.

Figure 5.2. Pollens.

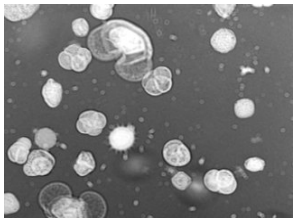


Fig. 5.3. Cropped enlarged region of pollens.

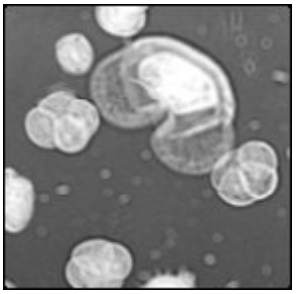


Figure 5.4. Despeckled cropped pollens.

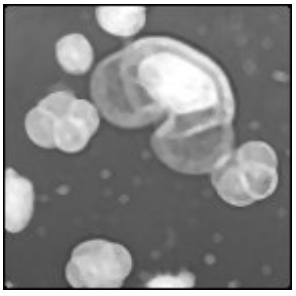
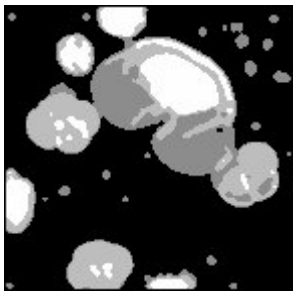


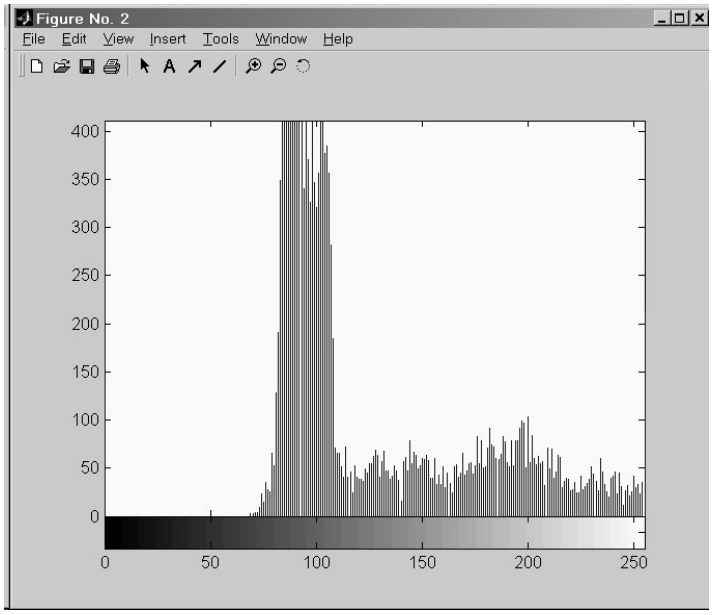
Figure 5.5. Segmented cropped pollens.



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 examing 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 and also that we need to use histogram equalization.

```
>> 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 ith 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 also use the Matlab function *T = graythresh(Im1)* function and then *Im2 = im2bw(Im1,T)* (converts to black and white).

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. Blocks of similar gray level can be joined to make larger regions. Region growing methods are usually computationally expensive and some can become unstable.

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.

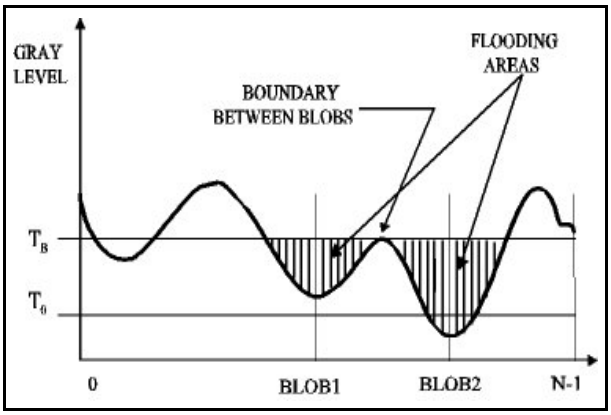
Interactive processing is where the user selects a point in a blob and an algorithm then starts at that point and grows a region by including all adjacent pixels whose gray levels are close to those of the selected pixels. Interactive methods can be very powerful because they rely on the extra knowledge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

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 over 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 $f(m\pm i,n\pm j)$, $0 \le i,j \le 1$, 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 $\{f(m,n): 0 \le n \le N-1\}$ and graph the inverted grayscale values 255 - f(m,n). We would not invert if the blobs were darker than the background. A threshold T_0 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_B 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.

Figure 5.7. Flooding (watershed).



***An Example: Watershed (Flooding) 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
>> Iwat = watershed(Iimpose);                //watershed on the valleys.
>> Irgb = label2rgb(Iwat);                   //color in segments for visualization
>> figure, imshow(Irgb);                     //display the watershed segmentation
```

