NISTIR 7750

Biological Cell Feature Identification by a Modified Watershed-Merging Algorithm

David E. Gilsinn
Kiran Bhadriraju
John T. Elliott

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7750

Biological Cell Feature Identification by a Modified Watershed-Merging Algorithm

David E. Gilsinn
Applied and Computational Mathematics Division
Information Technology Laboratory

Kiran Bhadriraju
John T. Elliott
Biochemical Science Division
Material Measurement Laboratory

November, 2010

U.S. Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Director

Biological Cell Feature Identification by a Modified Watershed-Merging Algorithm

David E. Gilsinn*
Kiran Bhadriraju †
John T. Elliott, ‡

December 3, 2010

Abstract

Biological cells are composed of many subsystems and organelles. The subsystem called the cytoskeleton is composed of long rod-shaped filaments. They give the cell form and help attach the cell to the substrate and neighbors. One type of the filaments is made up of a protein called actin. In studying cells biologists use microscopes that can be focused at different levels and can be automated to take multiple images. Various stain treatments are used to bring out various cell characteristics. In this study a computational method is studied that automatically isolates the actin structures in cells. The algorithm begins by segmenting the cell with a scheme called a watershed. Due to noise and a large number of local minima the number of resulting segmented regions can be large and not informative. The object then is to merge regions, based on nearness and common properties, into regions called features. The merging approach used here involves a graph representation called an adjacency graph and a search algorithm that finds the connected components in the graph. These connected components become the merged features.

Keywords: actin fibers; biological cells; depth-first search; merging; undirected graph; watershed.

1 Introduction

Biological cells are composed of many subsystems, called organelles. The subsystem of most interest in this work is called the cytoskeleton, which is composed of long rod-shaped molecules, or filaments, that are attached to each other and to other organelles. They give the cell form and help attach the cell to the substrate and neighbors. The cytoskeleton consists of three types of filaments: actin filaments, intermediate filaments, and microtubules. For a more complete discussion of the cell cytoskeleton and its structure see Bao and Suresh [2] and Ethier and Simmons [3], pp. 23-29. Graphically, actin filaments can be visually identified in specially stained cell images. However, to automatically identify cell features, such as actin filaments, is not straightforward. Modern microscopic systems can scan more cell samples than individuals can visually examine to compare, for

*Applied and Computational Mathematics Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg MD 20899-8910, david.gilsinn@nist.gov.

†Biochemical Science Division, Material Measurement Laboratory, National Institute of Standards and Technology, Gaithersburg MD 20899-8313, kiran.bhadriraju@nist.gov.

‡Biochemical Science Division, Material Measurement Laboratory, National Institute of Standards and Technology, Gaithersburg MD 20899-8313, john.elliott@nist.gov

example, cell lines so it is important that computational methods be developed that locate features like actin filaments so that changes done to actin shape within the cell edges can be automatically identified.

This work develops an initial approach to locating features in a cell image. It combines a method of image segmentation into what will be called regions, with an algorithm to merge regions into more defineable cell features. The algorithm developed is not the only approach to cell feature identification, but appears to be an initial workable solution to automatic cell feature identification. Further work is clearly needed in the future.

The feature identification process involves a multi-stage algorithm. The basic idea is to first apply an image segmentation algorithm, in the current study called watershedding, that provides as output a matrix the size of the image, called a label matrix. The matrix elements are uniquely numbered so that all the associated image pixels in a region are given a unique number identifying the region. Thus, for example, all of the pixels in region 3 are identified with 3 in the associated elements of the label matrix. The segmentation procedure usually generates a large number of regions because of all of the local minima and noise in the image. The idea behind the second stage of the algorithm is to merge regions with similar properties in order to isolate features. To do this, a measure is defined that is used to link regions with similar properties. In mathematical graph theory this measure is used to form what is called an *adjacency graph* where the graph nodes are the numbered regions and the link weights are the measures of adjacency. For a basic introduction to graph theory see Ore [8]. Once an adjacency graph is developed a tolerance is specified that tells which graph links are critical. It leads to a reduced graph. Not all nodes are connected by a path in the graph to every other node. Thus, given a node, a search of the graph is made for all the nodes linked by a path to that node. These node groups are called *connected components* and they ultimately are identified as the *features*. Once the connected components of the graph are identified, the node regions can then be merged by combining region pixels. Finally, a new label marix is developed that is used by a boundary tracing algorthm to identify the feature boundaries. This feature label matrix could also be used to assist in post-processing feature analysis. This report, however, will concentrate on a description of the implementation of the merging algorithm in MATLAB.

Combining watershed segmentation with region merging is not a new idea. See, for example, Haris et al. [6] and Pires et al. [9]. What is missing from many of these other studies, however, are details about code implementation. The current study is an attempt to fill the implementation gap and provide a detailed description of a code implementation using MATLAB. It is hoped that this will provide a basis for future extensions or constructive modifications of the algorithm and computer code.

It is assumed that the reader is familar with basic image processing terminology and is proficient with MATLAB. A working knowledge of the MATLAB Image Processsing Toolbox and some graph theory is also helpful.

The report is structured as follows. Each section will describe in detail a portion of the Watershed-Merge program given in Appendix A. Section 2 presents the order in which functions are called in the program. Section 3 describes the image contrast adjustment by histogram equalization, edge sharpening of the image by filtering, and finally the initial segmentation of the image by watershedding into regions. Section 4 describes the region formation in terms of a data type, called a structure, the formation of the region boundary lists, creating the region adjacency structure, and identifying of the closest adjacent neighbors. The section then introduces a graph theoretic formation of the adjacency structure that simplifies the feature and feature boundary formation. Section 5 presents images of the results, with features outlined along with a table of merging and timing results. Section 6 describes areas for future work. Section 7 acknowledges the image sources and Section 8 is the formal disclaimer of the use of commercial software.

2 Main Watershed-Merge Function

This function organizes the overall sequence of function calls. It calls each of the functions **Pre_processor, Watershed, Merge, feature_boundary, and Post_processor** in turn. The term function in MATLAB may be referred to in other programming languages as subroutines. Defining the main function as a function type allows all of the other functions to be included in the same file and allows for program portability.

3 Pre-processing an Image

Segmentation algorithms can tend to oversegment an image into a large number of regions that make the process of joining similar regions to form a common feature difficult and time consuming. Depending on the number of oversegmented regions one can run into out-of-memory errors or overrun allocated arrays. It is therefore important in the pre-processing of an image to obtain as few noise-based regions as possible. In this section we describe a modified watershed approach that produces an initial set of segmented regions and allows a merging algorithm to determine potential cell features in a reasonable amount of compute time.

3.1 Histogram Equalization

An image can be considered as a large matrix, $I(i,j)$, with M rows and N columns, where the element values, or pixels for picture elements, can vary from 0 to 255 for 8-bit images or 0 to 65535 for 16-bit images. Let $n_k,\ k = 0, \cdots, L-1$, where L is 255 for 8-bit images or 65535 for 16-bit images, be the number of pixels having intensity level k. If the range of intensity values is divided into intervals or 'bins' then a plot of the numbers of pixel values falling within the bin intervals is called a *histogram*. It shows the distribution of the intensity values in the image. In particular, a histogram is a discrete function $h(r_k) = n_k$ where r_k is the k-th intensity gray scale value and n_k is the number of pixels in the image with intensity value r_k. A normalized histogram is given by a function $p(r_k) = n_k/MN$ for $k = 0, \cdots, L-1$. $p(r_k)$ can be thought of as an estimate of the probability of occurence of the intensity value r_k.

The distribution of pixel intensities in an image can vary depending on the image acquisition system. If the pixel intensities are concentrated in the lower values the image shows up as 'dark', and if they are concentrated in the upper values it shows up as 'light'. The object of *histogram equalization* is to increase the intensity range as much as possible over the range of intensities. Figure 1 looks totally black because all of the figure intensities are concentrated in the low range. This is clear from the histogram of the intensities shown in Fig. 2. Notice the distribution of the intensities in the dark region of the gray scale color bar at the bottom of the figure. This accounts for the black image. The image is intended to be one of a biological cell, but, due to lack of contrast, it does not show in the image. The initial image used in this study is referenced as **cell1_MLCPPandfactin_red_0075**.

Histogram equalization is a process that modifies the histogram of an image in order to achieve a better overall contrast. It is shown in Gonzalez and Woods [4], pp. 122-128, that the histogram equalization of the histogram intensities can be given by

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j, \quad k = 0, \cdots, L-1, \tag{1}$$

Figure 1: Raw Figure without Histogram Equalization. Intensities are Distributed in the Low Range.

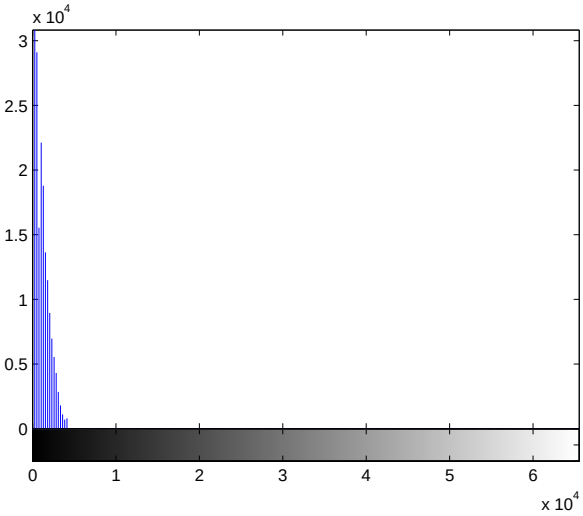

Figure 2: Histogram of the Raw Figure without Histogram Equalization. The corresponding intensities are shown as gray levels at the bottom of the figure.

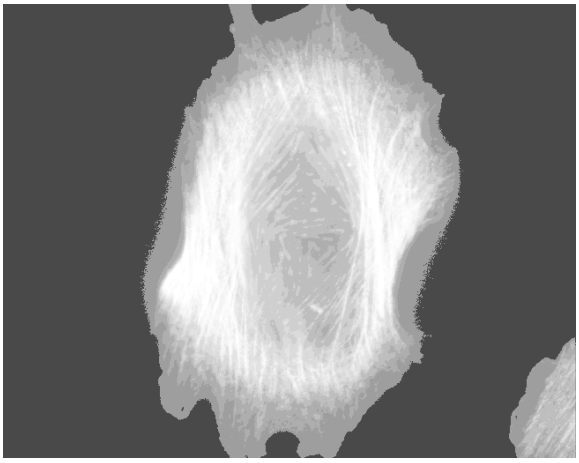

Figure 3: Raw Figure with Histogram Equalization. Intensities Redistributed across the Range.

where $n = n_0 + n_1 + \cdots + n_{L-1}$ is the total number of pixels and s_k is the redistributed number of pixels with intensities equal to r_k in the bins. The default number of bins used in MATLAB is taken to be 256. Figure 3 shows an image of the cell after histogram equalization. Note the redistribution of the pixel intensities towards the lighter side of the range in Fig. 4. If **I0** is the initial image then in MATLAB histogram equalization can be performed by the call **I1 = histeq(I0)**, where **I1** is the resulting image. It can be displayed by a call to **imshow(I1)** if desired.

In the next section we will sharpen the fibers a bit.

