This article will discuss simple image segmentation based on histograms and image thresholding.

Image segmentation is the process of dividing an image into regions or objects. It is the first step in the field of image analysis. Image processing displays images and alters them to make them look "better," while image analysis tries to discover what is in the image.

The basic idea of image segmentation is to group individual pixels (dots in the image) together into regions if they are similar. Similar can mean they are the same intensity (shade of gray), form a texture, line up in a row, create a shape, etc. There are many techniques available for image segmentation, and they vary in complexity, power, and area of application.

Histogram-Based Image Segmentation

Histogram-based image segmentation is one of the simplest and most often used segmentation techniques. It uses the histogram to select the gray levels for grouping pixels into regions. In a simple image there are two entities: the background and the object. The background is generally one gray level and occupies most of the image. Therefore, its gray level is a large peak in the histogram. The object or subject of the image is another gray level, and its gray level is another, smaller peak in the histogram.

Figure 1 shows an image example and Figure 2 shows its histogram. The tall peak at gray level 2 indicates it is the primary gray level for the background of the image. The secondary peak in the histogram at gray level 8 indicates it is the primary gray level for the object in the image. Figure 3 shows the image of Figure 1 with all the pixels except the 8s blanked out. The object is a happy face.
Figure 1 An image example

2222232221222212222
32222321250132123132
2258889777788888232
12988877707668882122
2288892326669893213
2127821222666665222
22002222220266600225
212212312236662321
3223085222266821222
21288883428882232
223288889888522121
221239888889223422
232227888882022122
2232232383212123234
2522121222222222222
2212222320222202102
2022232241212223221
2222122222222224222
21222221222222142

Figure 2 A histogram of the image in Figure 1
Figure 3 The image in Figure 1 with all the pixels except the 8's blanked out

This illustrates histogram-based image segmentation. The histogram will show us the gray levels of the background and the object. The largest peak represents the background and the next largest peak the object. We choose a threshold point in the valley between the two peaks and threshold the image. Thresholding takes any pixel whose value is on the object side of the point and sets it to one; it sets all others to zero. The histogram peaks and the valley between them are the keys.

The idea of histogram-based segmentation is simple, but there can be problems. Where should you set the threshold point for the image of Figure 1? If you choose the point mid-way between the two peaks (threshold point=5), you produce the image of Figure 4. This is not the happy face object desired. If you choose the valley floor values of 4 or 5 as the threshold point, you also have a poor result. The best threshold point would be 7, but how could you know that without using trial and error?

This example is difficult because there are only ten gray levels and the object (happy face) is small. In practice, the techniques discussed below will perform adequately, but there will be problems. Automatic techniques will fail, and you may have to resort to manual methods.
Preprocessing: Histogram Equalization and Histogram Smoothing

Histogram-based segmentation depends on the histogram of the image. Therefore, you must prepare the image and its histogram before analyzing it. The first step is histogram equalization (Phillips, August 1991). Histogram equalization attempts to alter an image so its histogram is flat and spreads out over the entire range of gray levels. The result is an image with better contrast.

Photograph 1 shows an aerial image of several house trailers with its histogram. The contrast is poor and it would be very difficult to find objects based on its histogram. Photograph 2 shows the result of performing histogram equalization. The contrast is much better and the histogram is spread out over the entire range of gray levels. Photograph 3 shows the result of performing high-pass filtering (Phillips, October 1992) on the image of photograph 2. It is easy to see the house trailers, sidewalks, trees, bushes, gravel roads, and parking lots.
The next preprocessing step is histogram smoothing. When examining a histogram, you look at peaks and valleys. Too many tall, thin, peaks and deep valleys will cause problems. Smoothing the histogram removes these spikes and fills in empty canyons while retaining the same basic shape of the histogram.

Figure 5 shows the result of smoothing the histogram given in Figure 2. You can still see the peaks corresponding to the object and background, but the peaks are shorter.
and the valleys are filled. **Photograph 4** shows another example of histogram smoothing. The histogram on the left is the original with several spikes and canyons. The histogram on the right has been smoothed. I will analyze this in the segmentation.