Figure 5.8. Cropped original. **Figure 5.9. Contrasted image.** **Figure 5.10. Complement image**.

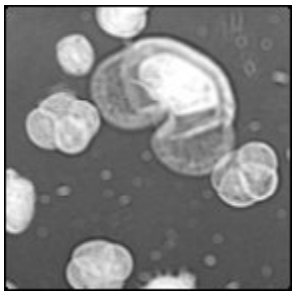
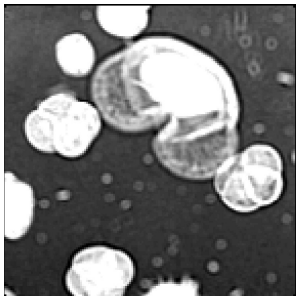
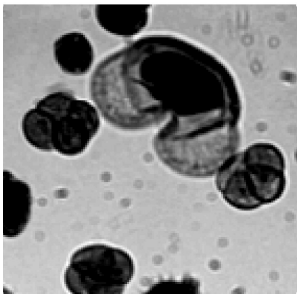
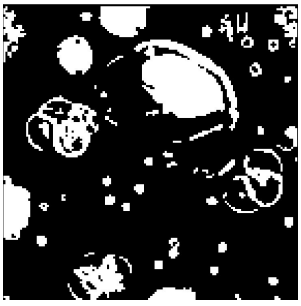
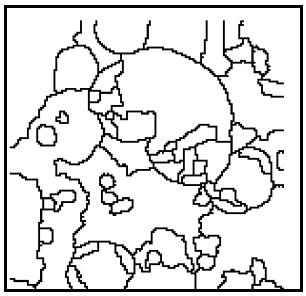




Fig. 5.11. Exaggerated gaps. **Figure 5.12. Watershed boundaries.** **Figure 5.13. Color segments**.


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 (afterwards we can merge regions that are similar). Examination of the histogram can provide a good value for a small number of 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 (256/32 = 8 and we choose midpoints of each interval of gray levels)

$$g_k = 4 + 8k,\ k = 0,..., 31 \tag{5.2}$$

Each pixel is tested to determine the cluster center (prototype) gray level 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 by deleting their prototype gray level. Then we take the average gray level of each cluster to be the new cluster prototype gray level. The remaining K clusters form regions, although they are not necessarily connected. At this point a merging process can be applied for any remaining clusters whose prototypes are very close (for this reason we need more clusters with prototype gray levels close to each other). Algorithm 5.1 does basic clustering.

Algorithm 5.1. Clustering into Regions

```
Step 1
    input K0;                          //initial number of clusters to be tried
    g = 256/K0;                        //get increment gray value
    for k = 0 to K0-1 do               //for each cluster center index
        g[k] = g/2 + g*k;              // compute the center (prototype) gray level
Step 2
    for m = 0 to M-1 do                //for all rows in image and
        for n = 0 to N-1 do            // for all columns
                dmin = 0.0;            // get minimum distance to a prototype
                for k = 0 to K0 do     // over all K0 prototypes
```

```
                d[k] = |f[m,n] - g[k]|      // by computing distance
                 if dmin > d[k] then        // and comparing min. distance with each distance
                      dmin = d[k];          //update current mininm distance
                      kmin = k;             // and record its index
              g[m,n] = g[kmin];             //finally, output the pixel at the winning gray level
```