3.2 Image Filtering and Feature Sharpening

Although histogram equalization leads to an image with a wider distribution of intensities it sometimes introduces some blurring of edges. There is definite edge blurring of the cytoskeleton fibers in Fig. 3 as a result of the histogram equalization. Image filters can be applied to sharpen the edges. In particular, we apply a filter called an *unsharp mask*. Unsharp masking is a method of edge enhancement in an image that involves subtracting a scaled blurred version of the original image from the original image. This has the affect of enhancing the edges in the image. It can be implemented by subtracting a 3×3 blurring mask from a scaled 3×3 mask with a one in the center and zeros elsewhere. This can be done in MATLAB with a call to the function **fspecial** followed by a call to **imfilter**. It is illustrated below, where **h** is the resulting mask and **0.2** is the scale factor for the mask. This scale factor was found to be optimum in McAndrew [7], pp. 105-107. The call **imfilter** applies the mask to the image. For a brief discussion of unsharp masking see McAndrew [7]. The image boundary is also padded with zeros so that the application of other image filters will not cause errors at the boundaries. This is done with the MATLAB function **padarray**.

```
%unsharp masking is edge crisping by subtracting a scaled blurred version
%of the image from the original
h = fspecial('unsharp',0.2);
I2 = imfilter(I1,h);
```

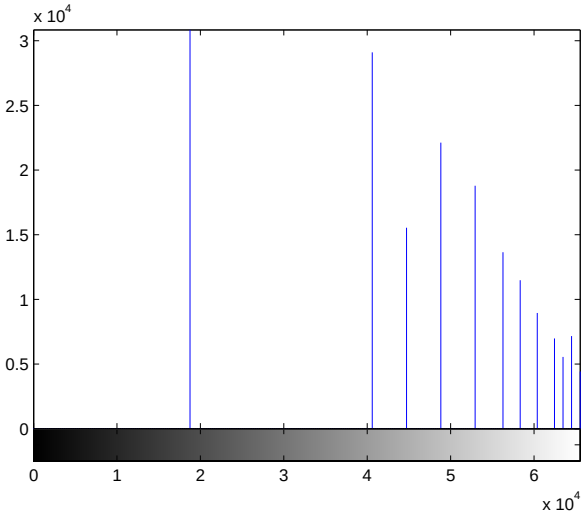

Figure 4: Histogram of the Raw Figure with Histogram Equalization.

```
%Display the sharpened image
figure, imshow(I2),impixelinfo;
%Pad the array boundary with zeros
I2 = padarray(I2,[1 1],0,'both');
```

An example of an unsharp filter is given in McAndrew [7], pp. 105-107 as

$$f = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{c} \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}, \tag{2}$$

where c is a constant chosen to provide the best sharpening result. In the code segment above $c = 0.2$. The result of applying the unsharp filter to the image in Fig. 3 is shown in Fig. 5. The cell fibers have been sharpened somewhat.

3.3 Image Segmentation by Watershedding

In order to understand watershedding, an image needs to be thought of as a surface with intensities as surface heights. The surface will have peaks and valleys as a result of the intensity distribution. If we think of water falling on the surface then it will collect in valleys around the local minima until it overflows into neighboring valleys. The overflows occur at the ridges that surround the valley around a local minimum. The watershed transform finds the local valleys and marks the ridges of the valleys. The MATLAB call to **watershed** produces a matrix the size of the original image in which each matrix element is associated with an image pixel. The matrix produced is called a *label matrix* and identifies all of the individual valleys around minima with a unique number with the elements in the label matrix associated with the pixels in each valley being given the same unique number. The ridges between valleys are labeled with zero. For a more complete discussion of the watershed transform see Gonzalez and Woods [4], pp. 769-776.

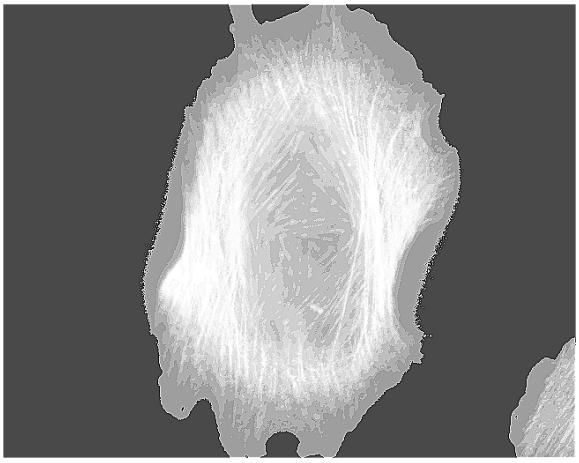

Figure 5: The Cell Image after the Unsharp Filter has been applied.

We illustrate below what the structure of a label matrix padded with zeros might look like for a simple example of a 12×12 image with 4 watershedded regions. Note the region boundaries separated by zeros and the outer edges padded with zeros. We will use this example throughout the text to illustrate some steps in the code.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 3 & 0 & 2 & 2 & 2 & 2 & 2 & 2 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 & 2 & 2 & 2 & 2 & 2 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 & 2 & 2 & 2 & 2 & 2 & 0 \\ 0 & 3 & 3 & 3 & 0 & 0 & 0 & 0 & 2 & 2 & 2 & 0 \\ 0 & 3 & 0 & 0 & 0 & 4 & 4 & 4 & 0 & 2 & 2 & 0 \\ 0 & 0 & 4 & 4 & 4 & 4 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 0 & 0 & 0 \\ 0 & 4 & 4 & 4 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

The one difficulty with applying the watershed transform directly to an image is that it can oversegment an image into too many regions that de-enhance image understanding and the isolation of significant features. This is illustrated in Fig. 6. Although the segmentation is valid, the oversegmentation observed in Fig. 6 is the result of a large number of local minima often caused by image noise. To control the oversegmentation we first apply a smoothing operation.

We will briefly discuss a method to control oversegmentation described in Gonzalez and Woods [4], pp. 776-778, and in Gonzalez et al. [5], pp. 422-424. Their approach to minimizing the number of valleys around minima is to note that many minima are shallow relative to neighboring pixels and should not be treated as legitimate deep valleys. Such minima are extraneous to the general feature identification. The method described in Gonzalez and Woods [4] and Gonzalez et al. [5] is based on a concept of *markers*. Markers are connected components of an image. There

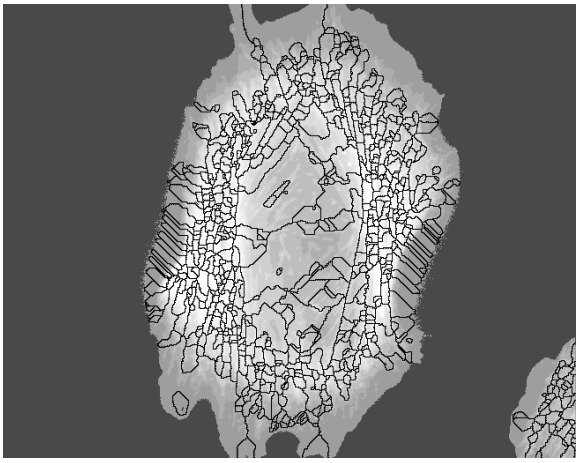

Figure 6: The Watershed Transform applied to the Original Cell Image.

are *internal markers* associated with objects of interest and *external markers* associated with the background. An effective way to mimimize the effects of small local mimima is to first apply a smoothing filter. Once the image has been smoothed the internal markers can be identified, where an internal marker has these characteristics (1) the region is surrounded by higher intensity pixels, (2) the region is connected, and (3) all points in the connected region have the same intensity. The internal markers can be identified using the Image Processing Toolbox (IPT) in MATLAB by a function called **imextendedmin**. This function computes the set of minima in an image that are lower than neighbors by a specified threshold. The threshold, in a sense, instructs the function to ignore all minima with intensities less than the threshold and makes the internal marker identify the locations of significant local minima. The process is called an *extended minima transform*. For a complete discussion of extended minima see Soille [10], pp. 203-204. In the case of 16-bit cell images with intensity range of 0 to 65535, a reasonable choice of threshold was found to be 10000. This threshold was found by applying the extended minima transform to multiple images and noting the reduction in the initial segmentation of the actin fiber regions.

Once the internal markers have been identified, the watershed algorithm in IPT can be applied. This is done in Gonzalez et al. [5], pp. 422-425, by first finding the pixels in the image generated by the **imextendedmin** call that are midway between each of the internal markers. This is done by applying the watershed algorithm to an image that is the distance function value between each pixel in the internal marker image and the nearest nonzero pixel. In the case below, the 'cityblock' metric is used, which is the sum of the lengths of the orthogonal edges of the right triangle formed between two pixel coordinates. Although there are multiple choices of metrics for the MATLAB function **bwdist**, this metric seems to produce good results for the cell images.

```
%Use markers to eliminate noise and shallow minima and then apply watershed
imin = imextendedmin(I2,10000);
L = watershed(bwdist(imin,'cityblock'));
```

The watershed boundaries here are the external markers. The resulting overlays to the original cell image can be displayed by the function **imshow** in IPT by giving the boundary zeros in the

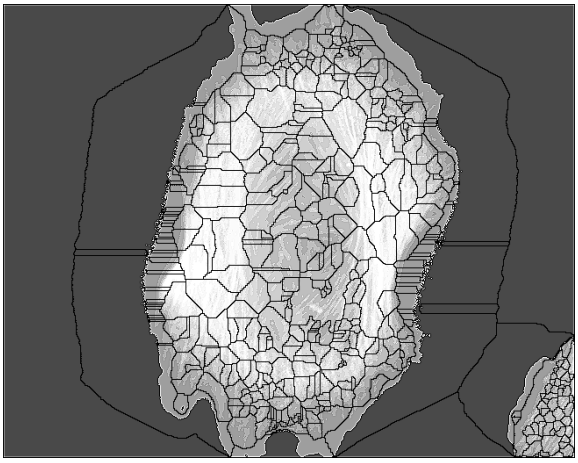

Figure 7: The Watershed Transform applied to the Distance Transform of the Internal Marker Image.

label matrix a very low intensity value. The segment of code below locates the label matrix elements that are zero. It then copies the original image to another one that is overlayed with low intensity pixel values at the pixels associated with zeros in the label matrix. The final pre-processed image is thus created by assigning a very low, in this case 1, intensity value to the watershed boundary pixels, identified by 0 in the label matrix L. For the original cell, this is given in Fig. 7. This process has reduced somewhat the segmented regions generated by applying the watershed algorithm directly to the original image and makes it easier to merge regions.

```
%Display the region boundaries as black overlays to I2
em = (L == 0);
I3 = I2;
I3(em) = 1;
figure, imshow(I3);
drawnow;
```

4 Region Merging

If we examine Fig. 7 closely we can note that many of the regions have similar properties. In particular, there are a number of regions that clearly cover areas of bundled actin filaments. These filaments surround a middle section of the cell. The middle section of the cell seems to have similar properties with lower intensities and not many filaments. It would then seem that we would like to merge regions that have high intensity filament structure, and also merge the regions covering the middle of the cell. The question is how best to do this.

One approach that has been used in the literature (Haris et al. [6]) is to link neighboring regions with common properties. A standard mathematical structure that can be employed as a computational tool that allows efficient representations of these relations is called a *graph* (see Ore [8]).

Graphs are graphical representations that join points, called *nodes*, with *links* that may or may not have some form of weight or distance associated with them. Nodes usually have numbers associated with them. Ordinarily, links with large weights are considered the most potentially significant candidates for merging. An example might be a street network where nodes are intersections and streets are the links with speed limits on them. Higher speeds mean shorter travel times. Another example might be a communication network with message capacities on the links. Larger capacities allow for higher volume message passing. In this study, however, we will take links with low weights to be significant. That is, adjacent regions with lower weights on their links will be candidates for merging.

