12 Object Recognition

Preview

We conclude the book with a discussion and development of several M-functions for region and/or boundary recognition, which in this chapter we call *objects* or *patterns*. Approaches to computerized pattern recognition may be divided into two principal areas: decision-theoretic and structural. The first category deals with patterns described using quantitative descriptors, such as length, area, texture, and many of the other descriptors discussed in Chapter 11. The second category deals with patterns best represented by symbolic information, such as strings, and described by the properties and relationships between those symbols, as explained in Section 12.4. Central to the theme of recognition is the concept of "learning" from sample patterns. Learning techniques for both decision-theoretic and structural approaches are implemented and illustrated in the material that follows.

12.1 Background

A *pattern* is an *arrangement of descriptors*, such as those discussed in Chapter 11. The name *feature* is used often in the pattern recognition literature to denote a descriptor. A *pattern class* is a family of patterns that share a set of common properties. Pattern classes are denoted $\omega_1, \omega_2, \ldots, \omega_W$, where W is the number of classes. Pattern recognition by machine involves techniques for assigning patterns to their respective classes—automatically and with as little human intervention as possible.

The two principal pattern arrangements used in practice are vectors (for quantitative descriptions) and strings (for structural descriptions). Pattern vectors are represented by bold lowercase letters, such as $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, and have the $n \times 1$ vector form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

where each component, x_i, represents the ith descriptor and n is the total number of such descriptors associated with the pattern. Sometimes it is necessary in computations to use row vectors of dimension $1 \times n$, obtained simply by forming the transpose, $\mathbf{x}^T$, of the preceding column vector.

The nature of the components of a pattern vector $\mathbf{x}$ depends on the approach used to describe the physical pattern itself. For example, consider the problem of automatically classifying alphanumeric characters. Descriptors suitable for a decision-theoretic approach might include measures such 2-D moment invariants or a set of Fourier coefficients describing the outer boundary of the characters.

In some applications, pattern characteristics are best described by structural relationships. For example, fingerprint recognition is based on the interrelationships of print features called *minutiae*. Together with their relative sizes and locations, these features are primitive components that describe fingerprint ridge properties, such as abrupt endings, branching, merging, and disconnected segments. Recognition problems of this type, in which not only quantitative measures about each feature but also the spatial relationships between the features determine class membership, generally are best solved by structural approaches.

The material in the following sections is representative of techniques for implementing pattern recognition solutions in MATLAB. A basic concept in recognition, especially in decision-theoretic applications, is the idea of pattern matching based on measures of distance between pattern vectors. Therefore, we begin the discussion with various approaches for the efficient computation of distance measures in MATLAB.

12.2 Computing Distance Measures in MATLAB

The material in this section deals with vectorizing distance computations that otherwise would involve `for` or `while` loops. Some of the vectorized expressions presented here are considerably more subtle than most of the examples in the previous chapters, so the reader is encouraged to study them in detail. The following formulations are based on a summary of similar expressions compiled by Acklam [2002].

The *Euclidean distance* between two n-dimensional (row or column) vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as the scalar

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| = \|\mathbf{y} - \mathbf{x}\| = [(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2]^{1/2}$$

This expression is simply the norm of the difference between the two vectors, so we compute it using MATLAB's function `norm`:

```
d = norm(x - y)
```

where x and y are vectors corresponding to $\mathbf{x}$ and $\mathbf{y}$ in the preceding equation for $d(\mathbf{x}, \mathbf{y})$. Often, it is necessary to compute a set of Euclidean distances between a vector $\mathbf{y}$ and each member of a vector *population* consisting of p, n-dimensional vectors arranged as the rows of a $p \times n$ matrix $\mathbf{X}$. For the dimensions to line up properly, $\mathbf{y}$ has to be of dimension $1 \times n$. Then the distance between $\mathbf{y}$ and each element of $\mathbf{X}$ is contained in the $p \times 1$ vector

```
d = sqrt(sum(abs(X - repmat(y, p, 1)).^2, 2))
```

where `d(i)` is the Euclidean distance between $\mathbf{y}$ and the ith row of $\mathbf{X}$ [i.e., `X(i, :)`]. Note the use of function `repmat` to duplicate row vector $\mathbf{y}$ p times and thus form a $p \times n$ matrix to match the dimensions of $\mathbf{X}$. The last 2 on the right of the preceding line of code indicates that `sum` is to operate along dimension 2; that is, to sum the elements along the horizontal dimension.

Suppose next that we have two vector populations $\mathbf{X}$, of dimension $p \times n$, and $\mathbf{Y}$, of dimension $q \times n$. The matrix containing the distances between rows of these two populations can be obtained using the expression

```
D = sqrt(sum(abs(repmat(permute(X, [1 3 2]), [1 q 1]) ...
          - repmat(permute(Y, [3 1 2]), [p 1 1])).^2, 3))
```

where `D(i,j)` is the Euclidean distance between the ith and jth rows of the populations; that is, the distance between `X(i,:)` and `Y(j,:)`.

The syntax for function `permute` in the preceding expression is


```
B = permute(A, order)
```

This function reorders the dimensions of `A` according to the elements of the vector `order` (the elements of this vector must be unique). For example, if `A` is a 2-D array, the statement `B = permute(A,[2 1])` simply interchanges the rows and columns of `A`, which is equivalent to letting `B` equal the transpose of `A`. If the length of vector `order` is greater than the number of dimensions of `A`, MATLAB processes the components of the vector from left to right, until all elements are used. In the preceding expression for `D`, `permute(X, [1 3 2])` creates arrays in the third dimension, each being a column (dimension 1) of `X`. Since there are n columns in `X`, n such arrays are created, with each array being of dimension $p \times 1$. Therefore, the command `permute(X, [1 3 2])` creates an array of dimension $p \times 1 \times n$. Similarly, the command `permute(Y, [3 1 2])` creates an array of dimension $1 \times q \times n$. Finally, the command `repmat(permute(X, [1 3 2]), [1 q 1])` duplicates q times each of the n columns produced by the `permute` function, thus creating an array of dimension $p \times q \times n$. Similar comments hold for the other command involving `Y`. Basically, the preceding expression for `D` is simply a vectorization of the expressions that would be written using `for` or `while` loops.

In addition to the expressions just discussed, we use in this chapter a distance measure from a vector $\mathbf{y}$ to the mean $\mathbf{m}_\mathbf{x}$ of a vector population, weighted inversely by the covariance matrix, $\mathbf{C}_\mathbf{x}$, of the population. This metric, called the *Mahalanobis distance*, is defined as

$$d(\mathbf{y}, \mathbf{m}_\mathbf{x}) = (\mathbf{y} - \mathbf{m}_\mathbf{x})^T \mathbf{C}_\mathbf{x}^{-1} (\mathbf{y} - \mathbf{m}_\mathbf{x})$$

The inverse matrix operation is the most time-consuming computational task required to implement the Mahalanobis distance. This operation can be optimized significantly by using MATLAB's matrix right division operator (/) introduced in Table 2.4 (see also the margin note in the following page). Expressions for $\mathbf{m}_x$ and $\mathbf{C}_x$ are given in Section 11.5.

Let $\mathbf{X}$ denote a population of p, n-dimensional vectors, and let $\mathbf{Y}$ denote a population of q, n-dimensional vectors, such that the vectors in both $\mathbf{X}$ and $\mathbf{Y}$ are the rows of these arrays. The objective of the following M-function is to compute the Mahalanobis distance between every vector in $\mathbf{Y}$ and the mean, $\mathbf{m}_x$:

mahalanobis

```
function d = mahalanobis(varargin)
%MAHALANOBIS Computes the Mahalanobis distance.
%   D = MAHALANOBIS(Y, X) computes the Mahalanobis distance between
%   each vector in Y to the mean (centroid) of the vectors in X, and
%   outputs the result in vector D, whose length is size(Y, 1). The
%   vectors in X and Y are assumed to be organized as rows. The
%   input data can be real of complex. The outputs are real
%   quantities.
%
%   D = MAHALANOBIS(Y, CX, MX) computes the Mahalanobis distance
%   between each vector in Y and the given mean vector, MX. The
%   results are output in vector D, whose length is size(Y, 1). The
%   vectors in Y are assumed to be organized as the rows of this
%   array. The input data can be real or complex. The outputs are
%   real quantities. In addition to the mean vector MX, the
%   covariance matrix CX of a population of vectors X also must be
%   provided. Use function COVMATRIX (Section 11.5) to compute MX and
%   CX.

% Reference: Acklam, P. J. [2002]. "MATLAB Array Manipulation Tips
% and Tricks." Available at
%     home.online.no/~pjacklam/matlab/doc/mtt/index.html
% or at
%     www.prenhall.com/gonzalezwoodseddins

param = varargin; % Keep in mind that param is a cell array.
Y = param{1};
ny = size(Y, 1); % Number of vectors in Y.

if length(param) == 2
   X = param{2};
   % Compute the mean vector and covariance matrix of the vectors
   % in X.
   [Cx, mx] = covmatrix(X);
elseif length(param) == 3 % Cov. matrix and mean vector provided.
   Cx = param{2};
   mx = param{3};
else
   error('Wrong number of inputs.')
end
```

```
mx = mx(:)'; % Make sure that mx is a row vector.

% Subtract the mean vector from each vector in Y.
Yc = Y - mx(ones(ny, 1), :);

% Compute the Mahalanobis distances.
d = real(sum(Yc/Cx.*conj(Yc), 2));
```

With A a square matrix, the MATLAB matrix operation `A/B` *is a more accurate (and generally faster) implementation of the operation* `B*inv(A)`. *Similarly,* `A\B` *is a preferred implementation of the operation* `inv(A)*B`. *See Table 2.4.*

The call to `real` in the last line of code is to remove "numeric noise," as we did in Chapter 4 after filtering an image. If the data are known to always be real, the code can be simplified by removing functions `real` and `conj`.

