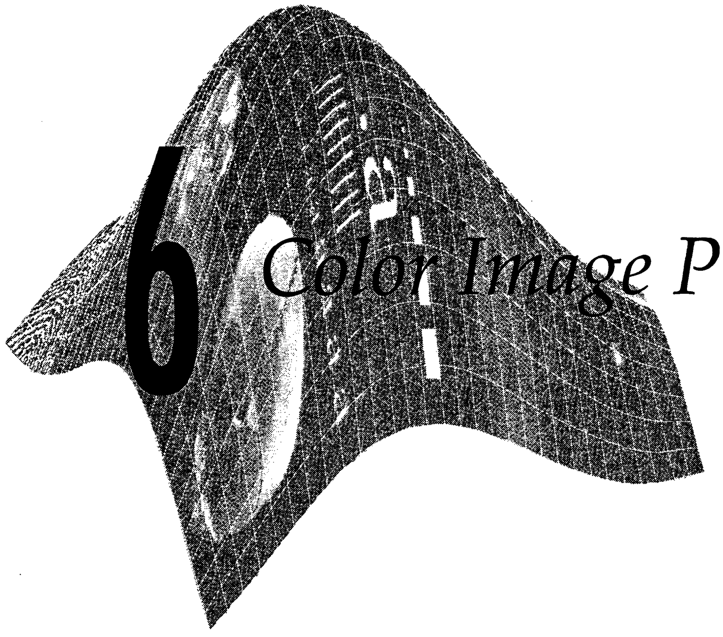


6 Color Image Processing

Preview

In this chapter we discuss fundamentals of color image processing using the Image Processing Toolbox and extend some of its functionality by developing additional color generation and transformation functions. The discussion in this chapter assumes familiarity on the part of the reader with the principles and terminology of color image processing at an introductory level.

6.1 Color Image Representation in MATLAB

As noted in Section 2.6, the Image Processing Toolbox handles color images either as indexed images or RGB (red, green, blue) images. In this section we discuss these two image types in some detail.

6.1.1 RGB Images

An RGB *color image* is an $M \times N \times 3$ array of *color pixels*, where each color pixel is a triplet corresponding to the red, green, and blue components of an RGB image at a specific spatial location (see Fig. 6.1). An RGB image may be viewed as a "stack" of three gray-scale images that, when fed into the red, green, and blue inputs of a color monitor, produce a color image on the screen. By convention, the three images forming an RGB color image are referred to as the red, green, and blue *component images*. The data class of the component images determines their range of values. If an RGB image is of class `double`, the range of values is [0, 1]. Similarly, the range of values is [0, 255] or [0, 65535] for RGB images of class `uint8` or `uint16`, respectively. The number of bits used to represent the pixel values of the component images determines the *bit depth* of an RGB image. For example, if each component image is an 8-bit image, the corresponding RGB image is said to be 24 bits deep. Generally, the number of bits in all component images is the same. In this case, the number of

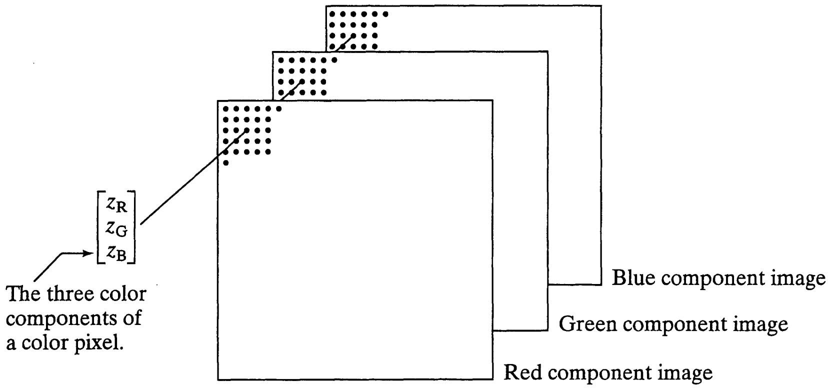


FIGURE 6.1 Schematic showing how pixels of an RGB color image are formed from the corresponding pixels of the three component images.

possible colors in an RGB image is $(2^b)^3$, where b is the number of bits in each component image. For the 8-bit case, the number is 16,777,216 colors.

Let fR, fG, and fB represent three RGB component images. An RGB image is formed from these images by using the cat (concatenate) operator to stack the images:

```
rgb_image = cat(3, fR, fG, fB)
```

cat

The order in which the images are placed in the operand matters. In general, cat(dim, A1, A2, . . .) concatenates the arrays along the dimension specified by dim. For example, if dim = 1, the arrays are arranged vertically, if dim = 2, they are arranged horizontally, and, if dim = 3, they are stacked in the third dimension, as in Fig. 6.1.

If all component images are identical, the result is a gray-scale image. Let rgb_image denote an RGB image. The following commands extract the three component images:

```
>> fR = rgb_image(:, :, 1);
>> fG = rgb_image(:, :, 2);
>> fB = rgb_image(:, :, 3);
```

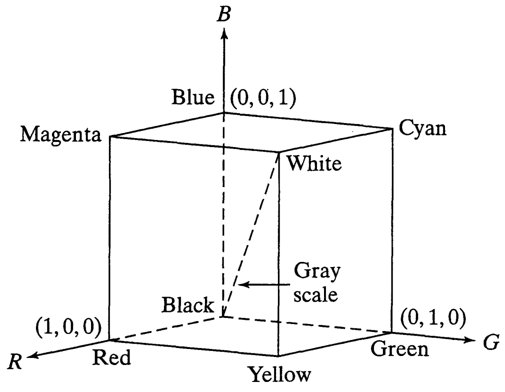
The RGB *color space* usually is shown graphically as an RGB color cube, as depicted in Fig. 6.2. The vertices of the cube are the *primary* (red, green, and blue) and *secondary* (cyan, magenta, and yellow) colors of light.

Often, it is useful to be able to view the color cube from any perspective. Function rgbcube is used for this purpose. The syntax is

```
rgbcube(vx, vy, vz)
```

rgbcube

Typing rgbcube(vx, vy, vz) at the prompt produces an RGB cube on the MATLAB desktop, viewed from point (vx, vy, vz). The resulting image can be saved to disk using function print, discussed in Section 2.4. The code for this function follows. It is self-explanatory.



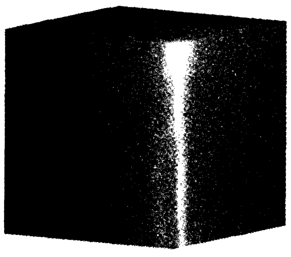


FIGURE 6.2 (a) Schematic of the RGB color cube showing the primary and secondary colors of light at the vertices. Points along the main diagonal have gray values from black at the origin to white at point (1, 1, 1). (b) The RGB color cube.

Function patch *creates filled, 2-D polygons based on specified property/value pairs. For more information about* patch, *see the MATLAB help page for this function.*

```
function rgbcube(vx, vy, vz)
%RGBCUBE Displays an RGB cube on the MATLAB desktop.
%   RGBCUBE(VX, VY, VZ) displays an RGB color cube, viewed from point
%   (VX, VY, VZ). With no input arguments, RGBCUBE uses (10, 10, 4)
%   as the default viewing coordinates. To view individual color
%   planes, use the following viewing coordinates, where the first
%   color in the sequence is the closest to the viewing axis, and the
%   other colors are as seen from that axis, proceeding to the right
%   right (or above), and then moving clockwise.
%
%     -------------------------------------------------------
%          COLOR PLANE                      ( vx,  vy,  vz)
%     -------------------------------------------------------
%          Blue-Magenta-White-Cyan          (  0,   0,  10)
%          Red-Yellow-White-Magenta         ( 10,   0,   0)
%          Green-Cyan-White-Yellow          (  0,  10,   0)
%          Black-Red-Magenta-Blue           (  0, -10,   0)
%          Black-Blue-Cyan-Green            (-10,   0,   0)
%          Black-Red-Yellow-Green           (  0,   0, -10)

% Set up parameters for function patch.
vertices_matrix = [0 0 0;0 0 1;0 1 0;0 1 1;1 0 0;1 0 1;1 1 0;1 1 1];
faces_matrix = [1 5 6 2;1 3 7 5;1 2 4 3;2 4 8 6;3 7 8 4;5 6 8 7];
colors = vertices_matrix;
% The order of the cube vertices was selected to be the same as
% the  order of the (R,G,B) colors (e.g., (0,0,0) corresponds to
% black, (1,1,1) corresponds to white, and so on.)

% Generate RGB cube using function patch.
patch('Vertices', vertices_matrix, 'Faces', faces_matrix, ...
     'FaceVertexCData', colors, 'FaceColor', 'interp', ...
     'EdgeAlpha', 0)

% Set up viewing point.
if nargin == 0
   vx = 10; vy = 10; vz = 4;
```

```
elseif nargin ~= 3
   error('Wrong number of inputs.')
end
axis off
view([vx, vy, vz])
axis square
```

6.1.2 Indexed Images

An *indexed* image has two components: a *data matrix* of integers, X, and a *colormap matrix*, map. Matrix map is an $m \times 3$ array of class double containing floating-point values in the range [0, 1]. The length, m, of the map is equal to the number of colors it defines. Each row of map specifies the red, green, and blue components of a single color. An indexed image uses "direct mapping" of pixel intensity values to colormap values. The color of each pixel is determined by using the corresponding value of integer matrix X as a pointer into map. If X is of class double, then all of its components with values less than or equal to 1 point to the first row in map, all components with value 2 point to the second row, and so on. If X is of class uint8 or uint16, then all components with value 0 point to the first row in map, all components with value 1 point to the second row, and so on. These concepts are illustrated in Fig. 6.3.

If three columns of map *are equal, then the colormap becomes a* grayscale map.

To display an indexed image we write

```
>> imshow(X, map)
```

or, alternatively,

```
>> image(X)
>> colormap(map)
```

A colormap is stored with an indexed image and is automatically loaded with the image when function imread is used to load the image.

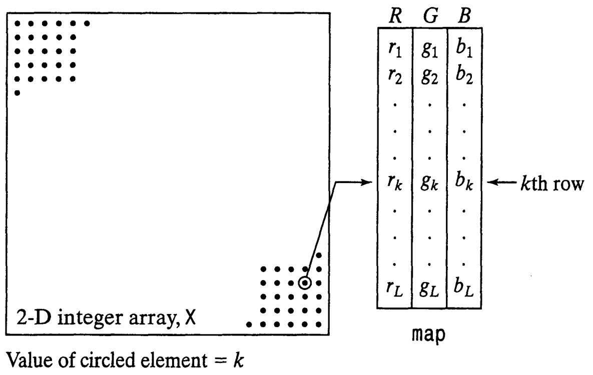


FIGURE 6.3 Elements of an indexed image. Note that the value of an element of integer array X determines the row number in the colormap. Each row contains an RGB triplet, and L is the total number of rows.

Sometimes it is necessary to approximate an indexed image by one with fewer colors. For this we use function imapprox, whose syntax is

imapprox

```
[Y, newmap] = imapprox(X, map, n)
```

This function returns an array Y with colormap newmap, which has at most n colors. The input array X can be of class uint8, uint16, or double. The output Y is of class uint8 if n is less than or equal to 256. If n is greater than 256, Y is of class double.

When the number of rows in map is less than the number of distinct integer values in X, multiple values in X are displayed using the same color in map. For example, suppose that X consists of four vertical bands of equal width, with values 1, 64, 128, and 256. If we specify the colormap map = [0 0 0; 1 1 1], then all the elements in X with value 1 would point to the first row (black) of the map and all the other elements would point to the second row (white). Thus, the command imshow(X, map) would display an image with a black band followed by three white bands. In fact, this would be true until the length of the map became 65, at which time the display would be a black band, followed by a gray band, followed by two white bands. Nonsensical image displays can result if the length of the map exceeds the allowed range of values of the elements of X.

There are several ways to specify a color map. One approach is to use the statement

```
>> map(k, :) = [r(k) g(k) b(k)]
```

where [r(k) g(k) b(k)] are RGB values that specify one row of a colormap. The map is filled out by varying k.

Table 6.1 lists the RGB values for some basic colors. Any of the three formats shown in the table can be used to specify colors. For example, the background color of a figure can be changed to green by using any of the following three statements:

whitebg

```
>> whitebg('g')
>> whitebg('green')
>> whitebg([0 1 0])
```

TABLE 6.1 RGB values of some basic colors. The long or short names (enclosed by quotes) can be used instead of the numerical triplet to specify an RGB color.

Long name	Short name	RGB values
Black	k	[0 0 0]
Blue	b	[0 0 1]
Green	g	[0 1 0]
Cyan	c	[0 1 1]
Red	r	[1 0 0]
Magenta	m	[1 0 1]
Yellow	y	[1 1 0]
White	w	[1 1 1]

Other colors in addition to the ones shown in Table 6.1 involve fractional values. For instance, `[.5 .5 .5]` is gray, `[.5 0 0]` is dark red, and `[.49 1 .83]` is aquamarine.

MATLAB provides several predefined color maps, accessed using the command

```
>> colormap(map_name)
```

colormap

which sets the colormap to the matrix `map_name`; an example is

```
>> colormap(copper)
```

where `copper` is one of the prespecified MATLAB colormaps. The colors in this map vary smoothly from black to bright copper. If the last image displayed was an indexed image, this command changes its colormap to `copper`. Alternatively, the image can be displayed directly with the desired colormap:

```
>> imshow(X, copper)
```

Table 6.2 lists some of the colormaps available in MATLAB. The length (number of colors) of these colormaps can be specified by enclosing the number in parentheses. For example, `gray(16)` generates a colormap with 16 shades of gray.

6.1.3 IPT Functions for Manipulating RGB and Indexed Images

Table 6.3 lists the IPT functions suitable for converting between RGB, indexed, and gray-scale images. For clarity of notation in this section, we use `rgb_image` to denote RGB images, `gray_image` to denote gray-scale images, `bw` to denote black and white images, and `X`, to denote the data matrix component of indexed images. Recall that an indexed image is composed of an integer data matrix and a colormap matrix.

Function `dither` is applicable both to gray-scale and color images. Dithering is a process used mostly in the printing and publishing industry to give the visual impression of shade variations on a printed page that consists of dots. In the case of gray-scale images, dithering attempts to capture shades of gray by producing a binary image of black dots on a white background (or vice versa). The sizes of the dots vary, from small dots in light areas to increasingly larger dots for dark areas. The key issue in implementing a dithering algorithm is a tradeoff between "accuracy" of visual perception and computational complexity. The dithering approach used in IPT is based on the Floyd-Steinberg algorithm (see Floyd and Steinberg [1975], and Ulichney [1987]). The syntax used by function `dither` for gray-scale images is

```
bw = dither(gray_image)
```

dither

where, as noted earlier, `gray_image` is a gray-scale image and `bw` is the dithered result (a binary image).

TABLE 6.2 Some of the MATLAB predefined colormaps.

Name	Description
`autumn`	Varies smoothly from red, through orange, to yellow.
`bone`	A gray-scale colormap with a higher value for the blue component. This colormap is useful for adding an "electronic" look to gray-scale images.
`colorcube`	Contains as many regularly spaced colors in RGB color space as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
`cool`	Consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
`copper`	Varies smoothly from black to bright copper.
`flag`	Consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
`gray`	Returns a linear gray-scale colormap.
`hot`	Varies smoothly from black, through shades of red, orange, and yellow, to white.
`hsv`	Varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions.
`jet`	Ranges from blue to red, and passes through the colors cyan, yellow, and orange.
`lines`	Produces a colormap of colors specified by the `ColorOrder` property and a shade of gray. Consult online help regarding function `ColorOrder`.
`pink`	Contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
`prism`	Repeats the six colors red, orange, yellow, green, blue, and violet.
`spring`	Consists of colors that are shades of magenta and yellow.
`summer`	Consists of colors that are shades of green and yellow.
`white`	This is an all white monochrome colormap.
`winter`	Consists of colors that are shades of blue and green.

TABLE 6.3 IPT functions for converting between RGB, indexed, and gray-scale intensity images.

Function	Purpose
`dither`	Creates an indexed image from an RGB image by dithering.
`grayslice`	Creates an indexed image from a gray-scale intensity image by multilevel thresholding.
`gray2ind`	Creates an indexed image from a gray-scale intensity image.
`ind2gray`	Creates a gray-scale intensity image from an indexed image.
`rgb2ind`	Creates an indexed image from an RGB image.
`ind2rgb`	Creates an RGB image from an indexed image.
`rgb2gray`	Creates a gray-scale image from an RGB image.

When working with color images, dithering is used principally in conjunction with function rgb2ind to reduce the number of colors in an image. This function is discussed later in this section.

