CHAPTER 15

IMAGE SEGMENTATION

WHAT WILL WE LEARN?

- What is image segmentation and why is it relevant?
- What is image thresholding and how is it implemented in MATLAB?
- What are the most commonly used image segmentation techniques and how do they work?

15.1 INTRODUCTION

Segmentation is one of the most crucial tasks in image processing and computer vision. As you may recall from our discussion in Chapter 1 (Section 1.5), image segmentation is the operation that marks the transition between *low-level image processing* and *image analysis*: the input of a segmentation block in a machine vision system is a preprocessed image, whereas the output is a representation of the regions within that image. This representation can take the form of the boundaries among those regions (e.g., when edge-based segmentation techniques are used) or information about which pixel belongs to which region (e.g., in clustering-based segmentation). Once an image has been segmented, the resulting individual regions (or objects) can be described, represented, analyzed, and classified with techniques such as the ones presented in Chapters 18 and 19.

Practical Image and Video Processing Using MATLAB®. By Oge Marques.

Segmentation is defined as the process of partitioning an image into a set of nonoverlapping regions whose union is the entire image. These regions should ideally correspond to objects and their meaningful parts, and background. Most image segmentation algorithms are based on one of two basic properties that can be extracted from pixel values—discontinuity and similarity—or a combination of them.

Segmentation of nontrivial images is a very hard problem—made even harder by nonuniform lighting, shadows, overlapping among objects, poor contrast between objects and background, and so on—that has been approached from many different angles, with limited success to this date. Many image segmentation techniques and algorithms have been proposed and implemented during the past 40 years and yet, except for relatively "easy" scenes, the problem of segmentation remains unsolved.

Figure 15.1 illustrates the problem. At the top, it shows the color and grayscale versions of a *hard* test image that will be used later in this chapter. Segmenting this image into its four main objects (Lego bricks) and the background is not a simple task for contemporary image segmentation algorithms, due to uneven lighting, projected shadows, and occlusion among objects. Attempting to do so without resorting to color information makes the problem virtually impossible to solve for the techniques described in this chapter.

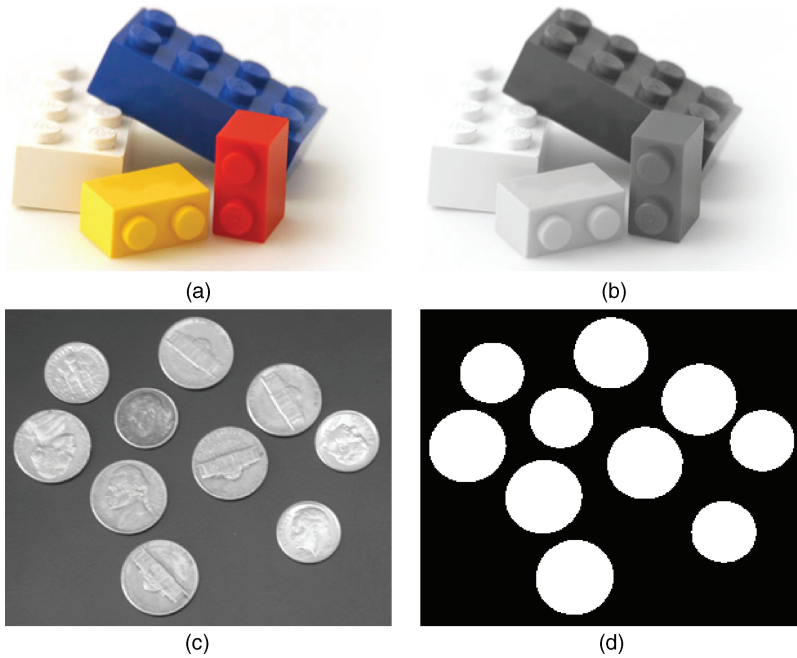


FIGURE 15.1 Test images for segmentation algorithms: (a) a *hard* test image and (b) its grayscale equivalent; (c) an easier test image (courtesy of MathWorks) and (d) the result of morphological preprocessing and thresholding.

The bottom part of Figure 15.1 shows another test image, which is considerably simpler and will probably lead to perfect segmentation with even the simplest techniques. Although the original image has a few imperfections (particularly on one coin that is significantly darker than the others), simple preprocessing operations such as region filling (Section 13.6.2) using `imfill` will turn it into an image suitable for global thresholding (Section 15.2.1) and subsequent labeling of the individual regions.

There is no underlying theory of image segmentation, only *ad hoc* methods, whose performance is often evaluated indirectly, based on the performance of the larger system to which they belong. Even though they share the same goal, image segmentation techniques can vary widely according to the type of image (e.g., binary, gray, color), choice of mathematical framework (e.g., morphology, image statistics, graph theory), type of features (e.g., intensity, color, texture, motion), and approach (e.g., top-down, bottom-up, graph-based).[1]

There is no universally accepted taxonomy for classification of image segmentation algorithms either. In this chapter, we have organized the different segmentation methods into the following categories:

- Intensity-based methods (Section 15.2), also known as *noncontextual* methods, work based on pixel distributions (i.e., histograms). The best-known example of intensity-based segmentation technique is *thresholding*.
- Region-based methods (Section 15.3), also known as *contextual* methods, rely on adjacency and connectivity criteria between a pixel and its neighbors. The best-known examples of region-based segmentation techniques are *region growing* and *split and merge*.
- Other methods, where we have grouped relevant segmentation techniques that do not belong to any of the two categories above. These include segmentation based on texture, edges, and motion, among others.[2]

15.2 INTENSITY-BASED SEGMENTATION

Intensity-based methods are conceptually the simplest approach to segmentation. They rely on pixel statistics—usually expressed in the form of a histogram (Chapter 9)—to determine which pixels belong to foreground objects and which pixels should be labeled as background. The simplest method within this category is *image thresholding*, which will be described in detail in the remaining part of this section.