Graphs are convenient tools for linking information. It sometimes happens that a complete description of a graph may not be necessary. For example, two nodes with similar properties may have a link between them with a very low weight. These two nodes could be merged. It is this idea that is behind the process of merging regions in a segmented image. Each region is considered to be a node and links are established between neighboring regions. Regions with similar properties are considered as candidates for merging. A graph representation of regions that link neighboring regions is called an *adjacency graph*. The approach to region merging taken in this work combines an adjacency graph representation of the relationship between regions, where adjacency is defined in terms of a distance function value assigned to each graph link, with a boundary detection algorithm to create the merged region boundaries.

The merging process begins by using the fact that the watershed algorithm produces a label matrix that assigns to each located region a unique number and identifies the boundary of the region with zeros in the label matrix (see (3)). In the first step the pixels in each region are located along with the boundaries of each region using an 8-mask, a 3×3 matrix centered about a pixel element in the label matrix. Next, neighboring regions are located by scanning around the boundary of each region with a 4-mask in order not to asssociate two regions as neighbors if there is only one common boundary point. A 4-mask is represented by the elements above, below and to the left and right sides of a pixel element in the label matrix. A distance function between any two regions is then constructed based on the difference between the average intensities of the two regions. Given that region neighbors and link distances are known, the adjacency graph can be built as a matrix in sparse form that has, for each row, a region and a neighbor along with the link distance. Thus the adjacency graph is represented in node-node-link weight form. The adjacency graph is in an undirected form in that it includes links from A to B and B to A, as well as links A to A. Two steps are then performed to compress the adjacency graph. The first is to eliminate all A to A links and the second is to eliminate all links with weights greater than a predefined tolerance. The adjacency matrix is then searched to form a linked list of regions to be merged. Finally, the linked list is used to join the linked region pixels into what is called a feature and a boundary algorithm is used to locate the external boundary of the features.

4.1 Initial Region Structure Formation

In order to understand how region information is stored one needs to know how an image is represented as a matrix. The origin of the coordinate system used to locate pixels begins in the upper left corner with the coordinate (1, 1). The x-coordinate points to the right and is indexed by matrix columns and the y-coordinate points down and is indexed by rows. This can be a bit confusing since in numerical matrix analysis one thinks in terms of rows first then columns, such as (i,j). But in imaging it is more convenient to think in terms of columns then rows, such as (j,i).

Before a region data structure can be defined, the number of regions found by **watershed** can be determined by the call

```
n_reg = max(max(L));
```

where L is the label matrix generated from the watershed call.

The data representation used for regions in this study is called a *structure*. These are MATLAB arrays with attached named *fields*. Structures are also commonly used in programming languages, such as C. Fields can contain different classes of data. For example, one field might contain text, another might contain a matrix. MATLAB allows the programmer to build a structure dynamically in that one does not have to preallocate the structure array lengths. To begin constructing the region structures here, a call is made to the function **regionprops** in the IPT, which builds a structure from the label matrix created by **watershed**. In the call below we want to store as fields the pixel list of a region as well as the number of pixels in the boundary pixel list. Since the **regionprops** function produces a limited number of fields, we will construct other fields dynamically as we need them.

```
R = regionprops(L,'PixelList','Perimeter');
```

If i is a region index then the initial region structure looks like

```
R(i).PixelList
R(i).Perimeter
```

The pixel list is refered to in (column, row) form that represents an (x,y) coordinate in the image framework.

In an earlier section we padded the original image with zeros around the boundary. When watershed is applied to this form of image in MATLAB it identifies the entire image boundary as region 1. We ignore region 1 by overwriting the 1's in the label matrix with zeros.

```
col = R(1).PixelList(:,1);
row = R(1).PixelList(:,2);
for rc = 1:length(col)
    L(row(rc),col(rc)) = 0;
end
```

We then add a pixel intensity field and an average intensity field to the region structure. In order to perform arithmetic on pixel elements we first convert the image to a double precision matrix. This code segment also points out the difference between image pixel coordinate references and matrix element references. Note that the (c,r) pixel reference is converted to an (r,c) reference in the matrix I2d in order to retrieve the matrix element for the pixel intensity. I2d is a matrix the size of the image I2 but in double precision form.

```
%Add the average region intensity to the region structure

for i = 2:n_reg
    sum = 0.0;
    Pixel_num = length(R(i).PixelList(:,1)); %get the number of pixels in R(i)
    for j = 1:Pixel_num
        c = R(i).PixelList(j,1);
        r = R(i).PixelList(j,2);
        R(i).Intensities(j) = I2d(r,c);
        sum = sum + 10^(-4)*I2d(r,c); %scaling 16 bit image pixels
```

```
    end
    R(i).avg_int = (sum/Pixel_num)*10^4;
end
```

This code adds to the region structures from regions 2 to n_reg the two new fields called Intensities and avg_int. The region structure now looks like

```
R(i).PixelList
R(i).Perimeter
R(i).Intensities
R(i).avg_int
```

In computing the average intensity we use the fact that intensity values are of the order of 10^4, in a **uint16** image, so that, if they are scaled, the sums can be formed with values in a much lower range and then scaled back after the scaled average has been computed. This is needed since the number of pixels in a region can be large. The sum of unscaled numbers could be very large, leading to potential numerical inaccuracies due to roundoff or overflow.

4.2 Building Region Boundary Lists

To build the region boundary lists, we first have to estimate an upper bound for all of the boundary perimeters. In fact, we double the maximum perimeter length in order to make sure that we do not overrun an array. This length is used in a temporary working array that is deleted once the region boundary lists have been formed. Note that we make sure that the maximum boundary length **BL** is an integer by using the MATLAB **fix** function.

```
for i = 2:n_reg
    Bdry_length(i) = R(i).Perimeter;
end

%Get an upper bound on the length of the boundary arrays

BL =2*(fix(max(Bdry_length))+1); %possible overestimate but is safe
```

We can now add a boundary list field to the region structure by scanning the region pixels with an 8-mask and locating the pixels in the label matrix with zero values neighboring region pixels. Points in a mask are referred to in compass notation: W, NW, N, NE, E, SE, S, SW. If a region pixel value is designated by (r,c) in the label matrix L, the NW corner of an 8-mask is (r-1,c-1), which uses the first element in the mask_offsets_8 array. Similarly the N point is (r-1,c) and this uses the second element in **mask_offsets_8** array. This continues around the mask. We scan the pixels in each region from 2 to n_reg. As we scan, we may pick up duplicate boundary array values, but these are eliminated using the MATLAB function **unique** that eliminates duplicate values in a set. The method involved with the mask scanning is based on ideas in the m-function **boundaries** given in Gonzalez et al. [5], pp. 557-560. The script below sets up an 8-mask pixel offset matrix. The pixel list for each region is scanned. A check is made so that the image boundary is not overrun. The boundary pixels found are put into a temporary array that is used in the function **unique** to eliminate duplicate pixels. Finally a boundary list is formed and added to the region structure. At the end, the boundary list is combined with the region pixel list to form a final region pixel list.

```
%Set up a temporary working array

Temp = zeros(BL,2);

%Next array to hold the lengths of the region boundaries

B = zeros(n_reg,1);

%Set up offsets from a pixel in (r,c) form to locate 8-mask entries around
%the (r,c) pixel.

mask_offsets_8 = [-1 -1; -1 0; -1 +1; 0 +1; +1 +1;...
    +1 0; +1 -1; 0 -1]; %for boundary detect mask

%Scan each region pixel to locate boundary pixels that are identified as
%zeros in the label matrix.

for i = 2:n_reg
    Pixel_num = length(R(i).PixelList(:,1)); %get the number of pixels in R(i)
    work_count = 0;
    for j = 1:Pixel_num
        c = R(i).PixelList(j,1);
        r = R(i).PixelList(j,2);
        %scan with 8-point mask to get all 0's around region
        %scan NW N NE E SE S SW W
        for k = 1:8 %need to check that indices are not out of range
            if (r + mask_offsets_8(k,1) <= 0) | (c + mask_offsets_8(k,2) <= 0)|...
                (r + mask_offsets_8(k,1) > mL) | (c + mask_offsets_8(k,2) > nL)
                continue;
            end
            if L(r + mask_offsets_8(k,1),c + mask_offsets_8(k,2))== 0
                %This will likely generate duplicate pixels due to
                %overlapping scan masks
                work_count = work_count + 1;
                Temp(work_count,1) = c + mask_offsets_8(k,2);
                Temp(work_count,2) = r + mask_offsets_8(k,1);
            end
        end
    end
      C = unique(Temp,'rows'); %Eliminates duplicate boundary points
      %Add a boundary list to the structure of R
      B(i,1) = length(C(:,1)); %Boundary length for R(i)
      R(i).BoundaryList = zeros(B(i),2);
      R(i).BoundaryList(1:B(i),1) = C(1:B(i),1);   %column
      R(i).BoundaryList(1:B(i),2) = C(1:B(i),2);   %row
      clear Temp C
end
```

```
%Join pixels and boundaries to essentially eliminate the zeros in
%the label matrix. The only zeros left are the image boundaries.
%Union also eliminates duplicates.

for i = 2:n_reg
    R(i).PixelList = union(R(i).PixelList,R(i).BoundaryList,'rows');
end
```

At this point we have extended the region structure by adding a boundary list.

```
R(i).PixelList
R(i).Perimeter
R(i).Intensities
R(i).avg_int
R(i).BoundaryList
```

We note that the union of the boundary pixels with region pixels would have the effect of eliminating the boundary zeros as shown in the matrix example below.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 & 2 & 2 & 0 \\ 0 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 & 2 & 2 & 0 \\ 0 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 & 2 & 2 & 0 \\ 0 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 2 & 2 & 2 & 0 \\ 0 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 2 & 2 & 0 \\ 0 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 2 & 2 & 0 \\ 0 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 2 & 0 \\ 0 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 1 & 1 & 0 \\ 0 & 4 & 4 & 4 & 4 & 4 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4}$$

4.3 Building a Region Adjacency Structure

We now have a boundary list for each region, although we ignore region 1 since the **watershed** call makes region 1 the image boundary. To form a neighbor structure for a region we trace the boundary of each region with a 4-mask and check the region numbers above, below, and to the sides of the mask. The scan procedure used is similar to the one used to find region boundaries, but in this case we use a 4-mask to identify adjacent regions. This mask eliminates neighboring regions that have one point of contact. As we scan around a region boundary we build an adjacency array and from it compute the number of neighbors and add a neighbor field to the region structure. We note that the array **B(i)** is computed in the previous script portion.

```
%At this point we have a boundary list and boundary length for each region
%We ignore region 1 since the watershed makes that the image boundary
%Next we need to establish an Adjacency Matrix but in sparse form.
%To do this we trace the boundary of each object with a 4-mask and check
%the numbers above, below and to the sides of the mask.
```

```
test = zeros(4,1);

%Set up mask_offset_4 for N E S W Use a 4-mask in order not to link regions
%with only one boundary point in common

mask_offsets_4 = [0 -1; 1 0; 0 1; -1 0]; %Mask to determine neighbors

for i = 2:n_reg

    Adjacency(n_reg,1) = 0; %Set up an adjacency array for a region

    for j = 1:B(i) %step around the length of the boundary for R(i)

        c = R(i).BoundaryList(j,1);
        r = R(i).BoundaryList(j,2);
        for k = 1:4
            rk = r + mask_offsets_4(k,1);
            ck = c + mask_offsets_4(k,2);
            if ((rk ~= 0) & (ck == 0)) | ((rk == 0) & (ck ~=0))...
                    | ((rk == 0) & (ck ==0))| rk > mL | ck > nL
                continue; %when at a boundary element or out of range
            else
                test(k,1) = L(rk,ck);
                for m = 2:n_reg
                    if test(k,1) == m; %may hit m more than once.
                        Adjacency(m,1) = 1;
                    end
                end
           end
        end
    end
   %We fill in the R(i) structure elements for numbers of neighbors
   sum_neigh = 0;
   for isum = 1:n_reg
       sum_neigh = sum_neigh + Adjacency(isum,1);
   end
   number_neighbors = sum_neigh;
   R(i).no_neighbors = number_neighbors;
   count = 0;
   for ii = 2:n_reg
       if (Adjacency(ii,1) ~= 0)
           count = count + 1;
           R(i).neighbor(count) = ii;
       end
   end
   clear Adjacency
end
```