Step 3

do until clusters do not change
eliminate clusters with no pixels
for each cluster find the average gray level of that cluster as its center gray level
repeat Step 2 above to assign pixels to clusters

Step 4

merge clusters whose gray levels are close (user selected closeness)

Segmentation with Weighted Fuzzy Expected Values. The *weighted fuzzy expected value* (WFEV), denoted by μ_F, 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 μ_F on a 3x3 neighborhood of p_5

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_5 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

$$w_i = \exp[(p_i - \mu_F)^2/(2\sigma^2)] \tag{5.3a}$$

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

$$\alpha_i = w_i/\sum_{(i=1,9)}w_i \quad (\sum_{(i=1,9)}\alpha_i = 1) \tag{5.3b}$$

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

$$\mu_F = \sum_{(i=1,9)}\alpha_i p_i = \sum_{(i=1,9)}w_i p_i \,/\, \sum_{(i=1,9)}w_i \tag{5.4}$$

The weighting is greatest for the pixels p_i that are the closest to μ_F (see Equation (5.3a)). However, μ_F is a function of μ_F, 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 or 5) iterations to be sufficiently accurate. Thus we compute

$$\mu_F^{(0)} = (1/9)\sum_{(i=1,9)}p_i \tag{5.5}$$

$$\vdots \qquad \vdots$$

$$\mu_F^{(r+1)} = \sum_{(i=1,9)}\{\exp[(p_i - \mu_F^{(r)})^2/(2s^2)]p_i\} \,/\, \sum_{(i=1,9)}w_i \tag{5.6}$$

After a pass through the image the result is a set of nbhds of a single gray level, which can then be merged into larger regions of similar gray level. The average on the region is then computed.

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

$$|Df(m,n)| = \{(\partial f/\partial x)^2 + (\partial f/\partial y)^2\}^{1/2} \quad (5.8)$$

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

$$|Df(m,n)| = \max\{|f(m,n) - f(m+1,n)|, |f(m,n) - f(m,n+1)|\} \quad (5.9a)$$

Another popular gradient edge detector is the *Roberts edge operator* given by

$$g(m,n) = \{[(f(m,n))^{1/2} - (f(m+1,n+1))^{1/2}]^2 + [(f(m+1,n))^{1/2} - (f(m,n+1))^{1/2}]^2\}^{1/2} \quad (5.9b)$$

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

$$\theta(m,n) = \arctan([\partial f/\partial x]/[\partial f/\partial y]) \quad (5.10)$$

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) > T2        then g(m,n) = 255;    //put output pixel to white
    else if D(m,n) ≤ T1   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 interporlate 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 kth 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 delected 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.

Figure 5.8. Simple Boundary Tracking.

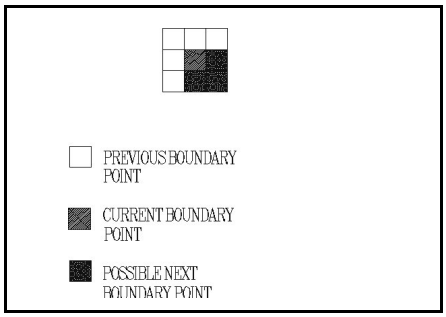
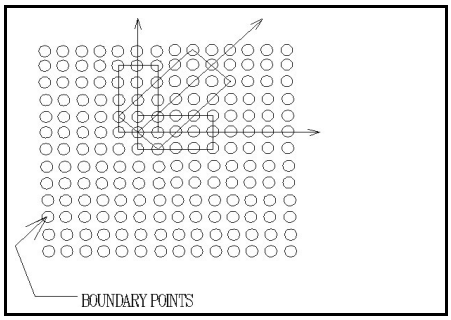


Figure 5.9. Boundary Tracking Beetle.