[1]The field of image segmentation research is still very active. Most recently published algorithms are far too complex to be included in this text, and their computational requirements often push MATLAB to its limits. Refer to "Learn More About It" section at the end of the chapter for useful pointers.

[2]Segmentation of color images will be discussed in Chapter 16.

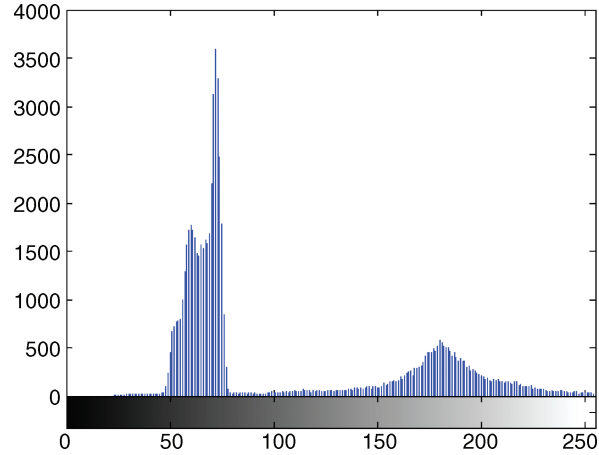


FIGURE 15.2 The histogram for the image in Figure 15.1c: an example of histogram suitable for partitioning using a single threshold.

15.2.1 Image Thresholding

The basic problem of thresholding is the conversion of an image with many gray levels into another image with fewer gray levels, usually only two. This conversion is usually performed by comparing each pixel intensity with a reference value (*threshold*, hence the name) and replacing the pixel with a value that means "white" or "black" depending on the outcome of the comparison. Thresholding is a very popular image processing technique, due to its simplicity, intuitive properties, and ease of implementation.

Thresholding an image is a common preprocessing step in machine visual systems in which there are relatively few objects of interest whose shape (silhouette) is more important than surface properties (such as texture) and whose average brightness is relatively higher or lower than the other elements in the image. The test image in Figure 15.1c is an example of image suitable for image thresholding. Its histogram (Figure 15.2) has two distinct modes, the narrowest and most prominent one (on the left) corresponding to background pixels, the broadest one (on the right) reflecting the intensity distribution of pixels corresponding to the coins.

Mathematically, the process of thresholding an input image $f(x, y)$ and producing a binarized version of it, $g(x, y)$, can be described as

$$g(x, y) = \begin{cases} 1 \text{ if } f(x, y) > T \\ 0 \text{ otherwise} \end{cases} \tag{15.1}$$

where T is the threshold. If the same value of T is adopted for the entire image, the process is called *global thresholding*. When the choice of value for T at a point of

coordinates (x, y) depends on statistical properties of pixel values in a neighborhood around (x, y), it will be referred to as *local* or *regional thresholding*.[3]

In MATLAB

The IPT has a function to convert a grayscale image into a binary (black-and-white) image, `im2bw`, that takes an image and a threshold value as input parameters. You will learn how to use it in Tutorial 15.1.

15.2.2 Global Thresholding

When the intensity distribution of an image allows a clear differentiation between pixels of two distinct predominant average gray levels, the resulting histogram has a bimodal shape (such as the one in Figure 15.2), which suggests that there may be a single value of T that can be used as a threshold for the entire image. For the case of a single image, the choice of the actual value of T can be done manually, in typical trial and error fashion, as follows:

1. Inspecting the image's histogram (using `imhist`).
2. Select an appropriate value for T.
3. Apply the selected value (using `im2bw`) to the image.
4. Inspect the results: if they are acceptable, save resulting image. Otherwise, make adjustments and repeat steps 2–4.

For the cases where many images need to be segmented using global thresholding, a manual, labor-intensive approach such as described above is not appropriate. An automated procedure for selecting T has to be employed. Gonzalez and Woods [GW08] proposed an iterative algorithm for this purpose, whose MATLAB implementation (based on [GWE04]) follows:

```
Id = im2double(I); % I is a uint8 grayscale image
T = 0.5*(min(Id(:)) + max(Id(:)));
deltaT = 0.01; % convergence criterion
done = false;
while ~done
     g = Id >= T;
     Tnext = 0.5*(mean(Id(g)) + mean(Id(~g)));
     done = abs(T - Tnext) < deltaT;
     T = Tnext;
end
```

[3]Techniques that rely on the spatial coordinates (x, y), often called *dynamic* or *adaptive thresholding*, have also been proposed in the literature. Since they cannot be considered purely "intensity-based," they have been left out of this discussion.

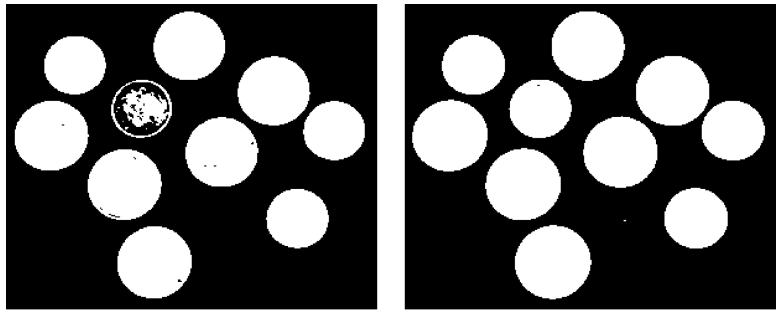


FIGURE 15.3 Image thresholding results for the image in Figure 15.1c using iterative threshold selection algorithm (a) and manually selected threshold (b).