We have now extended the structure to include the number of neighbors and the neighbor list.

```
R(i).PixelList
R(i).Perimeter
R(i).Intensities
R(i).avg_int
R(i).BoundaryList
R(i).no_neighbors
R(i).neighbor
```

4.4 Identifying Similar Neighbors

Although we have a list of all neighbors of a region, not all neighbors have similar characteristics. In order to merge regions into features with more unique characteristics, we need to define a measure of closeness. For this we define a distance function, d(i,j) between regions i and j, that satisfy the standard properties of a metric.

```
d(i,j) >= 0,
d(i,j) = 0 if and only if i = j,
d(i,j) = d(j,i),
d(i,j) <= d(i,h) + d(h,j).
```

The distance measure, **D(p,1)**, we use in this study involves differences between the average intensities of two regions' pixels. Since the image intensities are of the order 10^4 we scale quantities so that they fall in the interval [0,1]. The absolute values of the average intensity differences are used since the squares of the difference could create values of the order 10^8. We first extend the region structure by adding a distance value between each region i and all neighboring regions, **R(j).D_neighbor(i)**. Then we define a tolerance, **Tol**, that is a constraint on the measure of the closest neighbors in terms of region properties and extend the region structure by adding the closest neighbors to the structure. A similar measure has been used by Haris et al. [6]. Pires et al. [9] add a term involving a boundary gradient. Other distance functions could be defined based on such things as region texture measures, but the one used here seems to provide reasonable results and has shown success in the literature.

Once the distance function has been defined, the fields for the number of adjacent regions within a distance tolerance are computed. The scaled tolerance used involves the **median** scaled by **1/1.1**. This scale was determined experimentally and appears to produce the most meaningful feature formation. An adjacency matrix is formed with number of rows equal to the number of regions and column width the size of the list of maximum number of adjacent regions within tolerance to some region. This matrix is sparse and is used in a search algorithm to find all connected components in the adjacency graph. The neighbors within tolerance to region **j** are stored in **R(j).neigh_in_Tol(p)** and the number of the neighbors within tolerance are stored in **R(j).no_neigh_in_Tol**. This information is used to create the adjacency matrix, **adj**.

```
%The next step is to convert the region structure to a generalized
%Adjacency matrix in sparse form using a distance measure between regions
%This is where intensities  enter the picture. This section
%of code produces a matrix of paths from an origin nose to a final
%destination. Region merging will be performed by averaging values for the
%region features for the regions in each path.
```

```
%We will use a process of origin-destination link  matrices to create
%a final graph

p = 0; %need to accumulate over all regions
for j = 2:n_reg
    n_neigh = R(j).no_neighbors;
    for i = 1:n_neigh
        p = p+1;
        J(p,1) = j; %get column
        I(p,1) = R(j).neighbor(i); %get row
        %compute distance function as 10^(-4) times
        %average intensity differences since image intensities are
        %order 10^4
        D(p,1) = (10^(-4))*abs((R(j).avg_int - R(I(p)).avg_int));
        R(j).D_neighbor(i) = D(p,1);
    end
end
%At this point we need to estimate the merging tolerance.  Experimentation
%suggests the following is a working tolerance
Tol = median(D)/1.1;
%Add to the structure the number of neighbors within tolerance
for j = 2:n_reg
    n_neigh = R(j).no_neighbors;
    p = 0;
    for i = 1:n_neigh
        RD = R(j).D_neighbor(i);
        if (RD <= Tol)&(RD > eps)
            p = p+1;
            R(j).neigh_in_Tol(p) = R(j).neighbor(i);
        end
    end
    R(j).no_neigh_in_Tol = p;
end

%Make an adjacency list matrix in a modified linked form. Each row
%will begin with a region and the subsequent columns will list all of the
%regions within the distance tolerance of that region. This matrix will be
%used in a search algorithm during the merge phase. It is global.

no_adj = zeros(1,n_reg);
no_adj(1) = 0;
adj(1,1) = 0;
for i = 2:n_reg
    no_adj(i) = R(i).no_neigh_in_Tol;
    for j = 1:no_adj(i)
        adj(i,j) = R(i).neigh_in_Tol(j);
    end
```

```
end
```

4.5 Building an Undirected Graph Representation of the Region Adjacency

We can now represent the adjacency between regions using an undirected graph representation in terms of nodes and links. In the present case nodes will represent regions and links will be identified in terms of two nodes. Each link has a distance measure between nodes, or regions. The undirected graph is represented in terms of a sparse matrix where each row has the form of node-node-link weight, with the distance measure for the link weight. We then cull the matrix of all links of a node to itself and links with distances greater than the specified tolerance. Finally a list of all of the critical nodes is formed.

```
%We create an undirected graph of neighbors.

length_I = length(I);

%find all node pairs of the same value, such as 1 1 or 2 2, etc.

kill = zeros(1,length_I);
Index = find(D < eps);
kill(Index) = 1;
clear Index

%Form a reduced undirected node-node-link weight matrix by killing node
%pairs of equal value index

p = 0;
for i = 1:length_I
    if (kill(i) ~= 1)
        p = p+1;
        GD(p,1) = J(i);
        GD(p,2) = I(i);
        GD(p,3) = D(i);
    end
end

%Next eliminate links with link weights greater than Tol.
[ng,mg] = size(GD);
q = 0;
for j = 1:ng
     if(GD(j,3) < Tol)
        q = q+1;
        GDT(q,1) = GD(j,1);
        GDT(q,2) = GD(j,2);
        GDT(q,3) = GD(j,3);
    end
end
```

```
clear GD


%All of the critical nodes in the adjacency graph will appear in GDT(:,1).
%All other nodes in the set difference between 2:ndeg and GDT(:,1) will be
%taken as separate features. The critical nodes, called V, will be used in a
%subsequent search algorithm during the region merge phase.
V = unique(GDT(:,1));  %get the unique nodes.
```

4.6 Feature Formation

The vector **V** may not contain the indices of all regions. We identify those regions not appearing in **V** as single region features. Thus features contain one or more regions. In forming features we will begin by creating the single region features and then create the multiple region features. In the process of creating features we apply a graph search algorithm, **DFS**, for depth-first search. In doing so, we need to create an array that tells us whether we have visited a prior region in the search process so that we don't end up in a never terminating graph loop. This is the array **visited**. The array **Total_visited** lists all of the nodes visited. We update this array as we search through the graph for connected components. The connected components are identified as the features. Connected components are formed by starting at a node that has not been visited and going to each of its linked nodes in the **adj** matrix and then following down the graph from each of those nodes. It is clear that this becomes a recursive process that ends when there are no more linked nodes that have not been visited. The search algorithm is implemented in the depth-first search function **DFS**. Once the connected components for a node have been identified, a feature structure can be formed.

The depth-first search algorithm is based on the algorithm **SEARCH** in Aho et al. [1], p. 176-179. It takes as global variables **visited, no_adj, adj, V** and begins with a node called **start** and sets **visited(start)= 1**. The **start** node is always taken as the lowest index numbered region, say r, such that **Total_visited = 0**. Then it loops through each node adjacent to **start**. Let one be called **w**. If **visited(w) = 1** it goes on to another adjacent node to **start**. If not then it calls **DFS(w)** recursively. In effect it works down all path links for the adjacent node **w**. When all the path linked nodes to **w** have been visited, it proceeds to the next adjacent node to **start**. Again it continues until all possible path linked nodes to **start** have been visited. One of the significant outputs of this portion of the code is a new label matrix that labels the features in the same fashion that the region label matrix did for the regions. That is, all of the pixels in each feature are uniquely numbered.

```
%At this point we have an undirected graph with adjacency distances all
%within tolerance. We will begin the feature merging by marking those
%regions that have been visited during the merging process

Total_visited = zeros(1,n_reg);

%Find all regions that do not appear in GDT(:,1). First set
%Total_visited(1) = 1 to eliminate it from being a region

Total_visited(1) = 1;

%We can find all of the single node region features by a set difference
```

```
Total_nodes = 2:n_reg;
Single_reg = setdiff(Total_nodes,V);

%Form features for each of the Single-nodes

lSn = length(Single_reg);

%Make the first lSn features the same as the regions identified in the
%Single_nodes list. We form a feature structure that contains the feature
%pixels.

for i = 1:lSn
    Sn = Single_reg(i);
    lplist = length(R(Sn).PixelList(:,1)); %get no pixels in reg Sn
    %build the feature structure
    F(i).PixelList(1:lplist,1) = R(Sn).PixelList(1:lplist,1);
    F(i).PixelList(1:lplist,2) = R(Sn).PixelList(1:lplist,2);
    Total_visited(Sn) = 1;
end

%Initialize counter for the next multi-node features

p = lSn;
%Initialize a new label Matrix

NewL = zeros(mL,nL);

%Before creating features we need to find the regions that are to be merged
%for each feature

I = find(Total_visited == 0); %Find all nodes that haven't been visited

while ~isempty(I) %Continue as long as there are nodes that haven't been
                  %visited
    visited = zeros(1,n_reg);
    start = I(1); %Use the first available 0 visited node
    %The next call does a depth-first search of the undirected graph. The
    %output is the array visited of all nodes ultimately linked to the node
    %start. This generates the connected components to start in the graph.
    DFS(start);  %get the connected components linked to node start
    clear I %I will be updated below
    p = p+1;  %start a new feature.
    %Initialize the new feature
    Reg_visit = find(visited == 1); %Find the regions visited in the search
    lRv = length(Reg_visit);
    F(p).PixelList = R(Reg_visit(1)).PixelList;
    for i = 1:lRv %Join all the pixels of visited regions
        F(p).PixelList = union(F(p).PixelList,...
```

```
            R(Reg_visit(i)).PixelList,'rows');
    end
    Total_visited = Total_visited + visited; %add recent visited to
                                             %total visited
    I = find(Total_visited == 0);
end
no_features = p

%set the pixel values in F(q).PixelList in the array NewL.

for q = 1:no_features
   %form a new label matrix
    new_col = F(q).PixelList(:,1);
    new_row = F(q).PixelList(:,2);
    for nrc = 1:length(new_col)
        if (new_row(nrc) == 0) | (new_col(nrc) == 0)
            continue;
        end
        NewL(new_row(nrc),new_col(nrc)) = q;
    end
end
```

We now have the merged pixel list for each feature and a new label matrix for all of the features. The feature structure at this point consists of a pixel list.

```
F(i).PixelList
```

We next need to determine the boundaries of the merged regions. For this we apply a boundary tracking algorithm by Moore (see Gonzalez and Woods [4], p. 796-797).

4.7 Feature Boundary Identification

In order to locate the features on the original image, being able to identify the feature boundaries is critical. This code portion describes the implementation of the Moore boundary tracking algorithm used. The basic algorithm is rather simple. It starts in the upper left corner of a feature and sets that pixel as the first point of the boundary. It places an 8-mask over the point and begins a clockwise test of each element of the mask beginning with western point of the mask until the test hits a pixel of the feature. This point is put into the boundary list and an 8-mask is set over this point and the pixel just before the hit would be outside of the feature, so a clockwise test, beginning at the last outside point, of the elements of the 8-mask are made until a pixel hit of the feature is made. The process continues. The stopping criteria is a return to the initial pixel and a retest around the initial pixel. If the next hit is the same as the hit after beginning the algorithm, then the algorithm stops, otherwise it continues. There are cases in which the initial pixel is hit and the next hit is not the second pixel hit. The algorithm adds this new hit to the boundary list and continues. For a more complete discussion of the algorithm see Gonzalez and Woods [4], p. 796-798.