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 $|\partial f/\partial x|$ and $|\partial f/\partial y|$ over a neighborhood centered on the current boundary point. If the maximum of these is sufficiently high, then we make the tests

$$|\partial f/\partial x| > \alpha|\partial f/\partial y| \tag{5.11a}$$

$$|\partial f/\partial y| > \alpha|\partial f/\partial x| \tag{5.11b}$$

where α 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 situtation 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.

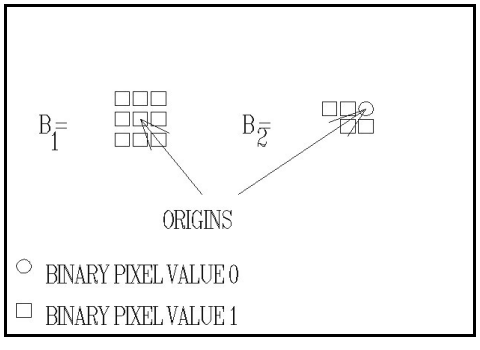
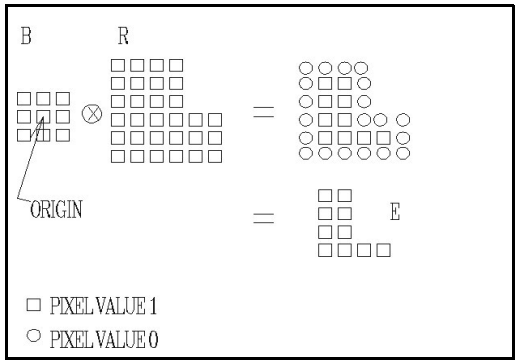


Figure 5.10. Erosion of R by B.



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

$$E = B \otimes R \tag{5.12}$$

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 $E \subseteq R$.

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: *the pixel p is included in D if and only if the block* B_p *intersects R*, i.e., whenever $B_p \cap R \neq \varnothing$. Figure 5.11 displays the operation of dilation, which is denoted via

$$D = B \oplus R \tag{5.13}$$

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* ~R in the image (~R is the set of all pixels in the image that do not belong to R) as described below.

Figure 5.11. Dilation of R to D.

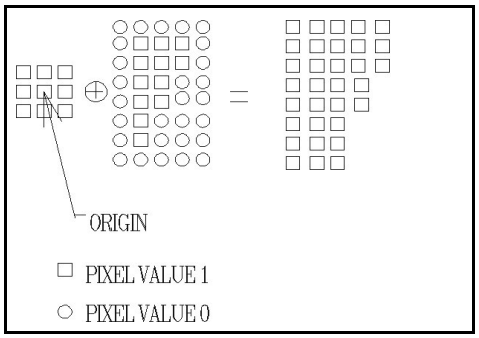
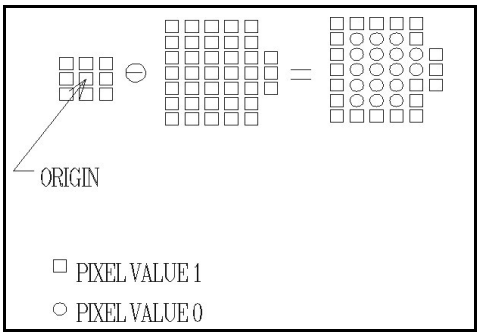


Figure 5.12. Boundarizing R.



