## Chapter 6 -Segmentation

### Segmentation is the first step in the analysis or automatic interpretation of an image.

### It partitions an image into distinct regions that contain objects or features of interest.

### Segmentation can be regarded as the process of grouping together pixels that have similar attributes.

* Segmentation is the intermediate stage between low level and high level image processing tasks. The former manipulates pixel values to correct defects or enhance the image, whereas the latter manipulates and analyses a group of pixels that represent a particular feature of interest.

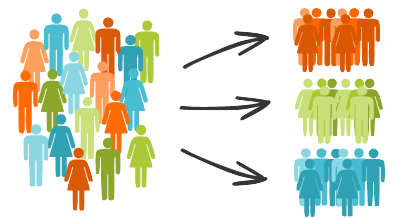


Fig. 6.1 Examples of segmentation

* Segmentation is used in a variety of applications such as
* Industrial inspection
* Optical character recognition
* Tracking of objects in a sequence of images
* Classification of terrain in remote sensing situations
* Detection and measurement of bone, tissue, etc. in medical image



Fig 6.2 Example of segmentation to identify people and bicycles



Segmentation techniques categories:

* based purely pixel values f(x,y)
* combination of pixel values and their locations; f(x,y) and (x,y).

**6.1 Thresholding**

This technique belongs to the first category, and is used in a variety of image processing operations.

* Thresholding: classifies pixels into two groups
* those at which some property (e.g. gray level) falls below a threshold,
* those at which the property equals or exceeds the threshold.

Because there are two outcomes, the resulting output image will be a binary image.

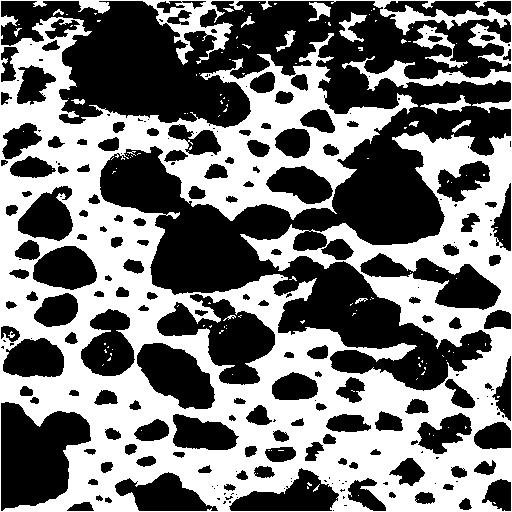
* In thresholding, the output image g(x,y) is obtained from the input image as

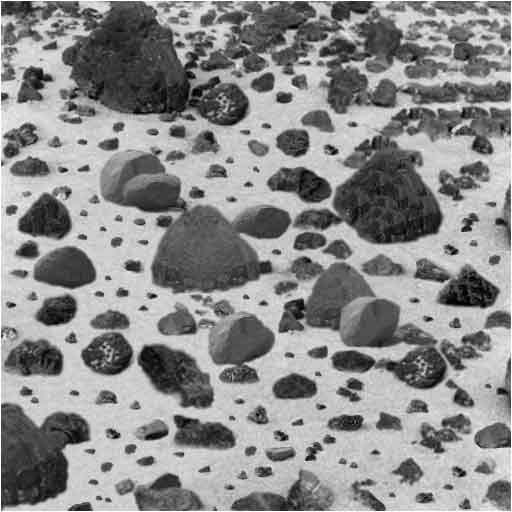
 (6.1)

Note: that thresholding can be performed in-place.

Thesholding T = 170, Lo = 0, Hi = 255 applied to detect rocks from the background







1. (b)

Fig. 6.3 Applying thresholding to a Mars terrain image to detect rocks.

In Matlab:

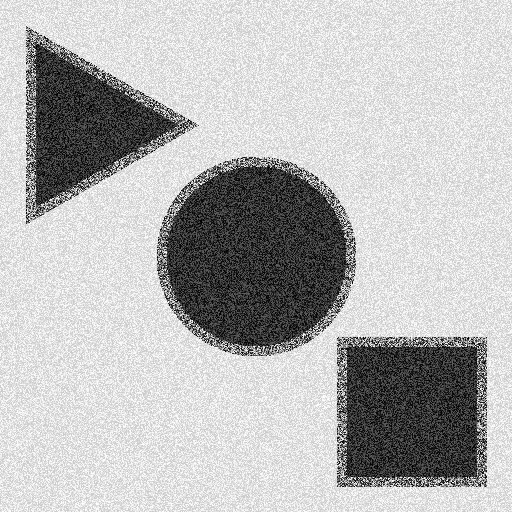
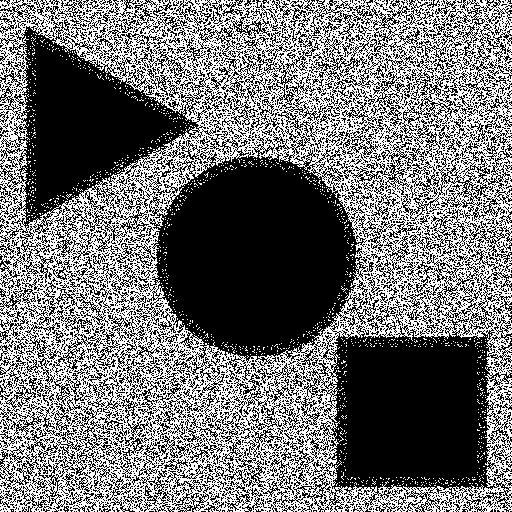
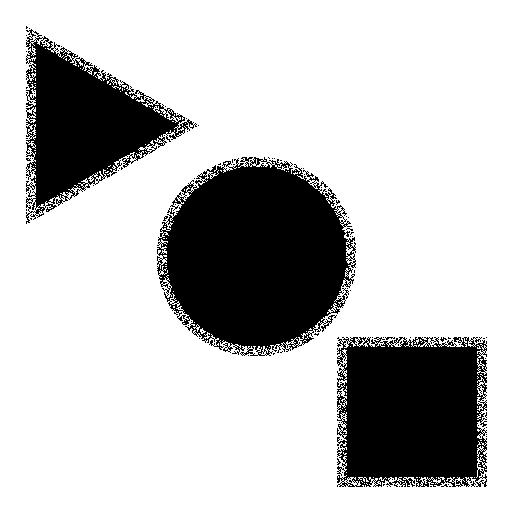
*>> rocks=imread(‘Mars.Rocks’);*

*>> imshow(rocks); figure, imshow(rocks(<128))*

* We can use two thresholds to define an acceptable range for the object to be detected. This is written as

 (6.2)

* The success of thresholding depends critically on the selection of the threshold.