12.3 Recognition Based on Decision-Theoretic Methods

Decision-theoretic approaches to recognition are based on the use of *decision* (also called *discriminant*) *functions*. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ represent an n-dimensional pattern vector, as discussed in Section 12.1. For W pattern classes, $\omega_1, \omega_2, \ldots, \omega_W$, the basic problem in decision-theoretic pattern recognition is to find W decision functions $d_1(\mathbf{x}), d_2(\mathbf{x}), \ldots, d_W(\mathbf{x})$ with the property that if a pattern $\mathbf{x}$ belongs to class ω_i, then

$$d_i(\mathbf{x}) > d_j(\mathbf{x}) \qquad j = 1, 2, \ldots, W; j \neq i$$

In other words, an unknown pattern $\mathbf{x}$ is said to belong to the ith pattern class if, upon substitution of $\mathbf{x}$ into all decision functions, $d_i(\mathbf{x})$ yields the largest numerical value. Ties are resolved arbitrarily.

The *decision boundary* separating class ω_i from ω_j is given by values of $\mathbf{x}$ for which $d_i(\mathbf{x}) = d_j(\mathbf{x})$ or, equivalently, by values of $\mathbf{x}$ for which

$$d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0$$

Common practice is to express the decision boundary between two classes by the single function $d_{ij}(\mathbf{x}) = d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0$. Thus $d_{ij}(\mathbf{x}) > 0$ for patterns of class ω_i and $d_{ij}(\mathbf{x}) < 0$ for patterns of class ω_j.

As will become clear in the following sections, finding decision functions entails estimating parameters from patterns that are representative of the classes of interest. Patterns used for parameter estimation are called *training patterns*, or *training sets*. Sets of patterns of known classes that are not used for training but are used instead to test the performance of a particular recognition approach are referred to as *test* or *independent* patterns or sets. The principal objective of Sections 12.3.2 and 12.3.4 is to develop various approaches for finding decision functions via the use of parameter estimation from training sets. Section 12.3.3 deals with matching by correlation, an approach that could be expressed in the form of decision functions but is traditionally presented in the form of direct image matching instead.

12.3.1 Forming Pattern Vectors

As noted at the beginning of this chapter, pattern vectors can be formed from quantitative descriptors, such as those discussed in Chapter 11 for regions and/or boundaries. For example, suppose that we describe a boundary by using

Fourier descriptors. The value of the ith descriptor becomes the value of x_i, the ith component of a pattern vector. In addition, we could append other components to pattern vectors. For instance, we could incorporate six additional components to the Fourier-descriptor by appending to each vector the six measures of texture in Table 11.2.

Another approach used quite frequently when dealing with (registered) multispectral images is to stack the images and then form vectors from corresponding pixels in the images, as illustrated in Fig. 11.24. The images are stacked by using function `cat`:

```
S = cat(3, f1, f2, ..., fn)
```

where `S` is the stack and `f1, f2, ..., fn` are the images from which the stack is formed. The vectors then are generated by using function `imstack2vectors` discussed in Section 11.5. See Example 12.2 for an illustration.

12.3.2 Pattern Matching Using Minimum-Distance Classifiers

Suppose that each pattern class, ω_j, is characterized by a mean vector $\mathbf{m}_j$. That is, we use the mean vector of each population of training vectors as being representative of that class of vectors:

$$\mathbf{m}_j = \frac{1}{N_j} \sum_{\mathbf{x} \in \omega_j} \mathbf{x} \qquad j = 1, 2, \ldots, W$$

where N_j is the number of training pattern vectors from class ω_j and the summation is taken over these vectors. As before, W is the number of pattern classes. One way to determine the class membership of an *unknown* pattern vector $\mathbf{x}$ is to assign it to the class of its closest prototype. Using the Euclidean distance as a measure of closeness (i.e., similarity) reduces the problem to computing the distance measures:

$$D_j(\mathbf{x}) = \|\mathbf{x} - \mathbf{m}_j\| \qquad j = 1, 2, \ldots, W$$

We then assign $\mathbf{x}$ to class ω_i if $D_i(\mathbf{x})$ is the smallest distance. That is, the smallest distance implies the best match in this formulation.

Suppose that all the mean vectors are organized as rows of a matrix $\mathbf{M}$. Then computing the distances from an arbitrary pattern $\mathbf{x}$ to all the mean vectors is accomplished by using the expression discussed in Section 12.2:

```
d = sqrt(sum(abs(M - repmat(x, W, 1)).^2, 2))
```

Because all distances are positive, this statement can be simplified by ignoring the `sqrt` operation. The minimum of `d` determines the class membership of pattern vector `x`:

```
>> class = find(d == min(d));
```

In other words, if the minimum of `d` is in its kth position (i.e., `x` belongs to the kth pattern class), then scalar `class` will equal `k`. If more than one minimum

exists, `class` would equal a vector, with each of its elements pointing to a different location of the minimum.

If, instead of a single pattern, we have a set of patterns arranged as the rows of a matrix, `X`, then we use an expression similar to the longer expression in Section 12.2 to obtain a matrix `D`, whose element `D(I, J)` is the Euclidean distance between the *i*th pattern vector in `X` and the *j*th mean vector in `M`. Thus, to find the class membership of, say, the *i*th pattern in `X`, we find the column location in row `i` of `D` that yields the smallest value. Multiple minima yield multiple values, as in the single-vector case discussed in the last paragraph.

It is not difficult to show that selecting the smallest distance is equivalent to evaluating the functions

$$d_j(\mathbf{x}) = \mathbf{x}^T\mathbf{m}_j - \frac{1}{2}\mathbf{m}_j^T\mathbf{m}_j \qquad j = 1, 2, \ldots, W$$

and assigning $\mathbf{x}$ to class ω_i if $d_i(\mathbf{x})$ yields the *largest* numerical value. This formulation agrees with the concept of a decision function defined earlier.

The decision boundary between classes ω_i and ω_j for a minimum distance classifier is

$$\begin{aligned} d_{ij}(\mathbf{x}) &= d_i(\mathbf{x}) - d_j(\mathbf{x}) \\ &= \mathbf{x}^T(\mathbf{m}_i - \mathbf{m}_j) - \frac{1}{2}(\mathbf{m}_i - \mathbf{m}_j)^T(\mathbf{m}_i + \mathbf{m}_j) = 0 \end{aligned}$$

The surface given by this equation is the perpendicular bisector of the line segment joining $\mathbf{m}_i$ and $\mathbf{m}_j$. For $n = 2$, the perpendicular bisector is a line, for $n = 3$ it is a plane, and for $n > 3$ it is called a *hyperplane.*

12.3.3 Matching by Correlation

Correlation is quite simple in principle. Given an image $f(x, y)$, the correlation problem is to find all places in the image that match a given subimage (also called a *mask* or *template*) $w(x, y)$. Typically, $w(x, y)$ is much smaller than $f(x, y)$. One approach for finding matches is to treat $w(x, y)$ as a spatial filter and compute the sum of products (or a normalized version of it) for each location of w in f, in exactly the same manner explained in Section 3.4.1. Then the best match (matches) of $w(x, y)$ in $f(x, y)$ is (are) the location(s) of the maximum value(s) in the resulting correlation image. Unless $w(x, y)$ is small, the approach just described generally becomes computationally intensive. For this reason, practical implementations of spatial correlation typically rely on hardware-oriented solutions.