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 ~R of R* in the image, i.e., whenever

$$B \cap R \neq \varnothing \text{ and } B \cap (\sim R) \neq \varnothing \tag{5.14}$$

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

$$(B \otimes (B \otimes (B \otimes ....(B \otimes R)..) \tag{5.15}$$

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;                                                          //first pass: construction
write original as binary image {f0(m,n)}                        //write output image as black and white
repeat                                                          //repeat the entire outer loop
    for m = 1 to M do                                           //for every row m and
       for n = 1 to N do                                        // for every column n
          fk+1(m,n) = f0(m,n) + min i,j{fk(i,j): (i-m)^2 + (j-n)^2 ≤ 1};//compute new skeleton pixel
   k = k+1;                                                     //increment count k
   compare = difference({fk(m,n)}, {fk-1(m,n)});                //false if difference exists
until compare;                                                  //stop if compare = TRUE
for m = 1 to M do                                               //second pass: select skeleton
   for n = 1 to N do                                            //for each row m and column n
    for i = 0 to 1 do                                           // for each i and j offsets
       for j = 0 to 1 do
            if (i-m)^2 + (j-n)^2 ≤ 1 and fk(m,n) > fk(i,j) select = true; //set select to true or false
            else                                                select = false;
    if select = true then   s(m,n) = 1;                         //s(m,n) is the skeleton
    else                    s(m,n) - 0;                         //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. $N_Z(p_0)$ 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.

* T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Comm. ACM*, vol. 27, no. 3, 236-239, 1984.

Figure 5.13. Skeletonizing.

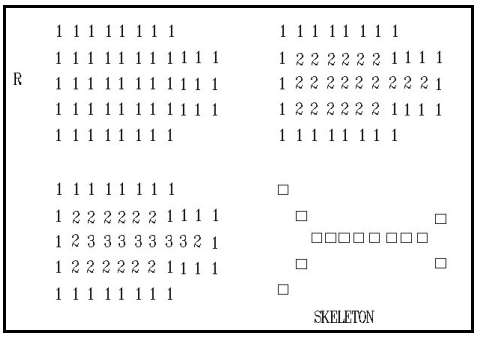
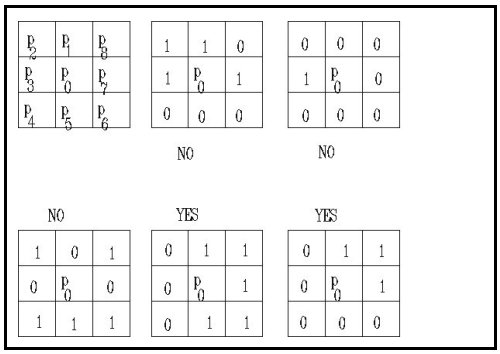


Figure 5.14. Thinning.



Algorithm 5.4. Binary Thinning

```
repeat                                          //repeat this loop until nothing is deleted
  deleted = 0;                                  //boolean, no deleted points yet
  for m = 1 to M do
    for n = 1 to N do                           //for each row m and column n
      p0 = f(m,n);                              //initialize the next pixel for testing
      if p0 = 1 then                            //if pixel is 1 then check for deletion (thinning)
          order nbhd;                           //examine nbhd, order pixels
          get C(p0);                            //compute value
          get NZ(p0);                           //compute value
          if ( (2 ≤ NZ(p0) ≤ 6)                 //in nonzeros number 2 to 6
              and (C(p0) = 1)                   //or the number of changes from 0 to 1 is 1
              and (p1p3p5 = 0)                  //or this product is 0
              and (p3p5p7 = 0))                 //or this product is 0
          then p1 = 0;                          //then delete the white point (set to black)
  change = difference();                        //true if image changed, else false
until (change = false);                         //stope if there is no change
```

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 1 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.

Figure 5.15. Blocks for Pruning.