The final output of this portion of the code is a new label matrix with feature boundaries identified by zeros in the same manner that the watershed segmentation algorithm identified boundaries of the segmented regions. This label matrix is used to plot the final image in the post-process portion of the code.

```
% Start the Moore Boundary Algorithm
[Nr,Nc] = size(NewL);
empty = 0;
ignore = zeros(no_features,1);
for p = 1:no_features
    %Finding the feature starting boundary point.
    %In Matlab arrays are stored by columns. Thus to find when NewL == p
    %the result returned is the set of all elements in linear order
    %starting with the upper left point of the feature. This point
    %represents the initial boundary point for the feature.
    L_index = min(find(NewL == p)); %Get the minimum of the linear indices
    if isempty(L_index) %If the index list is empty get a new feature
        empty = empty + 1;
        ignore(empty) = p;
        continue;
    end
    %Find the row and column in NewL for the minimum index
    r = mod(L_index, Nr);
    if r == 0
        r = Nr;
        c = L_index/Nr;
    else
        c = fix(L_index/Nr) + 1;
    end
    %We initialize the feature boundary structure
    F(p).BoundaryList(1,1) = c;
    F(p).BoundaryList(1,2) = r;
    b_cnt = 1;
    %By selection of (r,c) the western point (r,c-1) is not in the feature
    %or on the boundary of the pth feature. We test around (r,c) beginning
    %at (r,c-1) to find the next point that intersects the boundary. If no
    %intersection is found the feature is a single point. The function
    %Neigh8 generates the row and column offset given a mask index. It is
    %a look-up table.
    for index = 2:9
        [offset_r,offset_c] = Neigh8(index);
        if (NewL(r+offset_r,c+offset_c) ~= p)
            continue;
        elseif (NewL(r+offset_r,c+offset_c) == p)
            b_cnt = 2; %Store the second boundary hit for stopping
            F(p).BoundaryList(2,1) = c+offset_c;
            F(p).BoundaryList(2,2) = r+offset_r;
            %Get the point before the hit
            [back_off_r,back_off_c] = Neigh8(index-1);
            Last_non_boundary = [r+back_off_r, c+ back_off_c];
            r = r+offset_r;
            c = c+offset_c;
            break; %exit the search loop
```

```
        end
    end
    %If the search loop exits with index = 9 then it has returned to the
    %initial non-boundary point and this feature is a one point feature.
    if (index == 9)
        continue; %go to the next feature
    end
    %To continue locating the boundary we reset (r+offset_r,c+offset_c) to
    %(r,c) and start the scan around (r,c) from the Last_non_boundary
    %point. We set a check flag to be used as a feature boundary
    %termination check.
    check_cnt =-999;
    while (F(p).BoundaryList(b_cnt,1) ~= F(p).BoundaryList(2,1))|...
            (F(p).BoundaryList(b_cnt,2) ~= F(p).BoundaryList(2,2))|...
            (b_cnt ~= check_cnt + 1) %Termination criteria
        %search around the boundary point center (r,c) beginning with the
        %last non-boundary point. The function cal inv_Neigh8 takes a row
        %and column offset and produces the mask start index. It is a
        %look-up table.
        offset_r = Last_non_boundary(1) - r;
        offset_c = Last_non_boundary(2) - c;
        start_index = inv_Neigh8(offset_r,offset_c);
        for index = start_index+1:start_index+8
            [offset_r,offset_c] = Neigh8(index); %get offsets from (r,c)
            if (NewL(r+offset_r,c+offset_c) ~= p)
                continue;
            elseif (NewL(r+offset_r,c+offset_c) == p)
                b_cnt = b_cnt + 1;
                F(p).BoundaryList(b_cnt,1) = c+offset_c;
                F(p).BoundaryList(b_cnt,2) = r+offset_r;
                if (F(p).BoundaryList(b_cnt,1) == F(p).BoundaryList(1,1)) &...
                    (F(p).BoundaryList(b_cnt,2) == F(p).BoundaryList(1,2))
                    check_cnt = b_cnt;
                end
                %Get the point before the hit
                [back_off_r,back_off_c] = Neigh8(index-1);
                Last_non_boundary = [r+back_off_r, c+ back_off_c];
                %set up next boundary search center
                 r = F(p).BoundaryList(b_cnt,2);
                 c = F(p).BoundaryList(b_cnt,1);
                 break; %exit the local boundary search loop
            end
        end
    end
    %Delete duplicate points
    F(p).BoundaryList = unique(F(p).BoundaryList,'rows');
end
%To display the merged regions first form a label matrix with zeros at
```

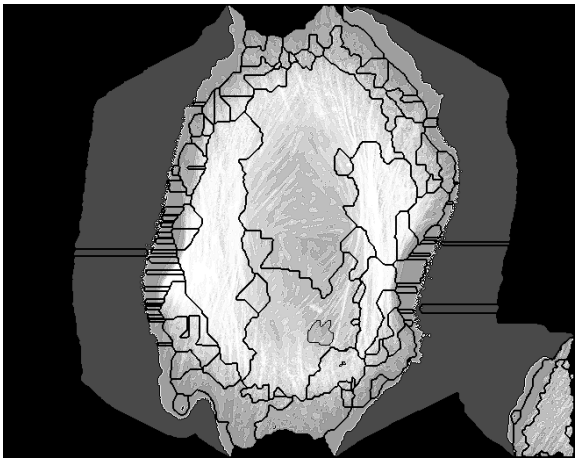

Figure 8: Merged Regions into Features for Cell 1.

```
%the feature boundaries
NewL_bdr = zeros(Nr,Nc);
NewL_bdr = NewL;
for p = 1:no_features
    ign = find(ignore == p);
    if ~isempty(ign)
        continue;
    end
    lFpB = length(F(p).BoundaryList(:,1));
    for q = 1:lFpB
        r = F(p).BoundaryList(q,2);
        c = F(p).BoundaryList(q,1);
        NewL_bdr(r,c) = 0;
    end
end
bdrs = (NewL_bdr == 0);
I2new = I2;
I2new(bdrs) = 1;
figure,imshow(I2new);
```

5 Results

Figure 8 shows the reduced number of regions compressed into regions that outline sections of the cell with common features. One of the outlined regions is the central portion of the cell that probably contins the nucleus. There are other regions that outline the dense actin fiber bundles. There are also regions that identify where the fibers spread out, and also regions that outline the boundary portion of the cell. The merging algorithm reduced 416 watershed segmented regions to 110.

Cell, Feature Region Nos. and Timings(s)							
Cell	N_1	N_2	T_1	T_2	T_3	T_4	T_5
1	416	110	0.29	0.73	31.92	1.10	0.04
2	1024	268	0.11	0.72	42.11	1.79	0.04
4	2009	435	0.12	0.73	71.89	2.71	0.04
5	319	75	0.12	0.72	48.37	0.59	0.04
6	1155	287	0.12	0.78	43.79	1.37	0.04
7	1373	343	0.11	0.74	27.35	1.60	0.04
8	2193	515	0.11	0.73	47.63	2.51	0.04

Table 1: Cell Regions, Features, and Program Timings

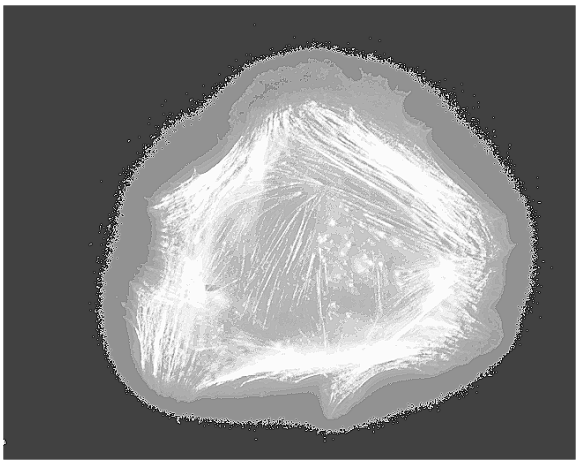

Figure 9: Raw Image for Cell 2.

We display below the image results of actin feature identification for several cells. We only include the raw cell images and the final feature identification images. The following are the reference file names for the cell images: cell2_MLCPPandfactin_red_0078, cell4_MLCPPandfactin_red_0084, cell5_MLCPPandfactin_red_0098, cell6_MLCPPandfactin_red_0101, cell7_MLCPPandfactin_red_0104, cell8_MLCPPandfactin_red_0107.

Table 1 has 8 columns representing cell number, watershed regions, feature regions, and particular program timings in seconds. Column 1 is the cell number associated with the file number. Column 2, N_1, is the number of watershed segmented regions. Column 3, N_2, is the number of final feature regions after merging. Column 4, T_1, is pre-process timing. Column 5, T_2, is Watershed timing. Column 6, T_3, is the merge process timing. Column 7, T_4, is the feature boundary formation by the Moore algorithm. Column 8, T_5, is the post-process timing. It is clear that future work must be aimed at timing reduction in the merging portion of the code.

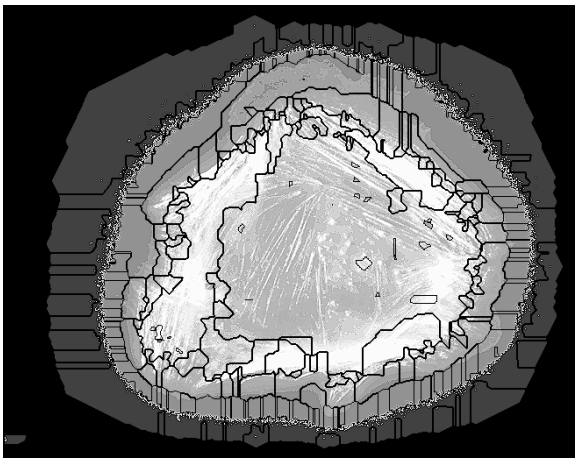

Figure 10: Feature Image for Cell 2.

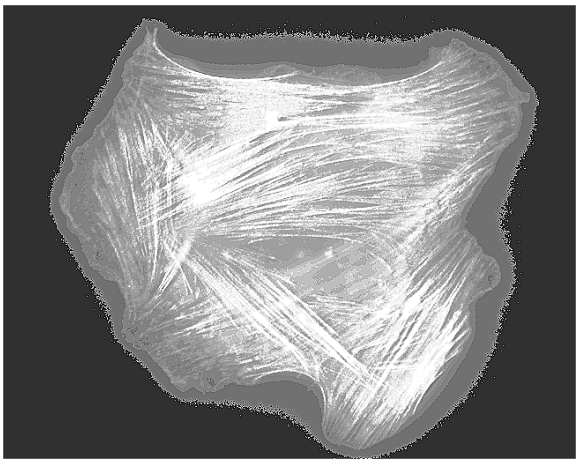

Figure 11: Raw Image for Cell 4.