Original Image T= 220 T = 130

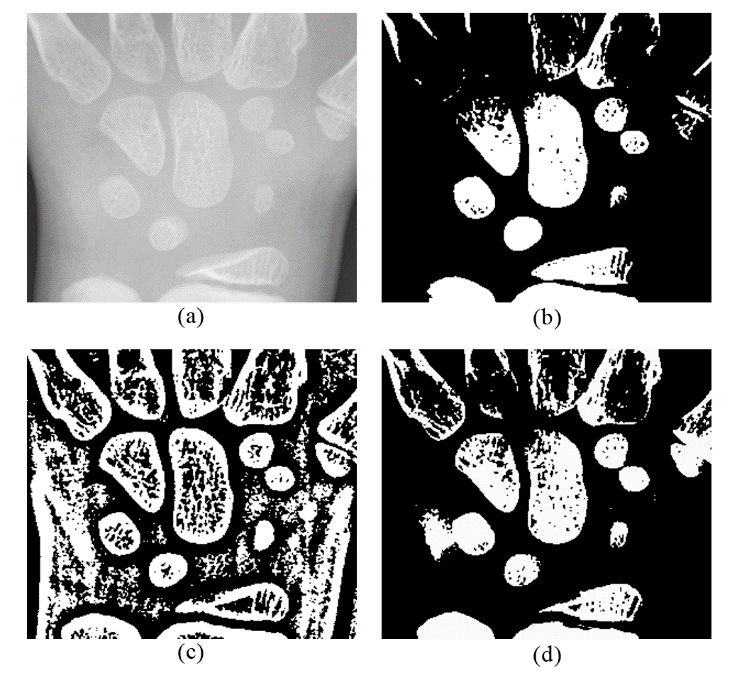


Fig. 6.4 The effects of various thresholding values

* For automatic determination of the threshold, one simple approach is to make T equal to the mean gray level of the image.
* The idea is that mean lies between the pixel values that belong to the region (objects of interest) and the background (other things not of interest).
* This approach is effective for images that have bright objects on a simple, dark background, or vice versa.
* A more sophisticated approach is to base the threshold value(s) on histogram analysis. If the histogram has distinct picks and valleys, then we may distinguish between two features of different gray level by thresholding at a point between histogram peaks corresponding to these features.

###### H

T1 T2

f

Fig. 6.5 Determining threshold values from the histogram.

Note: histogram segments overlap, and peaks/valleys are in general not so distinct due to smoothly varying gray level or color values in an object.

* To partly remedy this situation, an iterative method for automatic threshold selection can be used, as in the following algorithm. The method start with an initial guess of the threshold and improves this estimate by successive passes through the image.
* The initial value is usually found as the average of the gray level values of corner pixels which are assumed to be the background, and the average of all other pixels. Only few iterations are needed for the algorithm to converge to the final threshold.

**Iterative Threshold Selection**

#### Compute m1, the mean gray level of the corner pixels

#### Compute m2, the mean gray level of other pixels

Told = 0

Tnew = (m1+m2)/2

while (Tnew ! = Told)  **{**

m1 = mean gray level of pixels for which f(x,y) < Tnew

m2 = mean gray level of pixels for which f(x,y) >= Tnew

Told = Tnew

Tnew = (m1+ m2)/2

}

**Thresholding of Color Images**

The simplest approach to color images is to define three thresholds, one for each color: 

###### Rectangloid Thresholding

A particular colors in an RGB image can be visualized as a point in the 3-D color space. A blue threshold, for example, can be visualized as a plane perpendicular to the B axis and parallel to the R-G plane. Thus the three thresholds divide the RGB space into 8 rectangloid volumes, only one of which belongs to the object.

R



 G

B

Fig. 6.6 Thresholding in color space.

Fig. 6.7 shows thresholding where the red components in excess of 150 are set to white indicating the feature of interest while the other two components are set to black meaning that any blue or green component is set to zero.

(a) (b)

Fig 6.7 The original image, and detected region using color thresholding

**Spherical Thresholding:**

Another approach to thresholding is to define a spherical region in the RGB plane, instead of the above rectangloid regions. The center of the sphere is specified by the point , and the radius by. Any value within this sphere is set to a low value (say black), and outside the sphere is set to a high value (say white). This procedure is therefore formulated as

 (6.3)

where ,

and is the red component of the pixel at location x,y, and similarly the other two color components are defined. Fig. 6.8 show the results of this thresholding on the fish image of Fig. 6.7a with and =170.



### Fig. 6.8 The results of distance thresholding on Fig. 6.7a

Note that we can generalize (6.3) by defining different thresholds on each RGB component. In this case the defined volume becomes an ellipsoid in the RGB space.

Finally, we can employ histogram for as an aid to determine the threshold for each of the RGB components. Since 3-D histograms are difficult to visualize, the 2-D projections, e.g. in R-G plane, can be used for this purpose. The procedure is then to determine peaks and valleys as before in these 2-D planes.

**6.2 Canny Edge Detection and Linking**

The Canny Edge Detector is one of the most commonly used image processing tools, detecting edges in a very robust manner. It is a multi-step process, which can be implemented as a sequence of filters. The result can be used to detect regions in an image.

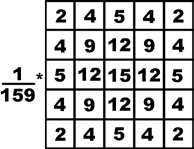
The following steps are also outlined in various online articles.

**Step 0: Convert to Grayscale**

There is no reason why you could not do Canny edge detection on a color image, but first convert the image to grayscale using some sort of RGB→grayscale.

**Step 1: Noise Reduction**

Usually noise reduction implies some sort of blurring operation. Most people use a Gaussian filter to do this. One suggestion is to use the following 5 × 5 filter:



Note that this 5 × 5 filter is roughly equivalent to the Gaussian filter. Note 15+4(12+9+4+4+5+2)= 159, thus (1/159) is a normalization.

**Step 2: Compute Gradient Magnitude and Angle**

Compute the gradients  and of the image in the x and y directions, using the Sobel mask. Then find the magnitude and phase as

 ; 

Round  to one of four directions 0◦, 45◦, 90◦, or 135◦. Obviously for edges, 180◦ = 0◦, 225◦ = 45◦, etc. this means in the ranges [−22.5◦...22.5◦] and [157.5◦ ...202.5◦ ] would “round” to 0◦. For a pictorial representation, each edge take on one of four colors: Here, the colors would repeat on the lower half of the circle

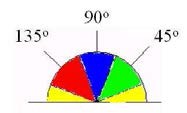


Fig. 6.9 The discretization of angles. Green around 225◦, blue around 270◦, and red around 315◦ .