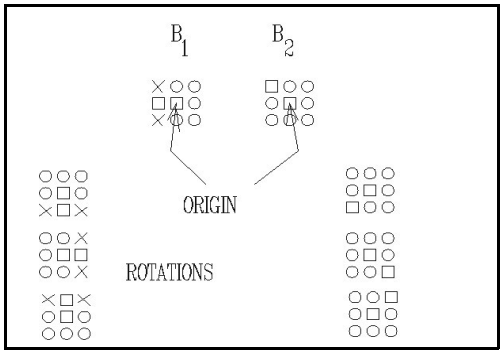
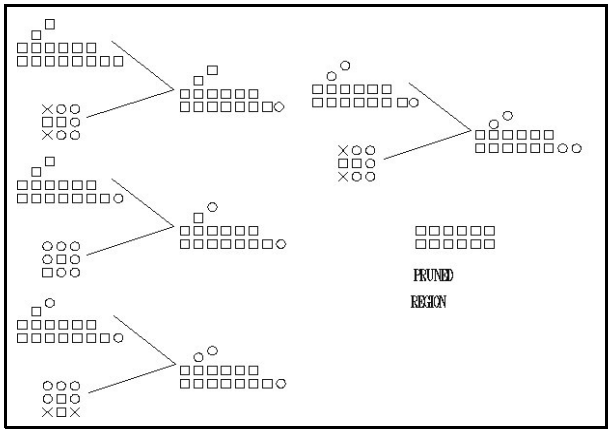


Figure 5.16. The Pruning Process.



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.

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." *Matlab* performs erosion, dilation, opening and closing on binary (0 and 1) images. However, *Lview Pro* performs these on grayscale images as well, which is what we use on the images below.

Figure 5.17. Original image.

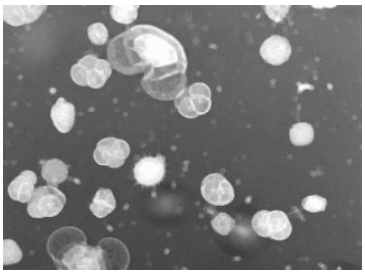


Figure 5.18. Eroded original.

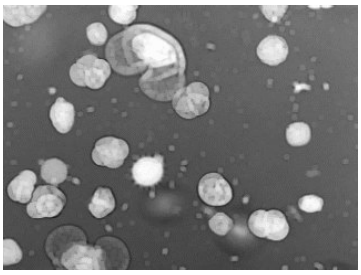


Figure 5.19. Dilated original.

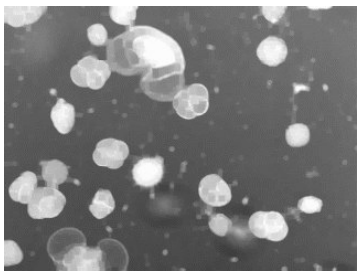


Figure 5.20. Opened original.

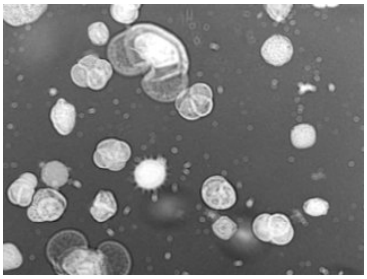


Figure 5.21. Closed original.

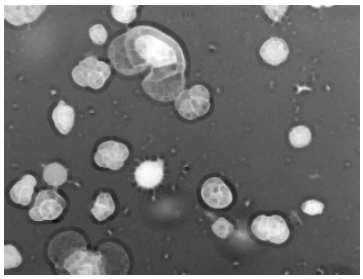


Figure 5.22. 4-times closed original.

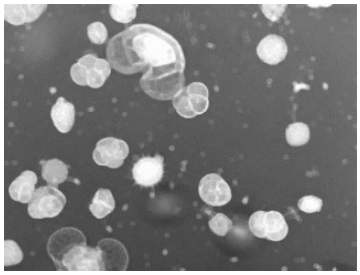


Fig. 5.23. 4-times: 2-dilations, 2-erosions.

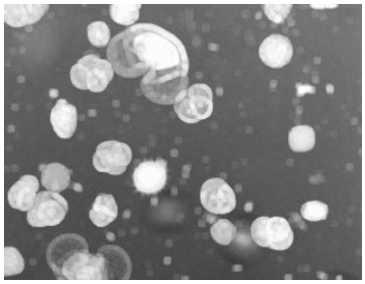
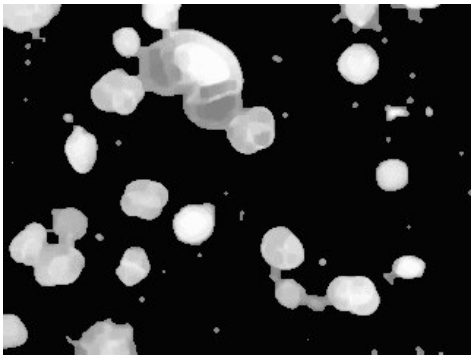


Figure 5.24. Multiple closures, blanking, enlarged.