Figure 15.3a shows the result of applying the threshold value to the image in Figure 15.1c. In this case, the maximum and minimum gray values in `Id` are 1 and 0.0902, respectively, `deltaT` was chosen to be 0.01, and it took the algorithm three iterations to arrive at the final result: $T = 0.4947$. For the sake of comparison, the results obtained with a manually selected $T = 0.25$ are shown in Figure 15.3b.

Optimal Thresholding Many *optimal* strategies for selecting threshold values have been suggested in the literature. These strategies usually rely on assumed statistical models and consist of modeling the thresholding problem as a statistical inference problem. Unfortunately, such statistical models usually cannot take into account important factors such as borders and continuity, shadows, nonuniform reflectance, and other perceptual aspects that would impact a human user making the same decision. Consequently, for most of the cases, manual threshold selection by humans will produce better results than statistical approaches would [BD00].

The most popular approach under this category was proposed by Otsu[4] in 1979 [Ots79] and implemented as an IPT function: `graythresh`. Applying that function to the image in Figure 15.1c results in an optimal value for $T = 0.4941$, which—in this particular case—is remarkably close to the one obtained with the (much simpler) iterative method described earlier ($T = 0.4947$). However, as shown in Figure 15.3, neither of these methods produce a better (from a visual interpretation standpoint) result than the manually chosen threshold.

15.2.3 The Impact of Illumination and Noise on Thresholding

Illumination and reflectance patterns play a critical role in thresholding. Even an easy input image (such as the `coins` image), which could be successfully segmented using global thresholding, poses a much harder challenge if the illumination pattern changes

[4] A detailed description of the approach is beyond the scope of this text. See Section 10.3.3 of [GW08] or Section 3.8.2 of [SS01] for additional information.

FIGURE 15.4 An example of uneven illumination pattern used to generate the image in Figure 15.5a.

from constant (uniform) to gradual (Figure 15.4). The resulting image (Figure 15.5a) is significantly darker overall and the corresponding histogram (Figure 15.5b) shows an expected shift to the left. Consequently, using the same value of threshold ($T = 25$) that produced very good results before (Figure 15.3b) will lead to an unacceptable binarized image (Figure 15.5c).

Noise can also have a significant impact on thresholding, as illustrated in Figure 15.5(d–f). In this case a Gaussian noise of mean zero and variance 0.03 has been applied to the image, resulting in the image in Figure 15.5d, whose histogram, shown in Figure 15.5e, has lost its original bimodal shape. The result of segmenting the image using $T = 0.25$ is shown in Figure 15.5f. Although not as bad as one could expect, it would need postprocessing (noise reduction) to be truly useful.

In summary, in both cases, the images changed significantly, their histograms lost their bimodal shape, and the originally chosen value for global threshold ($T = 0.25$) was no longer adequate. In addition, no other value could be easily chosen just by inspecting the histogram and following the trial and error procedure suggested earlier in this chapter.

15.2.4 Local Thresholding

Local (also called *adaptive*) thresholding uses block processing to threshold blocks of pixels, one at a time. The size of the block is usually specified by the user, with the two extreme conditions being avoided: blocks that are too small may require an enormous amount of processing time to compute, whereas large blocks may produce results that are not substantially better than the ones obtained with global thresholding.

In MATLAB

The block processing technique is implemented using the `blkproc` function. You will learn how to use this function in the context of local thresholding in Tutorial 15.1.

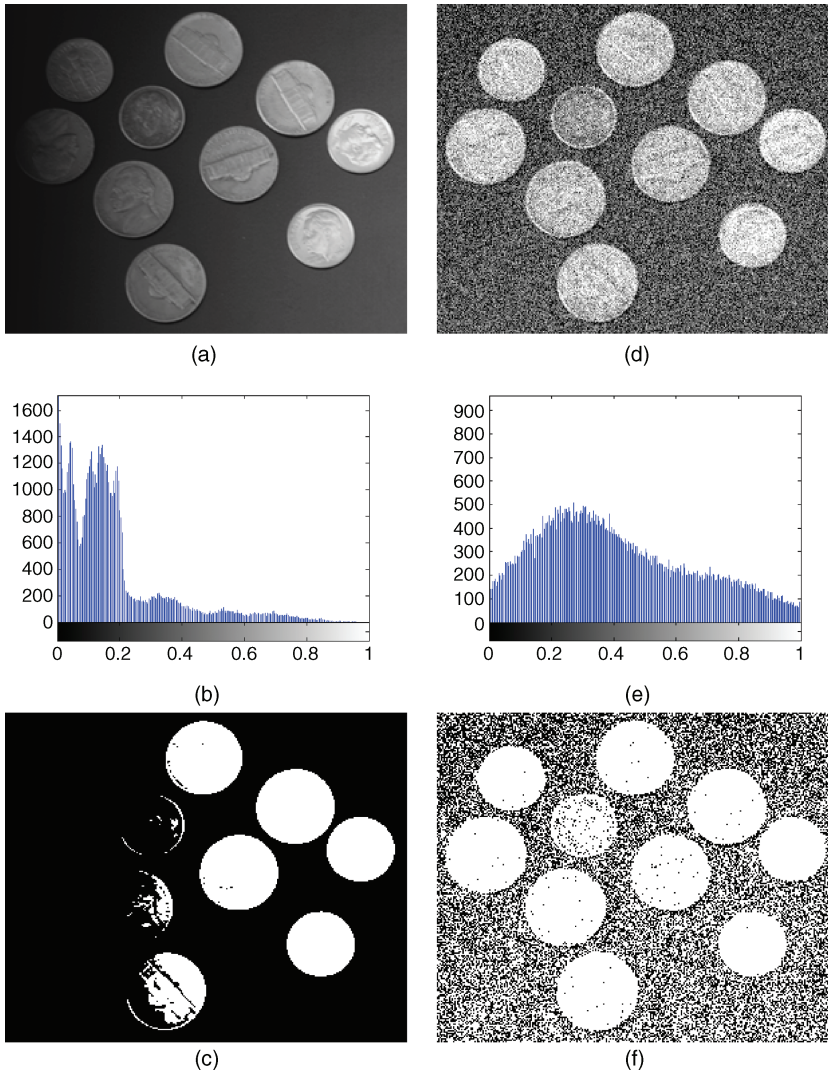
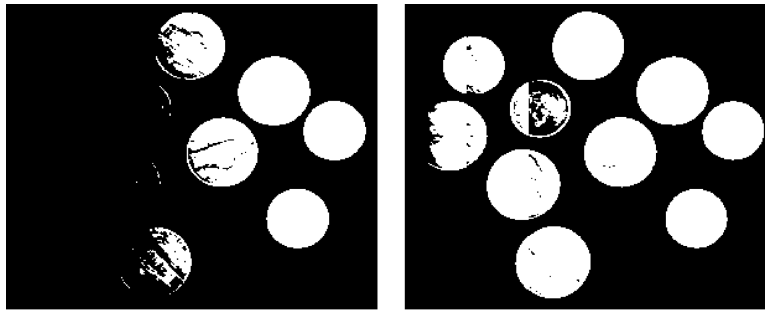