**Figure 5** *The result of smoothing the histogram given in Figure 2*

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

---

**Photograph 4** *Image portion with histogram and smoothed histogram*

Smoothing a histogram is an easy operation. It replaces each point with the average of it and its two neighbors. **Listing 1** shows the `smooth_histogram` function that performs this operation.
/* Listing 1 */

smooth_histogram(histogram)
unsigned long histogram[];
{
    int i;
    unsigned long new_hist[GRAY_LEVELS+1];

    zero_histogram(new_hist);

    new_hist[0] = (histogram[0] + histogram[1])/2;
    new_hist[GRAY_LEVELS] =
        (histogram[GRAY_LEVELS] +
            histogram[GRAY_LEVELS-1])/2;

    for(i=1; i<GRAY_LEVELS; i++)
        new_hist[i](histogram[i-1] +
                        histogram[i] +
                        histogram[i+1])/3;

    for(i=0; i<=GRAY_LEVELS; i++)
        histogram[i] = new_hist[i];
}

smooth_histo...
Thresholding and Region Growing Methods

There are two more topics common to all the methods of image segmentation: image thresholding and region growing. Image thresholding sets the pixels in the image to one or zero. Listing 2 shows the routine `threshold_image_array` that accomplishes this task.

**Listing 2 Function for image thresholding**

```c
/********************************************
* threshold_image_array(...
* This function thresholds an input image array
* and produces a binary output image array.
* If the pixel in the input array is between
* the hi and low values, then it is set to value.
* Otherwise, it is set to 0.
* ********************************************/
threshold_image_array(in_image, out_image, hi, low, value)
short hi, low, in_image[ROWS][COLS],
out_image[ROWS][COLS], value;
{
    int counter = 0, i, j;
    for(i=0; i<ROWS; i++)
        for(j=0; j<COLS; j++){
            if(in_image[i][j] >= low &&
               in_image[i][j] <= hi){
                out_image[i][j] = value;
                counter++;
            }
        } /* ends loop over j */
    } /* ends loop over i */
    printf("\n\tTIA> set %d points", counter);
} /* End of threshold_image_array */
/* End of File */
```

The difficult task is region growing. Figure 6 shows the result of thresholding Figure 1 correctly. The "object" in Figure 6 is a happy face. It comprises three different regions (two eyes and the smile). Region growing takes this image, groups the pixels in each separate region, and gives them unique labels. Figure 7 shows the result of region growing performed on Figure 6. Region growing grouped and labeled one eye as region one, the other eye as region two, and the smile as region three.
Figure 7 The result of region growing performed on Figure 6

Figure 8 shows the algorithm for region growing. It begins with an image array $g$ comprising zeros and pixels set to a value. The algorithm loops through the image array looking for a $g(i,j) == value$. When it finds such a pixel, it calls the `label_and_check_neighbor` routine. `label_and_check_neighbor` sets the pixel to $g_label$ (the region label) and examines the pixel's eight neighbors. If any of the neighbors equal $value$, they are pushed onto a stack. When control returns to the main algorithm, each pixel on the stack is popped and sent to `label_and_check_neighbor`. All the points on the stack equaled $value$, so you set them and check their neighbors. After setting all the pixels in the first region, you increment $g_label$ and move on looking for the next region.
Figure 8 Pseudocode for region growing

1. Given an image g with m rows and n columns
   \[ g(i,j) \] for i=1,m j=1,n
   \[ g(i,j) = \text{value for object} \]
   \[ g(i,j) = 0 \text{ for background} \]
2. set g_label=2 this is the label value
3. for (i=0; i<m; i++)
   scan ith row
   for (j=0; j<n; j++)
     check jth element
     stack_empty = true
     if g(i,j) == value
       label_and_check_neighbor(g(i,j),g_label)
     while stack_empty = false do
       pop element (i,j) off the stack
       label_and_check_neighbor(g(i,j),g_label)
     end while
     g_label = g_label + 1
   end of checking jth element
   end of scanning ith row
4. The End

---------------------------------------
procedure label_and_check_neighbor(g(r,e), g_label)
  g(r,e) = g_label
  for (R=r-1; r<=r+1; R++)
    for (E=e-1; e<=e+1; e++)
      if g(R,E) == value then
        push (R,E) onto the stack
        stack_empty = false
      end if
    end loop over E
  end loop over R
end procedure label_and_check_neighbor