For prototyping, an alternative approach is to implement correlation in the frequency domain, making use of the correlation theorem, which, like the convolution theorem discussed in Chapter 4, relates spatial correlation to the product of the image transforms. Letting " $\circ$ " denote correlation and "$*$" the complex conjugate, the correlation theorem states that

$$f(x, y) \circ w(x, y) \Leftrightarrow F(u, v)H^*(u, v)$$

In other words, spatial correlation can be obtained as the inverse Fourier transform of the product of the transform of one function times the conjugate of the transform of the other. Conversely, it follows that

$$f(x, y)w^*(x, y) \Leftrightarrow F(u, v) \circ H(u, v)$$

This second aspect of the correlation theorem is included for completeness. It is not used in this chapter.

Implementation of the first correlation result in the form of an M-function is straightforward, as the following code shows.

dftcorr

```
function g = dftcorr(f, w)
%DFTCORR 2-D correlation in the frequency domain.
%   G = DFTCORR(F, W) performs the correlation of a mask, W, with
%   image F. The output, G, is the correlation image, of class
%   double. The output is of the same size as F. When, as is
%   generally true in practice, the mask image is much smaller than
%   G, wraparound error is negligible if W is padded to size(F).
[M, N] = size(f);
f = fft2(f);
w = conj(fft2(w, M, N));
g = real(ifft2(w.*f));
```

EXAMPLE 12.1: Using correlation for image matching.

■ Figure 12.1(a) shows an image of Hurricane Andrew, in which the eye of the storm is clearly visible. As an example of correlation, we wish to find the location of the best match in (a) of the eye image in Fig. 12.1(b). The image is of size 912 × 912 pixels; the mask is of size 32 × 32 pixels. Figure 12.1(c) is the result of the following commands:

```
>> g = dftcorr(f, w);
>> gs = gscale(g);
>> imshow(gs)
```

The blurring evident in the correlation image of Fig. 12.1(c) should not be a surprise because the image in 12.1(b) has two dominant, nearly constant regions, and thus behaves similarly to a lowpass filter.

The feature of interest is the location of the best match, which, for correlation, implies finding the location(s) of the highest value in the correlation image:

```
>> [I, J] = find(g == max(g(:)))
I =
    554
J =
    203
```

In this case the highest value is unique. As explained in Section 3.4.1, the coordinates of the correlation image correspond to displacements of the template, so coordinates `[I, J]` correspond to the location of the bottom, left corner of

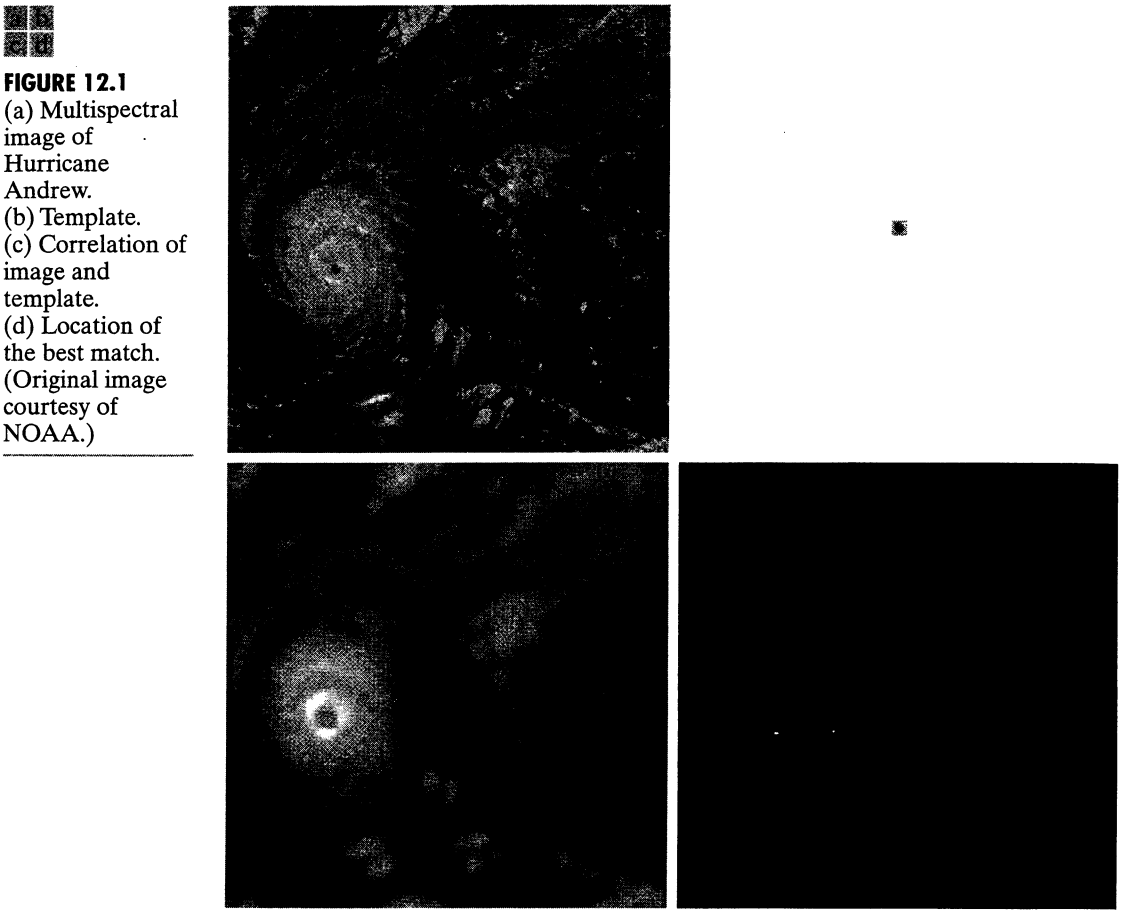


FIGURE 12.1
(a) Multispectral image of Hurricane Andrew.
(b) Template.
(c) Correlation of image and template.
(d) Location of the best match.
(Original image courtesy of NOAA.)

See Fig. 3.14 for an explanation of the mechanics of correlation.

the template. If the template were so located on top of the image, we would find that the template aligns quite closely with the eye of the hurricane at those coordinates. Another approach for finding the locations of the matches is to threshold the correlation image near its maximum, or threshold its scaled version, `gs`, whose highest value is known to be 255. For example, the image in Fig. 12.1(d) was obtained using the command

```
>> imshow(gs > 254)
```

Aligning the bottom, left corner of the template with the small white dot in Fig. 12.1(d) again reveals that the best match is near the eye of the hurricane. ■

12.3.4 Optimum Statistical Classifiers

The well-known Bayes classifier for a 0-1 loss function (Gonzalez and Woods [2002]) has decision functions of the form

$$d_j(\mathbf{x}) = p(\mathbf{x}/\omega_j)P(\omega_j) \quad j = 1, 2, \ldots, W$$

where $p(\mathbf{x}/\omega_j)$ is the probability density function (PDF) of the pattern vectors of class ω_j, and $P(\omega_j)$ is the probability (a scalar) that class ω_j occurs. As before, given an unknown pattern vector, the process is to compute a total of W decision functions and then assign the pattern to the class whose decision function yielded the largest numerical value. Ties are resolved arbitrarily.

The case when the probability density functions are (or are assumed to be) Gaussian is of particular practical interest. The n-dimensional Gaussian PDF has the form

$$p(\mathbf{x}/\omega_j) = \frac{1}{(2\pi)^{n/2}|\mathbf{C}_j|^{1/2}} e^{-\frac{1}{2}[(\mathbf{x}-\mathbf{m}_j)^T\mathbf{C}_j^{-1}(\mathbf{x}-\mathbf{m}_j)]}$$

where $\mathbf{C}_j$ and $\mathbf{m}_j$ are the covariance matrix and mean vector of the pattern population of class ω_j, and $|\mathbf{C}_j|$ is the determinant of $\mathbf{C}_j$.