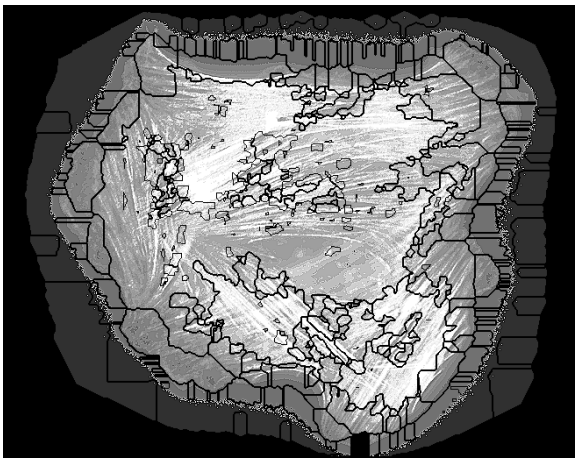

Figure 12: Feature Image for Cell 4.

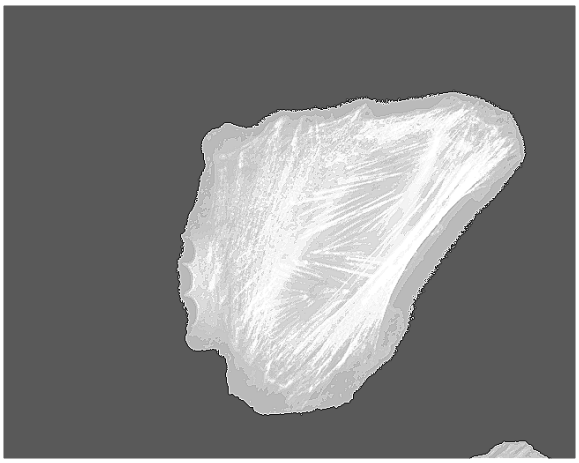

Figure 13: Raw Image for Cell 5.

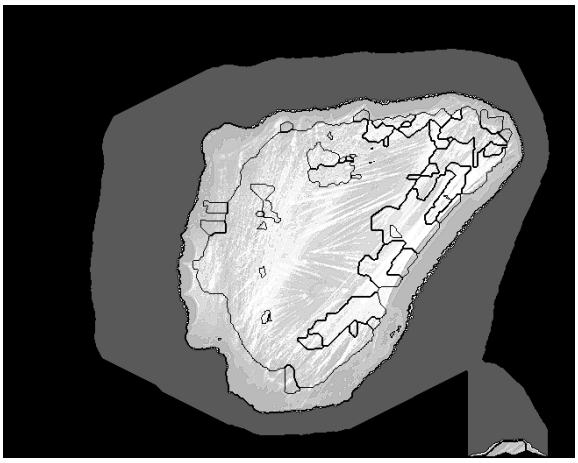

Figure 14: Feature Image for Cell 5.

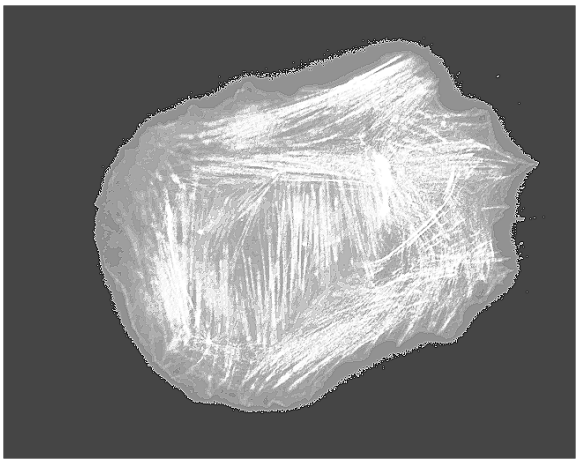

Figure 15: Raw Image for Cell 6.

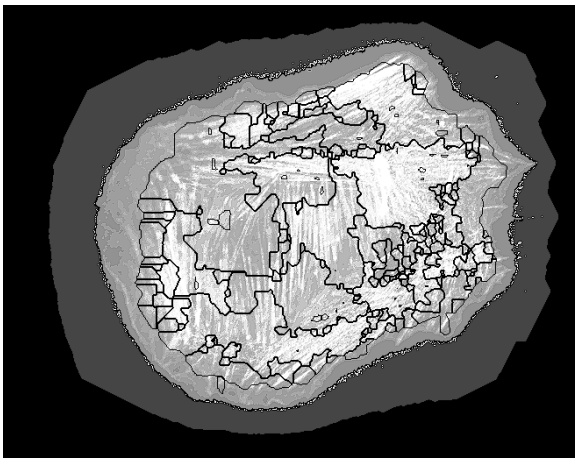

Figure 16: Feature Image for Cell 6.

Figure 17: Raw Image for Cell 7.

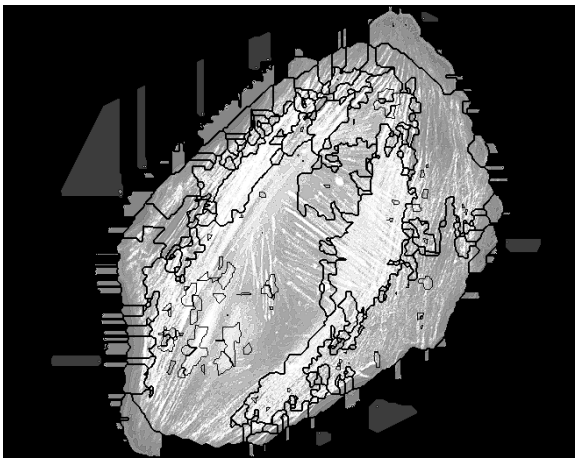

Figure 18: Feature Image for Cell 7.

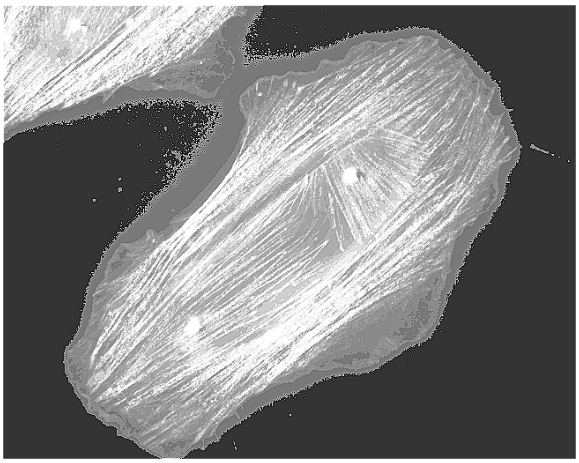

Figure 19: Raw Image for Cell 8.

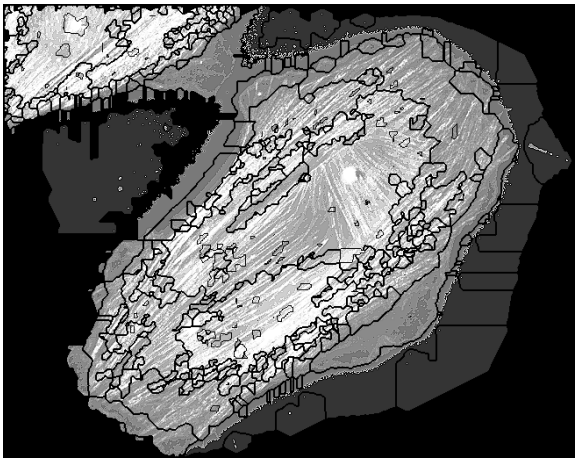

Figure 20: Feature Image for Cell 8.

6 Future Work

The current code is intended as a benchmark in that its results can be compared against those of other feature identification algorithms. There are several directions that future work can take. First, there are some code inefficiencies that need to be adjusted. Other segmentation algorithms could be implemented but their outputs would have to include a region label matrix. Other distance measures for region adjacency could be experimented with as well as other distance tolerances. A multiple pass through the algorithm might further feature compression. The current code is structured as a one-pass algorithm. One other possible efficiency might be to use a modified version of the Moore boundary tracking algorithm in the process of constructing the initial region boundaries.

7 Acknowledgement

The author wishes to thank Prof. Dianne P. O'Leary of the Department of Computer Science, University of Maryland, College Park, for some helpful editorial suggestions and some stimulating discussions on future code enhancements.

8 Disclaimer

Certain commercial software products are identified in this paper in order to adequately specify the computational procedures. Such identification does not imply recommendation or endoresement by the National Institute of Standards and Technology nor does it imply that the software products identified are necessarily the best available for the purpose.

References

[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, (1976).

[2] Bao, G., Suresh, S., 'Cell and molecular mechanics of biological materials', *nature materials*, **2**, (2003), 715-725.

[3] Ethier, C. R., Simmons, D. A., *Introductory Biomechanics: From Cells to Organisms*, Cambridge University Press, Cambridge (2007).

[4] Gonzalez, R. C., Woods, R. E., *Digital Image Processing*. Pearson Prentice Hall, Upper Saddle River, NJ (2008).

[5] Gonzalez, R. C., Woods, R. E., Eddins, S. L., *Digital Image Processing Using MATLAB*. Pearson Prentice Hall, Upper Saddle River, NJ (2004).

[6] Haris, K., Efstratiadis, S. N., Maglaveras, N., Katsaggelos, A. K., 'Hybrid Image Segmentation Using Watersheds and Fast Region Merging', *IEEE Transactions on Image Processing*, **7**, 12, (1998), 1684-1699.

[7] McAndrew, A., *Digital Image Processing with MATLAB*, Thomson Course Technology, Australia (2004).

[8] Ore, O., *Graphs and Their Uses*, Random House, New York (1963).

[9] Pires, R. L. V. P. M., De Smet, P., Philips, W., 'Watershed segmentation and region merging', *Visual Communications and Image Processing 2004*, (Ed. Panchanathan, S. and Vasudev, B.), Proc. SPIE-IS&T Electronic Imaging, SPIE Vol 5308, (2004), 1127-1135.

[10] Soille, P., *Morphological Image Analysis: Principles and Applications*, Springer-Verlag, Berlin (2004).