Function grayslice has the syntax

```
X = grayslice(gray_image, n)
```

grayslice

This function produces an indexed image by thresholding gray_image with threshold values

$$\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$$

As noted earlier, the resulting indexed image can be viewed with the command imshow(X, map) using a map of appropriate length [e.g., jet(16)]. An alternate syntax is

```
X = grayslice(gray_image, v)
```

where v is a vector whose values are used to threshold gray_image. When used in conjunction with a colormap, grayslice is a basic tool for pseudocolor image processing, where specified gray intensity bands are assigned different colors. The input image can be of class uint8, uint16, or double. The threshold values in v must between 0 and 1, even if the input image is of class uint8 or uint16. The function performs the necessary scaling.

Function gray2ind, with syntax

```
[X, map] = gray2ind(gray_image, n)
```

gray2ind

scales, then rounds image gray_image to produce an indexed image X with colormap gray(n). If n is omitted, it defaults to 64. The input image can be of class uint8, uint16, or double. The class of the output image X is uint8 if n is less than or equal to 256, or of class uint16 if n is greater than 256.

Function ind2gray, with the syntax

```
gray_image = ind2gray(X, map)
```

ind2gray

converts an indexed image, composed of X and map, to a gray-scale image. Array X can be of class uint8, uint16, or double. The output image is of class double.

The syntax of interest in this chapter for function rgb2ind has the form

```
[X, map] = rgb2ind(rgb_image, n, dither_option)
```

rgb2ind

where n determines the length (number of colors) of map, and dither_option can have one of two values: 'dither' (the default) dithers, if necessary, to achieve better color resolution at the expense of spatial resolution; conversely, `'nodither'` maps each color in the original image to the closest color in the new map (depending on the value of `n`). No dithering is performed. The input image can be of class `uint8`, `uint16`, or `double`. The output array, `X`, is of class `uint8` if `n` is less than or equal to 256; otherwise it is of class `uint16`. Example 6.1 shows the effect that dithering has on color reduction.

Function `ind2rgb`, with syntax

ind2rgb

```
rgb_image = ind2rgb(X, map)
```

converts the matrix `X` and corresponding colormap `map` to RGB format; `X` can be of class `uint8`, `uint16`, or `double`. The output RGB image is an $M \times N \times 3$ array of class `double`.

Finally, function `rgb2gray`, with syntax

rgb2gray

```
gray_image = rgb2gray(rgb_image)
```

converts an RGB image to a gray-scale image. The input RGB image can be of class `uint8`, `uint16`, or `double`; the output image is of the same class as the input.

EXAMPLE 6.1: Illustration of some of the functions in Table 6.3.

■ Function `rgb2ind` is quite useful for reducing the number of colors in an RGB image. As an illustration of this function, and of the advantages of using the dithering option, consider Fig. 6.4(a), which is a 24-bit RGB image, `f`. Figures 6.4(b) and (c) show the results of using the commands

```
>> [X1, map1] = rgb2ind(f, 8, 'nodither');
>> imshow(X1, map1)
```

and

```
>> [X2, map2] = rgb2ind(f, 8, 'dither');
>> figure, imshow(X2, map2)
```

Both images have only 8 colors, which is a significant reduction in the number of possible colors in `f`, which, for a 24-bit RGB image exceeds 16 million, as mentioned earlier. Figure 6.4(b) has noticeable false contouring, especially in the center of the large flower. The dithered image shows better tonality, and considerably less false contouring, a result of the "randomness" introduced by dithering. The image is a little blurred, but it certainly is visually superior to Fig. 6.4(b).

The effects of dithering are usually better illustrated with gray-scale images. Figures 6.4(d) and (e) were obtained using the commands

```
>> g = rgb2gray(f);
>> g1 = dither(g);
>> figure, imshow(g); figure, imshow(g1)
```

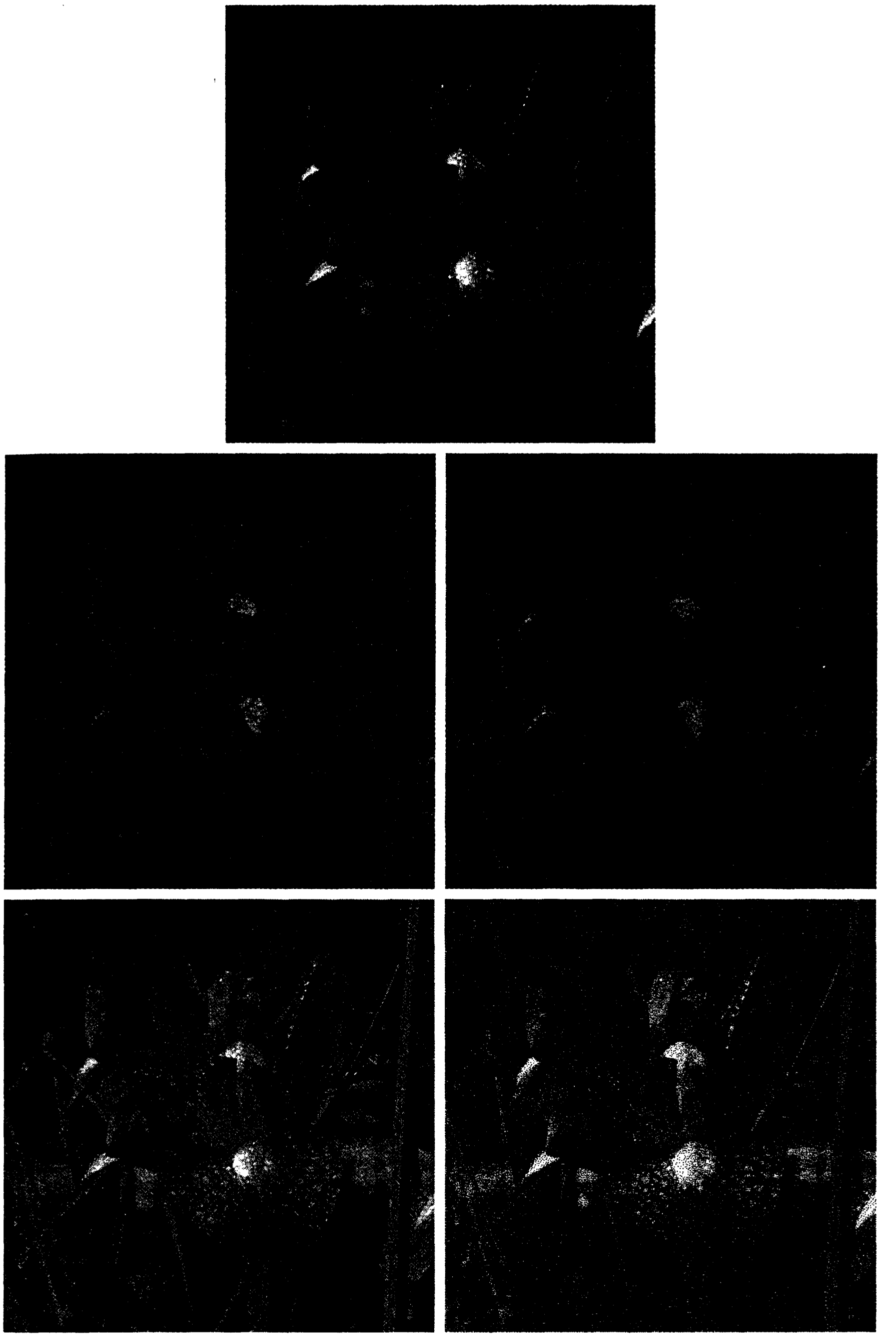


FIGURE 6.4
(a) RGB image.
(b) Number of colors reduced to 8 without dithering.
(c) Number of colors reduced to 8 with dithering.
(d) Gray-scale version of (a) obtained using function `rgb2gray`.
(e) Dithered gray-scale image (this is a binary image).

The image in Fig. 6.4(e) is a binary image, which again represents a significant degree of data reduction. By looking at Figs. 6.4(c) and (e), it is clear why dithering is such a staple in the printing and publishing industry, especially in situations (such as in newspapers) where paper quality and printing resolution are low. ■

6.2 Converting to Other Color Spaces

As explained in the previous section, the toolbox represents colors as RGB values, directly in an RGB image, or indirectly in an indexed image, where the colormap is stored in RGB format. However, there are other color spaces (also called *color models*) whose use in some applications may be more convenient and/or appropriate. These include the NTSC, YCbCr, HSV, CMY, CMYK, and HSI color spaces. The toolbox provides conversion functions from RGB to the NTSC, YCbCr, HSV and CMY color spaces, and back. Functions for converting to and from the HSI color space are developed later in this section.

6.2.1 NTSC Color Space

The NTSC color system is used in television in the United States. One of the main advantages of this format is that gray-scale information is separate from color data, so the same signal can be used for both color and monochrome television sets. In the NTSC format, image data consists of three components: *luminance* (Y), *hue* (I), and *saturation* (Q), where the choice of the letters YIQ is conventional. The luminance component represents gray-scale information, and the other two components carry the color information of a TV signal. The YIQ components are obtained from the RGB components of an image using the transformation

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Note that the elements of the first row sum to 1 and the elements of the next two rows sum to 0. This is as expected because for a gray-scale image all the RGB components are equal, so the I and Q components should be 0 for such an image. Function `rgb2ntsc` performs the transformation:

rgb2ntsc

```
yiq_image = rgb2ntsc(rgb_image)
```

where the input RGB image can be of class `uint8`, `uint16`, or `double`. The output image is an $M \times N \times 3$ array of class `double`. Component image `yiq_image(:, :, 1)` is the luminance, `yiq_image(:, :, 2)` is the hue, and `yiq_image(:, :, 3)` is the saturation image.

Similarly, the RGB components are obtained from the YIQ components using the transformation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

IPT function `ntsc2rgb` implements this equation:

```
rgb_image = ntsc2rgb(yiq_image)
```

ntsc2rgb

Both the input and output images are of class `double`.

6.2.2 The YCbCr Color Space

The YCbCr color space is used widely in digital video. In this format, luminance information is represented by a single component, Y, and color information is stored as two color-difference components, Cb and Cr. Component Cb is the difference between the blue component and a reference value, and component Cr is the difference between the red component and a reference value (Poynton [1996]). The transformation used by IPT to convert from RGB to YCbCr is

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.000 \\ 112.000 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The conversion function is

```
ycbcr_image = rgb2ycbcr(rgb_image)
```

rgb2ycbcr

The input RGB image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input. A similar transformation converts from YCbCr back to RGB:

```
rgb_image = ycbcr2rgb(ycbcr_image)
```

ycbcr2rgb

The input YCbCr image can be of class `uint8`, `uint16`, or `double`. The output image is of the same class as the input.

To see the transformation matrix used to convert from YCbCr to RGB, type the following command at the prompt:
`>> edit ycbcr2rgb`.

6.2.3 The HSV Color Space

HSV (hue, saturation, value) is one of several color systems used by people to select colors (e.g., of paints or inks) from a color wheel or palette. This color system is considerably closer than the RGB system to the way in which humans experience and describe color sensations. In artist's terminology, hue, saturation, and value refer approximately to tint, shade, and tone.

The HSV color space is formulated by looking at the RGB color cube along its gray axis (the axis joining the black and white vertices), which results in the hexagonally shaped color palette shown in Fig. 6.5(a). As we move along the vertical (gray) axis in Fig. 6.5(b), the size of the hexagonal plane that is perpendicular to the axis changes, yielding the volume depicted in the figure. Hue is

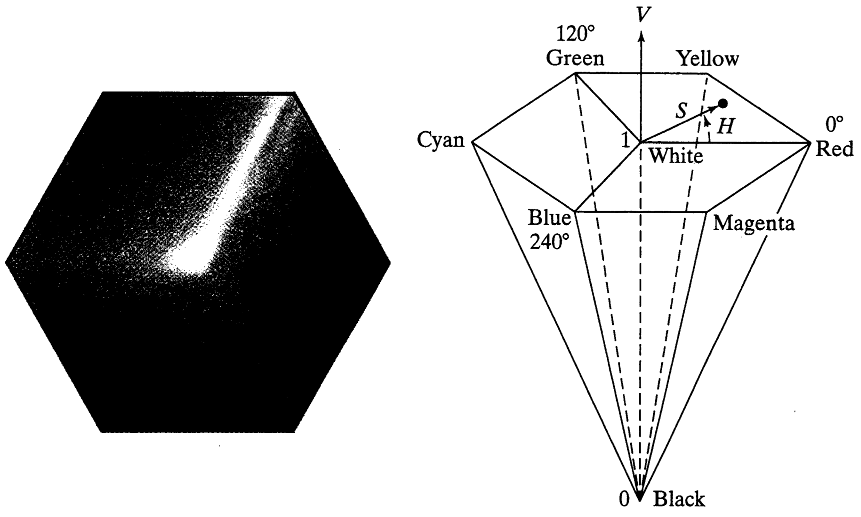


FIGURE 6.5 (a) The HSV color hexagon. (b) The HSV hexagonal cone.

expressed as an angle around a color hexagon, typically using the red axis as the 0° axis. Value is measured along the axis of the cone. The $V = 0$ end of the axis is black. The $V = 1$ end of the axis is white, which lies in the center of the full color hexagon in Fig. 6.5(a). Thus, this axis represents all shades of gray. Saturation (purity of the color) is measured as the distance from the V axis.

The HSV color system is based on cylindrical coordinates. Converting from RGB to HSV is simply a matter of developing the equations to map RGB values (which are in Cartesian coordinates) to cylindrical coordinates. This topic is treated in detail in most texts on computer graphics (e.g., see Rogers [1997]) so we do not develop the equations here.

The MATLAB function for converting from RGB to HSV is `rgb2hsv`, whose syntax is

rgb2hsv

```
hsv_image = rgb2hsv(rgb_image)
```

The input RGB image can be of class `uint8`, `uint16`, or `double`; the output image is of class `double`. The function for converting from HSV back to RGB is `hsv2rgb`:

hsv2rgb

```
rgb_image = hsv2rgb(hsv_image)
```

The input image must be of class `double`. The output also is of class `double`.

6.2.4 The CMY and CMYK Color Spaces

Cyan, magenta, and yellow are the secondary colors of light or, alternatively, the primary colors of pigments. For example, when a surface coated with cyan pigment is illuminated with white light, no red light is reflected from the surface. That is, the cyan pigment subtracts red light from reflected white light, which itself is composed of equal amounts of red, green, and blue light.

Most devices that deposit colored pigments on paper, such as color printers and copiers, require CMY data input or perform an RGB to CMY conversion internally. This conversion is performed using the simple equation

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

where the assumption is that all color values have been normalized to the range [0, 1]. This equation demonstrates that light reflected from a surface coated with pure cyan does not contain red (that is, $C = 1 - R$ in the equation). Similarly, pure magenta does not reflect green, and pure yellow does not reflect blue. The preceding equation also shows that RGB values can be obtained easily from a set of CMY values by subtracting the individual CMY values from 1.

In theory, equal amounts of the pigment primaries, cyan, magenta, and yellow should produce black. In practice, combining these colors for printing produces a muddy-looking black. So, in order to produce true black (which is the predominant color in printing), a fourth color, *black*, is added, giving rise to the CMYK color model. Thus, when publishers talk about "four-color printing," they are referring to the three-colors of the CMY color model plus black.

Function `imcomplement` introduced in Section 3.2.1 can be used to convert from RGB to CMY:

```
cmy_image = imcomplement(rgb_image)
```

We use this function also to convert a CMY image to RGB:

```
rgb_image = imcomplement(cmy_image)
```

6.2.5 The HSI Color Space

With the exception of HSV, the color spaces discussed thus far are not well suited for *describing* colors in terms that are practical for human interpretation. For example, one does not refer to the color of an automobile by giving the percentage of each of the pigment primaries composing its color.

When humans view a color object, we tend to describe it by its hue, saturation, and brightness. *Hue* is an attribute that describes a pure color (e.g., pure yellow, orange, or red), whereas *saturation* gives a measure of the degree to which a pure color is diluted by white light. *Brightness* is a subjective descriptor that is practically impossible to measure. It embodies the achromatic notion of *intensity* and is a key factor in describing color sensation. We do know that intensity (gray level) is a most useful descriptor of monochromatic images. This quantity definitely is measurable and easily interpretable.

The color space we are about to present, called the *HSI* (hue, saturation, intensity) *color space*, decouples the intensity component from the color-carrying information (hue and saturation) in a color image. As a result, the HSI model is an ideal tool for developing image-processing algorithms based on color descriptions that are natural and intuitive to humans who, after all, are the developers and users of these algorithms. The HSV color space is somewhat similar, but its focus is on presenting colors that are meaningful when interpreted in terms of a color artist's palette.