FIGURE 15.5 Effect of illumination (left) and noise (right) on thresholding. See text for details.

■ EXAMPLE 15.1

In this example, we divide the image in Figure 15.5a into six vertical slices, treating each of them separately and using `graythresh` to choose a different value of threshold for each slice. The resulting image (Figure 15.6b) is significantly better than the one we would have obtained using a single value ($T = 0.3020$, also calculated using



(a) (b)

FIGURE 15.6 Local thresholding. Using a single threshold for the entire image (a) and dividing it up into six slices and choosing a different threshold for each vertical slice (b).

`graythresh`) for the entire image (Figure 15.6a). The left portion of Figure 15.6b gives away some hints at where the slices were placed.

15.3 REGION-BASED SEGMENTATION

Region-based segmentation methods are based on the fact that a pixel cannot be considered a part of an object or not based solely on its gray value (as intensity-based methods do). They incorporate measures of connectivity among pixels in order to decide whether these pixels belong to the same region (or object) or not.

Mathematically, region-based segmentation methods can be described as a systematic way to partition an image I into n regions, $R_1, R_2, \cdots, R_n$, such that the following properties hold [GW08]:

1. $\bigcup_{i=1}^{n} R_i = I$.
2. R_i is a connected region, $i = 1, 2, \cdots, n$.
3. $R_i \cap R_j = \emptyset$ for all i and j, $i \neq j$.
4. $P(R_i) = \text{TRUE}$ for $i = 1, 2, \cdots, n$.
5. $P(R_i \cup R_j) = \text{FALSE}$ for any adjacent regions R_i and R_j.

Here $P(R_i)$ is a logical predicate defined over the points in set R_i and $\emptyset$ is the empty set.

The first property states that the segmentation will be complete, that is, each pixel in the image will be labeled as belonging to one of the n regions. Property 2 requires that all points within a region be 4- or 8- connected. Property 3 states that the regions cannot overlap. Property 4 states which criterion must be satisfied so that a pixel is granted membership in a certain region, for example, all pixel values must be within a certain range of intensities. Finally, property 5 ensures that two adjacent regions are different in the sense of predicate P.

These logical predicates are also called *homogeneity criteria*, $H(R_i)$. Some of the most common homogeneity criteria for grayscale images are as follows [Umb05]:

- *Pure Uniformity*: All pixel values in a region are the same.
- *Local Mean Relative to Global Mean*: The average intensity in a region is significantly greater (or smaller) than the average gray level in the whole image.
- *Local Standard Deviation Relative to Global Mean*: The standard deviation of the pixel intensities in a region is less than a small percentage of the average gray level in the whole image.
- *Variance*: At least a certain percentage of the pixels in a region are within two standard deviations of the local mean.
- *Texture*: All four quadrants within a region have comparable texture.

15.3.1 Region Growing

The basic idea of region growing methods is to start from a pixel and grow a region around it, as long as the resulting region continues to satisfy a homogeneity criterion. It is, in that sense, a bottom-up approach to segmentation, which starts with individual pixels (also called *seeds*) and produces segmented regions at the end of the process.

The key factors in region growing are as follows:

- *The Choice of Similarity Criteria*: For monochrome images, regions are analyzed based on intensity levels (either the gray levels themselves or the measures that can easily be calculated from them, for example, moments and texture descriptors[5]) and connectivity properties.
- *The Selection of Seed Points*: These can be determined interactively (if the application allows) or based on a preliminary cluster analysis of the image, used to determine groups of pixels that share similar properties, from which a seed (e.g., corresponding to the centroid of each cluster) can be chosen.
- *The Definition of a Stopping Rule*: A region should stop growing when there are no further pixels that satisfy the homogeneity and connectivity criteria to be included in that region.