The grow and label_and_check_neighbor functions follow the region-growing algorithm step for step. The only unusual item is the stack. There are several ways to implement a stack. I used a simple array and a file. I did this because a region could be as large as \( ROWS \times COLS \) (10,000 in the C Image Processing System), and the stack must be large enough to hold that many points. I push points onto the stack array by putting them in the array and moving the top of stack pointer. When the stack array becomes full, I write it to the stack file. If the array fills again, I push the array onto the top of the stack file. I pop points off the stack array by reading them and decrementing the top of stack pointer. When the stack array is empty, I pop an array off the stack file.
Histogram-Based Segmentation Techniques

There are four segmentation techniques: the manual technique, histogram peak technique, histogram valley technique, and an adaptive technique.

1- The Manual Technique

In the manual technique the user inspects an image and its histogram manually. Trial and error come into play and the result is as good as you want it to be.

I used Photograph 4 as the input for all the segmentation examples. Photograph 5, Photograph 6, and Photograph 7 show the result of segmentation using three different thresholds. The result in Photograph 5 used a high of 255 and a low of 125. The segmentation included the white gravel roads as well as the house trailers and sidewalks. The result in Photograph 6 used a high of 255 and a low of 175. The gravel roads begin to disappear, but the house trailers and sidewalks remain. Photograph 7 shows the result using a high of 255 and a low of 225. This segmentation only finds the four dominant house trailers. Which answer is correct? That depends on what you wanted to find. Photograph 5 Threshold of Photograph 4 with high=225 and low=125

![Photograph 5](image)

Photograph 6 Threshold of Photograph 4 with high=255 and low=175

![Photograph 6](image)

Prepared by Dr Mohamed Berbar
Photograph 7 Threshold of Photograph 4 with high = 255 and low = 225

Note, all image segmentations will appear rough. You can perform additional processing to make the result more pleasing to the eye, but that is not the purpose of segmentation. The purpose is to break the image into pieces so later computer processing can interpret their meaning. The output is for computer not human consumption. Also note how difficult it is for the computer, even with manual aid, to find objects that are trivial for humans to see. Anyone could trace over the input image and outline the objects better than the segmentation process.

Listing 4 shows the code that implements manual segmentation. The function \texttt{manual_threshold_segmentation} has the same form as the other application functions in this series. It creates the output image file if it does not exist, reads in the input data, thresholds it (Listing 2), grows regions if requested (Listing 3), and writes the output.

**Listing 3 Functions that implement region growing**

```c
/*****************************
* grow(...
* This function is an object detector.
* Its input is an binary image array
* containing 0's and value's.
* It searches through the image and connects
* the adjacent values.
/*****************************/

/*****************************/
* Now begin the process of growing
* regions.
/*****************************/

/*****************************/
* Search for the first pixel of
* a region.
/*****************************/

/*****************************/
```

Prepared by Dr Mohamed Berbar
*/ If the stack is not empty,
* pop the coordinates of
* the pixel off the stack
* and check its 8 neighbors.
*******************************************************************************

/*********************
********/

/************************************
* label_and_check_neighbors(...
* This function labels a pixel with an object
* label and then checks the pixel's 8
* neighbors. If any of the neighbors are
* set, then they are also labeled.
*******************************************************************************/

/**********************
************

/************************************
* Look at the 8 neighbors of the
* point r,e.
* Ensure the points you are checking
* are in the image, i.e. not less
* than zero and not greater than
* ROWS-1 or COLS-1.
*******************************************************************************/

/************************************
* push_data_onto_stack_file(...
* This function takes the stack array
* and pushes it onto the stack file.
*******************************************************************************/

/************************************
* Copy the elements to be stored to the
* stack file into holder
*******************************************************************************/

/************************************
* Move the elements of the stack down
*******************************************************************************/

/************************************
* Fill the top of the stack with zeros
*******************************************************************************/

/************************************
* Store the holder array into the stack file.
* Open the stack file for writing in binary
* mode. If the file does not exist it will be
* created. If the file does exist it will be
* over written.
* PUSH - IF first_time == 1 then write to stack
* ELSE write to stack.bak
* append stack onto stack.bak
* copy stack.bak to stack
* this has the effect of writing
* to the beginning of the stack.
Listing 4 Code implementing manual segmentation