As discussed in Section 6.1.1, an RGB color image is composed of three monochrome intensity images, so it should come as no surprise that we should be able to extract intensity from an RGB image. This becomes quite clear if we take the color cube from Fig. 6.2 and stand it on the black, (0, 0, 0), vertex, with the white vertex, (1, 1, 1), directly above it, as Fig. 6.6(a) shows. As noted in connection with Fig. 6.2, the intensity is along the line joining these two vertices. In the arrangement shown in Fig. 6.6, the line (intensity axis) joining the black and white vertices is vertical. Thus, if we wanted to determine the intensity component of any color point in Fig. 6.6, we would simply pass a plane *perpendicular* to the intensity axis and containing the color point. The intersection of the plane with the intensity axis would give us an intensity value in the range [0, 1]. We also note with a little thought that the saturation (purity) of a color increases as a function of distance from the intensity axis. In fact, the saturation of points on the intensity axis is zero, as evidenced by the fact that all points along this axis are gray.

In order to see how hue can be determined from a given RGB point, consider Fig. 6.6(b), which shows a plane defined by three points, (black, white, and cyan). The fact that the black and white points are contained in the plane tells us that the intensity axis also is contained in the plane. Furthermore, we see that *all* points contained in the plane segment defined by the intensity axis and the boundaries of the cube have the *same* hue (cyan in this case). This is because the colors inside a color triangle are various combinations or mixtures of the three vertex colors. If two of those vertices are black and white, and the third is a color point, all points on the triangle must have the same hue since the black and white components do not contribute to changes in hue (of course, the intensity and saturation of points in this triangle do change). By rotating the shaded plane about the vertical intensity axis, we would obtain different hues. From these concepts we arrive at the conclusion that the hue, saturation, and intensity values required to form the HSI space can be obtained from the RGB color cube. That is, we can convert any RGB point to a corresponding point is the HSI color model by working out the geometrical formulas describing the reasoning just outlined in the preceding discussion.

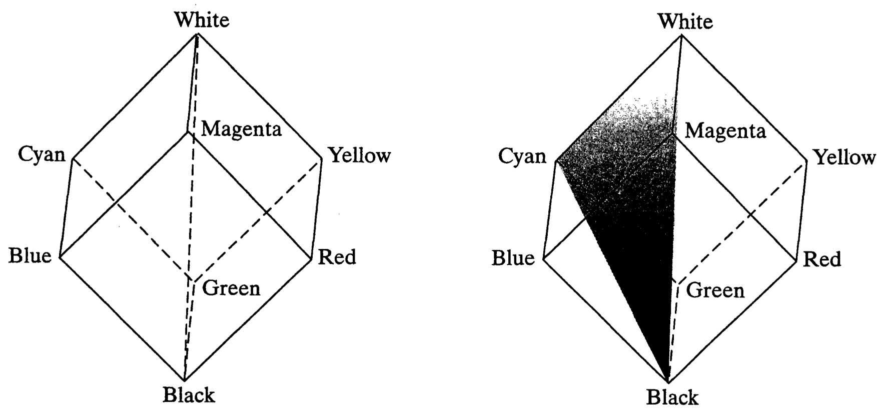


FIGURE 6.6 Relationship between the RGB and HSI color models.

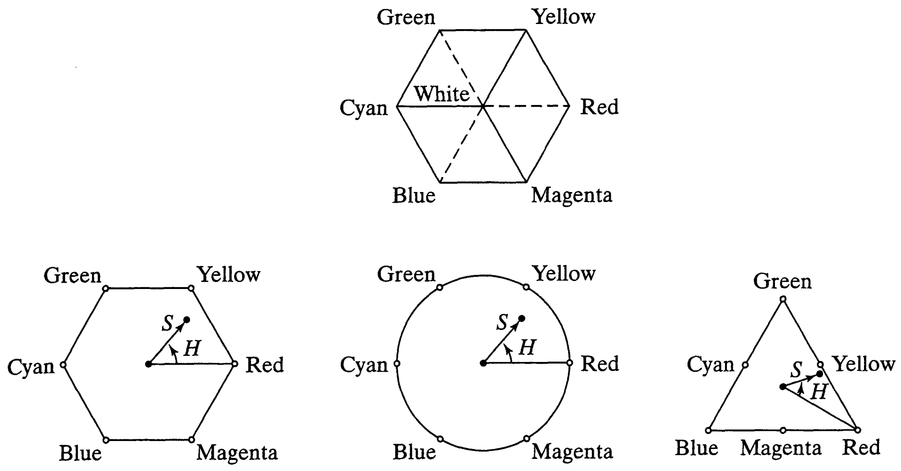


FIGURE 6.7 Hue and saturation in the HSI color model. The dot is an arbitrary color point. The angle from the red axis gives the hue, and the length of the vector is the saturation. The intensity of all colors in any of these planes is given by the position of the plane on the vertical intensity axis.

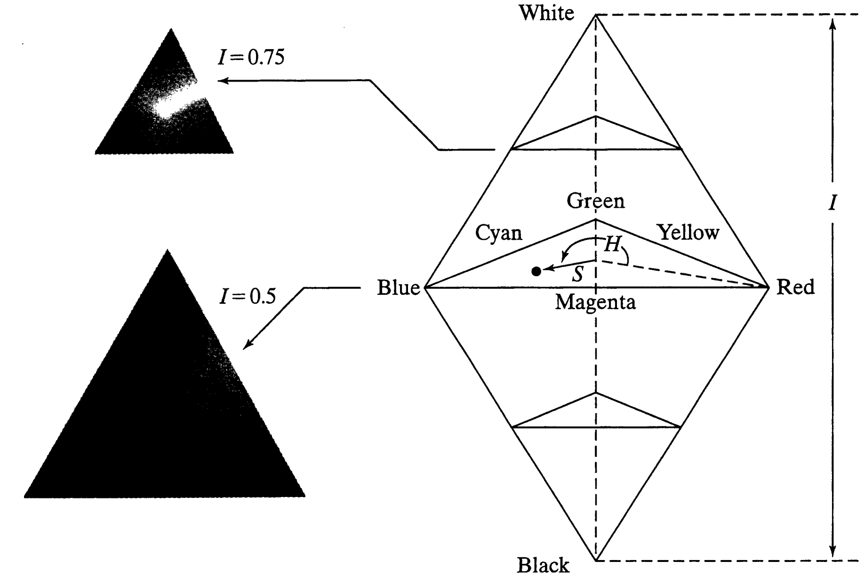
Based on the preceding discussion, we see that the HSI space consists of a vertical intensity axis and the locus of color points that lie on a plane perpendicular to this axis. As the plane moves up and down the intensity axis, the boundaries defined by the intersection of the plane with the faces of the cube have either a triangular or hexagonal shape. This can be visualized more readily by looking at the cube down its gray-scale axis, as shown in Fig. 6.7(a). In this plane we see that the primary colors are separated by 120°. The secondary colors are 60° from the primaries, which means that the angle between secondary colors also is 120°.

Figure 6.7(b) shows the hexagonal shape and an arbitrary color point (shown as a dot). The hue of the point is determined by an angle from some reference point. Usually (but not always) an angle of 0° from the red axis designates 0 hue, and the hue increases counterclockwise from there. The saturation (distance from the vertical axis) is the length of the vector from the origin to the point. Note that the origin is defined by the intersection of the color plane with the vertical intensity axis. The important components of the HSI color space are the vertical intensity axis, the length of the vector to a color point, and the angle this vector makes with the red axis. Therefore, it is not unusual to see the HSI plane defined is terms of the hexagon just discussed, a triangle, or even a circle, as Figs. 6.7(c) and (d) show. The shape chosen is not important because any one of these shapes can be warped into one of the other two by a geometric transformation. Figure 6.8 shows the HSI model based on color triangles and also on circles.

Converting Colors from RGB to HSI

In the following discussion we give the RGB to HSI conversion equations without derivation. See the book Web site (the address is listed in Section 1.5) for a detailed derivation of these equations. Given an image in RGB color format, the H component of each RGB pixel is obtained using the equation

$$H = \begin{cases} \theta & \text{if } B \leq G \\ 360 - \theta & \text{if } B > G \end{cases}$$



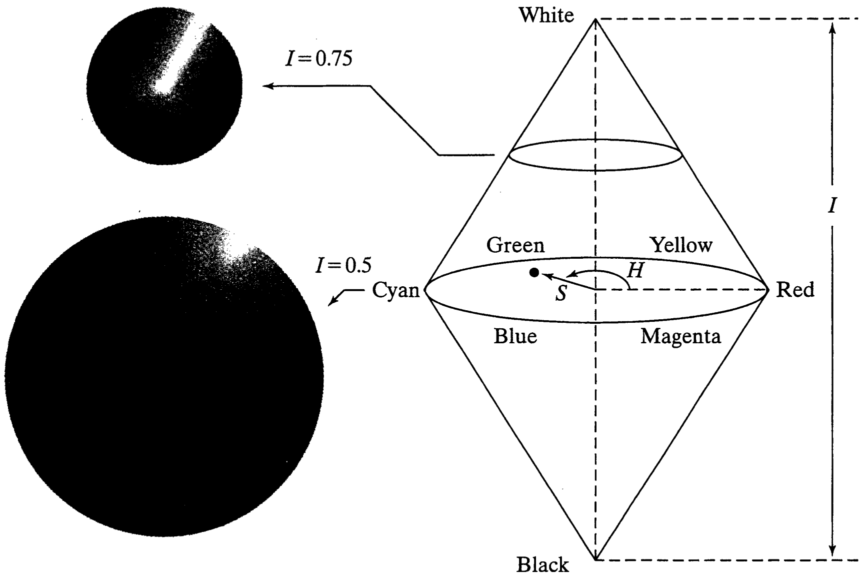


FIGURE 6.8 The HSI color model based on (a) triangular and (b) circular color planes. The triangles and circles are perpendicular to the vertical intensity axis.

with

$$\theta = \cos^{-1}\left\{\frac{\frac{1}{2}[(R-G)+(R-B)]}{[(R-G)^2+(R-B)(G-B)]^{1/2}}\right\}$$

The saturation component is given by

$$S = 1 - \frac{3}{(R+G+B)}[\min(R,G,B)]$$

Finally, the intensity component is given by

$$I = \frac{1}{3}(R+G+B)$$

It is assumed that the RGB values have been normalized to the range [0, 1], and that angle θ is measured with respect to the red axis of the HSI space, as indicated in Fig. 6.7. Hue can be normalized to the range [0, 1] by dividing by 360° all values resulting from the equation for H. The other two HSI components already are in this range if the given RGB values are in the interval [0, 1].

Converting Colors from HSI to RGB

Given values of HSI in the interval [0, 1], we now find the corresponding RGB values in the same range. The applicable equations depend on the values of H. There are three sectors of interest, corresponding to the 120° intervals in the separation of primaries (see Fig. 6.7). We begin by multiplying H by 360°, which returns the hue to its original range of [0°, 360°].

RG sector ($0° \leq H < 120°$): When H is in this sector, the RGB components are given by the equations

$$B = I(1-S)$$

$$R = I\left[1 + \frac{S\cos H}{\cos(60° - H)}\right]$$

and

$$G = 3I - (R+B)$$

GB sector ($120° \leq H < 240°$): If the given value of H is in this sector, we first subtract 120° from it:

$$H = H - 120°$$

Then the RGB components are

$$R = I(1-S)$$

$$G = I\left[1 + \frac{S\cos H}{\cos(60° - H)}\right]$$

and

$$B = 3I - (R + G)$$

BR sector ($240° \le H \le 360°$): Finally, if H is in this range, we subtract 240° from it:

$$H = H - 240°$$

Then the RGB components are

$$G = I(1 - S)$$

$$B = I\left[1 + \frac{S \cos H}{\cos(60° - H)}\right]$$

and

$$R = 3I - (G + B)$$

Use of these equations for image processing is discussed later in this chapter.

An M-function for Converting from RGB to HSI

The following function,

rgb2hsi

```
hsi = rgb2hsi(rgb)
```

implements the equations just discussed for converting from RGB to HSI. To simplify the notation, we use rgb and hsi to denote RGB and HSI images, respectively. The documentation in the code details the use of this function.

```
function hsi = rgb2hsi(rgb)
%RGB2HSI Converts an RGB image to HSI.
%   HSI = RGB2HSI(RGB) converts an RGB image to HSI. The input image
%   is assumed to be of size M-by-N-by-3, where the third dimension
%   accounts for three image planes: red, green, and blue, in that
%   order. If all RGB component images are equal, the HSI conversion
%   is undefined. The input image can be of class double (with values
%   in the range [0, 1]), uint8, or uint16.
%
%   The output image, HSI, is of class double, where:
%      hsi(:, :, 1) = hue image normalized to the range [0, 1] by
%                     dividing all angle values by 2*pi.
%      hsi(:, :, 2) = saturation image, in the range [0, 1].
%      hsi(:, :, 3) = intensity image, in the range [0, 1].

% Extract the individual component immages.
rgb = im2double(rgb);
r = rgb(:, :, 1);
g = rgb(:, :, 2);
b = rgb(:, :, 3);

% Implement the conversion equations.
num = 0.5*((r - g) + (r - b));
```

```
den = sqrt((r - g).^2 + (r - b).*(g - b));
theta = acos(num./(den + eps));

H = theta;
H(b > g) = 2*pi - H(b > g);
H = H/(2*pi);

num = min(min(r, g), b);
den = r + g + b;
den(den == 0) = eps;
S = 1 - 3.* num./den;

H(S == 0) = 0;

I = (r + g + b)/3;

% Combine all three results into an hsi image.
hsi = cat(3, H, S, I);
```

An M-function for Converting from HSI to RGB

The following function,

```
rgb = hsi2rgb(hsi)
```

hsi2rgb

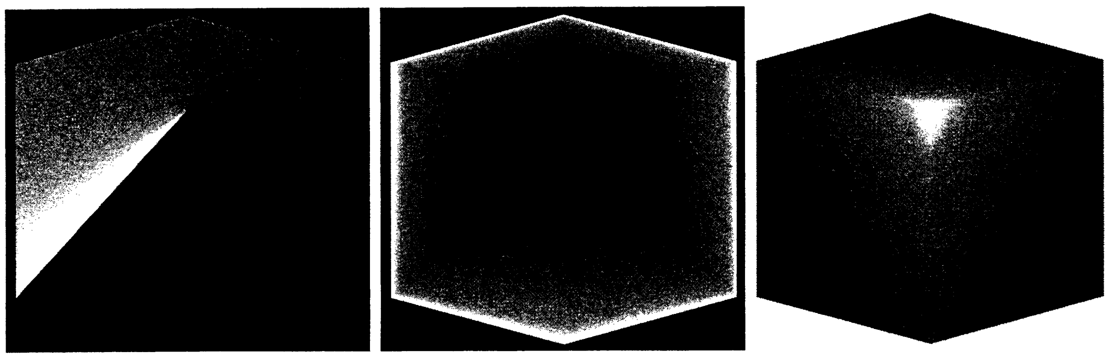
implements the equations for converting from HSI to RGB. The documentation in the code details the use of this function.

```
function rgb = hsi2rgb(hsi)
%HSI2RGB Converts an HSI image to RGB.
%   RGB = HSI2RGB(HSI) converts an HSI image to RGB, where HSI
%   is assumed to be of class double with:
%      hsi(:, :, 1) = hue image, assumed to be in the range
%                     [0, 1] by having been divided by 2*pi.
%      hsi(:, :, 2) = saturation image, in the range [0, 1].
%      hsi(:, :, 3) = intensity image, in the range [0, 1].
%
%   The components of the output image are:
%      rgb(:, :, 1) = red.
%      rgb(:, :, 2) = green.
%      rgb(:, :, 3) = blue.

% Extract the individual HSI component images.
H = hsi(:, :, 1) * 2 * pi;
S = hsi(:, :, 2);
I = hsi(:, :, 3);

% Implement the conversion equations.
R = zeros(size(hsi, 1), size(hsi, 2));
G = zeros(size(hsi, 1), size(hsi, 2));
B = zeros(size(hsi, 1), size(hsi, 2));

% RG sector (0 <= H < 2*pi/3).
idx = find( (0 <= H) & (H < 2*pi/3));
B(idx) = I(idx) .* (1 - S(idx));
```

```
R(idx) = I(idx) .* (1 + S(idx) .* cos(H(idx)) ./ ...
                                        cos(pi/3 - H(idx)));
G(idx) = 3*I(idx) - (R(idx) + B(idx));

% BG sector (2*pi/3 <= H < 4*pi/3).
idx = find( (2*pi/3 <= H) & (H < 4*pi/3) );
R(idx) = I(idx) .* (1 - S(idx));
G(idx) = I(idx) .* (1 + S(idx) .* cos(H(idx) - 2*pi/3) ./ ...
                         cos (pi - H(idx)));
B(idx) = 3*I(idx) - (R(idx) + G(idx));

% BR sector.
idx = find( (4*pi/3 <= H) & (H <= 2*pi));
G(idx) = I(idx) .* (1 - S(idx));
B(idx) = I(idx) .* (1 + S(idx) .* cos(H(idx) - 4*pi/3) ./ ...
                                         cos(5*pi/3 - H(idx)));
R(idx) = 3*I(idx) - (G(idx) + B(idx));

% Combine all three results into an RGB image. Clip to [0, 1] to
% compensate for floating-point arithmetic rounding effects.
rgb = cat(3, R, G, B);
rgb = max(min(rgb, 1), 0);
```

EXAMPLE 6.2:
Converting from RGB to HSI.