**Step 3: Non-Maximum Suppression**

When using the Sobel filter, the edges it finds can be either very thick or very narrow depending on the intensity across the edge and how much the image was blurred first. One would like to have edges that are only one pixel wide. The “non-maximal suppression” step keeps only those pixels on an edge with the highest gradient magnitude. These maximal magnitudes should occur right at the edge boundary, and the gradient magnitude should fall off with distance from the edge.

|  |  |  |
| --- | --- | --- |
| (x-1,y-1) | (x-1,y) | (x-1,y+1) |
| (x,y-1) | (x,y) | (x,y+1) |
| (x+1,y-1) | (x+1,y) | (x+1,y+1) |

M(-1,-1) M(-1,0) M(-1,1)

M(0,-1) M(0,0) M(0,1)

M(1,-1) M(1,0) M(1,1) M(1,1)

M(-1,1) M(0,1) M(1,1)

M(-1,-1) M(-1,0) M(-1,1)

M(0,-1) M(0,0) M(0,1)

M(1,-1) M(1,0) M(1,1) M(1,1)

M(-1,1) M(0,1) M(1,1)

g(x,y)

So, three pixels in a 3 × 3 area around pixel (x, y) are examined

* If = 0◦ , then the pixels (x + 1, y), (x, y), and (x − 1, y) are examined.
* If = 90◦, then the pixels (x, y + 1), (x, y), and (x, y − 1) are examined.
* If = 45◦, then the pixels (x + 1, y + 1), (x, y), and (x − 1, y − 1) are examined.
* If = 135◦, then the pixels (x + 1, y − 1), (x, y), and (x − 1, y + 1) are examined.

If pixel (x, y) has the highest gradient magnitude of the three pixels examined, it is kept as an edge. If one of the other two pixels has a higher gradient magnitude, then pixel (x, y) is not on the “center” of the edge and will not be classified as an edge pixel.

**Step 4: Hysteresis Thresholding**

Some of the edges detected by Steps 1–3 will not actually be valid, but will just be noise. We would like to filter this noise out. Eliminating pixels whose gradient magnitude *g* falls below some threshold removes the worst of this problem, but it introduces a new problem.

A simple threshold may actually remove valid parts of a connected edge, leaving a disconnected final edge image. This happens in regions where the edge’s gradient magnitude fluctuates between just above and just below the threshold. *Hysteresis* is one way of solving this problem. Instead of choosing a single threshold, two thresholds  and  are used. Pixels with a gradient magnitude *g* <  are discarded immediately. However, pixels with < *g* <  are only kept if they form a continuous edge line with pixels with high gradient magnitude (i.e. ). This is a little hard to implement. However, one can implement a partially correct version:

* If pixel (x, y) has gradient magnitude *g* <  discard the edge (write out black).
* If pixel (x, y) has gradient magnitude *g* >  keep the edge (write out white).
* If pixel (x, y) has gradient magnitude between < *g* <  and any of its neighbors in a 3 × 3 region around it have gradient magnitudes greater than , keep the edge (write out white).
* If none of pixel (x, y)’s neighbors have high gradient magnitudes but at least one falls between < *g* <  , search the 5 × 5 region to see if any of these pixels have a magnitude *g* > . If so, keep the edge (write out white). Else, discard the edge (write out black). The following function applies the Canny edge detector to the image Im, specifies the Canny method, using Sigma as the standard deviation of the Gaussian filter. The default of Sigma is sqrt(2); the size of the filter is chosen automatically, based Sigma.

*>> edge(Im,'canny',Thresh,Sigma)*

The method can be used without choosing Threshold and Sigma as

*>> edge(Im,'canny')*

**Example**



1. Original Image (b) Sobel Edge Detector



(c ) Laplacian after Gaussinan filtering (d) Canny Edge Detector

Fig. 6.10 Canny edge deytection stages

**Example 2**



1. Original (b) Grayleveled and smoothed (c) Gradient Image



(d) Non-maximum suppressed (e) Final

Fig. 6.11 An example of Canny edge detector procedure

**6.3 Hough Transform for Region Detection**

Hough transform allows finding regions that can be described by mathematical relationships, such as lines, circles and elipses.

Consider a set of points on a line in x-y plane (image space) described by



The above line can be described in the parameter space as



A point say in image space corresponds to line in parameter space

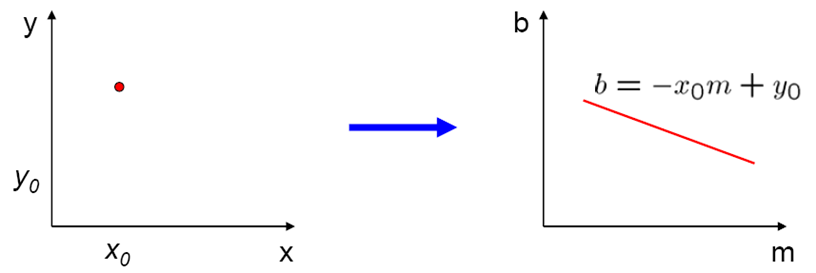
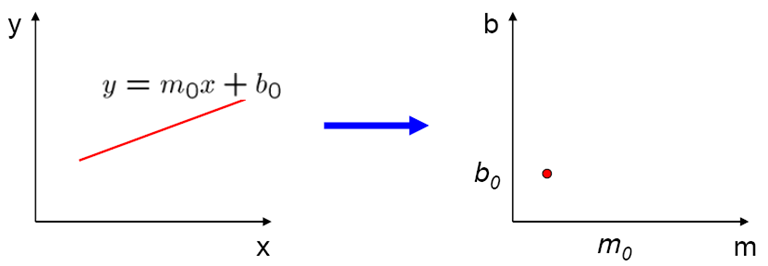


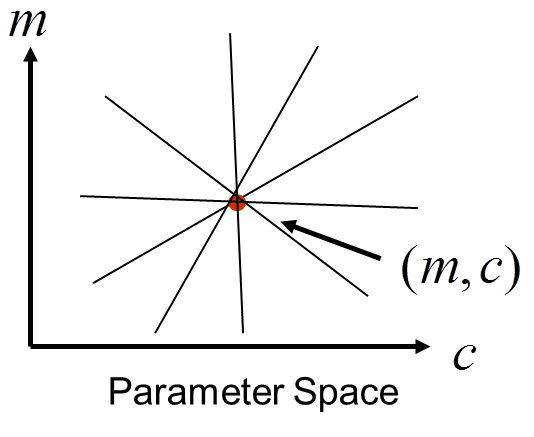
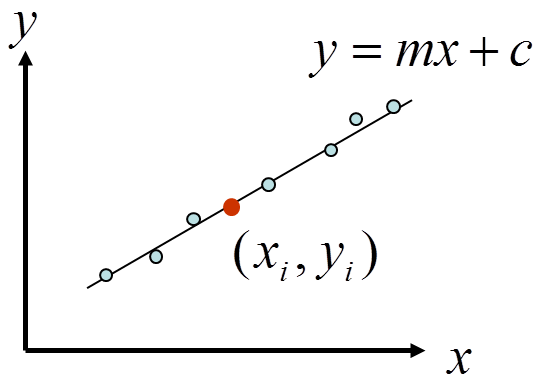
Fig. 6.12 Mapping of a point from x-y space into Hough space

Similarly a point in parameter space maps into a line in x-y space



6.13 Mapping of a line into Hough space

Now consider multiple points in (x-y) space that map into multiple lines in Hough space.

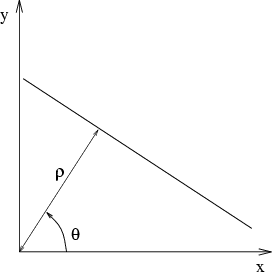


6.14 Mapping of a points from (x-y) space into Hough space. Note *c=b*

There are two problems with the line description in the form of  (or ).