Because the logarithm is a monotonically increasing function, choosing the largest $d_j(\mathbf{x})$ to classify patterns is equivalent to choosing the largest $\ln[d_j(\mathbf{x})]$, so we can use instead decision functions of the form

$$\begin{aligned} d_j(\mathbf{x}) &= \ln[p(\mathbf{x}/\omega_j)P(\omega_j)] \\ &= \ln p(\mathbf{x}/\omega_j) + \ln P(\omega_j) \end{aligned}$$

where the logarithm is guaranteed to be real because $p(\mathbf{x}/\omega_j)$ and $P(\omega_j)$ are nonnegative. Substituting the expression for the Gaussian PDF gives the equation

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \frac{n}{2}\ln 2\pi - \frac{1}{2}\ln|\mathbf{C}_j| - \frac{1}{2}[(\mathbf{x} - \mathbf{m}_j)^T\mathbf{C}_j^{-1}(\mathbf{x} - \mathbf{m}_j)]$$

The term $(n/2)\ln 2\pi$ is the same positive constant for all classes, so it can be deleted, yielding the decision functions

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \frac{1}{2}\ln|\mathbf{C}_j| - \frac{1}{2}[(\mathbf{x} - \mathbf{m}_j)^T\mathbf{C}_j^{-1}(\mathbf{x} - \mathbf{m}_j)]$$

for $j = 1, 2, \ldots, W$. The term inside the brackets is recognized as the Mahalanobis distance discussed in Section 12.2, for which we have a vectorized implementation. We also have an efficient method for computing the mean and covariance matrix from Section 11.5, so implementing the Bayes classifier for the multivariate Gaussian case is straightforward, as the following function shows.

bayesgauss

```
function d = bayesgauss(X, CA, MA, P)
%BAYESGAUSS Bayes classifier for Gaussian patterns.
%   D = BAYESGAUSS(X, CA, MA, P) computes the Bayes decision
%   functions of the patterns in the rows of array X using the
%   covariance matrices and and mean vectors provided in the arrays
%   CA and MA. CA is an array of size n-by-n-by-W, where n is the
%   dimensionality of the patterns and W is the number of
%   classes. Array MA is of dimension n-by-W (i.e., the columns of MA
%   are the individual mean vectors). The location of the covariance
%   matrices and the mean vectors in their respective arrays must
%   correspond. There must be a covariance matrix and a mean vector
```

```
%   for each pattern class, even if some of the covariance matrices
%   and/or mean vectors are equal. X is an array of size K-by-n,
%   where K is the total number of patterns to be classified (i.e.,
%   the pattern vectors are rows of X). P is a 1-by-W array,
%   containing the probabilities of occurrence of each class. If
%   P is not included in the argument list, the classes are assumed
%   to be equally likely.
%
%   The output, D, is a column vector of length K. Its Ith element is
%   the class number assigned to the Ith vector in X during Bayes
%   classification.

d = [ ]; % Initialize d.
error(nargchk(3, 4, nargin)) % Verify correct no. of inputs.
n = size(CA, 1);             % Dimension of patterns.

% Protect against the possibility that the class number is
% included as an (n+1)th element of the vectors.
X = double(X(:, 1:n));
W = size(CA, 3); % Number of pattern classes.
K = size(X, 1);  % Number of patterns to classify.
if nargin == 3
   P(1:W) = 1/W; % Classes assumed equally likely.
else
   if sum(P) ~= 1
      error('Elements of P must sum to 1.');
   end
end

% Compute the determinants.
for J = 1:W
   DM(J) = det(CA(:, :, J));
end

% Compute inverses, using right division (IM/CA), where IM =
% eye(size(CA, 1)) is the n-by-n identity matrix. Reuse CA to
% conserve memory.
IM = eye(size(CA,1));
for J = 1:W
   CA(:, :, J) = IM/CA(:, :, J);
end

% Evaluate the decision functions. The sum terms are the
% Mahalanobis distances discussed in Section 12.2.
MA = MA'; % Organize the mean vectors as rows.
for I = 1:K
   for J = 1:W
      m = MA(J, :);
      Y = X - m(ones(size(X, 1), 1), :);
      if P(J) == 0
         D(I, J) = -Inf;
      else
         D(I, J) = log(P(J)) - 0.5*log(DM(J)) ...
                   - 0.5*sum(Y(I, :)*(CA(:, :, J)*Y(I, :)'));
```

`eye(n)` *returns the* `n` × `n` *identity matrix;* `eye(m, n)` *or* `eye([m n])` *returns an* `m` × `n` *matrix with 1s along the diagonal and 0s elsewhere. The syntax* `eye(size(A))` *gives the same result as the previous format, with* `m` *and* `n` *being the number of rows and columns in* `A`*, respectively.*

```
      end
   end
end

% Find the maximum in each row of D. These maxima
% give the class of each pattern:
for I = 1:K
   J = find(D(I, :) == max(D(I, :)));
   d(I, :) = J(:);
end

% When there are multiple maxima the decision is
% arbitrary. Pick the first one.
d = d(:, 1);
```

EXAMPLE 12.2: Bayes classification of multispectral data.

■ Bayes recognition is used frequently for automatically classifying regions in multispectral imagery. Figure 12.2 shows the first four images from Fig. 11.25 (three visual bands and one infrared band). As a simple illustration, we apply the Bayes classification approach to three types (classes) of regions in these images: water, urban, and vegetation. The pattern vectors in this example are formed by the method discussed in Sections 11.5 and 12.3.1, in which corresponding pixels in the images are organized as vectors. We are dealing with four images, so the pattern vectors are four dimensional.

To obtain the mean vectors and covariance matrices, we need samples representative of each pattern class. A simple way to obtain such samples interactively is to use function `roipoly` (see Section 5.2.4) with the statement

```
>> B = roipoly(f);
```

where `f` is any of the multispectral images and `B` is a binary mask image. With this format, image `B` is generated interactively on the screen. Figure 12.2(e) shows a composite of three mask images, `B1`, `B2`, and `B3`, generated using this method. The numbers 1, 2, and 3 identify regions containing samples representative of water, urban development, and vegetation, respectively.

Next we obtain the vectors corresponding to each region. The four images already are registered spatially, so they simply are concatenated along the third dimension to obtain an image stack:

```
>> stack = cat(3, f1, f2, f3, f4);
```

where `f1` thorough `f4` are the four images in Figs. 12.2(a) through (d). Any point, when viewed through these four images, corresponds to a four-dimensional pattern vector (see Fig. 11.24). We are interested in the vectors contained in the three regions shown in Fig. 12.2(e), which we obtain by using function `imstack2vectors` discussed in Section 11.5:

```
>> [X, R] = imstack2vectors(stack, B);
```

where `X` is an array whose rows are the vectors, and `R` is an array whose rows are the locations (2-D region coordinates) corresponding to the vectors in `X`.