■ Figure 6.9 shows the hue, saturation, and intensity components of an image of an RGB cube on a white background, similar to the image in Fig. 6.2(b). Figure 6.9(a) is the hue image. Its most distinguishing feature is the discontinuity in value along a 45° line in the front (red) plane of the cube. To understand the reason for this discontinuity, refer to Fig. 6.2(b), draw a line from the red to the white vertices of the cube, and select a point in the middle of this line. Starting at that point, draw a path to the right, following the cube around until you return to the starting point. The major colors encountered on this path are yellow, green, cyan, blue, magenta, and back to red. According to Fig. 6.7, the value of hue along this path should increase



a b c

FIGURE 6.9 HSI component images of an image of an RGB color cube. (a) Hue, (b) saturation, and (c) intensity images.

from 0° to 360° (i.e., from the lowest to highest possible values of hue). This is precisely what Fig. 6.9(a) shows because the lowest value is represented as black and the highest value as white in the figure.

The saturation image in Fig. 6.9(b) shows progressively darker values toward the white vertex of the RGB cube, indicating that colors become less and less saturated as they approach white. Finally, every pixel in the intensity image shown in Fig. 6.9(c) is the average of the RGB values at the corresponding pixel in Fig. 6.2(b). Note that the background in this image is white because the intensity of the background in the color image is white. It is black in the other two images because the hue and saturation of white are zero. ■

6.3 The Basics of Color Image Processing

In this section we begin the study of processing techniques applicable to color images. Although they are far from being exhaustive, the techniques developed in the sections that follow are illustrative of how color images are handled for a variety of image-processing tasks. For the purposes of the following discussion we subdivide color image processing into three principal areas: (1) *color transformations* (also called *color mappings*); (2) *spatial processing* of individual color planes; and (3) *color vector processing*. The first category deals with processing the pixels of each color plane based strictly on their values and not on their spatial coordinates. This category is analogous to the material in Section 3.2 dealing with intensity transformations. The second category deals with spatial (neighborhood) filtering of *individual* color planes and is analogous to the discussion in Sections 3.4 and 3.5 on spatial filtering.

The third category deals with techniques based on processing all components of a color image simultaneously. Because full-color images have at least three components, color pixels really are vectors. For example, in the RGB system, each color point can be interpreted as a vector extending from the origin to that point in the RGB coordinate system (see Fig. 6.2).

Let **c** represent an arbitrary vector in RGB color space:

$$\mathbf{c} = \begin{bmatrix} c_R \\ c_G \\ c_B \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

This equation indicates that the components of **c** are simply the RGB components of a color image at a point. We take into account the fact that the color components are a function of coordinates (x, y) by using the notation

$$\mathbf{c}(x, y) = \begin{bmatrix} c_R(x, y) \\ c_G(x, y) \\ c_B(x, y) \end{bmatrix} = \begin{bmatrix} R(x, y) \\ G(x, y) \\ B(x, y) \end{bmatrix}$$

For an image of size $M \times N$, there are MN such vectors, $\mathbf{c}(x, y)$, for $x = 0, 1, 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$.

In some cases, equivalent results are obtained whether color images are processed one plane at a time or as vector quantities. However, as explained in

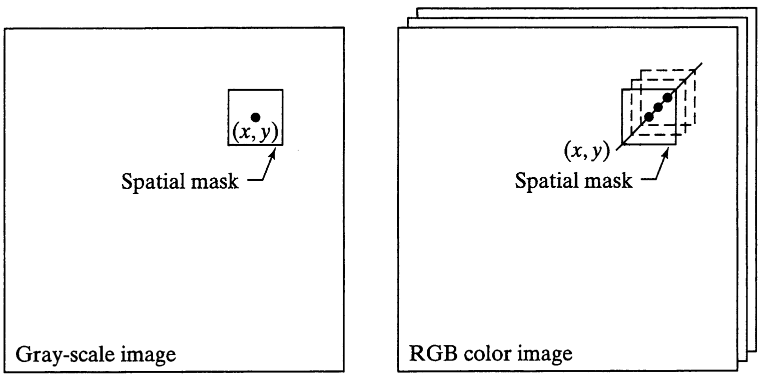


FIGURE 6.10 Spatial masks for gray-scale and RGB color images.

more detail in Section 6.6, this is not always the case. In order for independent color component and vector-based processing to be equivalent, two conditions have to be satisfied: First, the process has to be applicable to both vectors and scalars. Second, the operation on each component of a vector must be independent of the other components. As an illustration, Fig. 6.10 shows spatial neighborhood processing of gray-scale and full-color images. Suppose that the process is neighborhood averaging. In Fig. 6.10(a), averaging would be accomplished by summing the gray levels of all the pixels in the neighborhood and dividing by the total number of pixels in the neighborhood. In Fig. 6.10(b) averaging would be done by summing all the vectors in the neighborhood and dividing each component by the total number of vectors in the neighborhood. But each component of the average vector is the sum of the pixels in the image corresponding to that component, which is the same as the result that would be obtained if the averaging were done on the neighborhood of each component image individually, and then the color vector were formed.

6.4 Color Transformations

The techniques described in this section are based on processing the color components of a color image or intensity component of a monochrome image within the context of a single color model. For color images, we restrict attention to transformations of the form

$$s_i = T_i(r_i), \quad i = 1, 2, \ldots, n$$

where r_i and s_i are the color components of the input and output images, n is the dimension of (or number of color components in) the color space of r_i, and the T_i are referred to as *full-color transformation* (or *mapping*) functions.

If the input images are monochrome, then we write an equation of the form

$$s_i = T_i(r), \quad i = 1, 2, \ldots, n$$

where r denotes gray-level values, s_i and T_i are as above, and n is the number of color components in s_i. This equation describes the mapping of gray levels into arbitrary colors, a process frequently referred to as a *pseudocolor transformation* or *pseudocolor mapping*. Note that the first equation can be used to process monochrome images in RGB space if we let $r_1 = r_2 = r_3 = r$. In either case, the equations given here are straightforward extensions of the intensity transformation equation introduced in Section 3.2. As is true of the transformations in that section, all n pseudo- or full-color transformation functions $\{T_1, T_2, \ldots, T_n\}$ are independent of the spatial image coordinates (x, y).

Some of the gray-scale transformations introduced in Chapter 3, like `imcomplement`, which computes the negative of an image, are independent of the gray-level content of the image being transformed. Others, like `histeq`, which depends on gray-level distribution, are adaptive, but the transformation is fixed once the necessary parameters have been estimated. And still others, like `imadjust`, which requires the user to select appropriate curve shape parameters, are often best specified interactively. A similar situation exists when working with pseudo- and full-color mappings—particularly when human viewing and interpretation (e.g., for color balancing) are involved. In such applications, the selection of appropriate mapping functions is best accomplished by directly manipulating graphical representations of candidate functions and viewing their combined effect (in real time) on the images being processed.

Figure 6.11 illustrates a simple but powerful way to specify mapping functions graphically. Figure 6.11(a) shows a transformation that is formed by linearly interpolating three *control points* (the circled coordinates in the figure); Fig. 6.11(b) shows the transformation that results from a cubic spline interpolation of the same three points; and Figs. 6.11(c) and (d) provide more complex linear and cubic spline interpolations, respectively. Both types of interpolation are supported in MATLAB. Linear interpolation is implemented by using

```
z = interp1q(x, y, xi)
```

interp1q

which returns a column vector containing the values of the linearly interpolated 1-D function `z` at points `xi`. Column vectors `x` and `y` specify the horizontal and vertical coordinate pairs of the underlying control points. The elements of `x` must increase monotonically. The length of `z` is equal to the length of `xi`. Thus, for example,

```
>> z = interp1q([0 255]', [0 255]', [0: 255]')
```

produces a 256-element one-to-one mapping connecting control points (0, 0) and (255, 255)—that is, `z = [0 1 2 . . . 255]'`.

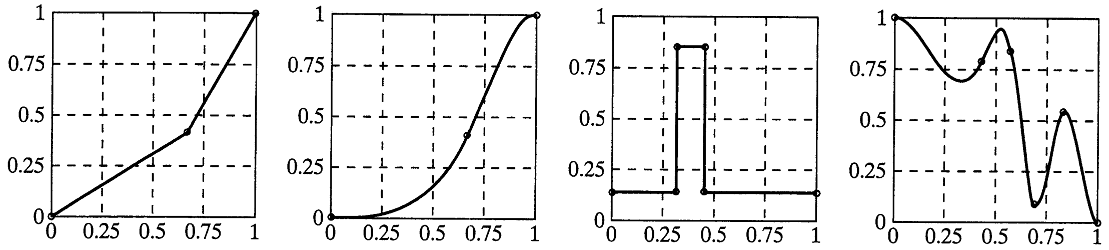


FIGURE 6.11 Specifying mapping functions using control points: (a) and (c) linear interpolation, and (b) and (d) cubic spline interpolation.

In a similar manner, cubic spline interpolation is implemented using the `spline` function,

spline

```
z = spline(x, y, xi)
```

where variables `z`, `x`, `y`, and `xi` are as described in the previous paragraph for `interp1q`. However, the `xi` must be distinct for use in function `spline`. Moreover, if `y` contains two more elements than `x`, its first and last entries are assumed to be the end slopes of the cubic spline. The function depicted in Fig. 6.11(b), for example, was generated using zero-valued end slopes.

The specification of transformation functions can be made interactive by graphically manipulating the control points that are input to functions `interp1q` and `spline` and displaying in real time the results of the transformation functions on the images being processed. The `ice` (interactive color editing) function does precisely this. Its syntax is

ice

```
g = ice('Property Name', 'Property Value', . . .)
```

The development of function `ice`, *given in Appendix B, is a comprehensive illustration of how to design a graphical user interface (GUI) in MATLAB.*

where `'Property Name'` and `'Property Value'` must appear in pairs, and the dots indicate repetitions of the pattern consisting of corresponding input pairs. Table 6.4 lists the valid pairs for use in function `ice`. Some examples are given later in this section.

With reference to the `'wait'` parameter, when the `'on'` option is selected either explicitly or by default, the output `g` is the processed image. In this case, `ice` takes control of the process, including the cursor, so nothing can be typed on the command window until the function is closed, at which time the final result is image `g`. When `'off'` is selected, `g` is the *handle*[†] of the processed image, and control is returned immediately to the command window; therefore, new commands can be typed with the `ice` function still active. To obtain the properties of an image with handle `g` we use the `get` function

get

```
h = get(g)
```

This function returns all properties and applicable current values of the graphics object identified by the handle `g`. The properties are stored in structure `h`,

TABLE 6.4 Valid inputs for function `ice`.

Property Name	Property Value
`'image'`	An RGB or monochrome input image, `f`, to be transformed by interactively specified mappings.
`'space'`	The color space of the components to be modified. Possible values are `'rgb'`, `'cmy'`, `'hsi'`, `'hsv'`, `'ntsc'` (or `'yiq'`), and `'ycbcr'`. The default is `'rgb'`.
`'wait'`	If `'on'` (the default), `g` is the mapped input image. If `'off'`, `g` is the handle of the mapped input image.

[†]Whenever MATLAB creates a graphics object, it assigns an identifier (called a *handle*) to the object, used to access the object's properties. Graphics handles are useful when modifying the appearance of graphs or creating custom plotting commands by writing M-files that create and manipulate objects directly.

so typing `h` at the prompt lists all the properties of the processed image (see Sections 2.10.6 and 11.1.1 for an explanation of structures). To extract a particular property, we type `h.PropertyName`.

Letting `f` denote an RGB or monochrome image, the following are examples of the syntax of function `ice`:

```
>> ice                                  % Only the ice graphical
                                        % interface is displayed.
>> g = ice('image', f);                 % Shows and returns the mapped
                                        % image g.
>> g = ice('image', f, 'wait', 'off');  % Shows g and returns
                                        % the handle.
>> g = ice('image', f, 'space', 'hsi'); % Maps RGB image f in HSI space.
```

Note that when a color space other than RGB is specified, the input image (whether monochrome or RGB) is transformed to the specified space before any mapping is performed. The mapped image is then converted to RGB for output. The output of `ice` is always RGB; its input is always monochrome or RGB. If we type `g = ice('image', f)`, an image and graphical user interface (GUI) like that shown in Fig. 6.12 appear on the MATLAB desktop. Initially,

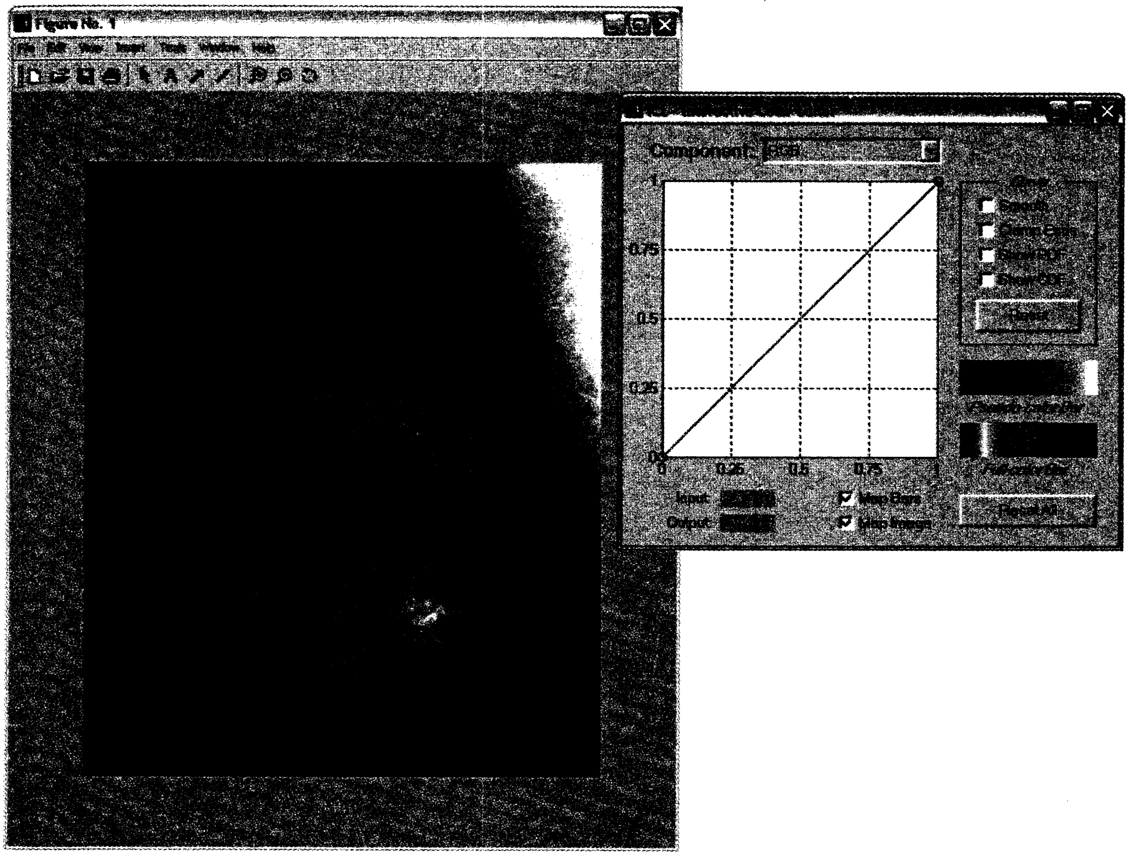


FIGURE 6.12 The typical opening windows of function `ice`. (Image courtesy of G. E. Medical Systems.)

TABLE 6.5 Manipulating control points with the mouse.

Mouse Action[†]	Result
Left Button	Move control point by pressing and dragging.
Left Button + Shift Key	Add control point. The location of the control point can be changed by dragging (while still pressing the Shift Key).
Left Button + Control Key	Delete control point.

[†] For three button mice, the left, middle, and right buttons correspond to the move, add, and delete operations in the table.

the transformation curve is a straight line with a control point at each end. Control points are manipulated with the mouse, as summarized in Table 6.5. Table 6.6 lists the function of the other GUI components. The following examples show typical applications of function `ice`.