It can be described in algorithmic form as follows [Eff00]:

```
Let f(x,y) be the input image
Define a set of regions R1, R2, ..., Rn, each consisting of a
          single seed pixel
repeat
   for i = 1 to n do
      for each pixel p at the border of Ri do
         for all neighbors of p do
```

[5]These will be discussed in Chapter 18.

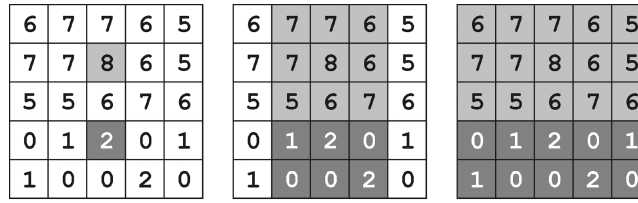


FIGURE 15.7 Region growing: (a) seed pixels; (b) first iteration; (c) final iteration.

```
            Let (x,y) be the neighbor's coordinates
            Let Mi be the mean gray level of pixels in Ri
            if the neighbor is unassigned and
                      |f(x,y) - Mi| <= Delta then
               Add neighbor to Ri
               Update Mi
            end if
         end for
      end for
   end for
until no more pixels can be assigned to regions
```

■ EXAMPLE 15.2

Figure 15.7 shows an example of region growing on small test image, where the logical predicate for uniformity is given by

$$P(R_i) = \begin{cases} \text{TRUE} & \text{if } |f(x, y) - \mu_i| \leq \Delta \\ \text{FALSE} & \text{otherwise} \end{cases} \tag{15.2}$$

where μ_i is the average intensity of all pixels in R_i except the reference pixel at (x, y) and Δ is a user-selected threshold. In this example, $\Delta = 3$. Figure 15.7 shows the seed pixels (a), and the results of the first (b) and last (c) iterations.

■ EXAMPLE 15.3

Figure 15.8 shows the results of applying a region growing algorithm[6] to the two test images originally introduced in Figure 15.1. Part (a) shows the *hard* input image with the seed points (specified interactively by the user) overlaid. Part (b) shows the results of using region growing (pseudocolored for easier visualization): four regions plus background. A quick inspection shows that the results are virtually useless, primarily

[6]The region growing algorithm used in this example appears in [GWE04].

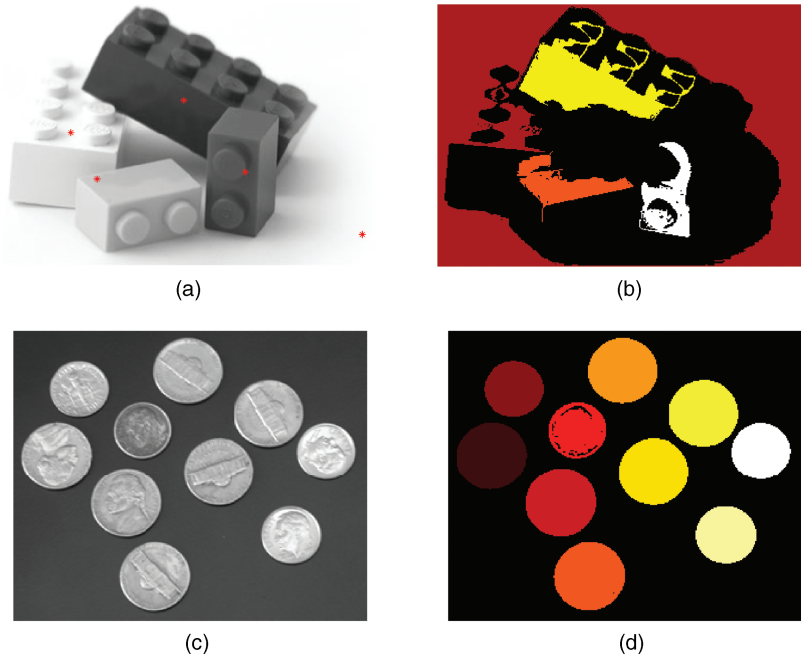


FIGURE 15.8 Region growing results for two test images. See text for details.

due to poor contrast between the two darker and two brighter bricks and the influence of projected shadows.

For comparison purposes, we ran the same algorithm with an *easy* input image (Figure 15.8c) in *unsupervised mode*, that is, without specifying any points to be used as seeds (or even the total number of regions that the algorithm should return). The results (Figure 15.8d) are good, comparable to the ones obtained using global thresholding earlier in this chapter.

Limitations of Region Growing The basic region growing algorithm described in this section has several limitations, listed below [Eff00]:

- It is not very stable: significantly different results are obtained when switching between 4-connectivity and 8-connectivity criteria.
- Segmentation results are very sensitive to choice of logical uniformity predicate.
- The number of seeds provided by the user may not be sufficient to assign every pixel to a region.
- If two or more seeds that should belong to the same region are incorrectly provided to the algorithm, it will be forced to create distinct regions around them although only one region should exist.

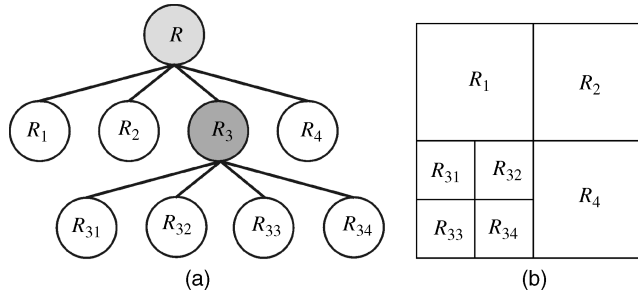


FIGURE 15.9 The quadtree data structure used in the split and merge segmentation algorithm (a) and the corresponding regions in the image (b).