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.

***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 is has a lot more power. We show it here on black and white images, which is a common application.

```
>> 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);                             //structuring element/block 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.

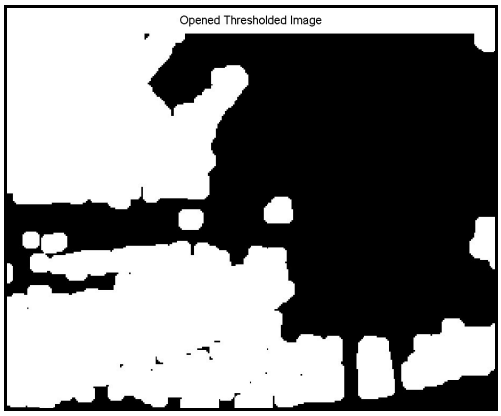


Figure 5.30. Closed thresholded palm image.

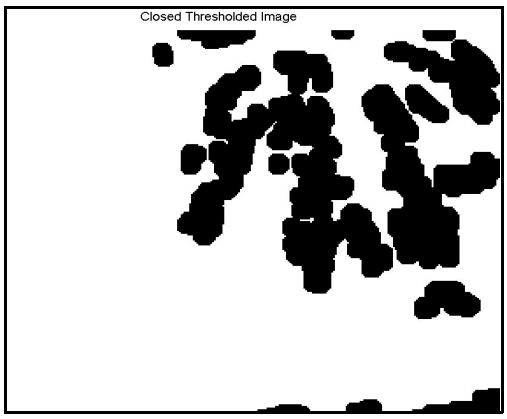


Figure 5.27 shows the original palm image and Figure 5.28 displays its thresholded result. The result was opened and closed with Matlab with the results shown in Figures 5.29 and 5.30, respectively.

Matlab *Examples of Boundarizing 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.

As an example of skeletonizing, we use the same black and white image *circles.tif* from *Matlab* and the same function *bwmorph()* but this time we select the morphology string parameter *skel* rather than *remove*. The *Matlab* code is given below.

```
>> Ibw1 = imread('circles.tif');          //read in binary (black and white) image file
>> imshow(Ibw1);                          //show binary file
>> Ibw2 = bwmorph(Ibw1,'remove');         //boundarize original by removing blob interiors
>> Ibw3 = bwmorph(Ibw1,'skel',Inf);       //skeletonize original
>> figure, imshow(Ibw2);                  //show boundaries
>> figure, imshow(Ibw3);                  //show skeleton
```

Figure 5.31. Original black & white image.

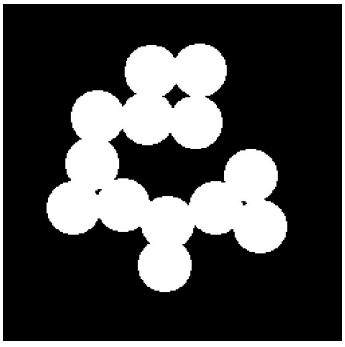


Figure 5.32. Boundaries of original image.

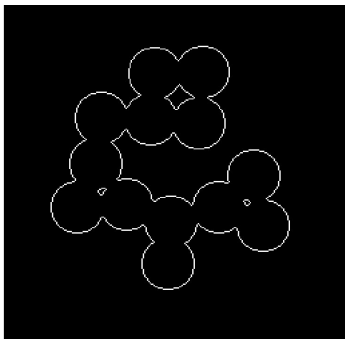
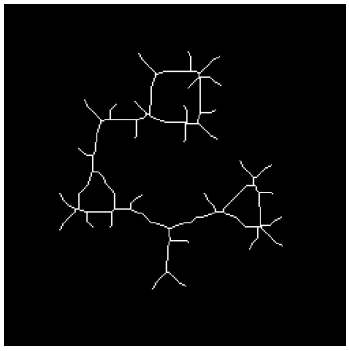


Fig. 5.33. Skeleton of original image.



The *structure element* we used above was a circular one 6 pixels across, but we can also use a square of r pixels on each side. Consider