EXAMPLE 6.3: Inverse mappings: monochrome negatives and color complements.

■ Figure 6.13(a) shows the `ice` interface after the default RGB curve of Fig. 6.12 is modified to produce an inverse or negative mapping function. To create the new mapping function, control point (0, 0) is moved (by clicking and dragging it to the upper-left corner) to (0, 1), and control point (1, 1) is moved similarly to coordinate (1, 0). Note how the coordinates of the cursor are displayed in red in the Input/Output boxes. Only the RGB map is modified; the

TABLE 6.6 Function of the checkboxes and pushbuttons in the `ice` GUI.

GUI Element	Function
Smooth	Checked for cubic spline (smooth curve) interpolation. If unchecked, piecewise linear interpolation is used.
Clamp Ends	Checked to force the starting and ending curve slopes in cubic spline interpolation to 0. Piecewise linear interpolation is not affected.
Show PDF	Display probability density function(s) [i.e., histogram(s)] of the image components affected by the mapping function.
Show CDF	Display cumulative distribution function(s) instead of PDFs. (Note: PDFs and CDFs cannot be displayed simultaneously.)
Map Image	If checked, image mapping is enabled; otherwise it is not.
Map Bars	If checked, pseudo- and full-color bar mapping is enabled; otherwise the unmapped bars (a gray wedge and hue wedge, respectively) are displayed.
Reset	Initialize the currently displayed mapping function and uncheck all curve parameters.
Reset All	Initialize all mapping functions.
Input/Output	Shows the coordinates of a *selected* control point on the transformation curve. Input refers to the horizontal axis, and Output to the vertical axis.
Component	Select a mapping function for interactive manipulation. In RGB space, possible selections include R, G, B, and RGB (which maps all three color components). In HSI space, the options are H, S, I, and HSI, and so on.

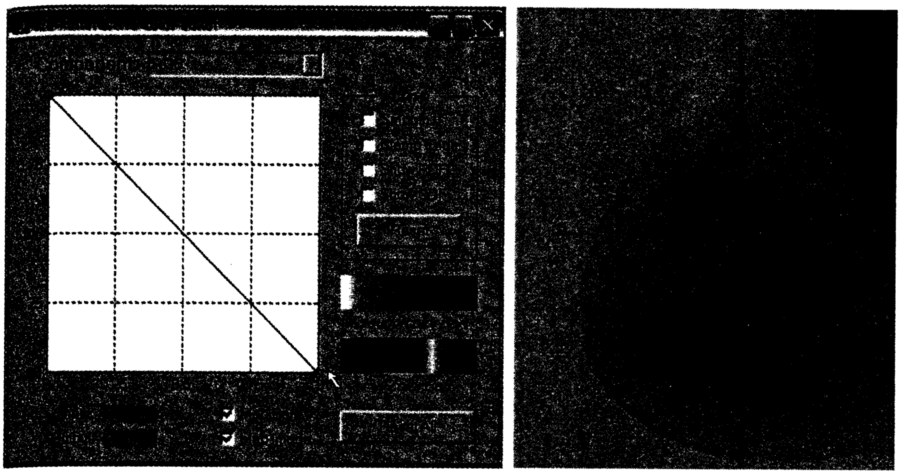


FIGURE 6.13 (a) A negative mapping function, and (b) its effect on the monochrome image of Fig. 6.12.

individual R, G, and B maps are left in their 1:1 default states (see the Component entry in Table 6.6). For monochrome inputs, this guarantees monochrome outputs. Figure 6.13(b) shows the monochrome negative that results from the inverse mapping. Note that it is identical to Fig. 3.3(b), which was obtained using the `imcomplement` function. The pseudocolor bar in Fig. 6.13(a) is the "photographic negative" of the original gray-scale bar in Fig. 6.12.

Default (i.e., 1:1) mappings are not shown in most examples.

Inverse or negative mapping functions also are useful in color processing. As can be seen in Figs. 6.14(a) and (b), the result of the mapping is reminiscent of conventional color film negatives. For instance, the red stick of chalk in the bottom row of Fig. 6.14(a) is transformed to cyan in Fig. 6.14(b)—the *color complement* of red. The complement of a primary color is the mixture of the other two primaries (e.g., cyan is blue plus green). As in the gray-scale case, color complements are useful for enhancing detail that is embedded in dark regions of color—particularly when the regions are dominant in size. Note that the *Full-color Bar* in Fig. 6.13(a) contains the complements of the hues in the *Full-color Bar* of Fig. 6.12. ■

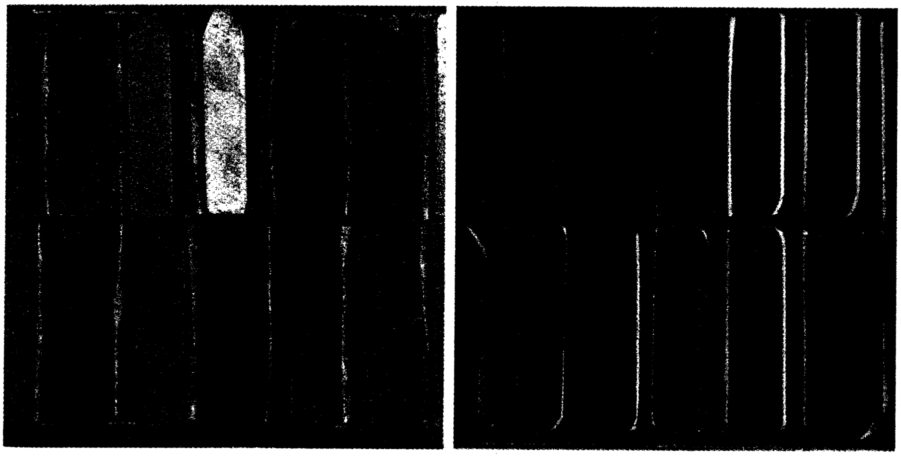


FIGURE 6.14 (a) A full color image, and (b) its negative (color complement).

EXAMPLE 6.4: Monochrome and color contrast enhancement.

■ Consider next the use of function `ice` for monochrome and color contrast manipulation. Figures 6.15(a) through (c) demonstrate the effectiveness of `ice` in processing monochrome images. Figures 6.15(d) through (f) show similar effectiveness for color inputs. As in the previous example, mapping functions that are not shown remain in their default or 1 : 1 state. In both processing sequences, the Show PDF checkbox is enabled. Thus, the histogram of the aerial photo in (a) is displayed under the gamma-shaped mapping function (see Section 3.2.1) in (c); and three histograms are provided in (f) for the color image in (d)—one for each of its three color components. Although the S-shaped mapping function in (f) increases the contrast of the image in (d) [compare it to (e)], it also has a slight effect on hue. The small change of color is virtually imperceptible in (e), but is an obvious result of the mapping, as can be seen in the mapped full-color reference bar in (f). Recall from the previous example that equal changes to the three components of an RGB image can have a dramatic effect on color (see the color complement mapping in Fig. 6.14). ■

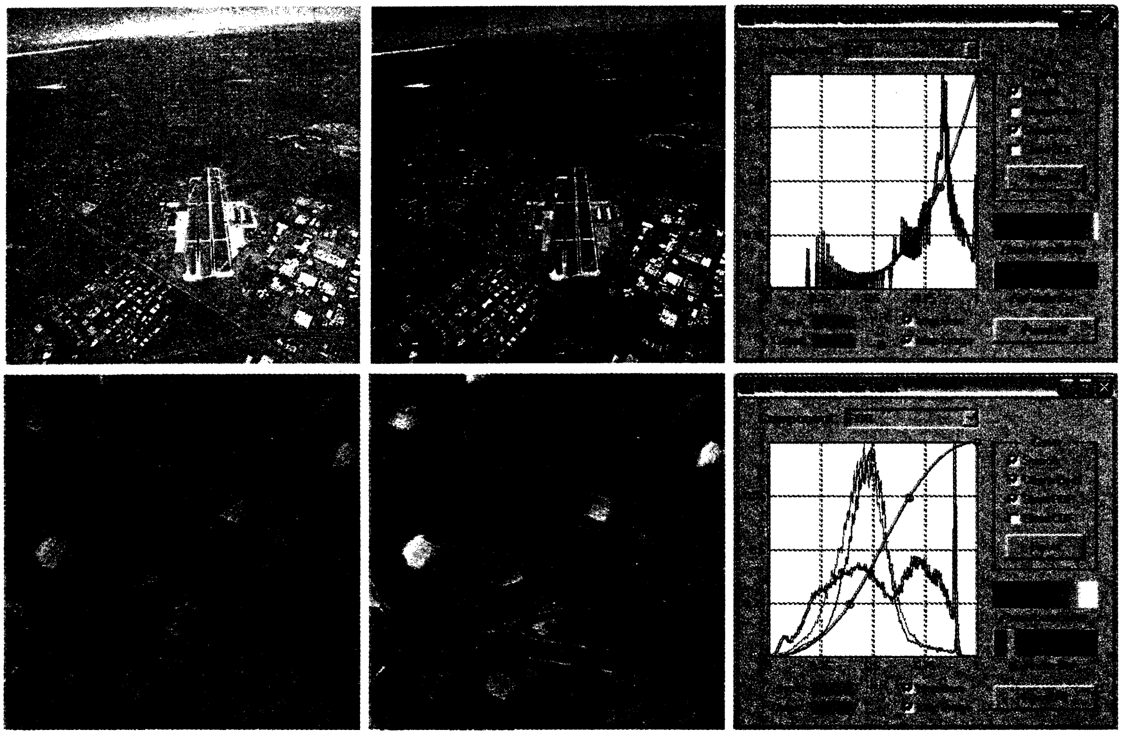


FIGURE 6.15 Using function `ice` for monochrome and full color contrast enhancement: (a) and (d) are the input images, both of which have a "washed-out" appearance; (b) and (e) show the processed results; (c) and (f) are the `ice` displays. (Original monochrome image for this example courtesy of NASA.)

The red, green, and blue components of the input images in Examples 6.3 and 6.4 are mapped identically—that is, using the same transformation function. To avoid the specification of three identical functions, function `ice` provides an "all components" function (the RGB curve when operating in the RGB color space) that is used to map all input components. The remaining examples demonstrate transformations in which the three components are processed differently.

EXAMPLE 6.5: Pseudocolor mappings.

■ As noted earlier, when a monochrome image is represented in the RGB color space and the resulting components are mapped independently, the transformed result is a pseudocolor image in which input image gray levels have been replaced by arbitrary colors. Transformations that do this are useful because the human eye can distinguish between millions of colors—but relatively few shades of gray. Thus, pseudocolor mappings are used frequently to make small changes in gray level visible to the human eye or to highlight important gray-scale regions. In fact, the principal use of pseudocolor is human visualization—the interpretation of gray-scale events in an image or sequence of images via gray-to-color assignments.

Figure 6.16(a) is an X-ray image of a weld (the horizontal dark region) containing several cracks and porosities (the bright white streaks running through the middle of the image). A pseudocolor version of the image in shown in

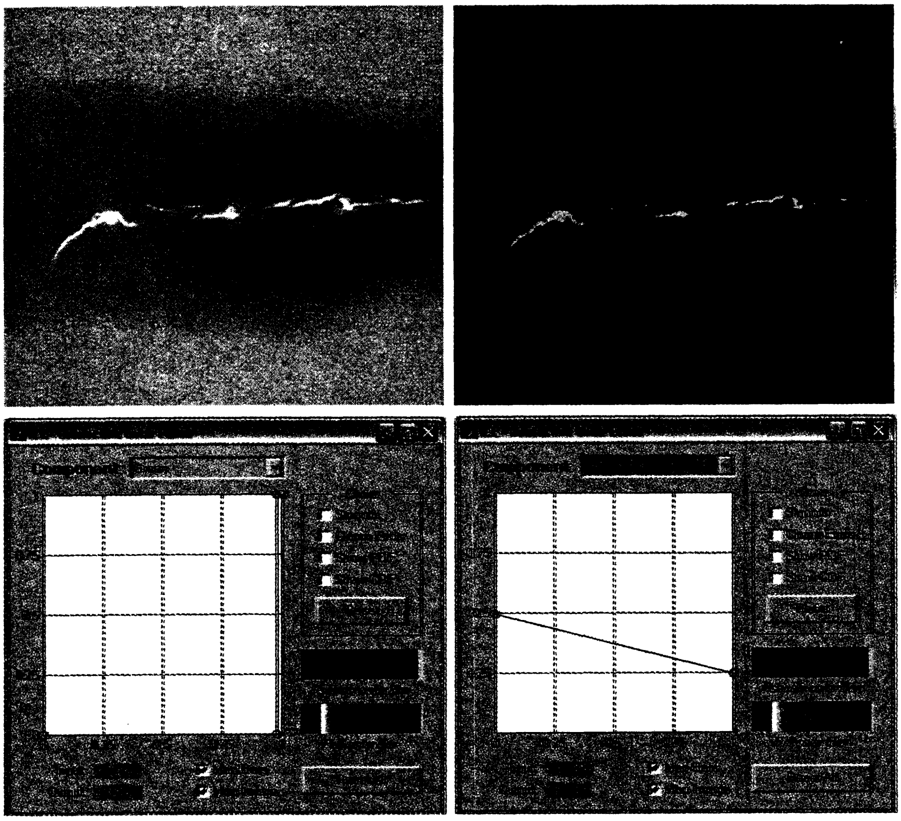


FIGURE 6.16 (a) X-ray of a defective weld; (b) a pseudocolor version of the weld; (c) and (d) mapping functions for the green and blue components. (Original image courtesy of X-TEK Systems, Ltd.)

Fig. 6.16(b); it was generated by mapping the green and blue components of the RGB-converted input using the mapping functions in Figs. 6.16(c) and (d). Note the dramatic visual difference that the pseudocolor mapping makes. The GUI pseudocolor reference bar provides a convenient visual guide to the composite mapping. As can be seen in Figs. 6.16(c) and (d), the interactively specified mapping functions transform the black-to-white gray scale to hues between blue and red, with yellow reserved for white. The yellow, of course, corresponds to weld cracks and porosities, which are the important features in this example. ■

EXAMPLE 6.6: Color balancing.

■ Figure 6.17 shows an application involving a full-color image, in which it is advantageous to map an image's color components independently. Commonly called *color balancing* or *color correction*, this type of mapping has been a mainstay of high-end color reproduction systems but now can be performed on most desktop computers. One important use is photo enhancement. Although color imbalances can be determined objectively by analyzing—with a color spectrometer—a known color in an image, accurate visual assessments are possible when white areas, where the RGB or CMY components should be equal, are present. As can be seen in Fig. 6.17, skin tones also are excellent samples for visual assessments because humans are highly perceptive of proper skin color.

Figure 6.17(a) shows a CMY scan of a mother and her child with an excess of magenta (keep in mind that only an RGB version of the image can be displayed by MATLAB). For simplicity and compatibility with MATLAB, function `ice` accepts only RGB (and monochrome) inputs as well—but can

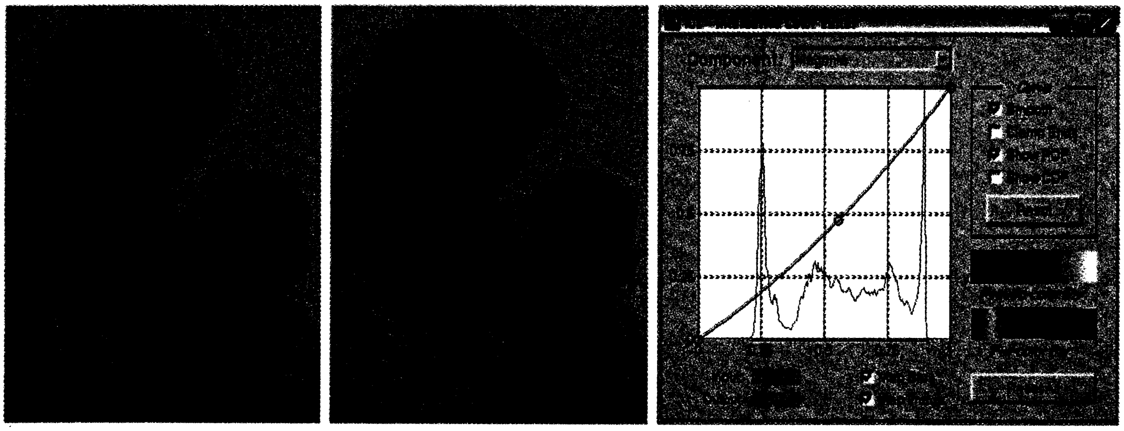


FIGURE 6.17 Using function `ice` for color balancing: (a) an image heavy in magenta; (b) the corrected image; and (c) the mapping function used to correct the imbalance.

process the input in a variety of color spaces, as detailed in Table 6.4. To interactively modify the CMY components of RGB image `f1`, for example, the appropriate `ice` call is

```
>> f2 = ice('image', f1, 'space', 'CMY');
```

As Fig. 6.17 shows, a small decrease in magenta had a significant impact on image color. ■

EXAMPLE 6.7: Histogram based mappings.