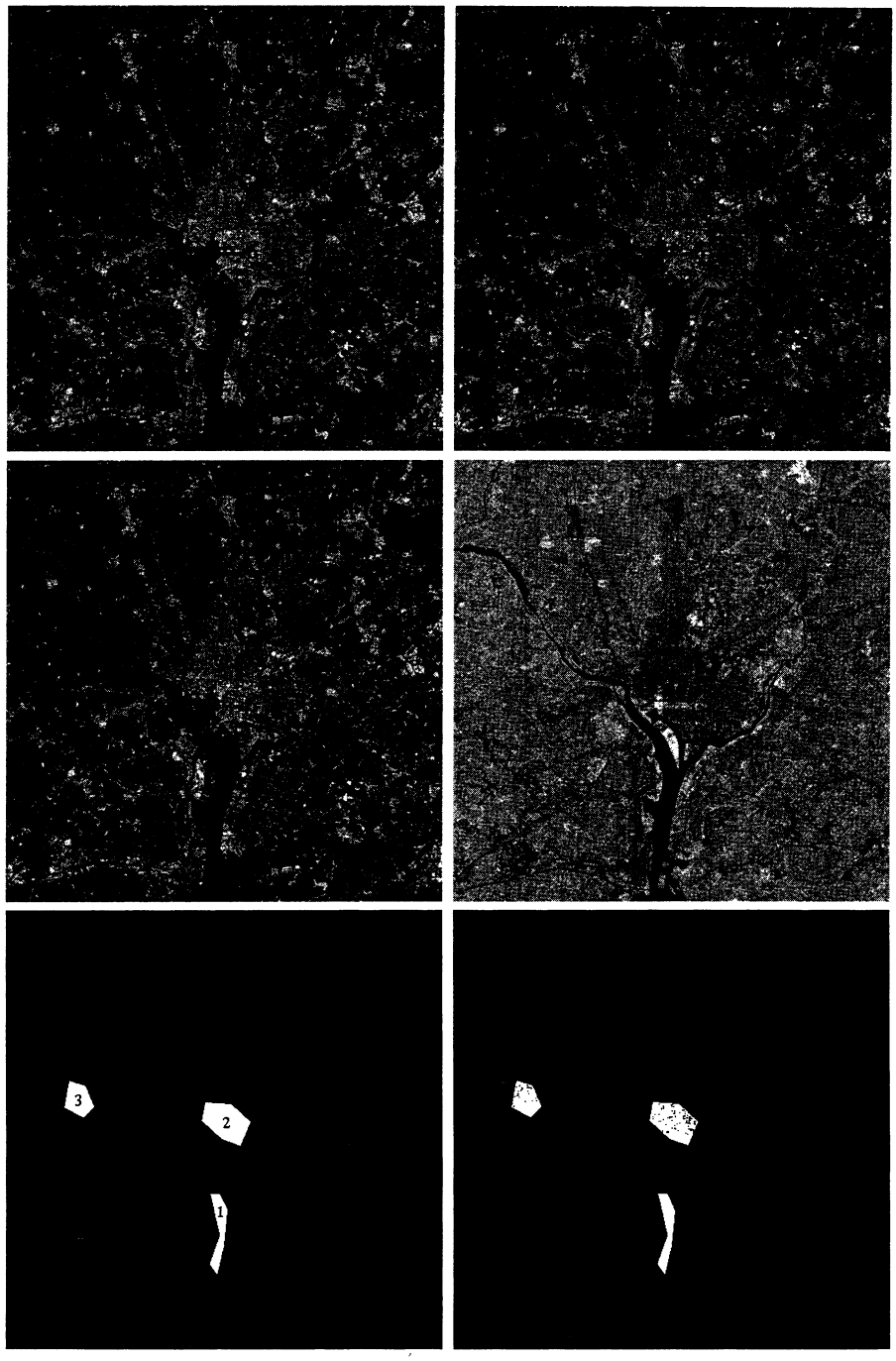


FIGURE 12.2 Bayes classification of multispectral data. (a)–(c) Images in the blue, green, and red visible wavelengths. (d) Infrared image. (e) Mask showing sample regions of water (1), urban development (2), and vegetation (3). (f) Results of classification. The black dots denote points classified incorrectly. The other (white) points in the regions were classified correctly. (Original images courtesy of NASA.)

Using `imstack2vectors` with the three masks `B1`, `B2`, and `B3` yielded three vector sets, `X1`, `X2`, and `X3`, and three sets of coordinates, `R1`, `R2`, and `R3`. Then three subsets `Y1`, `Y2`, and `Y3` were extracted from the `X`'s to use as training samples to estimate the covariance matrices and mean vectors. The `Y`'s were generated by skipping every other row of `X1`, `X2`, and `X3`. The covariance matrix and mean vector of the vectors in `Y1` were obtained with the command

```
>> [C1, m1] = covmatrix(Y1);
```

and similarly for the other two classes. Then we formed arrays `CA` and `MA` for use in `bayesgauss` as follows:

```
>> CA = cat(3, C1, C2, C3);
>> MA = cat(2, m1, m2, m3);
```

The performance of the classifier with the training patterns was determined by classifying the training sets:

```
>> dY1 = bayesgauss(Y1, CA, MA);
```

and similarly for the other two classes. The number of misclassified patterns of class 1 was obtained by writing

```
>> IY1 = find(dY1 ~= 1);
```

Finding the class into which the patterns were misclassified is straightforward. For instance, `length(find(dY1 == 2))` gives the number of patterns from class 1 that were misclassified into class 2. The other pattern sets were handled in a similar manner.

Table 12.1 summarizes the recognition results obtained with the training and independent pattern sets. The percentage of training and independent patterns recognized correctly was about the same with both sets, indicating stability in the parameter estimates. The largest error in both cases was with patterns from the urban area. This is not unexpected, as vegetation is present there also (note that no patterns in the urban or vegetation areas were misclassified as water). Figure 12.2(f) shows as black dots the points that were misclassified and as white dots the points that were classified correctly in each region. No black dots are readily visible in region 1 because the 7 misclassified points are very close to, or on, the boundary of the white region.

Additional work would be required to design an operable recognition system for multispectral classification. However, the important point of this example is the ease with which such a system could be prototyped using MATLAB and IPT functions, complemented by some of the functions developed thus far in the book. ■

TABLE 12.1 Bayes classification of multispectral image data.

Training Patterns						Independent Patterns					
	No. of	Classified into Class			%		No. of	Classified into Class			%
Class	Samples	1	2	3	Correct	Class	Samples	1	2	3	Correct
1	484	482	2	0	99.6	1	483	478	3	2	98.9
2	933	0	885	48	94.9	2	932	0	880	52	94.4
3	483	0	19	464	96.1	3	482	0	16	466	96.7

12.3.5 Adaptive Learning Systems

The approaches discussed in Sections 12.3.1 and 12.3.3 are based on the use of sample patterns to estimate the statistical parameters of each pattern class. The minimum-distance classifier is specified completely by the mean vector of each class. Similarly, the Bayes classifier for Gaussian populations is specified completely by the mean vector and covariance matrix of each class of patterns.

In these two approaches, training is a simple matter. The training patterns of each class are used to compute the parameters of the decision function corresponding to that class. After the parameters in question have been estimated, the structure of the classifier is fixed, and its eventual performance will depend on how well the actual pattern populations satisfy the underlying statistical assumptions made in the derivation of the classification method being used.

As long as the pattern classes are characterized, at least approximately, by Gaussian probability density functions, the methods just discussed can be quite effective. However, when this assumption is not valid, designing a statistical classifier becomes a much more difficult task because estimating multivariate probability density functions is not a trivial endeavor. In practice, such decision-theoretic problems are best handled by methods that yield the required decision functions directly via training. Then making assumptions regarding the underlying probability density functions or other probabilistic information about the pattern classes under consideration is unnecessary.

The principal approach in use today for this type of classification is based on neural networks (Gonzalez and Woods [2002]). The scope of implementing neural networks suitable for image-processing applications is not beyond the capabilities of the functions available to us in MATLAB and IPT. However, this effort would be unwarranted in the present context because a comprehensive neural-networks toolbox has been available from The MathWorks for several years.

12.4 Structural Recognition

Structural recognition techniques are based generally on representing objects of interest as strings, trees, or graphs and then defining descriptors and recognition rules based on those representations. The key difference between decision-theoretic and structural methods is that the former uses quantitative descriptors expressed in the form of numeric vectors. Structural techniques, on the other hand, deal principally with symbolic information. For instance, suppose that ob-

ject boundaries in a given application are represented by minimum-perimeter polygons. A decision-theoretic approach might be based on forming vectors whose elements are the numeric values of the interior angles of the polygons, while a structural approach might be based on defining symbols for *ranges* of angle values and then forming a string of such symbols to describe the patterns.

Strings are by far the most common representation used in structural recognition, so we focus on this approach in this section. As will become evident shortly, MATLAB has an extensive set of functions specialized for string manipulation.

12.4.1 Working with Strings in MATLAB

In MATLAB, a string is a one-dimensional array whose components are the numeric codes for the characters in the string. The characters displayed depend on the character set used in encoding a given font. The *length* of a string is the number of characters in the string, including spaces. It is obtained using the familiar function `length`. A string is defined by enclosing its characters in single quotes (a textual quote within a string is indicated by two quotes).

Table 12.2 lists the principal MATLAB functions that deal with strings.[†] Considering first the general category, function `blanks` has the syntax:


```
s = blanks(n)
```

It generates a string consisting of `n` blanks. Function `cellstr` creates a cell array of strings from a character array. One of the principal advantages of storing strings in cell arrays is that it eliminates the need to pad strings with blanks to create character arrays with rows of equal length (e.g., to perform string comparisons). The syntax


```
c = cellstr(S)
```

places the rows of the character array `S` into separate cells of `c`. Function `char` is used to convert back to a string matrix. For example, consider the string matrix


```
>> S = [' abc'; 'defg'; 'hi  ']   % Note the blanks.
S =
     abc
    defg
    hi
```

Typing `whos S` at the prompt displays the following information:

```
>> whos S
    Name        Size        Bytes       Class
    S           3x4         24          char array
```

[†]Some of the string functions discussed in this section were introduced in earlier chapters.

TABLE 12.2 MATLAB's string-manipulation functions.