* The parameters are unbounded, e.g. *m* and *b* can be very large positive or negative.
* Vertical lines require infinite *m*.

Alternative representation is polar coordinates: 



Therefore the parameter space is now . A vertical line has . And both  and  have limited ranges. Now a point in (x-y) maps into a sinusoidal curve in parameter (Hough) space . Note that since the three points in the figure below are on the same line therefore they intersect at one point in the Hough space (why?).

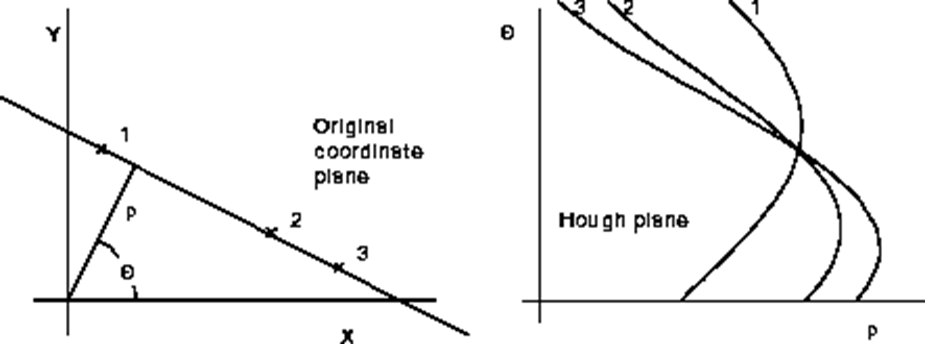


Fig. 6.15 Alternative representation of a line in (x-y) space and corresponding curves in Hough space

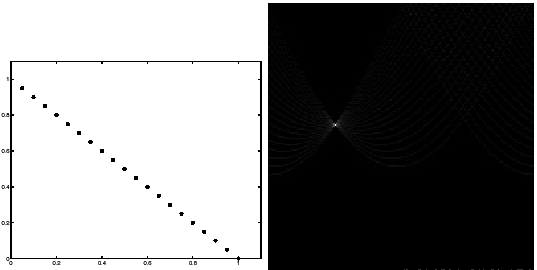


Fig. 6.16 A collection of points in (x-y) or image space and their curves in Hough space. Note all curves intersect at one point.

When points are not on the same line, the curves in Hough space intersect at different points as shown below.

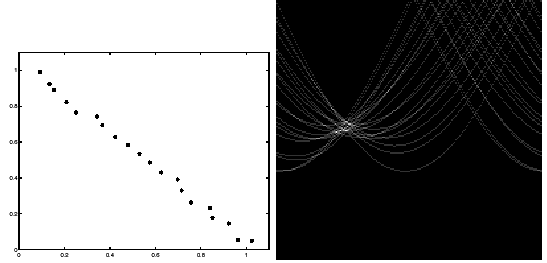


Fig. 6.17 Points not exactly on the same line due to noise.

The Hough space is divided into a number of cells to obtain an accumulator using the following algorithm.

* Apply edge detection and threshold so weak edges are removed
* Quantize the Hough transform space, identify the maximum and minimum values of to determine number of cells.
* Generate an accumulator array and set all values to zero
* For all edge points  in the gradient image

use gradient direction for 

compute  from the equation

increment by one

* For all cells in 

search for the maximum value of 

calculate the equation of the line

To reduce the effect of noise more than one element (elements in a neighborhood) in the accumulator array are increased

Now consider detection of circles of the form

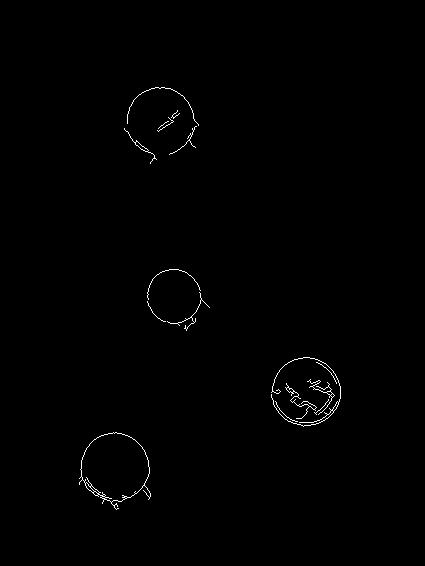


where *(a,b)* is the center of the circle and *r* is its radius. The equation has three parameters *a, b, r*. The curve obtained in the Hough space for each edge point will be a circular cone. Point of intersection of the cones gives the parameters *a, b, r* . In this case the accumulator is three dimensional.

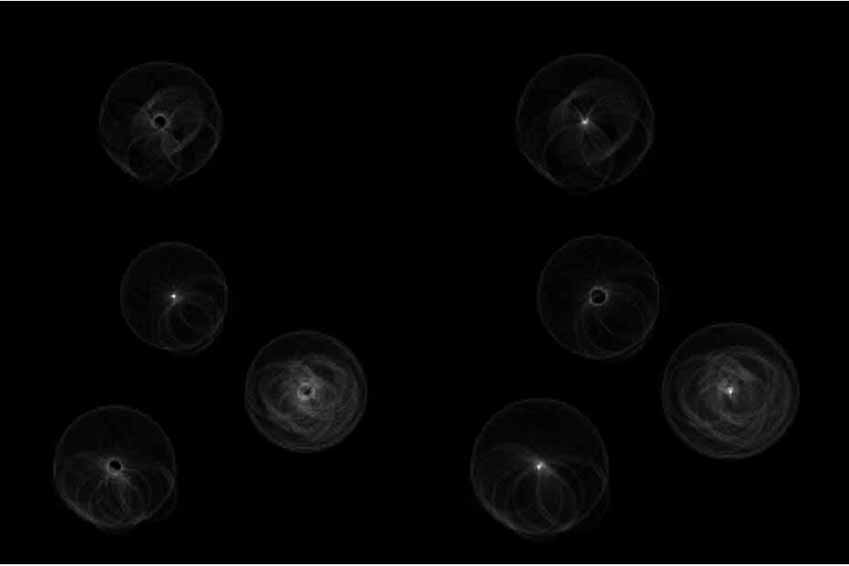
The equation of the circle can also be written as



If the radius of the circle is known, the center coordinates *(a,b)*can be found. In this case the accumulator is *A(a,b).*



1. (b)



(c ) (d)

Fig.6.18 (a) Original image, (b) Canny edge detection, (c) Hough space, (d) circles detected by Hough transform.

* 1. **Segmentation Based Similarity or Discontinuity**
* Thresholding techniques discussed above, group pixels according to some global criterion such as gray level. Two pixels maybe far apart in an image and still be classified as belonging to the same region if their values satisfy the thresholding criterion.
* Another class of segmentation techniques is based on discontinuity or similarity of pixels in the image.
* Techniques based on discontinuity attempt to partition the image by detecting sudden changes. Edge detection fall into these techniques.
* Techniques based on similarity attempt to identify regions by grouping pixels that are “connected” and satisfy certain similarity criterion. We must first explain connectivity of pixels.