■ Histogram equalization is a gray-level mapping process that seeks to produce monochrome images with uniform intensity histograms. As discussed in Section 3.3.2, the required mapping function is the cumulative distribution function (CDF) of the gray levels in the input image. Because color images have multiple components, the gray-scale technique must be modified to handle more than one component and associated histogram. As might be expected, it is unwise to histogram equalize the components of a color image independently. The result usually is erroneous color. A more logical approach is to spread color intensities uniformly, leaving the colors themselves (i.e., the hues) unchanged.

Figure 6.18(a) shows a color image of a caster stand containing cruets and shakers. The transformed image in Fig. 6.18(b), which was produced using the HSI transformations in Figs. 6.18(c) and (d), is significantly brighter. Several of the moldings and the grain of the wood table on which the caster is resting are now visible. The intensity component was mapped using the function in Fig. 6.18(c), which closely approximates the CDF of that component (also displayed in the figure). The hue mapping function in Fig. 6.18(d) was selected to improve the overall color perception of the intensity-equalized result. Note that the histograms of the input and output image's hue, saturation, and intensity components are shown in Figs. 6.18(e) and (f), respectively. The hue components are virtually identical (which is desirable), while the intensity and saturation components were altered. Finally note that, to process an RGB image in the HSI color space, we included the input property name/value pair `'space'/'hsi'` in the call to `ice`. ■

The output images generated in the preceding examples in this section are of type RGB and class `uint8`. For monochrome results, as in Example 6.3, all three components of the RGB output are identical. A more compact representation can be obtained via the `rgb2gray` function of Table 6.3 or by using the command

```
f3 = f2(:, :, 1);
```

where `f2` is an RGB image generated by `ice` and `f3` is a standard MATLAB monochrome image.

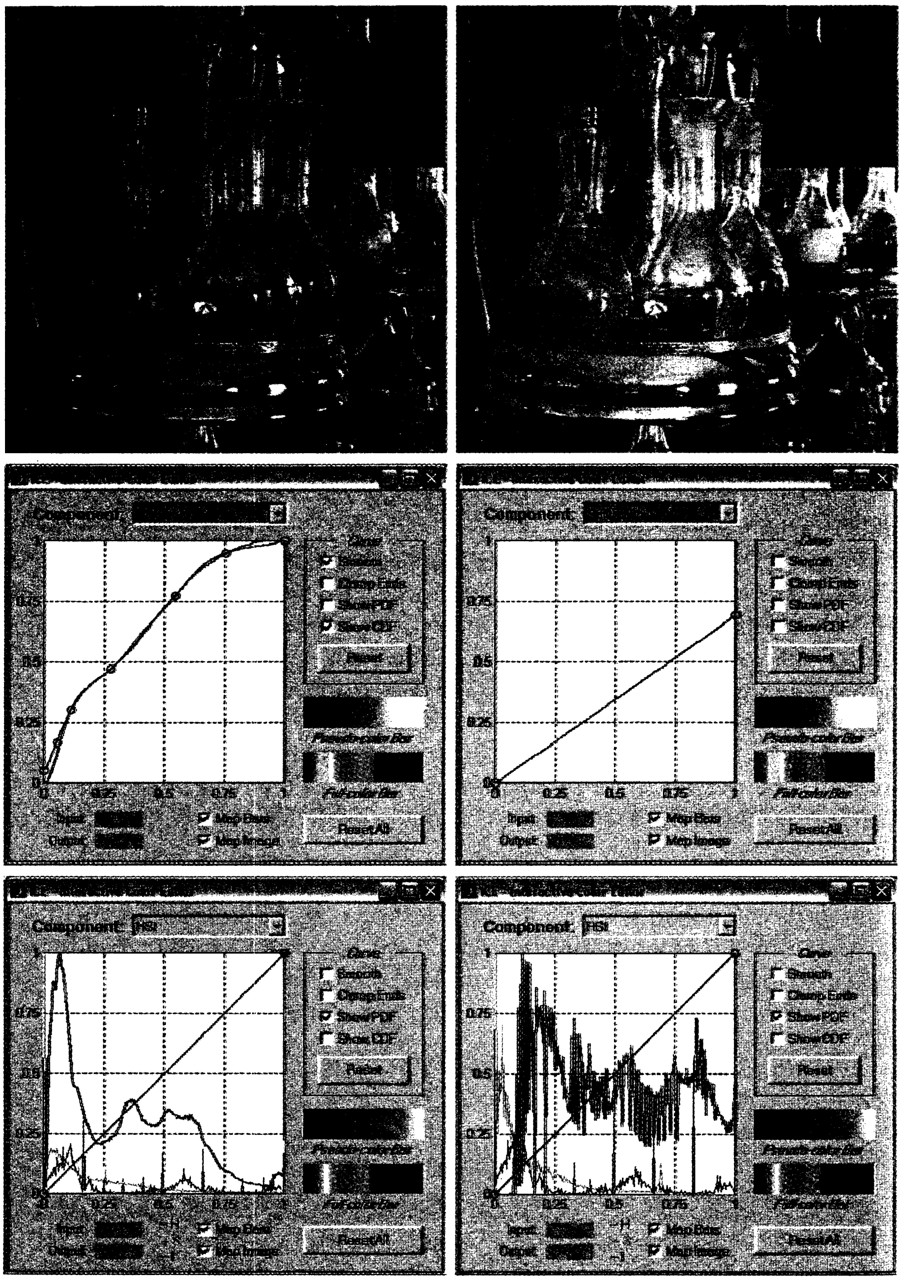


FIGURE 6.18 Histogram equalization followed by saturation adjustment in the HSI color space: (a) input image; (b) mapped result; (c) intensity component mapping function and cumulative distribution function; (d) saturation component mapping function; (e) input image's component histograms; and (f) mapped result's component histograms.

6.5 Spatial Filtering of Color Images

The material in Section 6.4 deals with color transformations performed on single image pixels of single color component planes. The next level of complexity involves performing spatial neighborhood processing, also on single image planes. This breakdown is analogous to the discussion on intensity transformations in Section 3.2, and the discussion on spatial filtering in Sections 3.4 and 3.5. We introduce spatial filtering of color images by concentrating mostly on RGB images, but the basic concepts are applicable to other color models as well. We illustrate spatial processing of color images by two examples of linear filtering: image smoothing and image sharpening.

6.5.1 Color Image Smoothing

With reference to Fig. 6.10(a) and the discussion in Sections 3.4 and 3.5, smoothing (spatial averaging) of a monochrome image can be accomplished by multiplying all pixel values by the corresponding coefficients in the spatial mask (which are all 1s) and dividing by the total number of elements in the mask. The process of smoothing a full-color image using spatial masks is shown in Fig. 6.10(b). The process (in RGB space for example) is formulated in the same way as for gray-scale images, except that instead of single pixels we now deal with vector values in the form shown in Section 6.3.

Let S_{xy} denote the set of coordinates defining a neighborhood centered at (x, y) in a color image. The average of the RGB vectors in this neighborhood is

$$\bar{\mathbf{c}}(x, y) = \frac{1}{K} \sum_{(s,t) \in S_{xy}} \mathbf{c}(s, t)$$

where K is the number of pixels in the neighborhood. It follows from the discussion in Section 6.3 and the properties of vector addition that

$$\bar{\mathbf{c}}(x, y) = \begin{bmatrix} \dfrac{1}{K} \displaystyle\sum_{(s,t) \in S_{xy}} R(s, t) \\ \dfrac{1}{K} \displaystyle\sum_{(s,t) \in S_{xy}} G(s, t) \\ \dfrac{1}{K} \displaystyle\sum_{(s,t) \in S_{xy}} B(s, t) \end{bmatrix}$$

We recognize each component of this vector as the result that we would obtain by performing neighborhood averaging on each individual component image, using standard gray-scale neighborhood processing. Thus, we conclude that smoothing by neighborhood averaging can be carried out on an independent component basis. The results would be the same as if neighborhood averaging were carried out directly in color vector space.

As discussed in Section 3.5.1, IPT linear spatial filters for image smoothing are generated with function `fspecial`, with one of three options: `'average'`, `'disk'`, and `'gaussian'` (see Table 3.4). Once a filter has been generated, filtering is performed by using function `imfilter`, introduced in Section 3.4.1.

Conceptually, smoothing an RGB color image, fc, with a linear spatial filter consists of the following steps:

1. Extract the three component images:

```
>> fR = fc(:, :, 1); fG = fc(:, :, 2); fB = fc(:, :, 3);
```

2. Filter each component image individually. For example, letting w represent a smoothing filter generated using fspecial, we smooth the red component image as follows:

```
>> fR_filtered = imfilter(fR, w);
```

and similarly for the other two component images.

3. Reconstruct the filtered RGB image:

```
>> fc_filtered = cat(3, fR_filtered, fG_filtered, fB_filtered);
```

However, we can perform linear filtering of RGB images in MATLAB using the same syntax employed for monochrome images, allowing us to combine the preceding three steps into one:

```
>> fc_filtered = imfilter(fc, w);
```

EXAMPLE 6.8: Color image smoothing.

■ Figure 6.19(a) shows an RGB image of size 1197 × 1197 pixels and Figs. 6.19(b) through (d) are its RGB component images, extracted using the procedure described in the previous paragraph. Figures 6.20(a) through (c) show the three HSI component images of Fig. 6.19(a), obtained using function rgb2hsi.

Figure 6.21(a) shows the result of smoothing the image in Fig. 6.19(a) using function imfilter with the 'replicate' option and an 'average' filter of size 25 × 25 pixels. The averaging filter was large enough to produce a significant degree of blurring. A filter of this size was selected to demonstrate the difference between smoothing in RGB space and attempting to achieve a similar result using only the intensity component of the image after it had been converted to the HSI color space. Figure 6.21(b) was obtained using the commands:

```
>> h = rgb2hsi(fc);
>> H = h(:, :, 1); S = h(:, :, 2); I =  h(:, :, 3);
>> w = fspecial('average', 25);
>> I_filtered = imfilter(I, w, 'replicate');
>> h = cat(3, H, S, I_filtered);
>> f = hsi2rgb(h);
>> f = min(f, 1); % RGB images must have values in the range [0, 1].
>> imshow(f)
```

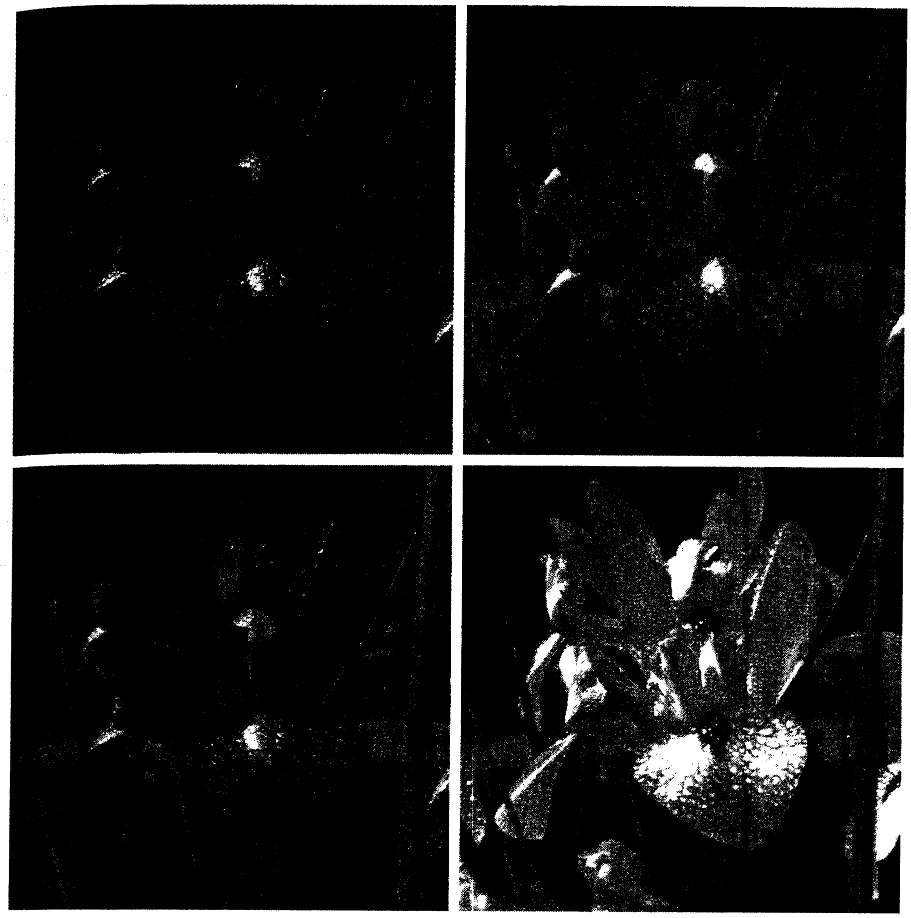


FIGURE 6.19 (a) RGB image; (b) through (d) are the red, green and blue component images, respectively.

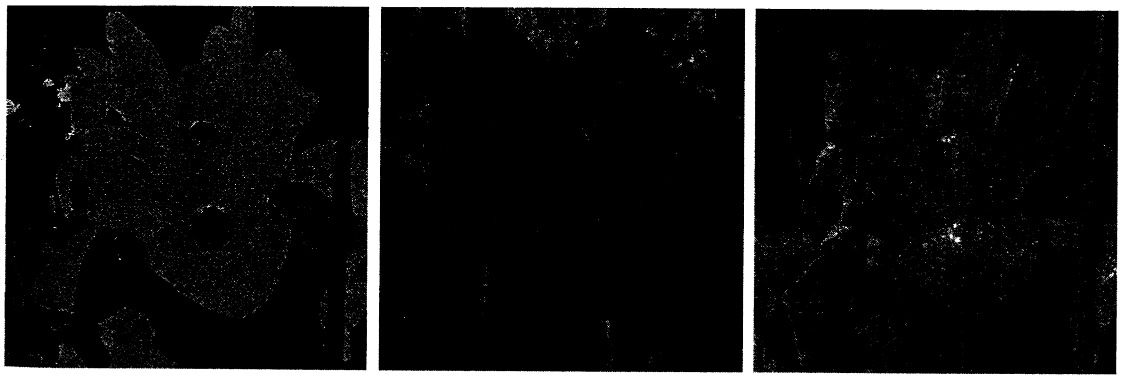
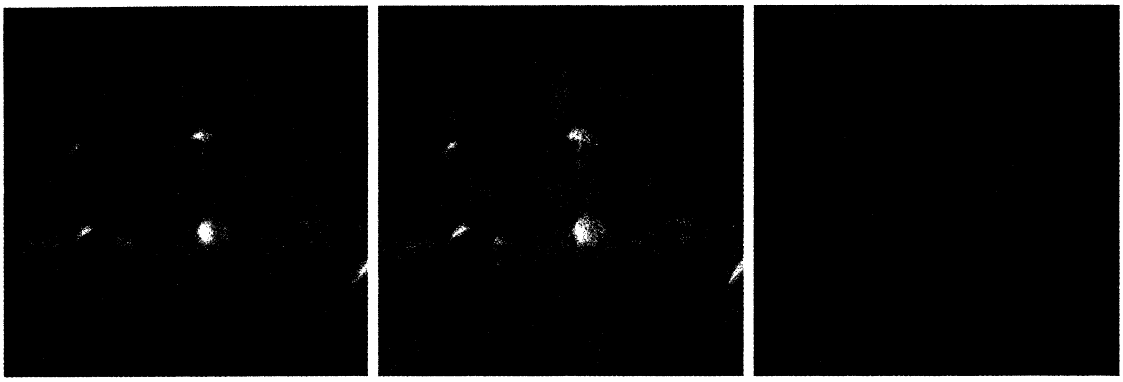


FIGURE 6.20 From left to right: hue, saturation, and intensity components of Fig. 6.19(a).



a b c

FIGURE 6.21 (a) Smoothed RGB image obtained by smoothing the R, G, and B image planes separately. (b) Result of smoothing only the intensity component of the HSI equivalent image. (c) Result of smoothing all three HSI components equally.

Clearly, the two filtered results are quite different. For example, in addition to the image being less blurred, note the green border on the top part of the flower in Fig. 6.21(b). The reason for this is simply that the hue and saturation components were not changed while the variability of values of the intensity components was reduced significantly by the smoothing process. A logical thing to try would be to smooth all three components using the same filter. However, this would change the relative relationship between values of the hue and saturation, thus producing nonsensical colors, as Fig. 6.21(c) shows.