    /*******************************************************************************/
    /* manual_threshold_segmentation(...) */
    /* This function segments an image using thresholding */
    /* given the hi and low values of the threshold */
    /* by the calling routine. It reads in an image */
    /*******************************************************************************/
* and writes the result to the output image.
* If the segment parameter is 0, you only
* threshold the array - you do not segment.
**********************************************************/

manual_threshold_segmentation(in_name, out_name,
     the_image, out_image,
     il, ie, ll, le,
     hi, low, value, segment)

char    in_name[], out_name[];
int     il, ie, ll, le, segment;
short   hi, low, the_image[ROWS][COLS],
        out_image[ROWS][COLS], value;
{
int     length, width;
struct tiff_header_struct image_header;

if(does_not_exist(out_name)){
    printf("\n\nMTS> output file does not exist %s", out_name);
    read_tiff_header(in_name, &image_header);
    round_off_image_size(&image_header, &length, &width);
    image_header.image_length = length*ROWS;
    image_header.image_width  = width*COLS;
    create_allocate_tiff_file(out_name, &image_header, out_image);
}  /* ends if does_not_exist */

read_tiff_image(in_name, the_image, il, ie, ll, le);
threshold_image_array(the_image, out_image, hi, low, value);
if(segment == 1)
    grow(out_image, value);
write_array_into_tiff_image(out_name, out_image, il, ie, ll, le);
}  /* ends manual_threshold_segmentation */

/* End of File */

manual_threshold_segmentation  has the usual inputs (image file names, image arrays,
etc.) as well as the high and low threshold values, and the value and segment parameters.
The value parameter specifies the value at which to set a pixel if it falls between the high
and low thresholds. You usually set value equal to 1 since those pixels outside the high-
low range are set to zero. The segment parameter specifies whether or not to grow regions
after thresholding. Sometimes you only want to threshold an image and not grow regions.
The two operations are identical except for the last step. If segment  == 1, you call the
region-growing routines.

Manual segmentation is good for fine tuning and getting a feel for the operation. Its trial-
and-error nature, however, makes it time consuming and impractical for many
applications. You need techniques that examine the histogram and select threshold values
automatically.

Prepared by Dr Mohamed Berbar
2- Histogram Peak Technique

The first such technique uses the peaks of the histogram. This technique finds the two peaks in the histogram corresponding to the background and object of the image. It sets the threshold halfway between the two peaks. Look back at the smoothed histogram in Figure 5. The background peak is at 2 and the object peak is at 7. The midpoint is 4, so the low threshold value is 4 and the high is 9.

The peaks technique is straightforward except for two items. In the histogram in Figure 5, you'll note the peak at 7 is the fourth highest peak. The peaks at 1 and 3 are higher, but they are part of the background mountain of the histogram and do not correspond to the object. When you search the histogram for peaks, you must use a peak spacing to ensure the highest peaks are separated. If you did not, then you would choose 2 as the background peak and 1 as the object peak. Figure 9 shows the disastrous effect of this.

The second item to watch carefully is determining which peak corresponds to the background and which corresponds to the object. Suppose an image had the histogram shown in Figure 10. Which peak corresponds to the background? The peak for gray level 8 is the highest, but it corresponds to the object not the background. The reason is the mountain surrounding the peak at gray level 2 has a much greater area than that next to gray level 8. Therefore, gray levels 0 through 6 occupy the vast majority of the image, and they are the background.

Listing 5 shows the source code to implement the peaks technique. peak_threshold_segmentation is the primary function. It checks for the existence of the output image, reads the input image, and calculates and smoothes the histogram. Next, it calls new functions to find the histogram peaks and determine the high and low threshold values. Finally, it thresholds the image, performs region growing if desired, and writes the result image to the output file.

The functions find_peaks and insert_into_peaks in Listing 5 analyze the histogram to find the peaks for the object and background. These functions build a list of histogram peaks. There are several ways to do this. I used an array of values. find_peaks loops through the histogram and calls insert_into_peaks, which puts the histogram values in the proper place in the array. find_peaks ends by looking at the spacing between the largest peaks to ensure we do not have a disaster such as shown in Figure 9.

The function peaks_high_low takes the two peaks from find_peaks and calculates the high- and low-threshold values for segmentation. peaks_high_low examines the mountains next to the peaks as illustrated in Figure 10. It then finds the mid-point between the peaks and sets the high and low threshold values.