**Pixel Connectivity**

In general there are two types of neighborhoods for a pixel. The 4-neighbors of the pixel at (x,y) consist of pixels located at (x –1, y), (x+1, y), (x, y –1) and (x, y+1), as shown in Fig. 6.19a. The 8-neighbors of the pixel at (x,y) are all surrounding pixels which are located at (x+i, y+j), i = –1,0,+1 ;

j = –1,0,+1, as shown in Fig. 6.19b.

(x,y)

1. (b)

Fig. 6.19 The 4– and 8–neighbors of a pixel

A 4-connected path from a pixel to another pixel  is the sequence of pixels  such that  is a 4-neighbor of  for . This path is called an 4-connected path if  is an 4-neighbor of .

Now in order to find a region, we apply thresholding and then pick 4-connected (or 8-connected) path from the thresholded pixels. As an example, consider the following image where the dark pixels are those that have gray levels more than a threshold. However, only a set number of dark pixels are selected for forming a region if 4-neighbor path is to be considered.

**Region Similarity**

The similarity of pixels in a region ***R*** is identified by a uniformity predicate P(R) such as

 (6.4)

where ***f(i,j)*** and ***f(k,l)*** are the coordinates of pixels in region ***R***. Note that that above does not mean that all pixel values in the region R are within  of each other. In fact, with (6.4) the difference in gray levels between pixel in a region can be much larger than  (why?).

A similar predicate is

 (6.5)

where  is the mean values of all pixels within the region collected so far. Equation (6.4)-(6.5) can be generalized to cope with color images where we now compute the distance in RGB space between the colors of neighboring pixels, or between the color of a pixel and mean color for the region.

* 1. **Region Growing**

Region growing starts by planting a set of seed pixels each of which is then grown to a uniform connected region. A pixel is added to a region if and only if

* has not been assign to any other region
* is a neighbor of that region
* the new region created by addition of the pixel is still uniform.

### Region Growing Algorithm

Define a set of regions R1, R2,…, Rm each consisting a single seed pixel

while (no more pixels being assigned to region) {

for(i = 1; i<m; i++)

for(each pixel p at border of Ri)

for(all neighbors of p) {

Let x,y be the neighbor coordinates

Let mu\_i be mean gray level of pixels in Ri collected so far

if ( (neighbor is unassigned) && (abs (f(x,y) – mu\_i) )<= delta) {

add neighbor to Ri

update mu\_i }

}

}

7

7

6

6

5

5

0

0

0

0

5

6

7

7

7

6

6

7

8

6

5

5

6

7

5

1

1

0

1

0

0

2

0

1

0

8

8

5

5

1

1

1

1

7

7

6

6

7

7

0

0

1

1

7

7

7

7

7

8

6

5

5

6

1

1

0

0

1

0

2

2

6

6

6

6

7

7

0

0

5

6

5

6

1

1

5

5

0

0

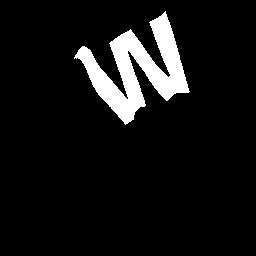
Fig. 6.20 Region growing

Fig. 6.20 shows an example of region growing using (6.5) with  and 8-neighbor connectivity where the seeds are placed in pixels (2,1) and (2,3).

Matlab function: regiongrowing(Image,x,y,reg\_maxdist)

Region growing can be very effective in finding regions of interest. Fig. 6.21b is the result of region growing on the 256 by 256 image in Fig. 6.21a where one seed is placed at the location (128, 100) using 4-connectivity and 30. It is seen the region of interest, e.g. W, has been nicely segmented from the remaining of the image. This would not have been possible with thresholding (why?).

Note: region growing suffers from a number of limitations. For example region growing with 4-connectivity can produce very different results from 8-connectivity. Also, the results obtained can be very sensitive to the choice of uniformity predicate, seed location, and .

Seed at (128, 100), 30

(a) (b)

Seed at (128,140), 30 Seed at (128,140), 50

Fig. 6.21 Effect of seed location and 

For a successful and complete segmentation, the following criteria must be satisfied.

* All pixels must be assigned to regions.
* Each pixel must belong to a single region only.
* Each region must be a connected set of pixels.
* Each region must be uniform.
* Any pair of adjacent regions must be non-uniform.

Region growing satisfies the third and fourth criteria but not the others. It fails to satisfy the first and second criteria because the number of seed is usually insufficient to create regions for every pixel.

See Appendix 1 for a Matlab implementation of region growing. Also check out the following link.

<https://www.mathworks.com/videos/search.html?q=image+segmentation&page=1>

**The Split and Merge**

A complete segmentation is possible if we initially consider the entire image as a single region. Then we test this region for uniformity predicate, and if it is false, then the region is split or divided into sub-regions, each of which is tested for uniformity. The procedure iterates until all regions are uniform, or desired number of regions have been established.

A common strategy is to divide the image into 4 regions recursively, and check that for the region Ri , P(Ri) is true. Each of these quadrants is in turn divided into 4 quadrants. At each stage the adjacent regions are checked, and if they have identical quality, then they are merged. Therefore, each split is followed by a test of adjacent regions and a possible merge. For example if Ri and Rj are two adjacent regions, then they are combined into a larger region if the uniformity predicate is true for the union of these two regions, i.e. P(Ri  Rj) = TRUE.