In general, as the size of the mask decreases, the differences obtained when filtering the RGB component images and the intensity component of the HSI equivalent image also decrease. ■

6.5.2 Color Image Sharpening

Sharpening an RGB color image with a linear spatial filter follows the same procedure outlined in the previous section, but using a sharpening filter instead. In this section we consider image sharpening using the Laplacian (see Section 3.5.1). From vector analysis, we know that the Laplacian of a vector is defined as a vector whose components are equal to the Laplacian of the individual scalar components of the input vector. In the RGB color system, the Laplacian of vector **c** introduced in Section 6.3 is

$$\nabla^2[\mathbf{c}(x, y)] = \begin{bmatrix} \nabla^2 R(x, y) \\ \nabla^2 G(x, y) \\ \nabla^2 B(x, y) \end{bmatrix}$$

which, as in the previous section, tells us that we can compute the Laplacian of a full-color image by computing the Laplacian of each component image separately.

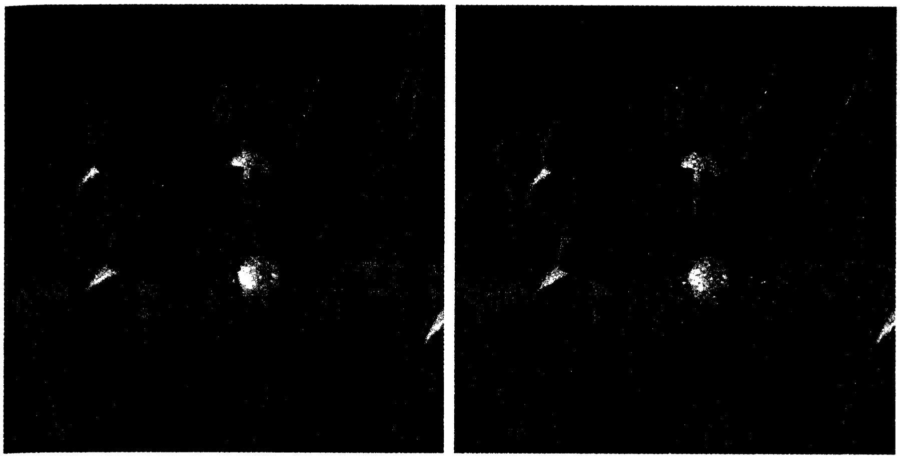


FIGURE 6.22 (a) Blurred image. (b) Image enhanced using the Laplacian, followed by contrast enhancement using function `ice`.

EXAMPLE 6.9: Color image sharpening.

■ Figure 6.22(a) shows a slightly blurred version, `fb`, of the image in Fig. 6.19(a), obtained using a 5 × 5 averaging filter. To sharpen this image we used the Laplacian filter mask

```
>> lapmask = [1 1 1; 1 -8 1; 1 1 1];
```

Then, as in Example 3.9, the enhanced image was computed and displayed using the commands

```
>> fen = imsubtract(fb, imfilter(fb, lapmask, 'replicate'));
>> imshow(fen)
```

where we combined the two required steps into a single command. As in the previous section, RGB images were treated exactly as monochrome images (i.e., with the same calling syntax) when using `imfilter`. Figure 6.22(b) shows the result. Note the significant increase in sharpness of features such as the water droplets, the veins in the leaves, the yellow centers of the flowers, and the green vegetation in the foreground. ■

6.6 Working Directly in RGB Vector Space

As mentioned in Section 6.3, there are cases in which processes based on individual color planes are not equivalent to working directly in RGB vector space. This is demonstrated in this section, where we illustrate vector processing by considering two important applications in color image processing: color edge detection and region segmentation.

6.6.1 Color Edge Detection Using the Gradient

The gradient of a 2-D function, $f(x, y)$, is defined as the vector

$$\nabla \mathbf{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}$$

The magnitude of this vector is

$$\begin{aligned} \nabla f &= \text{mag}(\nabla \mathbf{f}) = [G_x^2 + G_y^2]^{1/2} \\ &= [(\partial f/\partial x)^2 + (\partial f/\partial y)^2]^{1/2} \end{aligned}$$

Often, this quantity is approximated by absolute values:

$$\nabla f \approx |G_x| + |G_y|$$

This approximation avoids the square and square root computations, but still behaves as a derivative (i.e., it is zero in constant areas, and has a magnitude proportional to the degree of change in areas whose pixel values are variable). It is common practice to refer to the magnitude of the gradient simply as "the gradient."

A fundamental property of the gradient vector is that it points in the direction of the maximum rate of change of f at coordinates (x, y). The angle at which this maximum rate of change occurs is

$$\alpha(x, y) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

It is customary to approximate the derivatives by differences of pixel values over small neighborhoods in an image. Figure 6.23(a) shows a neighborhood of size 3×3, where the z's indicate pixel values. An approximation of the partial derivatives in the x (vertical) direction at the center point of the region (i.e., z_5) is given by the difference

$$G_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

FIGURE 6.23 (a) A small neighborhood. (b) and (c) Sobel masks used to compute the gradient in the x (vertical) and y (horizontal) directions, respectively, with respect to the center point of the neighborhood.

Similarly, the derivative in the y direction is approximated by the difference

$$G_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

These two quantities are easily computed at all points in an image by convolving (using `imfilter`) the image separately with the two masks shown in Figs. 6.23(b) and (c), respectively. Then, an approximation of the corresponding gradient image is obtained by summing the absolute value of the two filtered images. The masks just discussed are the Sobel masks mentioned in Table 3.4, which can be generated using function `fspecial`.

The gradient computed in the manner just described is one of the most frequently-used methods for edge detection in gray-scale images, as discussed in more detail in Chapter 10. Our interest at the moment is in computing the gradient in RGB color space. However, the method just derived is applicable in 2-D space but does not extend to higher dimensions. The only way to apply it to RGB images would be to compute the gradient of each component color image and then combine the results. Unfortunately, as we show later in this section, this is not the same as computing edges in RGB vector space directly.

The problem, then, is to define the gradient (magnitude and direction) of the vector **c** defined in Section 6.3. The following is one of the various ways in which the concept of a gradient can be extended to vector functions. Recall that for a scalar function, $f(x, y)$, the gradient is a vector pointing in the direction of maximum rate of change of f at coordinates (x, y).

Let **r**, **g**, and **b** be unit vectors along the R, G, and B axis of RGB color space (see Fig. 6.2), and define the vectors

$$\mathbf{u} = \frac{\partial R}{\partial x}\mathbf{r} + \frac{\partial G}{\partial x}\mathbf{g} + \frac{\partial B}{\partial x}\mathbf{b}$$

and

$$\mathbf{v} = \frac{\partial R}{\partial y}\mathbf{r} + \frac{\partial G}{\partial y}\mathbf{g} + \frac{\partial B}{\partial y}\mathbf{b}$$

Let the quantities g_{xx}, g_{yy}, and g_{xy} be defined in terms of the dot product of these vectors, as follows:

$$g_{xx} = \mathbf{u}\cdot\mathbf{u} = \mathbf{u}^T\mathbf{u} = \left|\frac{\partial R}{\partial x}\right|^2 + \left|\frac{\partial G}{\partial x}\right|^2 + \left|\frac{\partial B}{\partial x}\right|^2$$

$$g_{yy} = \mathbf{v}\cdot\mathbf{v} = \mathbf{v}^T\mathbf{v} = \left|\frac{\partial R}{\partial y}\right|^2 + \left|\frac{\partial G}{\partial y}\right|^2 + \left|\frac{\partial B}{\partial y}\right|^2$$

and

$$g_{xy} = \mathbf{u}\cdot\mathbf{v} = \mathbf{u}^T\mathbf{v} = \frac{\partial R}{\partial x}\frac{\partial R}{\partial y} + \frac{\partial G}{\partial x}\frac{\partial G}{\partial y} + \frac{\partial B}{\partial x}\frac{\partial B}{\partial y}$$

Keep in mind that R, G, and B, and consequently the g's, are functions of x and y. Using this notation, it can be shown (Di Zenzo [1986]) that the direction of maximum rate of change of $\mathbf{c}(x, y)$ as a function (x, y) is given by the angle

$$\theta(x, y) = \frac{1}{2}\tan^{-1}\left[\frac{2g_{xy}}{(g_{xx} - g_{yy})}\right]$$

and that the value of the rate of change (i.e., the magnitude of the gradient) in the directions given by the elements of $\theta(x, y)$ is given by

$$F_\theta(x, y) = \left\{\frac{1}{2}[(g_{xx} + g_{yy}) + (g_{xx} - g_{yy})\cos 2\theta + 2g_{xy}\sin 2\theta]\right\}^{1/2}$$

Note that $\theta(x, y)$ and $F_\theta(x, y)$ are images of the same size as the input image. The elements of $\theta(x, y)$ are simply the angles at each point that the gradient is calculated, and $F_\theta(x, y)$ is the gradient image.

Because $\tan(\alpha) = \tan(\alpha \pm \pi)$, if θ_0 is a solution to the preceding $\tan^{-1}$ equation, so is $\theta_0 \pm \pi/2$. Furthermore, $F_\theta(x, y) = F_{\theta+\pi}(x, y)$, so F needs to be computed only for values of θ in the half-open interval $[0, \pi)$. The fact that the $\tan^{-1}$ equation provides two values 90° apart means that this equation associates with each point (x, y) a pair of orthogonal directions. Along one of those directions F is maximum, and it is minimum along the other, so the final result is generated by selecting the maximum at each point. The derivation of these results is rather lengthy, and we would gain little in terms of the fundamental objective of our current discussion by detailing it here. The interested reader should consult the paper by Di Zenzo [1986] for details. The partial derivatives required for implementing the preceding equations can be computed using, for example, the Sobel operators discussed earlier in this section.

The following function implements the color gradient for RGB images (see Appendix C for the code):

colorgrad

```
[VG, A, PPG] = colorgrad(f, T)
```

where `f` is an RGB image, `T` is an optional threshold in the range [0, 1] (the default is 0); `VG` is the RGB vector gradient $F_\theta(x, y)$; `A` is the angle image $\theta(x, y)$, in radians; and `PPG` is the gradient formed by summing the 2-D gradients of the individual color planes (generated for comparison purposes). These latter gradients are $\nabla R(x, y)$, $\nabla G(x, y)$, and $\nabla B(x, y)$, where the ∇ operator is as defined earlier in this section. All the derivatives required to implement the preceding equations are implemented in function `colorgrad` using Sobel operators. The outputs `VG` and `PPG` are normalized to the range [0, 1] by `colorgrad` and they are thresholded so that `VG(x, y) = 0` for values less than or equal to `T` and `VG(x, y) = VG(x, y)` otherwise. Similar comments apply to `PPG`.

EXAMPLE 6.10: RGB edge detection using function `colorgrad`.

■ Figures 6.24(a) through (c) show three simple monochrome images which, when used as RGB planes, produced the color image in Fig. 6.24(d). The objectives of this example are (1) to illustrate the use of function `colorgrad`, and (2) to show that computing the gradient of a color image by combining the gradients of its individual color planes is quite different from computing the gradient directly in RGB vector space using the method just explained.

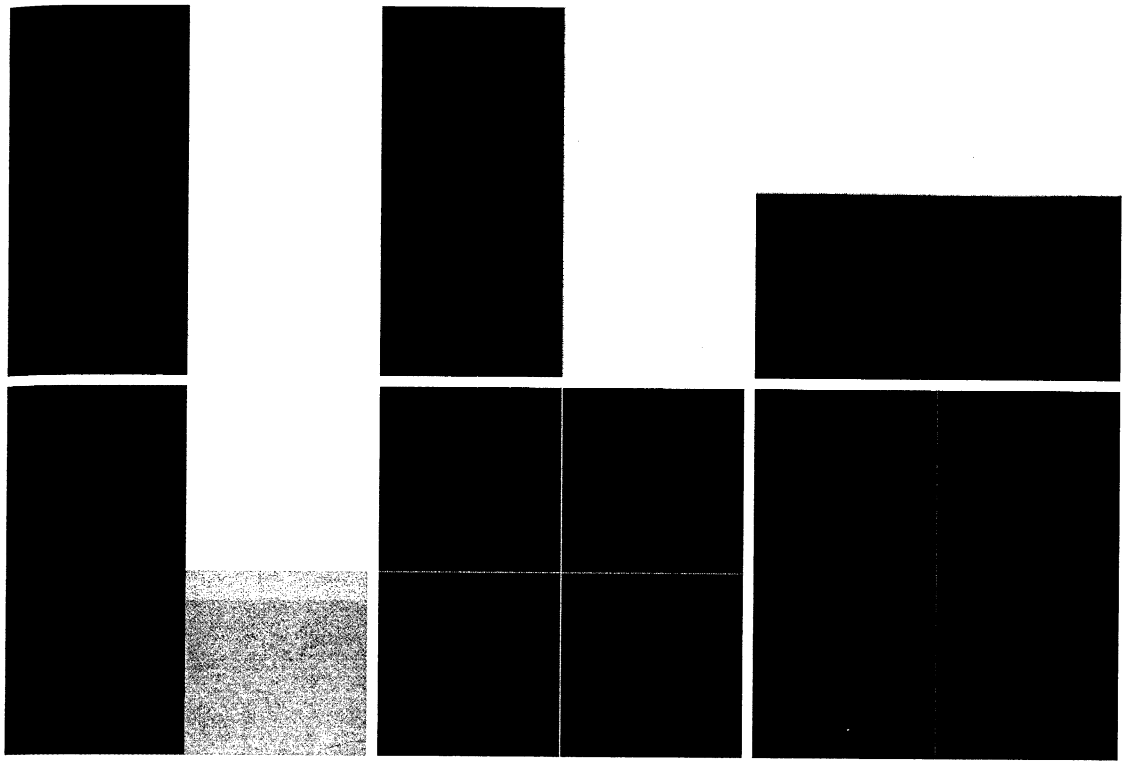


FIGURE 6.24 (a) through (c) RGB component images (black is 0 and white is 255). (d) Corresponding color image. (e) Gradient computed directly in RGB vector space. (f) Composite gradient obtained by computing the 2-D gradient of each RGB component image separately and adding the results.

Letting f represent the RGB image in Fig. 6.24(d), the command

```
>> [VG, A, PPG] = colorgrad(f);
```

produced the images VG and PPG shown in Figs. 6.24(e) and (f). The most important difference between these two results is how much weaker the horizontal edge in Fig. 6.24(f) is than the corresponding edge in Fig. 6.24(e). The reason is simple: The gradients of the red and green planes [Figs. 6.24(a) and (b)] produce two vertical edges, while the gradient of the blue plane yields a single horizontal edge. Adding these three gradients to form PPG produces a vertical edge with twice the intensity as the horizontal edge.

On the other hand, when the gradient of the color image is computed directly in vector space [Fig. 6.24(e)], the ratio of the values of the vertical and horizontal edges is $\sqrt{2}$ instead of 2. The reason again is simple: With reference to the color cube in Fig. 6.2(a) and the image in Fig. 6.24(d) we see that the vertical edge in the color image is between a blue and white square and a black and yellow square. The distance between these colors in the color cube is $\sqrt{2}$, but the distance between black and blue and yellow and white (the horizontal edge) is only 1. Thus the ratio of the vertical to the horizontal differences is $\sqrt{2}$. If edge accuracy is an issue, and especially when a threshold is used, then the difference between these two approaches can be significant. For example, if we had used a threshold of 0.6, the horizontal line in Fig. 6.24(f) would have disappeared.