Photograph 8 shows the result of applying the peaks technique to the image of Photograph 4. The peaks technique found the two peaks at 255 and 77. The mid-point is 166, so the high threshold is 255 and the low threshold is 166. This is a reasonably good segmentation of Photograph 4.
3- Histogram Valley Technique

The second automatic technique uses the peaks of the histogram, but concentrates on the valley between them. Instead of setting the mid-point arbitrarily half way between the two peaks, the valley technique searches between the two peaks to find the lowest valley.

Look back at the histogram of Figure 10. The peaks are at gray levels 2 and 8 and the peaks technique would set the midpoint at 5. In contrast, the valley technique searches from 2 through 8 to find the lowest valley. In this case, the "valleypoint" is at gray level 7.

Listing 6 shows the code that implements the valley technique. The primary function is valley_threshold_segmentation. It checks for the output file, reads the input image, calculates and smooths the histogram, and finds the peaks as peak_threshold_segmentation did. It finds the valley-point via the functions valley_high_low, find_valley_point, and insert_into_deltas, find_valley_point starts at one peak and goes to the next inserting the histogram values into a deltas array via the insert_into_deltas function. This uses an array to create a list of deltas in the same manner as insert_into_peaks did in Listing 5. Once you have the valleypoint, valley_high_low checks the mountains around the peaks to ensure you associate the peaks with the background and object correctly.

Photograph 9 shows the result of applying the valley technique to the image in Photograph 4. It found the peaks at 77 and 255 and went from 77 up to 255 looking for the lowest valley. It pinpointed the lowest valley at gray level 241.

4- Adaptive Histogram Technique

The final technique uses the peaks of the histogram in a first pass and adapts itself to the objects found in the image in a second pass (Castleman 1979). In the first pass, the adaptive technique calculates the histogram for the entire image. It smoothes the histogram and uses the peaks technique to find the high and low threshold values.

In the second pass, the technique works on each ROWSxCOLS area of the image individually. In each area, it segments using the high and low values found during the first pass. Then, it calculates the mean value for all the pixels segmented into background and object. It uses these means as new peaks and calculates new high and low threshold values for that ROWSxCOLS area. Now, it segments that area again using the new values.

Listing 7 shows the code that implements the adaptive technique with adaptive_threshold_segmentation being the primary function. It is very similar to the peak_threshold_segmentation function of Listing 5 in that it uses all that code for its first pass. The second pass starts by calling threshold_and_find_means. This function thresholds the image array into background and object and calculates the mean pixel value for each. The second pass continues by using peaks_high_low to find new threshold value.

Prepared by Dr Mohamed Berbar
values based on the background and object means. Finally, you threshold the image using these new threshold values.

Photograph 10 shows the result of applying the adaptive technique to the image of Photograph 4. The first pass found the high- and low-threshold values to be 255 and 166. On the left side of the photograph, the second pass thresholded the image array and found the background mean to be 94 and the object mean to be 205. The new threshold values were 255 and 149. On the right side of the photograph, the background and object means were 84 and 200 and the new threshold values were 255 and 142.

**Integrating Histogram-Based Segmentation into the C Image Processing System**

Listing 8 shows the new code for the main routine of CIPS. I've added the options of thresholding and segmentation using the four techniques discussed above in cases 16 and 17. Listed next are the changes to the CIPS main menu and the two functions that interact with the user to obtain the processing options.

Listing 9 shows a stand-alone application program for thresholding and segmenting entire image files. It is command-line driven and calls the functions shown in the earlier listings.

**Conclusions**

This installment in the series introduced image segmentation. This is the first step in locating and labeling the contents of an image. The techniques discussed work on simple images with good contrast and gray level separation between the object and background. You will need other techniques to attack more complex images.