Category	Function Name	Explanation
General	`blanks`	String of blanks.
	`cellstr`	Create cell array of strings from character array. Use function `char` to convert back to a character string.
	`char`	Create character array (string).
	`deblank`	Remove trailing blanks.
	`eval`	Execute string with MATLAB expression.
String tests	`iscellstr`	True for cell array of strings.
	`ischar`	True for character array.
	`isletter`	True for letters of the alphabet.
	`isspace`	True for whitespace characters.
String operations	`lower`	Convert string to lowercase.
	`regexp`	Match regular expression.
	`regexpi`	Match regular expression, ignoring case.
	`regexprep`	Replace string using regular expression.
	`strcat`	Concatenate strings.
	`strcmp`	Compare strings (see Section 2.10.5).
	`strcmpi`	Compare strings, ignoring case.
	`strfind`	Find one string within another.
	`strjust`	Justify string.
	`strmatch`	Find matches for string.
	`strncmp`	Compare first `n` characters of strings.
	`strncmpi`	Compare first `n` characters, ignoring case.
	`strread`	Read formatted data from a string. See Section 2.10.5 for a detailed explanation.
	`strrep`	Replace a string within another.
	`strtok`	Find token in string.
	`strvcat`	Concatenate strings vertically.
	`upper`	Convert string to uppercase.
String to number conversion	`double`	Convert string to numeric codes.
	`int2str`	Convert integer to string.
	`mat2str`	Convert matrix to a string suitable for processing with the `eval` function.
	`num2str`	Convert number to string.
	`sprintf`	Write formatted data to string.
	`str2double`	Convert string to double-precision value.
	`str2num`	Convert string to number (see Section 2.10.5).
	`sscanf`	Read string under format control.
Base number conversion	`base2dec`	Convert base B string to decimal integer.
	`bin2dec`	Convert binary string to decimal integer.
	`dec2base`	Convert decimal integer to base B string.
	`dec2bin`	Convert decimal integer to binary string.
	`dec2hex`	Convert decimal integer to hexadecimal string.
	`hex2dec`	Convert hexadecimal string to decimal integer.
	`hex2num`	Convert IEEE hexadecimal to double-precision number.

Note in the first command line that two of the three strings in S have trailing blanks because all rows in a string matrix must have the same number of characters. Note also that no quotes enclose the strings in the output because S is a character array. The following command returns a 3 × 1 cell array:

```
>> c = cellstr(S)
c =
    ' abc'
    'defg'
    'hi'
>> whos c
    Name      Size      Bytes      Class
    c         3x1       294        cell array
```

where, for example, `c(1) = ' abc'`. Note that quotes appear around the strings in the output, and that the strings have no trailing blanks. To convert back to a string matrix we let

```
Z = char(c)
Z =
     abc
    defg
    hi
```

Function `eval` evaluates a string that contains a MATLAB expression. The call `eval(expression)` executes `expression`, a string containing any valid MATLAB expression. For example, if `t` is the character string `t = '3^2'`, typing `eval(t)` returns a 9.

The next category of functions deals with string tests. A 1 is returned if the funtion is `true`; otherwise the value returned is 0. Thus, in the preceding example, `iscellstr(c)` would return a 1 and `iscellstr(S)` would return a 0. Similar comments apply to the other functions in this category.

String operations are next. Functions `lower` (and `upper`) are self explanatory. They are discussed in Section 2.10.5. The next three functions deal with *regular expressions*,[†] which are sets of symbols and syntactic elements used commonly to match patterns of text. A simple example of the power of regular expressions is the use of the familiar wildcard symbol " * " in a file search. For instance, a search for image*.m in a typical search command window would return all the M-files that begin with the word "image." Another example of the use of regular expressions is in a search-and-replace function that searches for an instance of a given text string and replaces it with another. Regular expressions are formed using *metacharacters*, some of which are listed in Table 12.3.

[†]Regular expressions can be traced to the work of American mathematician Stephen Kleene, who developed regular expressions as a notation for describing what he called "the algebra of regular sets."

TABLE 12.3 Some of the metacharacters used in regular expressions for matching. See the `regular expressions` help page for a complete list.

Metacharacters	Usage
`.`	Matches any one character.
`[ab...]`	Matches any one of the characters, (`a`, `b`, ...), contained within the brackets.
`[^ab...]`	Matches any character except those contained within the brackets.
`?`	Matches any character zero or one times.
`*`	Matches the preceding element zero or more times.
`+`	Matches the preceding element one or more times.
`{num}`	Matches the preceding element `num` times.
`{min, max}`	Matches the preceding element at least `min` times, but not more than `max` times.
`|`	Matches either the expression preceding or following the metacharacter `|`.
`^chars`	Matches when a string begins with `chars`.
`chars$`	Matches when a string ends with `chars`.
`\<chars`	Matches when a word begins with chars.
`chars\>`	Matches when a word ends with chars.
`\<word\>`	Exact word match.

In the context of this discussion, a "word" is a substring within a string, preceded by a space or the beginning of the string, and ending with a space or the end of the string. Several examples are given in the following paragraph.

Function `regexp` matches a regular expression. Using the basic syntax


```
idx = regexp(str, expr)
```

returns a row vector, `idx`, containing the indices (locations) of the substrings in `str` that match the regular expression string, `expr`. For example, suppose that `expr = 'b.*a'`. Then the expression `idx = regexp(str, expr)` would mean find matches in string `str` for any `b` that is followed by any character (as specified by the metacharacter ".") any number of times, including zero times (as specified by *), followed by an `a`. The indices of any locations in `str` meeting these conditions are stored in vector `idx`. If no such locations are found, then `idx` is returned as the empty matrix.

A few more examples of regular expressions for `expr` should clarify these concepts. The regular expression `'b.+a'` would be as in the preceding example, except that "any number of times, including zero times" would be replaced by "one or more times." The expression `'b[0–9]'` means any `b` followed by any number from 0 to 9; the expression `'b[0–9]*'` means any `b` followed by any number from 0 to 9 any number of times; and `'b[0–9]+'` means `b` followed by any number from 0 to 9 one or more times. For example, if `str = 'b0123c234bcd'`, the preceding three instances of `expr` would give the following results: `idx = 1`; `idx = [1 10]`; and `idx = 1`.

As an example of the use of regular expressions for recognizing object characteristics, suppose that the boundary of an object has been coded with a four-directional Freeman chain code [see Fig. 11.1(a)], stored in string `str`, so that

```
>> str
str =
000300333222221111
```

Suppose also that we are interested in finding the locations in the string where the direction of travel turns from east (0) to south (3), and stays there for at least two increments, but no more than six increments. This is a "downward step" feature in the object, larger than a single transition, which may be due to noise. We can express these requirements in terms of the following regular expression:

```
>> expr = '0[3]{2, 6}';
```

Then

```
>> idx = regexp(str, expr)
idx =
    6
```

The value of `idx` identifies the point in this case where a `0` is followed by three 3s. More complex expressions are formed in a similar manner.

Function `regexpi` behaves in the manner just described for `regexp`, except that it ignores character (upper and lower) case. Function `regexprep`, with syntax

```
s = regexprep(str, expr, replace)
```

replaces with string `replace` all occurrences of the regular expression `expr` in string, `str`. The new string is returned. If no matches are found `regexprep` returns `str`, unchanged.

Function `strcat` has the syntax

```
C = strcat(S1, S2, S3, ...)
```

This function concatenates (horizontally) corresponding rows of the character arrays `S1`, `S2`, `S3`, and so on. All input arrays must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array. If any of the inputs is a cell array of strings, `strcat` returns a cell array of strings formed by concatenating corresponding elements of `S1`, `S2`, `S3`, and so on. The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be character arrays. Trailing spaces in character array inputs are ignored and do not appear in the output. This is not true for inputs that are cell arrays of strings. To preserve trailing spaces the familiar concatenation syntax based on square brackets, `[S1 S2 S3 ...]`, should be used. For example,

```
>> a = 'hello '   % Note the trailing blank space.
>> b = 'goodbye'
>> strcat(a, b)
ans =
    hellogoodbye
[a b]
ans =
    hello goodbye
```

Function `strvcat`, with syntax


```
S = strvcat(t1, t2, t3, ...)
```

forms the character array `S` containing the text strings (or string matrices) `t1,t2,t3, ...` as rows. Blanks are appended to each string as necessary to form a valid matrix. Empty arguments are ignored. For example, using the strings `a` and `b` in the previous example,

```
>> strvcat(a, b)
ans =
    hello
    goodbye
```

Function `strcmp`, with syntax


```
k = strcmp(str1, str2)
```

compares the two strings in the argument and returns 1 (`true`) if the strings are identical. Otherwise it returns a 0 (`false`). A more general syntax is

```
K = strcmp(S, T)
```

where either `S` or `T` is a cell array of strings, and `K` is an array (of the same size as `S` and `T`) containing 1s for the elements of `S` and `T` that match, and 0s for the ones that do not. `S` and `T` must be of the same size (or one can be a scalar cell). Either one can also be a character array with the proper number of rows.

Function `strcmpi` performs the same operation as `strcmp`, but it ignores character case.