A Watershed_Merge MATLAB Program

```
function Main_Watershed_Merge
%MAIN_WATERSHED_MERGE - This function simply calls the functions that
%create image regions, merge them into features, and displays the final
%image
%
%Author:
%   David E. Gilsinn
%   Mathematical and Computational Sciences Division
%   National Institute of Standards and Technology
%   100 Bureau Drive, Stop 8910
%   Gaithersburg, MD  20899-8910
%
%*************************************************************************
I2 = Pre_processor; %Get the padded filtered cell image
[L,I2] = Watershed(I2); %Get the label matrix of the region boundaries
[NewL,F, no_features] = Merge(L, I2); %Merge the regions into features
```

```
NewL_bdr = feature_boundary(F, NewL, no_features); %Locate the feature boundaries
Post_processor(I2, NewL_bdr); %Plot the feature boundaries as image overlays

%**************************************************************************
%Main functions
%**************************************************************************
function I2 = Pre_processor
%PRE_PROCESSOR - Loads a cell image, sharpens the edges and pads the
%                boundary with zeros
%
%Output:
%   I2 - filtered image with padded boundary
%   Plotted image
%
%Author:
%   David E. Gilsinn
%   Mathematical and Computational Sciences Division
%   National Institute of Standards and Technology
%   100 Bureau Drive, Stop 8910
%   Gaithersburg, MD  20899-8910
%
%**************************************************************************
%Load a 16 bit image
%I0 = imread('cell1_MLCPPandfactin_red_0075.tif'); %16 bit image
%I0 = imread('cell2_MLCPPandfactin_red_0078.tif'); %16 bit image
%I0 = imread('cell3_MLCPPandfactin_red_0081.tif'); %16 bit image
%I0 = imread('cell4_MLCPPandfactin_red_0084.tif'); %16 bit image
%I0 = imread('cell5_MLCPPandfactin_red_0098.tif'); %16 bit image
%I0 = imread('cell6_MLCPPandfactin_red_0101.tif'); %16 bit image
%I0 = imread('cell7_MLCPPandfactin_red_0104.tif'); %16 bit image
I0 = imread('cell8_MLCPPandfactin_red_0107.tif'); %16 bit image
%Do a histogram equalization
I1 = histeq(I0);
%unsharp masking is edge crisping by subtracting a scaled blurred version
%of the image from the original image
h = fspecial('unsharp',0.2);
I2 = imfilter(I1,h);
%Display the sharpened image
figure, imshow(I2),impixelinfo;
%Pad the array boundary with zeros
I2 = padarray(I2,[1 1],0,'both');

%**************************************************************************
function [L,I2] = Watershed(I2)
%WATERSHED - This function makes use of watershed segmentation of an image
%into regions. The function initially applies an image extended
%minimization. See the Soille reference.
%
```

```
%Input:
%   I2 - Image padded with zeros around the image boundary
%
%Output:
%   I2 - The loaded image, assumed to be uint16
%   L - label matrix the size of I2 with watershed regions uniquely
%       identified with numbers. That is, each matrix element associated
%       with a region pixel is given the same number. Region boundaries are
%       identified with zeros
%
%Author:
%   David E. Gilsinn
%   Mathematical and Computational Sciences Division
%   National Institute of Standards and Technology
%   100 Bureau Drive, Stop 8910
%   Gaithersburg, MD  20899-8910
%
%*************************************************************************
%Use markers to eliminate noise and shallow minima and then apply watershed
imin = imextendedmin(I2,10000);
L = watershed(bwdist(imin,'cityblock'));
%Display the region boundaries as black overlays to I2
em = (L == 0);
I3 = I2;
I3(em) = 1;
figure, imshow(I3);
drawnow;

%*************************************************************************
function [NewL,F, no_features] = Merge(L, I2)
%MERGE - This function takes a label matrix of a zero padded image I2,
%develops a region structure array, finds the region boundaries and region
%neighbors within a specified tolerance distance, forms an adjacency array,
%and links neighbors along paths from a given region into feature regions.
%
%Input:
%   I2  -  Zero padded 16-bit image
%   L   -  Label matrix of regions identified with unique numbers
%
%Output:
%   NewL - Label matrix for the features
%   F    - Feature structure with associated fields
%   no_features - number of features
%
%Author:
%   David E. Gilsinn
%   Mathematical and Computational Sciences Division
%   National Institute of Standards and Technology
```

```
%   100 Bureau Drive, Stop 8910
%   Gaithersburg, MD  20899-8910
%
%*************************************************************************
global visited no_adj adj V %Used in depth-first search algorithm
[mL,nL] = size(L);
I2d = double(I2); %Convert to double precision for computational reasons
n_reg = max(max(L)) %Get the number of regions

%*************************************************************************
%Build the first fields of the region structure
%*************************************************************************

%Begin building the region structure array
%Get a list of the pixels and approximate boundary length of the objects
%PixelList is given in (c,r) form. Need to zero out region 1.
%Watershed makes the zeros around the boundary of the image region 1.

R = regionprops(L,'PixelList','Perimeter'); %R structure

%Zero out region 1, although this might not be necessary

col = R(1).PixelList(:,1);
row = R(1).PixelList(:,2);
for rc = 1:length(col)
    L(row(rc),col(rc)) = 0;
end

%We only work with region 2 to n_reg.
%Next find an upper bound estimate of the number of boundary pixels

for i = 2:n_reg
    Bdry_length(i) = R(i).Perimeter;
end

%Get an upper bound on the length of the boundary arrays

BL =2*(fix(max(Bdry_length))+1); %possible overestimate but is safe

%Add the intensities and average region intensity to the region structure
%Scale the 16 bit intensities so that sume can be formed in a reasonable
%numerical range. 16 bit intensities are of the order 10^4

for i = 2:n_reg
    sum = 0.0;
    Pixel_num = length(R(i).PixelList(:,1)); %get the number of pixels in R(i)
    for j = 1:Pixel_num
        c = R(i).PixelList(j,1);
```

```
        r = R(i).PixelList(j,2);
        R(i).Intensities(j) = I2d(r,c);
        sum = sum + 10^(-4)*I2d(r,c); %scaling 16 bit image pixels
    end
    R(i).avg_int = (sum/Pixel_num)*10^4;
end

%*************************************************************************
%Find the region boundary and add the field to the region structure
%*************************************************************************

%Set up a temporary working array

Temp = zeros(BL,2);

%Next array to hold the lengths of the region boundaries

B = zeros(n_reg,1);

%Set up offsets from a pixel in (r,c) form to locate 8-mask entries around
%the (r,c) pixel.

mask_offsets_8 = [-1 -1; -1 0; -1 +1; 0 +1; +1 +1;...
    +1 0; +1 -1; 0 -1]; %for boundary detect mask

%Scan each region pixel to locate boundary pixels that are identified as
%zeros in the label matrix.

for i = 2:n_reg
    Pixel_num = length(R(i).PixelList(:,1)); %get the number of pixels in R(i)
    work_count = 0;
    for j = 1:Pixel_num
        c = R(i).PixelList(j,1);
        r = R(i).PixelList(j,2);
        %scan with 8-point mask to get all 0's around region
        %scan NW N NE E SE S SW W
        for k = 1:8 %need to check that indices are not out of range
            if (r + mask_offsets_8(k,1) <= 0) | (c + mask_offsets_8(k,2) <= 0)|...
                (r + mask_offsets_8(k,1) > mL) | (c + mask_offsets_8(k,2) > nL)
                continue;
            end
            if L(r + mask_offsets_8(k,1),c + mask_offsets_8(k,2))== 0
                %This will likely generate duplicate pixels due to
                %overlapping scan masks
                work_count = work_count + 1;
                Temp(work_count,1) = c + mask_offsets_8(k,2);
                Temp(work_count,2) = r + mask_offsets_8(k,1);
            end
```

```
        end
    end
      C = unique(Temp,'rows'); %Eliminates duplicate boundary points
      %Add a boundary list to the structure of R
      B(i,1) = length(C(:,1)); %Boundary length for R(i)
      R(i).BoundaryList = zeros(B(i),2);
      R(i).BoundaryList(1:B(i),1) = C(1:B(i),1);   %column
      R(i).BoundaryList(1:B(i),2) = C(1:B(i),2);   %row
      clear Temp C  %clear working arrays so they can be reset for each region
end

%Join pixels and boundaries to essentially eliminate the zeros in
%the label matrix. The only zeros left are the image boundaries.
%Union also eliminates duplicates.

for i = 2:n_reg
    R(i).PixelList = union(R(i).PixelList,R(i).BoundaryList,'rows');
end

%************************************************************************
%Build the sparse form of the adjacency matrix
%************************************************************************

%At this point we have a boundary list and boundary length for each region
%We ignore region 1 since the watershed makes that the image boundary
%Next we need to establish an Adjacency Matrix but in sparse form.
%To do this we trace the boundary of each object with a 4-mask and check
%the numbers above, below and to the sides of the mask.

test = zeros(4,1);

%Set up mask_offset_4 for N E S W Use a 4-mask in order not to link regions
%with only one boundary point in common

mask_offsets_4 = [0 -1; 1 0; 0 1; -1 0]; %Mask to determine neighbors

for i = 2:n_reg

    Adjacency(n_reg,1) = 0; %Set up an adjacency array for a region

    for j = 1:B(i) %step around the length of the boundary for R(i)

        c = R(i).BoundaryList(j,1);
        r = R(i).BoundaryList(j,2);
        for k = 1:4
            rk = r + mask_offsets_4(k,1);
            ck = c + mask_offsets_4(k,2);
            if ((rk ~= 0) & (ck == 0)) | ((rk == 0) & (ck ~=0))...
```

```
                    | ((rk == 0) & (ck ==0))| rk > mL | ck > nL
                continue; %when at a boundary element or out of range
             else
                test(k,1) = L(rk,ck);
                for m = 2:n_reg
                    if test(k,1) == m; %may hit m more than once.
                        Adjacency(m,1) = 1;
                    end
                end
            end
         end
     end
   %We fill in the R(i) structure elements for numbers of neighbors
   sum_neigh = 0;
   for isum = 1:n_reg
       sum_neigh = sum_neigh + Adjacency(isum,1);
   end
   number_neighbors = sum_neigh;
   R(i).no_neighbors = number_neighbors;
   count = 0;
   for ii = 2:n_reg
       if (Adjacency(ii,1) ~= 0)
           count = count + 1;
           R(i).neighbor(count) = ii;
       end
   end
   clear Adjacency %Array is reset for a new region
end

%The next step is to convert the region structure to a generalized
%Adjacency matrix in sparse form using a distance measure between regions
%This is where intensities  enter the picture. This section
%of code produces a matrix of paths from an origin nose to a final
%destination. Region merging will be performed by averaging values for the
%region features for the regions in each path.

%We will use a process of origin-destination link  matrices to create
%a final graph

p = 0; %need to accumulate over all regions
for j = 2:n_reg
    n_neigh = R(j).no_neighbors;
    for i = 1:n_neigh
        p = p+1;
        J(p,1) = j; %get column
        I(p,1) = R(j).neighbor(i); %get row
        %compute distance function as 10^(-4) times
        %average intensity differences since image intensities are
```

```
        %order 10^4
        D(p,1) = (10^(-4))*abs((R(j).avg_int - R(I(p)).avg_int));
        R(j).D_neighbor(i) = D(p,1);
    end
end
%At this point we need to estimate the merging tolerance.  Experimentation
%suggests the following is a working tolerance
Tol = median(D)/1.1;
%Add to the structure the number of neighbors within tolerance
for j = 2:n_reg
    n_neigh = R(j).no_neighbors;
    p = 0;
    for i = 1:n_neigh
        RD = R(j).D_neighbor(i);
        if (RD <= Tol)&(RD > eps)
            p = p+1;
            R(j).neigh_in_Tol(p) = R(j).neighbor(i);
        end
    end
    R(j).no_neigh_in_Tol = p;
end

%Make an adjacency list matrix in a modified linked form. Each row
%will begin with a region and the subsequent columns will list all of the
%regions within the distance tolerance of that region. This matrix will be
%used in a search algorithm during the merge phase. It is global.

no_adj = zeros(1,n_reg);
no_adj(1) = 0;
adj(1,1) = 0;
for i = 2:n_reg
    no_adj(i) = R(i).no_neigh_in_Tol;
    for j = 1:no_adj(i)
        adj(i,j) = R(i).neigh_in_Tol(j);
    end
end

%************************************************************************