**References**


**Get Article Source Code**
Listing 3 Functions that implement region growing

/****************************/
* grow(...)                *
* This function is an object detector. *
* Its input is an binary image array *
* containing 0's and value's.  *
* It searches through the image and connects   *
* the adjacent values.                        *
****************************************************************************/

grow(binary, value)
short binary[ROWS][COLS],
    value;
{
    char name[80];

    int first_call,
        i,
        j,
        object_found,
        pointer,
        pop_i,
        pop_j,
        stack_empty,
        stack_file_in_use;

    short g_label, stack[STACK_SIZE][2];

    /***********************************
    * Now begin the process of growing
    * regions.                         
    *************************************/

    g_label        = 2;
    object_found  = 0;
    first_call    = 1;

    for(i=0; i<ROWS; i++){
        for(j=0; j<COLS; j++){
            stack file in use = 0;
            stack_empty = 1;
            pointer = -1;

            /*******************************
            * Search for the first pixel of
            * a region.                     
            *******************************/

            if(binary[i][j] == value)(
                label_and_check_neighbor(binary, stack, g_label,
                                            &stack_empty, &pointer, i, j,
                                            value, &stack_file_in_use,
                                            &first_call);
                object_found = 1;  
            )  /* ends if binary[i][j] == value */

            } /* ends for j<COLS; j++*/
        } /* ends for i<ROWS; i++*/
    } /* ends for i<ROWS; i++*/

Prepared by Dr Mohamed Berbar
/* If the stack is not empty, */
/* pop the coordinates of */
/* the pixel off the stack */
/* and check its 8 neighbors. */
*******************************************************************************/

while(stack_empty == 0){
    pop_i = stack[pointer][0]; /* POP */
    pop_j = stack[pointer][1]; /* OPERATION */
    --pointer;
    if(pointer <= 0){
        if(stack_file_in_use){
            pop_data_off_of_stack_file(
                stack,
                &pointer,
                &stack_file_in_use);
        } /* ends if stack_file_in_use */
        else{
            pointer  = 0;
            stack_empty = 1;
        } /* ends else stack file is not in use */
    } /* ends if point <= 0 */
    label_and_check_neighbor(binary,
                            stack, g_label,
                            &stack_empty,
                            &pointer, pop_i,
                            pop_j, value,
                            &stack_file_in_use,
                            &first_call);
} /* ends while stack_empty == 0 */

if(object_found == 1){
    object_found = 0;
    ++g_label;
} /* ends if object_found == 1 */

} /* ends loop over j */
} /* ends loop over i */

printf("\nGROW> found %d objects", g_label);

} /* ends grow */

*******************************************************************************/
/* label_and_check_neighbors(...) */
/* This function labels a pixel with an object */
/* label and then checks the pixel's 8 */
/* neighbors. If any of the neighbors are */
/* set, then they are also labeled. */
*******************************************************************************/

label_and_check_neighbor(binary_image, stack,
                            g_label, stack_empty,
                            pointer, r, e, value,
int e,
    *first_call,
    *pointer,
    r,
    *stack_empty,
    *stack_file_in_use;

short binary_image [ROWS] [COLS],
    g_label,
    stack[STACK_SIZE][2],
    value;
{
int already_labeled = 0,
    i, j;

if (binary_image[r][e] == g_label)
    already_labeled = 1;

binary_image[r][e] = g_label;

/****************************************************
* Look at the 8 neighbors of the
* point r,e.
* Ensure the points you are checking
* are in the image, i.e. not less
* than zero and not greater than
* ROWS-1 or COLS-1.
****************************************************/

for(i=(r-1); i<=(r+1); i++){
    for(j=(e-1); j<=(e+1); j++){

        if((i>=0) && (i<=ROWS-1) && (j>=0) && (j<=COLS-1)) {

            if(binary_image[i][j] == value){
                *pointer = *pointer + 1;
                stack[*pointer][0] = i; /* PUSH */
                stack[*pointer][1] = j; /* OPERATION */
                *stack_empty = 0;

                if(*pointer >= (STACK_SIZE - STACK_FILE_LENGTH)){
                    push_data_onto_stack_file(stack, pointer,
                        first_call);
                    *stack_file_in_use = 1;
                } /* ends if *pointer >= STACK_SIZE -
                    STACK_FILE_LENGTH*/

            } /* end of if binary_image == value */
        } /* end if i and j are on the image */
    } /* ends loop over i rows */
} /* ends loop over j columns */
} /* ends label_and_check_neighbors */