Function `strncmp`, with syntax


```
k = strncmp('str1', 'str2', n)
```

returns a logical `true` (1) if the first `n` characters of the strings `str1` and `str2` are the same, and returns a logical `false` (0) otherwise. Arguments `str1` and `str2` can be cell arrays of strings also. The syntax

```
R = strncmp(S, T, n)
```

where S and T can be cell arrays of strings, returns an array R the same size as S and T containing 1 for those elements of S and T that match (up to n characters), and 0 otherwise. S and T must be the same size (or one can be a scalar cell). Either one can also be a character array with the correct number of rows. The command strncmp is case sensitive. Any leading and trailing blanks in either of the strings are included in the comparison. Function strncmpi performs the same operation as strncmp, but ignores character case.

Function strfind, with syntax

```
I = strfind(str, pattern)
```

searches string str for occurrences of a *shorter* string, pattern, returning the starting index of each such occurrence in the double array, I. If pattern is not found in str, or if pattern is longer than str, then strfind returns the empty array, [ ].

Function strjust has the syntax

```
Q = strjust(A, direction)
```

where A is a character array, and direction can have the justification values 'right', 'left', and 'center'. The default justification is 'right'. The output array contains the same strings as A, but justified in the direction specified. Note that justification of a string implies the existence of leading and/or trailing blank characters to provide space for the specified operation. For instance, letting the symbol "□" represents a blank character, the string '□□abc' with two leading blank characters does not change under 'right' justification; becomes 'abc□□' with 'left' justification; and becomes the string '□abc□' with 'center' justification. Clearly, these operations have no effect on a string that does not contain any leading or trailing blanks.

Function strmatch, with syntax

```
m = strmatch('str', STRS)
```

looks through the rows of the character array or cell array of strings, STRS, to find strings that begin with string str, returning the matching row indices. The alternate syntax

```
m = strmatch('str', STRS, 'exact')
```

returns only the indices of the strings in STRS matching str exactly. For example, the statement

```
>> m = strmatch('max', strvcat('max', 'minimax', 'maximum'));
```

returns m = [1; 3] because rows 1 and 3 of the array formed by strvcat begin with 'max'. On the other hand, the statement

```
>> m = strmatch('max', strvcat('max', 'minimax', 'maximum'), 'exact');
```

returns `m = 1`, because only row 1 matches `'max'` exactly.

Function `strrep`, with syntax


```
r = strrep('str1', 'str2', 'str3')
```

replaces all occurrences of the string `str2` within string `str1` with the string `str3`. If any of `str1`, `str2`, or `str3` is a cell array of strings, this function returns a cell array the same size as `str1`, `str2`, and `str3`, obtained by performing a `strrep` using corresponding elements of the inputs. The inputs must all be of the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the correct number of rows. For example,

```
>> s = 'Image processing and restoration.';
>> str = strrep(s, 'processing', 'enhancement')
str =
    Image enhancement and restoration.
```

Function `strtok`, with syntax


```
t = strtok('str', delim)
```

returns the first token in the text string `str`, that is, the first set of characters before a delimiter in `delim` is encountered. Parameter `delim` is a vector containing delimiters (e.g., blanks, other characters, strings). For example,

```
>> str = 'An image is an ordered set of pixels';
>> delim = ['x'];
>> t = strtok(str, delim)
t =
    An
```

Note that function `strtok` terminates after the first delimiter is encountered. (i.e., a blank character in the example just given). If we change `delim` to `delim = ['x'],` then the output becomes

```
>> t = strtok(str, delim)
t =
    An image is an ordered set of pi
```

The next set of functions in Table 12.2 deals with conversions between strings and numbers. Function `int2str`, with syntax


```
str = int2str(N)
```

converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion. For example, `int2str(2 + 3)` is the *string* `'5'`. For matrix or vector inputs, `int2str` returns a string matrix:

```
>> str = int2str(eye(3))
ans =
    1   0   0
    0   1   0
    0   0   1
>> class(str)
ans =
    char
```

Function `mat2str`, with syntax

```
str = mat2str(A)
```

converts matrix `A` into a string, suitable for input to the `eval` function, using full precision. Using the syntax

```
str = mat2str(A, n)
```

converts matrix `A` using `n` digits of precision. For example, consider the matrix

```
>> A =  [1 2;3 4]
A =
    1    2
    3    4
```

The statement

```
>> b = mat2str(A)
```

produces

```
b =
    [1 2;3 4]
```

where `b` is a string of 9 characters, including the square brackets, spaces, and a semicolon. The command

```
>> eval(mat2str(A))
```

reproduces `A`. The other functions in this category have similar interpretations.

The last category in Table 12.2 deals with base number conversions. For example, function dec2base, with syntax


```
str = dec2base(d, base)
```

converts the decimal integer d to the specified base, where d must be a nonnegative integer smaller than 2^52, and base must be an integer between 2 and 36. The returned argument str is a string. For example, the following command converts 23_{10} to base 2 and returns the result as a string:

```
>> str = dec2base(23, 2)
str =
    10111
>> class(str)
ans =
    char
```

Using the syntax

```
str = dec2base(d, base, n)
```

produces a representation with at least n digits.

12.4.2 String Matching

In addition to the string matching and comparing functions in Table 12.2, it is often useful to have available measures of similarity that behave much like the distance measures discussed in Section 12.2. We illustrate this approach using a measure defined as follows.

Suppose that two region boundaries, a and b, are coded into strings $a_1a_2 \ldots a_m$ and $b_1b_2 \ldots b_n$, respectively. Let α denote the number of matches between these two strings, where a match is said to occur in the kth position if $a_k = b_k$. The number of symbols that do not match is

$$\beta = \max(|a|, |b|) - \alpha$$

where $|\arg|$ is the length (number of symbols) of the string in the argument. It can be shown that $\beta = 0$ if and only if a and b are identical strings.

A simple measure of similarity between a and b is the ratio

$$R = \frac{\alpha}{\beta} = \frac{\alpha}{\max(|a|, |b|) - \alpha}$$

This measure, proposed by Sze and Yang [1981], is infinite for a perfect match and 0 when none of the corresponding symbols in a and b match (α is 0 in this case).

Because matching is performed between corresponding symbols, it is required that all strings be "registered" in some position-independent manner in order for this method to make sense. One way to register two strings is to shift one string with respect to the other until a maximum value of R is obtained. This and other similar matching strategies can be developed using some of the string operations detailed in Table 12.2. Typically, a more efficient approach is to define the same starting point for all strings based on normalizing the boundaries with respect to size and orientation before their string representation is extracted. This approach is illustrated in Example 12.3.

The following M-function computes the preceding measure of similarity for two character strings.

strsimilarity

```
function R = strsimilarity(a, b)
%STRSIMILARITY Computes a similarity measure between two strings.
%   R = STRSIMILARITY(A, B) computes the similarity measure, R,
%   defined in Section 12.4.2 for strings A and B. The strings do not
%   have to be of the same length, but if one is shorter than other,
%   then it is assumed that the shorter string has been padded with
%   leading blanks so that it is brought into the necessary
%   registration prior to using this function. Only one of the
%   strings can have blanks, and these must be leading and/or
%   trailing blanks. Blanks are not counted when computing the length
%   of the strings for use in the similarity measure.

% Verify that a and b are character strings.
if ~ischar(a) | ~ischar(b)
   error('Inputs must be character strings.')
end

% Find any blank spaces.
I = find(a == ' ');
J = find(b == ' ');
LI = length(I); LJ = length(J);
if LI ~= 0 & LJ ~= 0
   error('Only one of the strings can contain blanks.')
end

% Pad the end of the appropriate string. It is assumed
% that they are registered in terms of their beginning
% positions.
a = a(:); b = b(:);
La = length(a); Lb = length(b);
if LI == 0 & LJ == 0
   if La > Lb
      b = [b; blanks(La - Lb)'];
   else
      a = [a; blanks(Lb - La)'];
   end
elseif isempty(I)
   Lb = length(b) - length(J);
   b = [b; blanks(La - Lb - LJ)'];
```

```
else
   La = length(a) - length(I);
   a = [a; blanks(Lb - La - LI)'];
end
% Compute the similarity measure.
I = find(a == b);
alpha = length(I);
den = max(La, Lb) - alpha;
if den == 0
   R = Inf;
else
   R = alpha/den;
end
```

EXAMPLE 12.3: Object recognition based on string matching.

■ Figures 12.3(a) and (d) show silhouettes of two samples of container bottles whose principal shape difference is the curvature of their sides. For purposes of differentiation, objects with the curvature characteristics of Fig. 12.3(a) are said to be from class 1. Objects with straight sides are said to be from class 2. The images are of size 372 × 288 pixels.