15.3.2 Region Splitting and Merging

Region splitting is a top-down approach to image segmentation. It starts from the entire image and partitions it into smaller subimages until each resulting region is considered homogeneous by some criteria. At the end of the process, it is guaranteed that the resulting regions satisfy the homogeneity criterion. It is possible, however, that two or more adjacent regions are similar enough that should be combined into one. This is the goal of the *merging* step: to merge two or more adjacent regions into one if they satisfy a homogeneity criterion.

The data structure most commonly used for this algorithm is the *quadtree*, a special type of tree in which each node (except for the leaves) has four children. Each leaf node in the quadtree corresponds to a region in the segmented image (Figure 15.9).

The split and merge segmentation algorithm can be described in algorithmic form as follows:

1. Define a logical uniformity predicate $P(R_i)$.
2. Compute $P(R_i)$ for each region.
3. Split into four disjoint quadrants any region R_i for which $P(R_i) = \text{FALSE}$.
4. Repeat steps 2 and 3 until all resulting regions satisfy the uniformity criterion, that is, $P(R_i) = \text{TRUE}$.
5. Merge any adjacent regions R_j and R_k for which $P(R_j \cup R_k) = \text{TRUE}$.
6. Repeat step 5 until no further merging is possible.

15.4 WATERSHED SEGMENTATION

In this section, we describe a popular application of the morphological watershed transform in image segmentation. The watershed transform is a morphological technique that derives its name from an expression in geography, where *watershed* is

defined as the ridge that divides areas drained by different river systems. A related term, *catchment* (or *drainage*) *basin*, is used to represent the geographical area that drains into a river or reservoir.

In morphological image processing, the watershed transform is used to represent regions in a segmented image (equivalent to catchment basins) and the boundaries among them (analogous to the ridge lines).

In MATLAB

The IPT function `watershed` implements the watershed transform. It takes an input image and (optionally) a connectivity criterion (4- or 8-connectivity) as input parameters and produces a labeled matrix (of the same size as the input image) as a result. Elements labeled 1 and higher belong to a unique watershed region, identified by their number, whereas elements labeled 0 do not belong to any watershed region.

15.4.1 The Distance Transform

The distance transform is a useful tool employed in conjunction with the watershed transform. It computes the distance from every pixel to the nearest nonzero-valued pixel. It is implemented in MATLAB by function `bwdist`, which allows specification of the distance method (Euclidean distance being the default) to be used.

■ EXAMPLE 15.4

This example shows the creation of a test matrix of size 5×5 and the results of computing the distance transform using `bwdist` and two different distance calculations: Euclidean and city block.

```
>> a = [0 1 1 0 1; 1 1 1 0 0; 0 0 0 1 0; 0 0 0 0 0; 0 1 0 0 0]

a =

     0     1     1     0     1
     1     1     1     0     0
     0     0     0     1     0
     0     0     0     0     0
     0     1     0     0     0

>> b = bwdist(a)
```

```
b =

    1.0000         0         0    1.0000         0
         0         0         0    1.0000    1.0000
    1.0000    1.0000    1.0000         0    1.0000
    1.4142    1.0000    1.4142    1.0000    1.4142
    1.0000         0    1.0000    2.0000    2.2361

>> b = bwdist(a,'cityblock')

b =

     1     0     0     1     0
     0     0     0     1     1
     1     1     1     0     1
     2     1     2     1     2
     1     0     1     2     3
```

■ **EXAMPLE 15.5**

Figure 15.10 shows an example of segmentation using watershed. It uses a binarized and postprocessed version of the coins test image as input (a). Part (b) shows the results of the distance transform calculations. Part (c) shows the ridge lines obtained as a result of applying the watershed transform. Finally, part (d) shows the overlap between parts (a) and (c), indicating how the watershed transform results lead to an excellent segmentation result in this particular case.

15.5 TUTORIAL 15.1: IMAGE THRESHOLDING

Goal

The goal of this tutorial is to learn to perform image thresholding using MATLAB and the IPT.

Objectives

- Learn how to visually select a threshold value using a heuristic approach.
- Explore the `graythresh` function for automatic threshold value selection.
- Learn how to implement adaptive thresholding.

What You Will Need

- `gradient_with_text.tif`

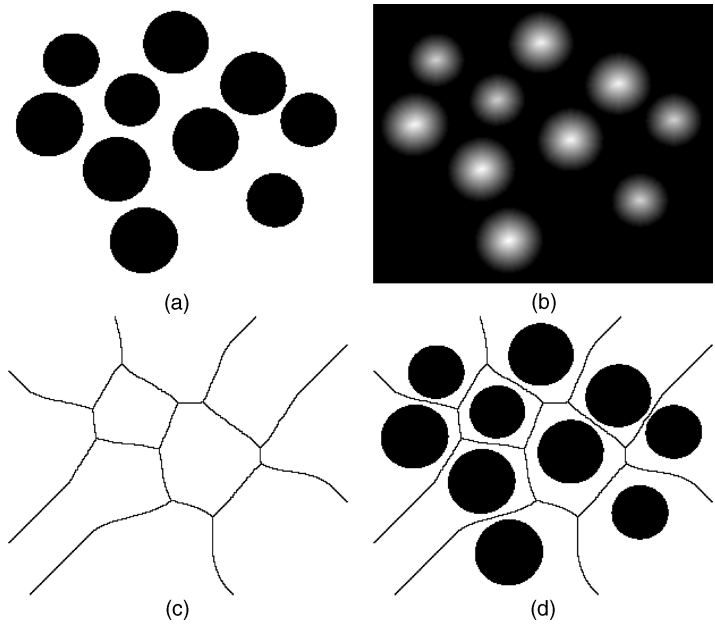


FIGURE 15.10 Segmentation using the morphological watershed transform: (a) complement of the image shown in Figure 15.3; (b) distance transform; (c) watershed ridge lines; (d) result of segmentation.