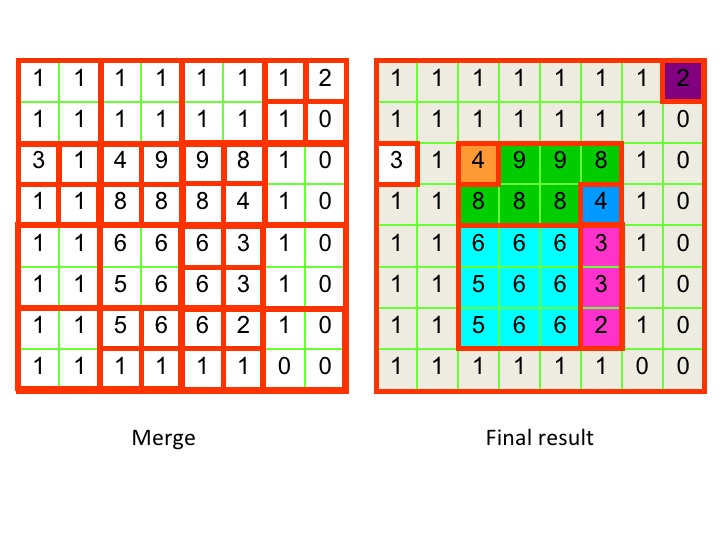


Fig. 6.22 Demonstration of split and merge

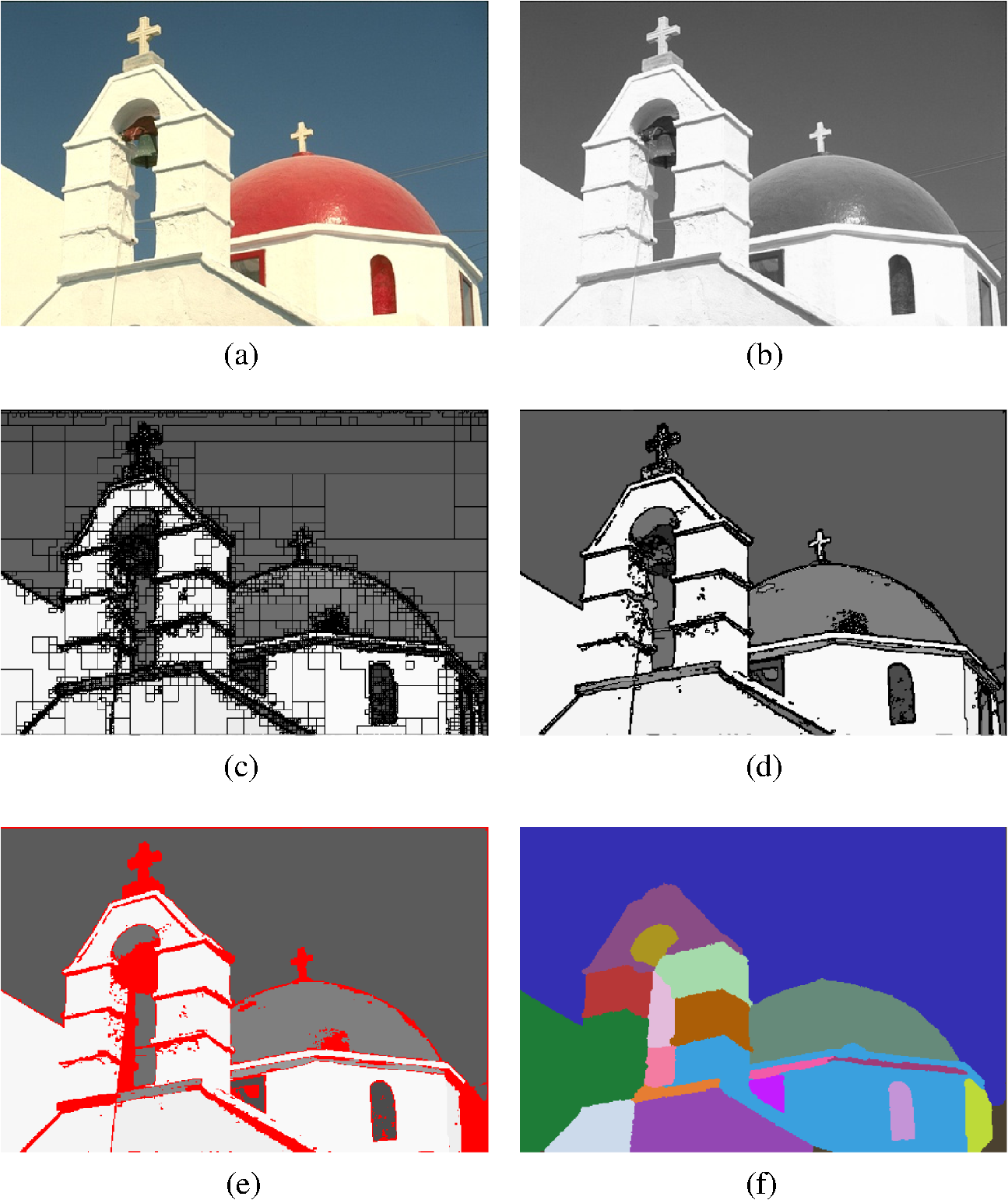


Fig. 6.23 Segmentation steps: (a) Original image, (b) feature description image, (c) image after the splitting process, (d) image after the merging process, (e) elimination of small regions, and the (f) resulting segmentation classes obtained after the region growing procedure.

* 1. **Region Representation**

Once the regions are detected, they must be stored for possible further processing by the computer.

**Occupancy Array**: The simplest representation is the spatial occupancy array which defines a membership function as

 (6.6)

This two-dimensional array can be easily accessed, unioned, merged or intersected by AND/OR logic operations. However it requires much space, and also does not represent the boundary of the region in a useful form.

**Y-axis**: A more compact representation than the above is Y-axis representation, which also allows all the above operations. Here each pixel row (Y) belonging to the region is encoded as a list of X coordinates of pixel going in and out of the region, i.e. (

Example: 0 1 2 3 4 5

x

(0,0,4), (1,1,3), (2,1,2),(3,2,5) 0 1

2

3

y

The Y-representation becomes inefficient when the region is long and thin along y axis (why?). In this case X-axis representation can be used.

**Quad Tree**: This representation is useful for the encoding occupancy. Each node of the tree has 4 children. The children represent regions. For example in the region shown in Fig. 6.22b, the quad tree is given in Fig. 6.22a. The gray nodes represent the mixed regions which must be split further. The black nodes represent pixels on the region, and white nodes are outside the regions.

S

(b)

(a)

Fig. 6.24 Quad tree representation of a region.

**Polylines:** In this representation boundary of a region is approximated by a number of line segments and the coordinates of those segments are stored as

{(x1, y1), (x2, y2), …, (xn, yn)}. For example, the region in Fig 6.25 is approximated by 13 line segments, which are then stored by the coordinates of the intersections of the lines segments.

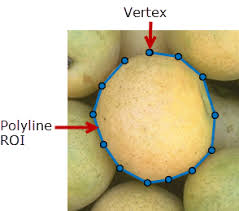


Fig. 6.25 polyline approximation of a region.

Any accuracy of representation can be achieved by increasing the number of line segments.

**Chain Code**: This is another representation, and is based on 4 or 8 directions (connectivity). A chain code can be generated by following the boundary in a clockwise direction and assigning directions to segments connecting pixels.

1 2

3 1

1. 0 4 0

5 7

3 6

Fig. 6.26 4- and 8-Chain code

For example starting at point S in Fig. 6.24, and following the boundary clockwise using 4-directions, we have 2233233001012101.

If the region is long, the resulting codes will be very long. Also, small disturbances due to noise will cause changes to the boundary shape. To overcome these difficulties, we can resample the boundary by selecting a larger grid.

**Fourier Descriptor**: This method is used to reduce the number of points describing a boundary to a much smaller number without loosing much precision. Suppose that the boundary of a region is represented by a set of point with their x-y coordinates, i.e.



where N is the number of points (pixels on the boundary). Fourier descriptor treats each point as a complex number, i.e.



A one dimensional Fourier transform is then applied to the  as follows



Now we retain only values of S(u), and set the remain values to zero. In other words,



which means that the high frequency contents of S(u) are removed. This Fourier descriptor is then stored. Typically, , and thus only a small number of points are stored. To get the boundary in the spatial domain, we perform the inverse Fourier transform, i.e.

 k = 0,1,…, N-1

Note that the spatial representation still has N points, and is a good approximation of .