To illustrate the effectiveness of measure R for differentiating between objects of classes 1 and 2, the boundaries of the objects were approximated by minimum-perimeter polygons using function `minperpoly` (see Section 11.2.2) with a cell size of 8. Figures 12.3(b) and (e) show the results. Then noise was added to the coordinates of each vertex of the polygons using function `randvertex` (the listing is included in Appendix C), which has the syntax

`randvertex`

```
[xn, yn] = randvertex(x, y, npix)
```

where `x` and `y` are column vectors containing the coordinates of the vertices of a polygon, `xn` and `yn` are the corresponding noisy coordinates, and `npix` is the maximum number of pixels by which a coordinate is allowed to be displaced in either direction. Five sets of noisy vertices were generated for each class using `npix = 5`. Figures 12.3(c) and (f) show typical results.

Strings of symbols were generated for each class by coding the interior angles of the polygons using function `polyangles` (see Appendix C for the code listing):

`polyangles`

```
>> angles = polyangles(x, y);
```

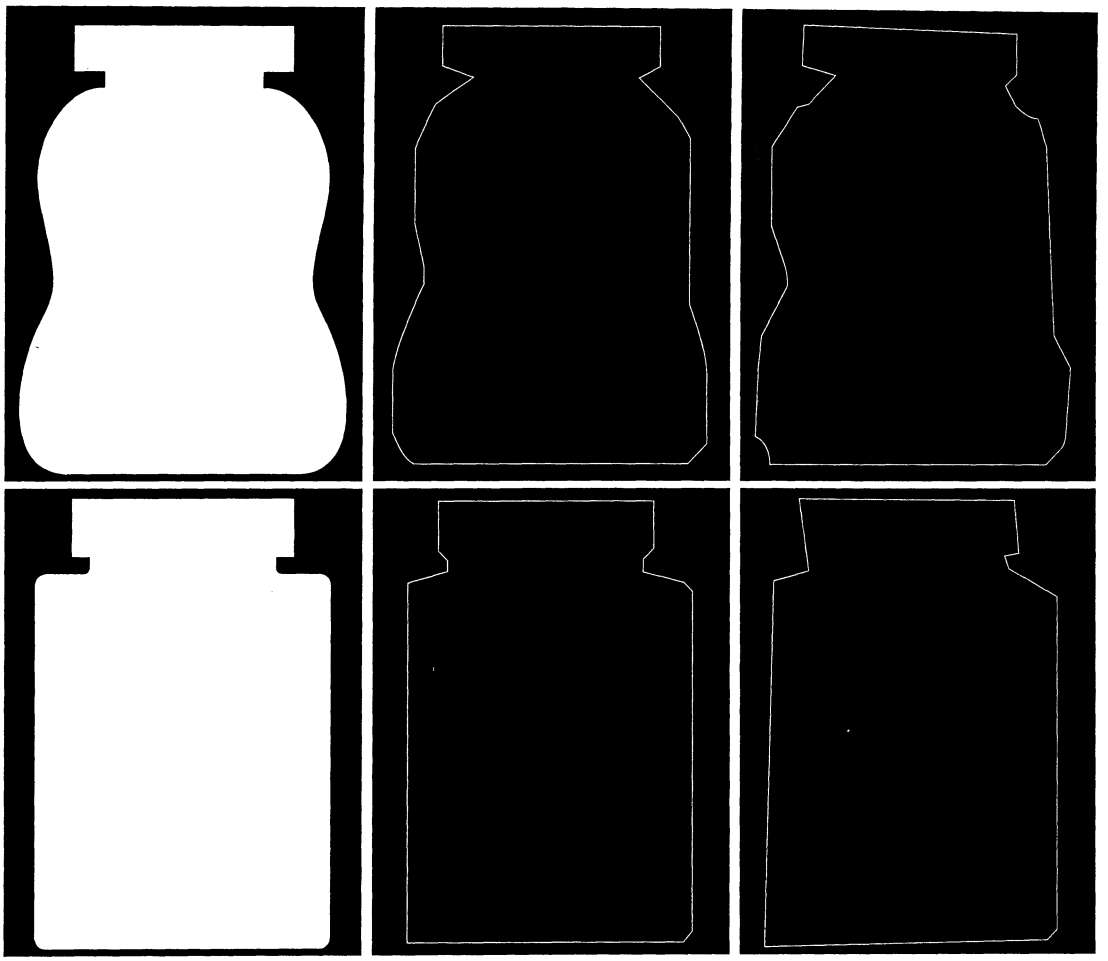
The `x` *and* `y` *inputs to function* `polyangles` *are vectors containing the x- and y-coordinates of the vertices of a polygon, ordered in the clockwise direction. The output is a vector containing the corresponding interior angles, in degrees.*

Then a string, `s`, was generated from a given `angles` array by quantizing the angles into 45° increments, using the statement

```
>> s = floor(angles/45) + 1;
```

This yielded a string whose elements were numbers between 1 and 8, with 1 designating the range $0° \leq \theta < 45°$, 2 designating the range $45° \leq \theta < 90°$, and so forth, where θ denotes an interior angle.

Because the first vertex in the output of `minperpoly` is always the top, left vertex of the boundary of the input, `B`, the first element of string `s` corresponds



a b c
d e f

FIGURE 12.3 (a) An object. (b) Its minimum perimeter polygon obtained using function `minperpoly` with a cell size of 8. (c) A typical noisy boundary. (d)–(f) The same sequence for another object.

to the interior angle of that vertex. This automatically registers the strings (if the objects are not rotated) because they all start at the top, left vertex in all images. The direction of the vertices output by `minperpoly` is clockwise, so the elements of `s` also are in that direction. Finally, each `s` was converted from a string of integers to a character string using the command

```
>> s = int2str(s);
```

In this example the objects are of comparable size and they are all vertical, so normalization of neither size nor orientation was required. If the objects had been of arbitrary size and orientation, we could have aligned them along

their principal directions by using the eigenvector transformation discussed at the end of Section 11.5. Then we could have used the bounding box in Section 11.4.1 to obtain the object dimensions for normalization purposes.

First, function `strsimilarity` was used to measure the similarity of all strings of class 1 between themselves. For instance, to compute the similarity between the first and second strings of class 1 we used the command

```
>> R = strsimilarity(s11, s12);
```

where the first subscript indicates class and the second a string number within that class. The results obtained using five typical strings are summarized in Table 12.4, where `Inf` indicates infinity (i.e., a perfect match, as discussed earlier). Table 12.5 shows the same type of computation involving five strings of class 2 against themselves. Table 12.6 shows values of the similarity measure between the strings of class 1 and class 2. Note that the values in this table are significantly lower than the entries in the two preceding tables, indicating that the R measure achieved a high degree of discrimination between the two classes of objects. In other words, measuring the similarity of strings against members of their own class showed significantly larger values of `R`, indicating a closer match than when strings were compared to members of the opposite class. ■

TABLE 12.4
Values of similarity measure, R, between the strings of class 1. (All values shown are ×10.)

R	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
s_{11}	Inf				
s_{12}	9.33	Inf			
s_{13}	26.25	12.31	Inf		
s_{14}	16.36	9.33	14.16	Inf	
s_{15}	22.22	14.17	14.01	19.02	Inf

TABLE 12.5
Values of similarity measure, R, between the strings of class 2. (All values shown are ×10.)

R	s_{21}	s_{22}	s_{23}	s_{24}	s_{25}
s_{21}	Inf				
s_{22}	10.00	Inf			
s_{23}	13.33	13.33	Inf		
s_{24}	7.50	13.31	18.00	Inf	
s_{25}	13.33	7.51	18.12	10.01	Inf

TABLE 12.6
Values of similarity measure, R, between the strings of classes 1 and 2. (All values shown are ×10.)

R	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
s_{21}	2.03	0.01	1.15	1.17	0.75
s_{22}	1.15	1.61	1.16	0.75	2.07
s_{23}	2.08	1.15	2.08	2.06	2.08
s_{24}	1.60	1.62	1.59	1.14	2.61
s_{25}	1.61	0.36	0.74	1.60	1.16

Summary

Starting with Chapter 9, our treatment of digital image processing began a transition from processes whose outputs are images to processes whose outputs are attributes about those images. Although the material in the present chapter is introductory in nature, the topics covered are fundamental to understanding the state of the art in object recognition. As mentioned in Section 1.2 at the onset of our journey, recognition of individual objects is a logical place at which to conclude this book.

Having finished study of the material in the preceding twelve chapters, the reader is now in the position of being able to master the fundamentals of how to prototype software solutions of image-processing problems using MATLAB and Image Processing Toolbox functions. What is even more important, the background and numerous new functions developed in the book constitute a basic blueprint on how to extend the power of MATLAB and IPT. Given the task-specific nature of most imaging problems, a clear understanding of this material enhances significantly the chances of arriving at successful solutions in a broad spectrum of image processing application areas.