Procedure

Global Thresholding

The first method of thresholding that we will explore involves visually analyzing the histogram of an image to determine the appropriate value of `T` (the threshold value).

1. Load and display the test image.

```
I = imread('coins.png');
figure, imshow(I), title('Original Image');
```

2. Display a histogram plot of the `coins` image to determine what threshold level to use.

```
figure, imhist(I), title('Histogram of Image');
```

Question 1 Which peak of the histogram represents the background pixels and which peak represents the pixels associated with the coins?

The histogram of the image suggests a bimodal distribution of grayscale values. This means that the objects in the image are clearly separated from the background.

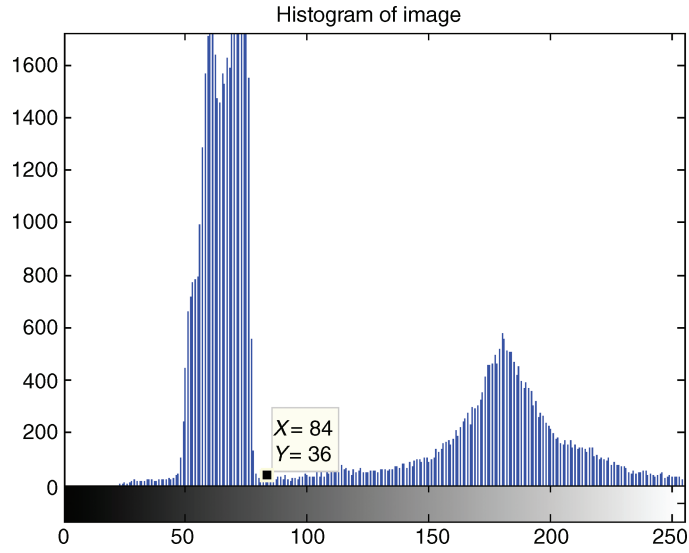


FIGURE 15.11 Histogram plot with data cursor selection.

We can inspect the `X` and `Y` values of the histogram plot by clicking the "inspect" icon and then selecting a particular bar on the graph.

3. Inspect the histogram near the right of the background pixels by activating the data cursor. To do so, click on the "inspect" icon.

Figure 15.11 illustrates what the selection should look like. The data cursor tool suggests that values between 80 and 85 could possibly be used as a threshold, since they fall immediately to the right of the leftmost peak in the histogram. Let us see what happens if we use a threshold value of 85.

4. Set the threshold value to 85 and generate the new image.

```
T = 85; I_thresh = im2bw(I,( T / 255));
figure, imshow(I_thresh), title('Threshold Image (heuristic)');
```

Question 2 What is the purpose of the `im2bw` function?

Question 3 Why do we divide the threshold value by 255 in the `im2bw` function call?

You may have noticed that several pixels—some white pixels in the background and a few black pixels where coins are located—do not belong in the resulting image. This small amount of noise can be cleaned up using the noise removal techniques discussed in chapter 12.

Question 4 Write one or more lines of MATLAB code to remove the noise pixels in the thresholded image.

The thresholding process we just explored is known as the *heuristic* approach. Although it did work, it cannot be extended to automated processes. Imagine taking on the job of thresholding a thousand images using the heuristic approach! MATLAB's IPT function `graythresh` uses Otsu's method [Ots79] for automatically finding the best threshold value.

5. Use the `graythresh` function to generate the threshold value automatically.

```
T2 = graythresh(I);
I_thresh2 = im2bw(I,T2);
figure, imshow(I_thresh2), title('Threshold Image (graythresh)');
```

Question 5 How did the `graythresh` function compare with the heuristic approach?

Adaptive Thresholding

Bimodal images are fairly easy to separate using basic thresholding techniques discussed thus far. Some images, however, are not as well behaved and require a more advanced thresholding technique such as *adaptive thresholding*. Take, for example, one of the images we used back in Tutorial 6.1: a scanned text document with a nonuniform gradient background.

6. Close all open figures and clear all workspace variables.
7. Load the `gradient_with_text` image and prepare a subplot.

```
I = imread('gradient_with_text.tif');
figure, imshow(I), title('Original Image');
```

Let us see what happens when we attempt to threshold this image using the techniques we have learned so far.

8. Globally threshold the image.

```
I_gthresh = im2bw(I,graythresh(I));
figure, imshow(I_gthresh), title('Global Thresholding');
figure, imhist(I), title('Histogram of Original');
```

As you may have noticed, we cannot pick one particular value to set as the threshold value because the image is clearly not bimodal. Adaptive thresholding may help us in this instance. To properly implement adaptive thresholding, we must use the `blkproc` function to perform an operation on small blocks of pixels one at a time.

In order to use the function, we must specify what is to be done on each block of pixels. This can be specified within a function that we will create manually. Let us first set up this function.

9. Close all open figures.
10. Start a new M-File in the MATLAB Editor.
11. Define the function as well as its input and output parameters in the first line.

```
function y = adapt_thresh(x)
```

This function will be used to define each new block of pixels in our image. Basically all we want to do is perform thresholding on each block individually, so the code to do so will be similar to the code we previously used for thresholding.