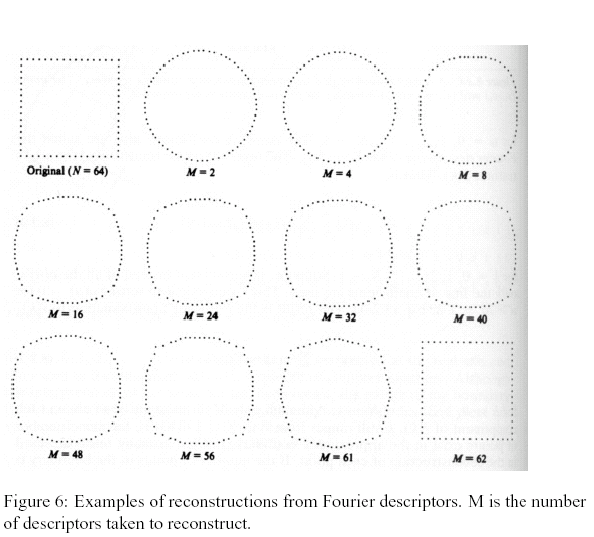


Fig. 6.27 Effect of increasing Fourier descriptor coefficients (M=No).

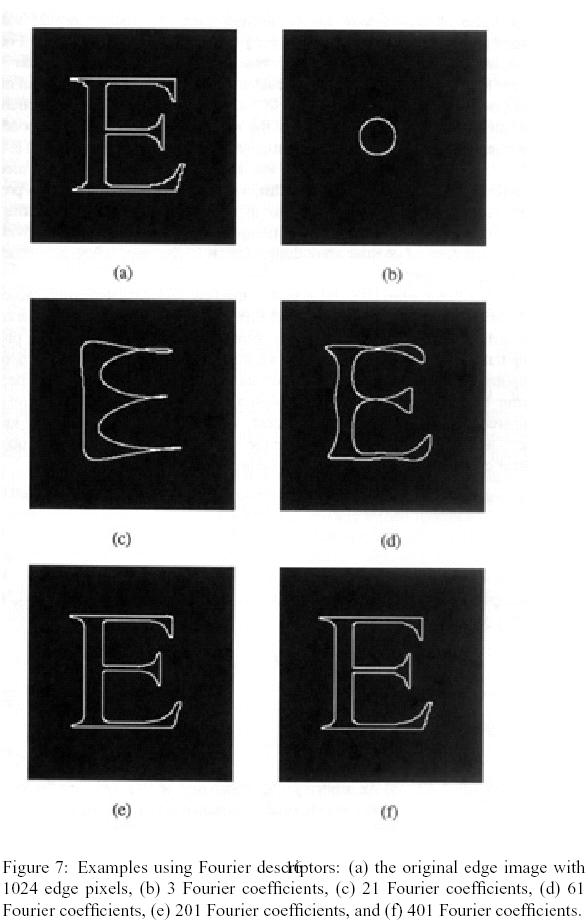
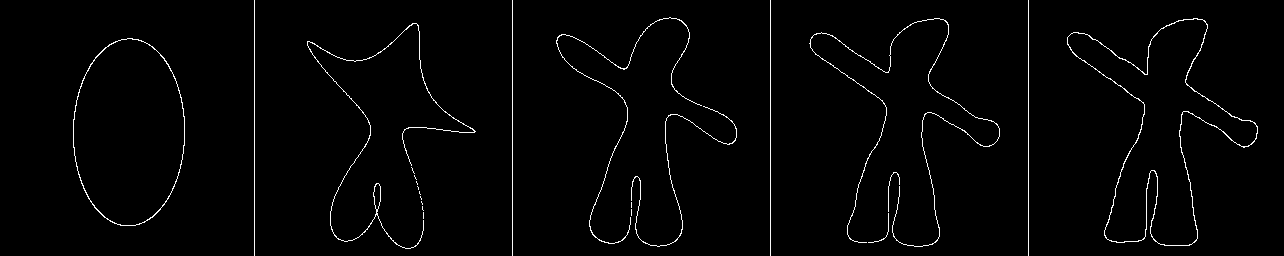


Fig. 6.28 Effect of increasing Fourier descriptor coefficients



Original shape



**0.1% 1% 2% 5% 100%**

Fig. 6.29 Original image and shapes after keeping the percentages indicated of the Fourier descriptor coefficients.

**6.7 Shape Measures of a Region**

There are several measures that identify the shape and size of a region.

**Area**: The total number of pixels in a region is the area of that region. Area can be deduced from the region representation. For example, if the polyline consists of m points with coordinates (x[i], y[i]), i=0,1, …, m-1, then the area of the region is obtained from (why?)

 (6.7)

**Eccentricity**: This is the ratio of maximum width to maximum height of a region as obtained from two perpendicular lines, i.e.  as shown in Fig. 6.28.

A

C D

B

##### Fig. 6.30 Definition of eccentricity

**Euler Number**:. The Euler number E is defined as

 (6.8)

where C is the number of connected components of the region, and H is the number of holes in the region. The Euler numbers for the regions (a)-(c) of Fig. 6.14 are, E =1–2 = –1, E =1– 0 = 1 and E = 1 – 1= 0, respectively.

(a) (b) (c)

Fig. 6.31 Regions

Euler number describes a shape property that is unaffected by any deformation, so long as there is no tearing or joining. This type of deformation is sometime called rubber sheet distortion.

**Signature**: This is a one dimensional function giving a measure of a boundary. Given a binary image function ***R(x,y)*** representing a region, then the horizontal and vertical signatures are defined, respectively, as

 and  (6.9)

**Centroid**: The centroid of a function ***f(x,y),*** which could be non-binary, is defined as

 (6.10)

where A is the area, and w and h are the width and height of the image, respectively. For a binary image (region) ***f(x,y) = 1*** if the the pixel ***(x,y)*** is in the region, otherwise ***f(x,y) = 0***.

**Moments**: These provide measure of distribution of values across an axis. The p and q moments are defined as

 (6.11)

Note that:



 (6.12)



The central moment is defined as

 (6.13)

That is the moments with the origin of axis located on the centroid.

**Circularity**: This is a measure of the roundedness of a region. Let  be pixels on the boundary of a region and determine the distance to this pixel from the centroid as

 (6.14)

If there are m points on the boundary, the average distance is

 (6.15)

Then the circularity is defined as the standard deviation, i.e.

 (6.16)

A perfectly circular region has  (why?). Higher values of  correspond to non-circular regions.

**Rectangularity**: This measure is defined as

 (6.17)

where A is the area of the region and L and W are the length and width of the rectangular bounding box. The maximum value of is equal to 1.

Fig. 6.32 Bounding box

**Appendix 1**

function J=regiongrowing(I,x,y,reg\_maxdist)

% This function performs "region growing" in an image from a specified

% seedpoint (x,y)

%

% J = regiongrowing(I,x,y,t)

%

% I : input image

% J : logical output image of region

% x,y : the position of the seedpoint (if not given uses function getpts)

% t : maximum intensity distance (defaults to 0.2)

%

% The region is iteratively grown by comparing all unallocated neighbouring pixels to the region.