/****************************************************
* push_data_onto_stack_file(...
* This function takes the stack array
* and pushes it onto the stack file.
*****************************************************************************************/

push_data_onto_stack_file(stack, pointer, first_call)
int *first_call, *pointer;
short stack[STACK_SIZE][2];
{
    char backup_file_name[MAX_NAME_LENGTH];
    FILE *backup_file_pointer, *stack_file_pointer;
    int diff, i;
    short holder[STACK_FILE_LENGTH][2];

    printf("\nSFO> Start of push_data_onto_stack ");

    diff = STACK_SIZE - STACK_FILE_LENGTH;

        /***************************************************************************/
        * Copy the elements to be stored to the
        * stack file into holder
        ***************************************************************************/

    for(i=0; i<STACK_FILE_LENGTH; i++)
    {
        holder[i][0] = stack[i][0];
        holder[i][1] = stack[i][1];
    }

        /***************************************************************************/
        * Move the elements of the stack down
        ***************************************************************************/

    for(i=0; i<diff; i++)
    {
        stack[i][0] = stack[i + STACK_FILE_LENGTH][0];
        stack[i][1] = stack[i + STACK_FILE_LENGTH][1];
    }

        /***************************************************************************/
        * Fill the top of the stack with zeros
        ***************************************************************************/

    for(i=diff; i<STACK_SIZE; i++)
    {
        stack[i][0] = 0;
        stack[i][1] = 0;
    }

    *pointer = *pointer - STACK_FILE_LENGTH;

        /***************************************************************************/
        * Store the holder array into the stack file.
        * Open the stack file for writing in binary
        * mode. If the file does not exist it will be
        * created. If the file does exist it will be
        * over written.
        * PUSH - IF first_time == 1 then write to stack
        * ELSE write to stack.bak
        * append stack onto stack.bak
        * copy stack.bak to stack
        * this has the effect of writing

Prepared by Dr Mohamed Berbar
* to the beginning of the stack.

if(*first_call == 1){
        *first_call = *first_call + 1;
        if((stack_file_pointer = fopen(STACK_FILE, "wb"))
          == NULL)
                printf("\n\nSFO> Could not open stack file");
          else{
                /*printf("\n\nSFO> Writing to stack file");*/
                fwrite(holder, sizeof(holder), 1, stack_file_pointer);
                fclose(stack_file_pointer);
        } /* ends else could not open stack_file */
}
else{ /* else stack file has been used already */
        strcpy(backup_file_name, STACK_FILE);
        append_string( ".bak\0", backup_file_name);
        if((backup_file_pointer =
          fopen(backup_file_name, "wb")) == NULL)
                printf("\n\nSFO> Could not open backup file");
          else{
                /*printf("\n\nSFO> Writing to backup file");*/
                fwrite(holder, sizeof(holder), 1, backup_file_pointer);
                fclose(backup_file_pointer);
        } /* ends else could not open backup_file */
        append_stack_files(backup_file_name, STACK_FILE, holder);
        copy_stack_files(backup_file_name, STACK_FILE, holder);
}
else{ /* else first_call != 1 */
        printf("--- End of push_data_onto_stack_file");
}

pop_data_off_of_stack_file(stack, pointer, stack_file_in_use)
int  *pointer, *stack_file_in_use;
short stack[STACK_SIZE][2];
{
        char  backup_file_name[MAX_NAME_LENGTH];
        FILE  *backup_file_pointer, *stack_file_pointer;
        int  i;
        long  write_counter;
        short holder[STACK_FILE_LENGTH][2],
                holder2[STACK_FILE_LENGTH][2];

        /*-----------------------------*/
        * POP - Read 1 time from stack
        * Copy the remainder of stack to
* stack.bak
* Copy stack.bak to stack
* This has the effect of popping off
* of the stack.
* Read the holder array from the stack file.
* Open the stack file for reading in binary
* mode.
* If it requires more than one write to
* copy the remainder of stack to
* stack.bak then there is still data in the
* stack file so set stack_file_in_use = 1.
* Else set it to 0.
* ****************************

printf("\nSFO> Start of pop_data_off_of_stack ");
write_counter = 0;

strcpy(backup_file_name, STACK_FILE);
append_string(".bak\0", backup_file_name);