12. Add this line of code under the function definition.

```
y = im2bw(x,graythresh(x));
```

When the function is called, it will be passed a small portion of the image, and will be stored in the variable `x`. We define our output variable `y` as a black and white image calculated by thresholding the input.

13. Save your function as `adapt_thresh.m` in the current directory.

We can now perform the operation using the `blkproc` function. We will adaptively threshold the image, 10×10 pixel blocks at a time.

14. Perform adaptive thresholding by entering the following command in the command window. Note that it may take a moment to perform the calculation, so be patient.

```
I_thresh = blkproc(I,[10 10],@adapt_thresh);
```

15. Display the original and new image.

```
figure
subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I_thresh), title('Adaptive Thresholding');
```

The output is not quite what we expected. If you look closely, however, the operation was successful near the text, but everywhere else it was a disaster. This suggests that we need to add an extra step to our function to compensate for this unwanted effect. Over the next few steps, let us examine the standard deviation of the original image where there *is* text, and where there is *not*.

16. Calculate the standard deviation of two 10×10 blocks of pixels; one where there is text and another where there is not.

```
std_without_text = std2(I(1:10, 1:10))
std_with_text = std2(I(100:110, 100:110))
```

Question 6 What is the difference between the standard deviation of the two blocks of pixels? Explain.

Since there is such a difference between a block with and without text, we can use this information to improve our function. Let us replace the one line of code we previously wrote with the new code that will include an `if` statement: if the standard deviation of the block of pixels is low, then simply label it as background; otherwise, perform thresholding on it. This change should cause the function to only perform thresholding where text exists. Everything else will be labeled as background.

17. Replace the last line of our function with the following code. Save the function after the alteration.

```
if std2(x) < 1
    y = ones(size(x,1),size(x,2));
else
    y = im2bw(x,graythresh(x));
end
```

Question 7 How does our function label a block of pixels as background?

18. Now rerun the block process (in the command window) to see the result.

```
I_thresh2 = blkproc(I,[10 10],@adapt_thresh);
figure, subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I_thresh2), title('Adaptive Thresholding');
```

Question 8 How does the output of the new function compare with the old?

Question 9 What is the main limitation of the adaptive thresholding function developed in this tutorial?

WHAT HAVE WE LEARNED?

- Image segmentation is the process of grouping image pixels into meaningful, usually connected, regions. It is an important (and often required) step in many image processing solutions. It establishes the transition between treating the image as a whole to processing individual relevant regions.

- Image segmentation is a hard image processing problem: the quality of the results will depend on the algorithm, careful selection of algorithm's parameters, and the input image.
- Thresholding is an image processing technique by which an input (grayscale) image is requantized to two gray levels, that is, converted to a binary image. Each pixel in the original image is compared with a threshold; the result of such comparison will determine whether the pixel will be converted to black or white. The simplest thresholding algorithm (*global thresholding*, `im2bw` in MATLAB) employs one value for the entire image.
- Image segmentation techniques can be classified in three main groups: *intensity-based methods* (e.g., thresholding), *region-based methods* (e.g., region growing and split and merge), and *other methods* (e.g., segmentation based on texture, edges, and motion).

LEARN MORE ABOUT IT

- Entire books have been written on the topic of image segmentation, for example, [Wan08] and [Zha06a].
- Many surveys on image segmentation have been published during the past 30 years, among them (in reverse chronological order): [Zha06b], [FMR$^+$02], [CJSW01], [PP93], [HS85], and [FM81].
- Chapters 6 and 7 of [SHB08] provide an extensive and very readable discussion of image segmentation algorithms. Several of these algorithms have been implemented in MATLAB [SKH08].
- Chapter 4 of [Dav04] is entirely devoted to thresholding techniques.
- Section 6.1.2 of [SHB08] discusses optimal thresholding techniques.
- The concept of thresholding can be extended to cases where the original image is to be partitioned in more than two regions, in what is known as *multiple thresholding*. Refer to Section 10.3.6 of [GW08] for a discussion on this topic.
- Chapters 4.7–4.9 of [Bov00a] discuss statistical, texture-based, and motion-based segmentation strategies, respectively.
- The discussion on watershed segmentation is expanded to include the use of gradients and markers in Section 10.5 of [GWE04].
- Comparing and evaluating different segmentation approaches can be a challenging task, for which no universally accepted benchmarks exist. This issue is discussed in [Zha96], [Zha01], and, more recently, [UPH05], expanded in [UPH07].

ON THE WEB

- Color-based segmentation using K-means clustering (IPT image segmentation demo)
  http://tinyurl.com/matlab-k-means

- Color-based segmentation using the L*a*b* color space (IPT image segmentation demo)
  http://tinyurl.com/matlab-lab
- Detecting a cell using image segmentation (IPT image segmentation demo)
  http://tinyurl.com/cell-seg
- Marker-controlled watershed segmentation (IPT image segmentation demo)
  http://tinyurl.com/watershed-seg
- Texture segmentation Using Texture Filters (IPT image segmentation demo)
  http://tinyurl.com/texture-seg
- University of Washington image segmentation demo
  http://www.cs.washington.edu/research/imagedatabase/demo/seg/

15.6 PROBLEMS

15.1 Explain in your own words why the image on the top right of Figure 15.1 is significantly harder to segment than the one on the bottom left of the same figure.

15.2 Modify the MATLAB code to perform iterative threshold selection on an input gray-level image (section 15.2.2) to include a variable that counts the number of iterations and an array that stores the values of T for each iteration.

15.3 Write a MATLAB script to demonstrate that thresholding techniques can be used to subtract the background of an image.