% The difference between a pixel's intensity value and the region's mean,

% is used as a measure of similarity. The pixel with the smallest difference

% measured this way is allocated to the respective region.

% This process stops when the intensity difference between region mean and

% new pixel become larger than a certain treshold (t)

%

% Example:

%

% I = im2double(imread('medtest.png'));

% x=198; y=359;

% J = regiongrowing(I,x,y,0.2);

% figure, imshow(I+J);

%

% Author: D. Kroon, University of Twente

if(exist('reg\_maxdist','var')==0), reg\_maxdist=0.2; end

if(exist('y','var')==0), figure, imshow(I,[]); [y,x]=getpts; y=round(y(1)); x=round(x(1)); end

J = zeros(size(I)); % Output

Isizes = size(I); % Dimensions of input image

reg\_mean = I(x,y); % The mean of the segmented region

reg\_size = 1; % Number of pixels in region

% Free memory to store neighbours of the (segmented) region

neg\_free = 10000; neg\_pos=0;

neg\_list = zeros(neg\_free,3);

pixdist=0; % Distance of the region newest pixel to the regio mean

% Neighbor locations (footprint)

neigb=[-1 0; 1 0; 0 -1;0 1];

% Start regiogrowing until distance between regio and posible new pixels become

% higher than a certain treshold

while(pixdist<reg\_maxdist&&reg\_size<numel(I))

% Add new neighbors pixels

for j=1:4,

% Calculate the neighbour coordinate

xn = x +neigb(j,1); yn = y +neigb(j,2);

% Check if neighbour is inside or outside the image

ins=(xn>=1)&&(yn>=1)&&(xn<=Isizes(1))&&(yn<=Isizes(2));

% Add neighbor if inside and not already part of the segmented area

if(ins&&(J(xn,yn)==0))

neg\_pos = neg\_pos+1;

neg\_list(neg\_pos,:) = [xn yn I(xn,yn)]; J(xn,yn)=1;

end

end

% Add a new block of free memory

if(neg\_pos+10>neg\_free), neg\_free=neg\_free+10000; neg\_list((neg\_pos+1):neg\_free,:)=0; end

% Add pixel with intensity nearest to the mean of the region, to the region

dist = abs(neg\_list(1:neg\_pos,3)-reg\_mean);

[pixdist, index] = min(dist);

J(x,y)=2; reg\_size=reg\_size+1;

% Calculate the new mean of the region

reg\_mean= (reg\_mean\*reg\_size + neg\_list(index,3))/(reg\_size+1);

% Save the x and y coordinates of the pixel (for the neighbour add proccess)

x = neg\_list(index,1); y = neg\_list(index,2);

% Remove the pixel from the neighbour (check) list

neg\_list(index,:)=neg\_list(neg\_pos,:); neg\_pos=neg\_pos-1;

end

% Return the segmented area as logical matrix

J=J>1;

Appendix 2

function g = splitmerge(f, mindim, fun)

%SPLITMERGE Segment an image using a split-and-merge algorithm.

% G = SPLITMERGE(F, MINDIM, @PREDICATE) segments image F by using a

% split-and-merge approach based on quadtree decomposition. MINDIM

% (a positive integer power of 2) specifies the minimum dimension

% of the quadtree regions (subimages) allowed. If necessary, the

% program pads the input image with zeros to the nearest square

% size that is an integer power of 2. This guarantees that the

% algorithm used in the quadtree decomposition will be able to

% split the image down to blocks of size 1-by-1. The result is

% cropped back to the original size of the input image. In the

% output, G, each connected region is labeled with a different

% integer.

%

% Note that in the function call we use @PREDICATE for the value of

% fun. PREDICATE is a function in the MATLAB path, provided by the

% user. Its syntax is

%

% FLAG = PREDICATE(REGION) which must return TRUE if the pixels

% in REGION satisfy the predicate defined by the code in the

% function; otherwise, the value of FLAG must be FALSE.

%

% The following simple example of function PREDICATE is used in

% Example 10.9 of the book. It sets FLAG to TRUE if the

% intensities of the pixels in REGION have a standard deviation

% that exceeds 10, and their mean intensity is between 0 and 125.

% Otherwise FLAG is set to false.

%

% function flag = predicate(region)

% sd = std2(region);

% m = mean2(region);

% flag = (sd > 10) & (m > 0) & (m < 125);

% Copyright 2002-2004 R. C. Gonzalez, R. E. Woods, & S. L. Eddins

% Digital Image Processing Using MATLAB, Prentice-Hall, 2004

% $Revision: 1.6 $ $Date: 2003/10/26 22:36:01 $

% Pad image with zeros to guarantee that function qtdecomp will

% split regions down to size 1-by-1.

Q = 2^nextpow2(max(size(f)));

[M, N] = size(f);

f = padarray(f, [Q - M, Q - N], 'post');

%Perform splitting first.

S = qtdecomp(f, @split\_test, mindim, fun);

% Now merge by looking at each quadregion and setting all its

% elements to 1 if the block satisfies the predicate.

% Get the size of the largest block. Use full because S is sparse.

Lmax = full(max(S(:)));

% Set the output image initially to all zeros. The MARKER array is

% used later to establish connectivity.

g = zeros(size(f));

MARKER = zeros(size(f));

% Begin the merging stage.

for K = 1:Lmax

[vals, r, c] = qtgetblk(f, S, K);

if ~isempty(vals)

% Check the predicate for each of the regions

% of size K-by-K with coordinates given by vectors

% r and c.

for I = 1:length(r)

xlow = r(I); ylow = c(I);

xhigh = xlow + K - 1; yhigh = ylow + K - 1;

region = f(xlow:xhigh, ylow:yhigh);

flag = feval(fun, region);

if flag

g(xlow:xhigh, ylow:yhigh) = 1;

MARKER(xlow, ylow) = 1;

end

end

end

end

% Finally, obtain each connected region and label it with a

% different integer value using function bwlabel.

g = bwlabel(imreconstruct(MARKER, g));

% Crop and exit

g = g(1:M, 1:N);

%-------------------------------------------------------------------%

function v = split\_test(B, mindim, fun)

% THIS FUNCTION IS PART OF FUNCTION SPLIT-MERGE. IT DETERMINES

% WHETHER QUADREGIONS ARE SPLIT. The function returns in v

% logical 1s (TRUE) for the blocks that should be split and

% logical 0s (FALSE) for those that should not.

% Quadregion B, passed by qtdecomp, is the current decomposition of

% the image into k blocks of size m-by-m.

% k is the number of regions in B at this point in the procedure.

k = size(B, 3);

% Perform the split test on each block. If the predicate function

% (fun) returns TRUE, the region is split, so we set the appropriate

% element of v to TRUE. Else, the appropriate element of v is set to

% FALSE.

v(1:k) = false;

for I = 1:k

quadregion = B(:, :, I);

if size(quadregion, 1) <= mindim

v(I) = false;

continue

end

flag = feval(fun, quadregion);

if flag

v(I) = true;

end

end