In practice, when interest is mostly on edge detection with no regard for accuracy, the two approaches just discussed generally yield comparable results. For example, Figs. 6.25(b) and (c) are analogous to Figs. 6.24(e) and (f). They were obtained by applying function `colorgrad` to the image in Fig. 6.25(a). Figure 6.25(d) is the difference of the two gradient images, scaled to the range [0, 1]. The maximum absolute difference between the two images is 0.2, which translates to 51 gray levels on the familiar 8-bit range [0, 255]. However, these two gradient images are quite close in visual appearance, with Fig. 6.25(b) being slightly brighter in some places (for reasons similar to those explained in the previous paragraph). Thus, for this type of analysis, the simpler approach of computing the gradient of each individual component generally is acceptable. In other circumstances (as in the inspection of color differences in automated machine inspection of painted products), the more accurate vector approach may be necessary. ■

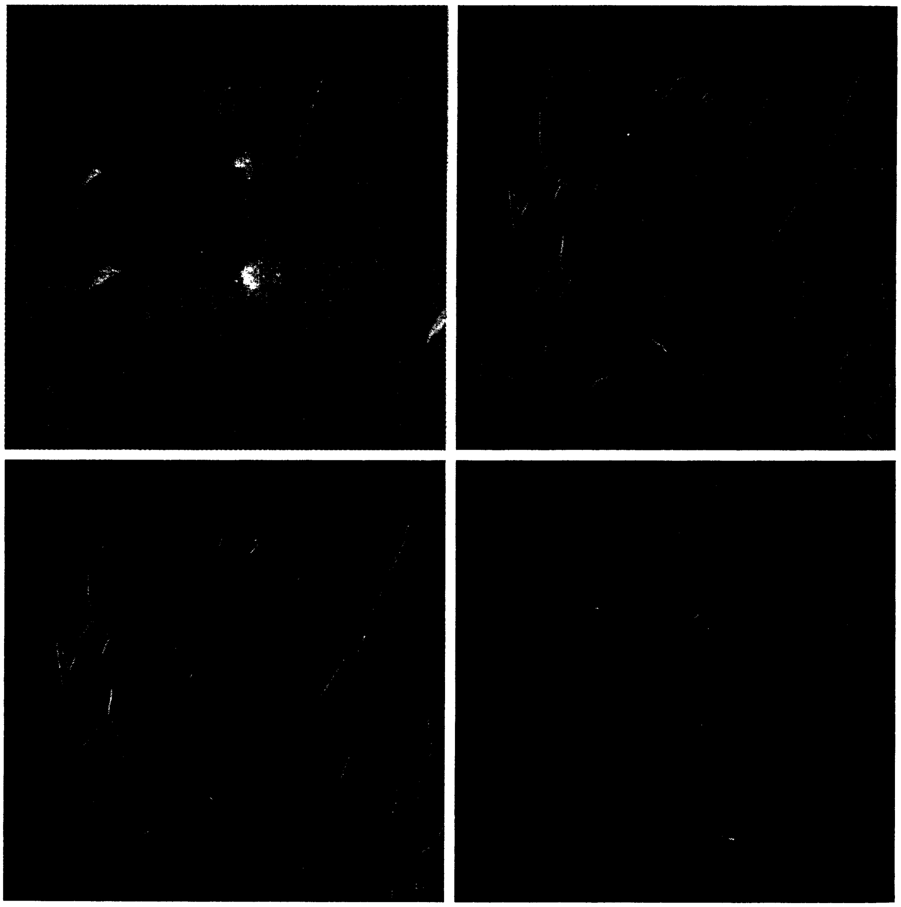


FIGURE 6.25 (a) RGB image. (b) Gradient computed in RGB vector space. (c) Gradient computed as in Fig. 6.24(f). (d) Absolute difference between (b) and (c), scaled to the range [0, 1].

6.6.2 Image Segmentation in RGB Vector Space

Segmentation is a process that partitions an image into regions. Although segmentation is the topic of Chapter 10, we consider color region segmentation briefly here for the sake of continuity. The reader will have no difficulty following the discussion.

Color region segmentation using RGB color vectors is straightforward. Suppose that the objective is to segment objects of a specified color range in an RGB image. Given a set of sample color points representative of a color (or range of colors) of interest, we obtain an estimate of the "average" or "mean" color that we wish to segment. Let this average color be denoted by the RGB column vector **m**. The objective of segmentation is to classify each RGB pixel in an image as having a color in the specified range or not. To perform this comparison, we need a measure of similarity. One of the simplest measures is the Euclidean distance. Let **z** denote an arbitrary point in RGB space. We say that **z** is *similar* to **m** if the distance between them is less than a specified threshold, T. The Euclidean distance between **z** and **m** is given by

$$\begin{aligned} D(\mathbf{z},\mathbf{m}) &= \|\mathbf{z}-\mathbf{m}\| \\ &= \left[(\mathbf{z}-\mathbf{m})^T(\mathbf{z}-\mathbf{m})\right]^{1/2} \\ &= \left[(z_R-m_R)^2+(z_G-m_G)^2+(z_B-m_B)^2\right]^{1/2} \end{aligned}$$

where $\|\cdot\|$ is the norm of the argument, and the subscripts R, G, and B, denote the RGB components of vectors **m** and **z**. The locus of points such that $D(\mathbf{z},\mathbf{m}) \leq T$ is a solid sphere of radius T, as illustrated in Fig. 6.26(a). By definition, points contained within, or on the surface of, the sphere satisfy the specified color criterion; points outside the sphere do not. Coding these two sets of points in the image with, say, black and white, produces a binary, segmented image.

Following convention, we use a superscript, T, to indicate vector or matrix transposition and a normal, inline, T to denote a threshold value. Care should be exercised not to confuse these unrelated uses of the same variable.

A useful generalization of the preceding equation is a distance measure of the form

$$D(\mathbf{z},\mathbf{m}) = \left[(\mathbf{z}-\mathbf{m})^T\mathbf{C}^{-1}(\mathbf{z}-\mathbf{m})\right]^{1/2}$$

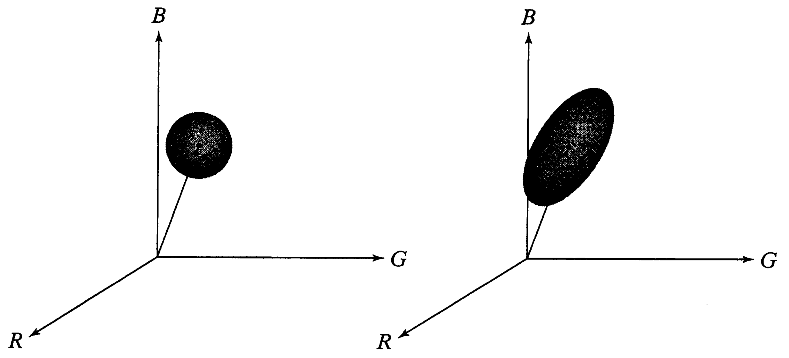


FIGURE 6.26 Two approaches for enclosing data in RGB vector space for the purpose of segmentation.

See Section 12.2 for a detailed discussion on efficient implementations for computing the Euclidean and Mahalanobis distances.

where **C** is the covariance matrix[†] of the samples representative of the color we wish to segment. This distance is commonly referred to as the *Mahalanobis distance*. The locus of points such that $D(\mathbf{z}, \mathbf{m}) \leq T$ describes a solid 3-D elliptical body [see Fig. 6.26(b)] with the important property that its principal axes are oriented in the direction of maximum data spread. When $\mathbf{C} = \mathbf{I}$, the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. Segmentation is as described in the preceding paragraph, except that the data are now enclosed by an ellipsoid instead of a sphere.

Segmentation in the manner just described is implemented by function `colorseg` (see Appendix C for the code), which has the syntax

`colorseg`

```
S = colorseg(method, f, T, parameters)
```

where `method` is either `'euclidean'` or `'mahalanobis'`, `f` is the RGB image to be segmented, and `T` is the threshold described above. The input parameters are either `m` if `'euclidean'` is chosen, or `m` and `C` if `'mahalanobis'` is chosen. Parameter `m` is the vector, **m**, described above, in either a row or column format, and `C` is the 3×3 covariance matrix, **C**. The output, `S`, is a two-level image (of the same size as the original) containing 0s in the points failing the threshold test, and 1s in the locations that passed the test. The 1s indicate the regions segmented from `f` based on color content.

EXAMPLE 6.11: RGB color image segmentation.

■ Figure 6.27(a) shows a pseudocolor image of a region on the surface of the Jupiter Moon Io. In this image, the reddish colors depict materials newly ejected from an active volcano, and the surrounding yellow materials are older sulfur deposits. This example illustrates segmentation of the reddish region using both options in function `colorseg`.

First we obtain samples representing the range of colors to be segmented. One simple way to obtain such a region of interest (ROI) is to use function `roipoly` described in Section 5.2.4, which produces a binary mask of a region selected interactively. Thus, letting `f` denote the color image in Fig. 6.27(a), the region in Fig. 6.27(b) was obtained using the commands

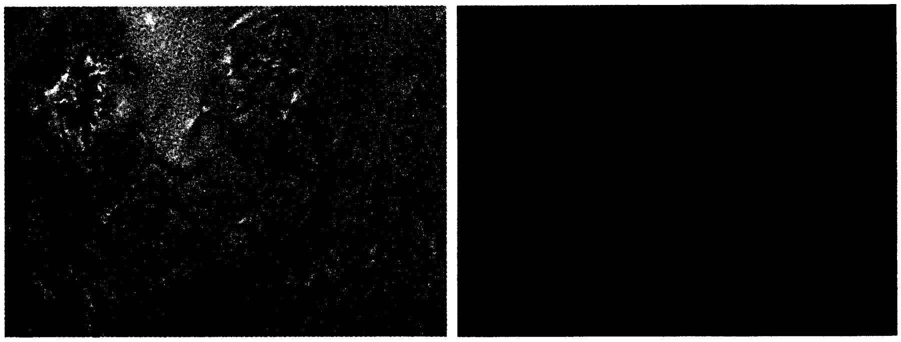


FIGURE 6.27 (a) Pseudocolor of the surface of Jupiter's Moon Io. (b) Region of interest extracted interactively using function `roipoly`. (Original image courtesy of NASA.)

[†]Computation of the covariance matrix of a set of vector samples is discussed in Section 11.5.

```
>> mask = roipoly(f);          % Select region interactively.
>> red = immultiply(mask, f(:, :, 1));
>> green = immultiply(mask, f(:, :, 2));
>> blue = immultiply(mask, f(:, :, 3));
>> g = cat(3, red, green, blue);
>> figure, imshow(g)
```

where mask is a binary image (the same size as f) with 0s in the background and 1s in the region selected interactively.

Next, we compute the mean vector and covariance matrix of the points in the ROI, but first the coordinates of the points in the ROI must be extracted.

```
>> [M, N, K] = size(g);
>> I = reshape(g, M * N, 3); % reshape is discussed in Sec. 8.2.2.
>> idx = find(mask);
>> I = double(I(idx, 1:3));
>> [C, m] = covmatrix(I); % See Sec. 11.5 for details on covmatrix.
```

The second statement rearranges the color pixels in g as rows of I, and the third statement finds the row indices of the color pixels that are not black. These are the non-background pixels of the masked image in Fig. 6.27(b).

The final preliminary computation is to determine a value for T. A good starting point is to let T be a multiple of the standard deviation of one of the color components. The main diagonal of C contains the variances of the RGB components, so all we have to do is extract these elements and compute their square roots:

```
>> d = diag(C);
>> sd = sqrt(d)'

   22.0643   24.2442   16.1806
```

diag

d = diag(C) *returns in vector* d *the main diagonal of matrix* C.

The first element of sd is the standard deviation of the red component of the color pixels in the ROI, and similarly for the other two components.

We now proceed to segment the image using values of T equal to multiples of 25, which is an approximation to the largest standard deviation: $T = 25, 50, 75, 100$. For the 'euclidean' option with $T = 25$, we use

```
>> E25 = colorseg('euclidean', f, 25, m);
```

Figure 6.28(a) shows the result, and Figs. 6.28(b) through (d) show the segmentation results with $T = 50, 75, 100$. Similarly, Figs. 6.29(a) through (d) show the results obtained using the 'mahalanobis' option with the same sequence of threshold values.

Meaningful results [depending on what we consider as *red* in Fig. 6.27(a)] were obtained with the 'euclidean' option when $T = 25$ and 50, but $T = 75$ and 100 produced significant oversegmentation. On the other hand, the results with the 'mahalanobis' option make a more sensible transition

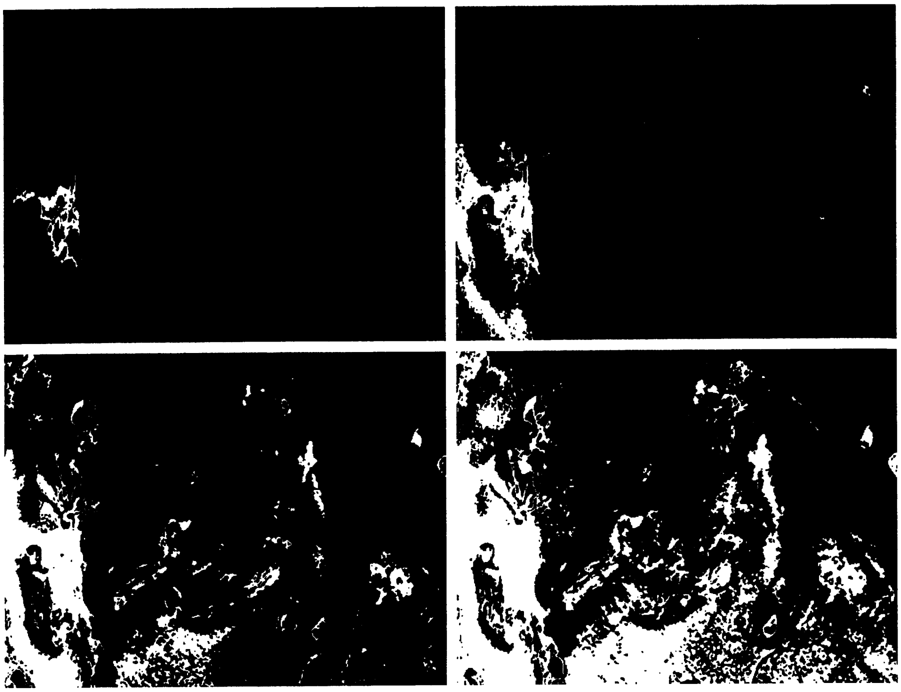


FIGURE 6.28 (a) through (d) Segmentation of Fig. 6.27(a) using option `'euclidean'` in function `colorseg` with $T = 25, 50, 75,$ and 100, respectively.

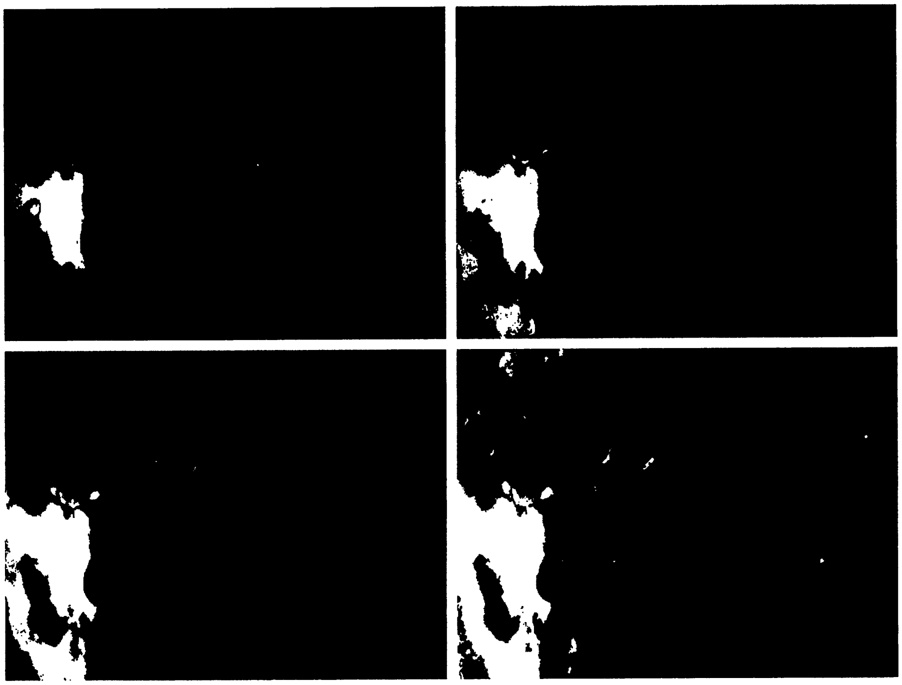


FIGURE 6.29 (a) through (d) Segmentation of Fig. 6.27(a) using option `'mahalanobis'` in function `colorseg` with $T = 25, 50, 75,$ and 100, respectively. Compare with Fig. 6.28.

for increasing values of T. The reason is that the 3-D color data spread in the ROI is fitted much better in this case with an ellipsoid than with a sphere. Note that in both methods increasing T allowed weaker shades of red to be included in the segmented regions, as expected. ■

Summary

The material in this chapter is an introduction to basic topics in the application and use of color in image processing, and on the implementation of these concepts using MATLAB, IPT, and the new functions developed in the preceding sections. The area of color models is broad enough so that entire books have been written on just this topic. The models discussed here were selected for their usefulness in image processing, and also because they provide a good foundation for further study in this area.

The material on pseudocolor and full-color processing on individual color planes provides a tie to the image processing techniques developed in the previous chapters for monochrome images. The material on color vector space is a departure from the methods discussed in those chapters, and highlights some important differences between gray-scale and full-color image processing. The techniques for color-vector processing discussed in the previous section are representative of vector-based processes that include median and other order filters, adaptive and morphological filters, image restoration, image compression, and many others.