%We create an undirected graph of neighbors using the adjacency matrix
%************************************************************************

%We create an undirected graph of neighbors.

length_I = length(I);

%find all node pairs of the same value, such as 1 1 or 2 2, etc.

kill = zeros(1,length_I);
```

```
Index = find(D < eps);
kill(Index) = 1;
clear Index

%Form a reduced undirected node-node-link weight matrix by killing node
%pairs of equal value index

p = 0;
for i = 1:length_I
    if (kill(i) ~= 1)
        p = p+1;
        GD(p,1) = J(i);
        GD(p,2) = I(i);
        GD(p,3) = D(i);
    end
end

%Next eliminate links with link weights greater than Tol.
[ng,mg] = size(GD);
q = 0;
for j = 1:ng
     if(GD(j,3) < Tol)
        q = q+1;
        GDT(q,1) = GD(j,1);
        GDT(q,2) = GD(j,2);
        GDT(q,3) = GD(j,3);
    end
end
clear GD


%All of the critical nodes in the adjacency graph will appear in GDT(:,1).
%All other nodes in the set difference between 2:ndeg and GDT(:,1) will be
%taken as separate features. The critical nodes, called V, will be used in a
%subsequent search algorithm during the region merge phase. It is global.
V = unique(GDT(:,1));  %get the unique nodes.

%*************************************************************************
%Build the merged features by doing a depth-first search of the unirected
%graph
%*************************************************************************

%At this point we have an undirected graph with adjacency distances all
%within tolerance. We will begin the feature merging by marking those
%regions that have been visited during the merging process

Total_visited = zeros(1,n_reg);
```

```
%Find all regions that do not appear in GDT(:,1). First set
%Total_visited(1) = 1 to eliminate it from being a region

Total_visited(1) = 1;

%We can find all of the single node region features by a set difference
%between all nodes and V.

Total_nodes = 2:n_reg;
Single_reg = setdiff(Total_nodes,V);

%Form features for each of the Single-nodes

lSn = length(Single_reg);

%Make the first lSn features the same as the regions identified in the
%Single_nodes list. We form a feature structure that contains the feature
%pixels.

for i = 1:lSn
    Sn = Single_reg(i);
    lplist = length(R(Sn).PixelList(:,1)); %get no pixels in reg Sn
    %build the feature structure
    F(i).PixelList(1:lplist,1) = R(Sn).PixelList(1:lplist,1);
    F(i).PixelList(1:lplist,2) = R(Sn).PixelList(1:lplist,2);
    Total_visited(Sn) = 1;
end

%Initialize counter for the next multi-node features

p = lSn;
%Initialize a new label Matrix

NewL = zeros(mL,nL);

%Before creating features we need to find the regions that are to be merged
%for each feature

I = find(Total_visited == 0); %Find all nodes that haven't been visited

while ~isempty(I) %Continue as long as there are nodes that haven't been
                  %visited
    visited = zeros(1,n_reg);
    start = I(1); %Use the first available 0 visited node
    %The next call does a depth-first search of the undirected graph. The
    %output is the array visited of all nodes ultimately linked to the node
    %start. This generates the connected components to start in the graph.
    %The search is done using the adjacency matrix adj.
```

```
    DFS(start);  %get the connected components linked to node start
    clear I %I will be updated below
    p = p+1;  %start a new feature.
    %Initialize the new feature.
    %The visited array guards against cycles in the graph search.
    Reg_visit = find(visited == 1); %Find the regions visited in the search
    lRv = length(Reg_visit);
    F(p).PixelList = R(Reg_visit(1)).PixelList;
    for i = 1:lRv %Join all the pixels of visited regions
        F(p).PixelList = union(F(p).PixelList,...
            R(Reg_visit(i)).PixelList,'rows');
    end
    Total_visited = Total_visited + visited; %add recent visited to
                                             %total visited
    I = find(Total_visited == 0);
end

%Set the number of connected components as the number of features

no_features = p

%set the pixel values in F(q).PixelList in the array NewL.

for q = 1:no_features
   %form a new label matrix
    new_col = F(q).PixelList(:,1);
    new_row = F(q).PixelList(:,2);
    for nrc = 1:length(new_col)
        if (new_row(nrc) == 0) | (new_col(nrc) == 0)
            continue;
        end
        NewL(new_row(nrc),new_col(nrc)) = q;
    end
end



%************************************************************************
%Support functions
%************************************************************************
%Depth-first graph search algorithm

function DFS(v)
%DFS - Depth-first-search algorithm
%See Aho-Hopcroft-Ullman, "The Design and Analysis of Computer Algorithms",
%pp 176-177
global visited no_adj adj V
visited(v) = 1;
```

```
na = no_adj(v);
for i = 1:na
    w = adj(v,i);
    if visited(w) == 0
        DFS(w);
    end
end

%8-mask offset look-up table

function [offset_r, offset_c] = Neigh8(index)
%NEIGH8 - Given an element index of an 8-mask this function produces the
%pixel offset from the central pixel. The indices are: W - 1, NW - 2,
%N - 3, NE - 4, E - 5, SE - 6, S - 7, SW - 8
icase = mod(index,8);
if icase == 0
    icase = 8;
end
switch icase
    case 1
        offset_r = 0;
        offset_c = -1;
    case 2
        offset_r = -1;
        offset_c = -1;
    case 3
        offset_r = -1;
        offset_c = 0;
    case 4
        offset_r = -1;
        offset_c = 1;
    case 5
        offset_r = 0;
        offset_c = 1;
    case 6
        offset_r = 1;
        offset_c = 1;
    case 7
        offset_r = 1;
        offset_c = 0;
    case 8
        offset_r = 1;
        offset_c = -1;
end

%8-mask index look-up table

function index = inv_Neigh8(offset_r,offset_c)
```

```
%INV_NEIGH8 - Given a pixel offset an index is produced
%This function begins with neighborhood offsets and produces the index
%number for an 8 Neighbor point. The indices are: W - 1, NW - 2, N - 3,
%NE - 4, E - 5, SE - 6, S - 7, SW - 8
if (offset_r == 0) & (offset_c == -1)
    index = 1;
elseif (offset_r == -1) & (offset_c == -1)
    index = 2;
elseif (offset_r == -1) & (offset_c == 0)
    index = 3;
elseif (offset_r == -1) & (offset_c == 1)
    index = 4;
elseif (offset_r == 0) & (offset_c == 1)
    index = 5;
elseif (offset_r == 1) & (offset_c == 1)
    index = 6;
elseif (offset_r == 1) & (offset_c == 0)
    index = 7;
elseif (offset_r == 1) & (offset_c == -1)
    index = 8;
end

%*************************************************************************
function NewL_bdr = feature_boundary(F, NewL, no_features)
%FEATURE_BOUNDARY - Form the feature boundaries and add a boundary field
%array. The algoritm is based on the Moore boundary algorithm. See
%Gonzalez, R. C., Woods, R. E., Digital Image Processing, Pearson-prentice
%Hall, Upper Saddle River, NJ, 2008, 796-798.
%
%Input:
%   F - Feature structure
%   NewL - Feature label matrix
%   no_features - number of features
%
%Output:
%   NewL-bdr - Feature label matrix with boundaries identified by zeros
%
%Author:
%   David E. Gilsinn
%   Mathematical and Computational Sciences Division
%   National Institute of Standards and Technology
%   100 Bureau Drive, Stop 8910
%   Gaithersburg, MD  20899-8910
%
%*************************************************************************

% Start the Moore Boundary Algorithm
[Nr,Nc] = size(NewL);
```

```
empty = 0;
ignore = zeros(no_features,1);
for p = 1:no_features
    %Finding the feature starting boundary point.
    %In Matlab arrays are stored by columns. Thus to find when NewL == p
    %the result returned is the set of all elements in linear order
    %starting with the upper left point of the feature. This point
    %represents the initial boundary point for the feature.
    L_index = min(find(NewL == p));%Get the minimum of the linear indices
    if isempty(L_index) %If the index list is empty get a new feature
        empty = empty + 1;
        ignore(empty) = p;
        continue;
    end
    %Find the row and column in NewL for the minimum index
    r = mod(L_index, Nr);
    if r == 0
        r = Nr;
        c = L_index/Nr;
    else
        c = fix(L_index/Nr) + 1;
    end
    %We initialize the feature boundary structure
    F(p).BoundaryList(1,1) = c;
    F(p).BoundaryList(1,2) = r;
    b_cnt = 1;
    %By selection of (r,c) the western point (r,c-1) is not in the feature
    %or on the boundary of the pth feature. We test around (r,c) beginning
    %at (r,c-1) to find the next point that intersects the boundary. If no
    %intersection is found the feature is a single point. The function
    %Neigh8 generates the row and column offset given a mask index. It is a
    %look-up table.
    for index = 2:9
        [offset_r,offset_c] = Neigh8(index);
        if (NewL(r+offset_r,c+offset_c) ~= p)
            continue;
        elseif (NewL(r+offset_r,c+offset_c) == p)
            b_cnt = 2;
            F(p).BoundaryList(2,1) = c+offset_c;
            F(p).BoundaryList(2,2) = r+offset_r;
            %Get the point before the hit
            [back_off_r,back_off_c] = Neigh8(index-1);
            Last_non_boundary = [r+back_off_r, c+ back_off_c];
            r = r+offset_r;
            c = c+offset_c;
            break; %exit the search loop
        end
    end
```

```
    %If the search loop exits with index = 9 then it has returned to the
    %initial non-boundary point and this feature is a one point feature.
    if (index == 9)
        continue; %go to the next feature
    end

    %To continue locating the boundary we reset (r+offset_r,c+offset_c) to
    %(r,c) and start the scan around (r,c) from the Last_non_boundary
    %point. We set a check flag to be used as a feature boundary
    %termination check.
    check_cnt =-999;
    while (F(p).BoundaryList(b_cnt,1) ~= F(p).BoundaryList(2,1))|...
            (F(p).BoundaryList(b_cnt,2) ~= F(p).BoundaryList(2,2))|...
            (b_cnt ~= check_cnt + 1)
        %search around the boundary point center (r,c) beginning with the
        %last non-boundary point
        offset_r = Last_non_boundary(1) - r;
        offset_c = Last_non_boundary(2) - c;
        start_index = inv_Neigh8(offset_r,offset_c);
        for index = start_index+1:start_index+8
            [offset_r,offset_c] = Neigh8(index); %get offsets from (r,c)
            if (NewL(r+offset_r,c+offset_c) ~= p)
                continue;
            elseif (NewL(r+offset_r,c+offset_c) == p)
                b_cnt = b_cnt + 1;
                F(p).BoundaryList(b_cnt,1) = c+offset_c;
                F(p).BoundaryList(b_cnt,2) = r+offset_r;
                if (F(p).BoundaryList(b_cnt,1) == F(p).BoundaryList(1,1)) &...
                    (F(p).BoundaryList(b_cnt,2) == F(p).BoundaryList(1,2))
                    check_cnt = b_cnt;
                end
                %Get the point before the hit
                [back_off_r,back_off_c] = Neigh8(index-1);
                Last_non_boundary = [r+back_off_r, c+ back_off_c];
                %set up next boundary search center
                 r = F(p).BoundaryList(b_cnt,2);
                 c = F(p).BoundaryList(b_cnt,1);
                 break; %exit the local boundary search loop
            end
        end
    end
    %Delete duplicate points
    F(p).BoundaryList = unique(F(p).BoundaryList,'rows');
end
%To display the merged regions first form a label matrix with zeros at
%the feature boundaries
NewL_bdr = zeros(Nr,Nc);
NewL_bdr = NewL;
```

```
for p = 1:no_features
    ign = find(ignore == p);
    if ~isempty(ign)
        continue;
    end
    lFpB = length(F(p).BoundaryList(:,1));
    for q = 1:lFpB
        r = F(p).BoundaryList(q,2);
        c = F(p).BoundaryList(q,1);
        NewL_bdr(r,c) = 0;
    end
end

%*************************************************************************
function Post_processor(I2, NewL_bdr)
%POST_PROCESSOR - post-process feature boundary label matrix
%                 Overlay boundaries on cell image
%
%Input:
%   I2 - image pixels
%   NewL_bdr - feature label matrix with boundaries identified by zeros
%Output:
%   Figure with black feature boundaries overlaying the original image
%
%Author:
%   David E. Gilsinn
%   Mathematical and Computational Sciences Division
%   National Institute of Standards and Technology
%   100 Bureau Drive, Stop 8910
%   Gaithersburg, MD  20899-8910
%
%*************************************************************************
bdrs = (NewL_bdr == 0);
I2new = I2;
I2new(bdrs) = 1;
figure,imshow(I2new);
```