if( (stack_file_pointer = fopen(STACK_FILE, "rb")) == NULL)
 printf("\nSFO> Could not open stack file");
else{
    /*printf("\n\nSFO> Reading from stack file");*/
    fread(holder, sizeof(holder),
    1, stack_file_pointer);
    backup_file_pointer =
        fopen(backup_file_name, "wb");
    while( fread(holder2, sizeof(holder2),
    1, stack_file_pointer) ){
        fwrite(holder2, sizeof(holder2),
        1, backup_file_pointer);
        ++write_counter;
    } /* ends while reading */
    if(write_counter > 0)
        *stack_file_in_use = 1;
    else
        *stack_file_in_use = 0;
    fclose(backup_file_pointer);
    fclose(stack_file_pointer);
} /* ends else could not open stack file */

copy_stack_files(backup_file_name,
    STACK_FILE, holder2);

for(i=0; i<STACK_FILE_LENGTH; i++){
    stack[i][0] = holder[i][0];
    stack[i][1] = holder[i][1];
}
*pointer = *pointer + STACK_FILE_LENGTH - 1;

printf("--- End of pop_data_off_of_stack");
} /* ends pop_data_off_of_stack_file */

/****************************************************
* append_stack_files(...

Prepared by Dr Mohamed Berbar
* Append the second file onto the end of the first.
***********************************************

append_stack_files(first_file, second_file, holder)
char first_file[], second_file[];
short holder[STACK_FILE_LENGTH][2];
{
  FILE  *first, *second;
  int  i;

  if((first = fopen(first_file, "r+b")) == NULL)
    printf("\n\nSFO> Cannot open file %s",
          first_file);

  if((second = fopen(second_file, "rb")) == NULL)
    printf("\n\nSFO> Cannot open file %s",
          second_file);

    /* Seek to the end of the first file and to the beginning of the second file. */

  fseek(first, OL, 2);
  fseek(second, OL, 0);

  while(fread(holder, sizeof(holder), 1, second) )
  {
    fwrite(holder, sizeof(holder), 1, first);
  }  /* ends while reading */

  fclose(first);
  fclose(second);
}
  
}  /*  ends append_stack_files */

/******************************************
* copy_stack_files...
* Copy the first file to the second.
******************************************/

copy_stack_files(first_file, second_file, holder)
char first_file[], second_file[];
short holder[STACK_FILE_LENGTH][2];
{
  FILE  *first, *second;
  int  i;

  if( (first = fopen(first_file, "rb")) == NULL)
    printf("\n\nSFO> Cannot open file %s",
          first_file);

  if( (second = fopen(second_file, "wb")) == NULL)
    printf("\n\nSFO> Cannot open file %s",
          second_file);

    /* Seek to the beginning of the first file. */

Prepared by Dr Mohamed Berbar
fseek(first, 0L, 0);
while( fread(holder, sizeof(holder), 1, first) ){  
    fwrite(holder, sizeof(holder), 1, second);
}  /* ends while reading */
fclose(first);
fclose(second);
}  /* ends copy_stack_files */
/* End of File */

Listing 4 Code implementing manual segmentation

/*******************************************************************************/
* manual_threshold_segmentation(...
* This function segments an image using thresholding
* given the hi and low values of the threshold
* by the calling routine. It reads in an image
* and writes the result to the output image.
* If the segment parameter is 0, you only
* threshold the array - you do not segment.
*******************************************************************************/
manual_threshold_segmentation(in_name, out_name,
    the_image, out_image,
    il, ie, ll, le,
    hi, low, value, segment)
char     in_name[], out_name[];
int     il, ie, ll, le, segment;
short   hi, low, the_image[ROWS][COLS],
        out_image[ROWS][COLS], value;
{
    int     length, width;
    struct tiff_header_struct image_header;
    if(does_not_exist(out_name)){
        printf("\n\nMTS> output file does not exist %s", out_name);
        read_tiff_header(in_name, &image_header);
        round_off_image_size(&image_header, &length, &width);
        image_header.image_length = length*ROWS;
        image_header.image_width  = width*COLS;
        create_allocate_tiff_file(out_name, &image_header, out_image);
    }  /* ends if does_not_exist */
    read_tiff_image(in_name, the_image, il, ie, ll, le);
    threshold_image_array(the_image, out_image, hi, low, value);
    if(segment == 1)
        grow(out_image, value);
    write_array_into_tiff_image(out_name, out_image, il, ie, ll, le);
}  /* ends manual_threshold_segmentation */

Prepared by Dr Mohamed Berbar