```
>> Im1 = imread('circles.tif');          //read in image
>> se   = strel('square', 5);            //create square element 5 pixels on each side
>> Im2 = imclose(Im1, se);               //close Im1 using block element se
>> Im3 = imopen(Im1,se);                 //open Im1 using block element se
>> imshow(Im1), title('Original Image');
>> figure, imshow(Im2), title('Closed Image');
>> figure, imshow(Im3), title('Opened Imge');
```

It is also useful to first dilate an original image, then erode the original image, and then subtract the eroded image from the dilated image to leave a strong boundary. But first it may be better to close the original image and use that as the original (it will have cleaner boundaries).\

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 special array where real values are stored.

Images are arrays that can not contain complex values, but must contain values of one of the following types.

double - double precision real values from 0.0 to 1.0

uint8 - unsigned integers of 8 bits (a byte)

uint16 - unsigned integers of 16 bits (2 bytes)

Additionally, the images may be

intensity images - the values are numbers of type double, uint8, or uint16 (Matlab can not process uint16 values so they must be converted to double or uint8 first)

binary images - each pixel has a value of 0 or 1 respectively for black and white and can be stored as value types uint8 or double (not uint16), but uint8 is the preferred data type

rgb (truecolor) images - these are now *truecolor* images that are stored as MxNx3 arrays of 8-bit

values (bytes), so that at pixel location (m,n) there would be 3 bytes stored with a byte for each of red, green and blue (2^{24} or 16 million colors). The values can be of type double (0.0 to 1.0), uint8 (0 to 255), or uint16 (0 to 65,535). The pixel color at (m,n) would be stored at the array positions (m,n,1), (m,n,2), (m,n,3) and each would be a byte for uint8 data type.

indexed images - these consist of a 2-dimensional array of indices of values that can be uint8, uint16, or double. For example an array of indices could be

```
. . . . . .
. . . . . .
. . . . . 17 20 18 49 84 82 82 82 112 112 79 . . . . . .
. . . . . .9   9    8 43 81 83 82 81 114 113 81 . . . . . .
. . . . . .
. . . . . .
```

where each uint8 integer is an index into a one dimensional array of color triples called a *colormap*. The colormap is an m *x* 3 array of m triples with each triple represent a red, a green and an blue value of double values (0.0 to 1.0).

Now consider a 320 *x* 240 image of type *index*. There is the index data array of MxN pixels with an integer at each location (m,n) as shown above. Let the index value be 117 (but it could be much larger than 255). Associated with it is a colormap array where at index 117 we find a triple of real values for red, green and blue. Thus we have

Index		Red	Green	Blue
:				
117	⇒	0.2357	0.5689	0.06295
:				

The indexed triple of real values are red, green and blue. This allows for millions of colors because the integer valued index can be very large (2^{16} or larger for double values).

binary images - each pixel value is a 0 or 1 stored as data type uint8 or double.

Conversion of Data Types in Images. *Matla*b 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.

```
>> 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 intensity, rgb or indexed image to binary (black and white)
rgb2gray()	converts rgb image to grayscale intensity image
rgb2ind()	converts rgb image to indexed image
ind2rgb()	converts indexed image to rgb image
gray2ind()	converts grayscale intensity image to indexed image
ind2gray()	converts from indexed image to grayscale intensity image

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

$$b(R) = R \sim (B \otimes R)$$

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

```
0 0 0        x 0 0
x 1 x        1 1 0
1 1 1        1 1 x
```

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.

```
                  1
     1    1      1
1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

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