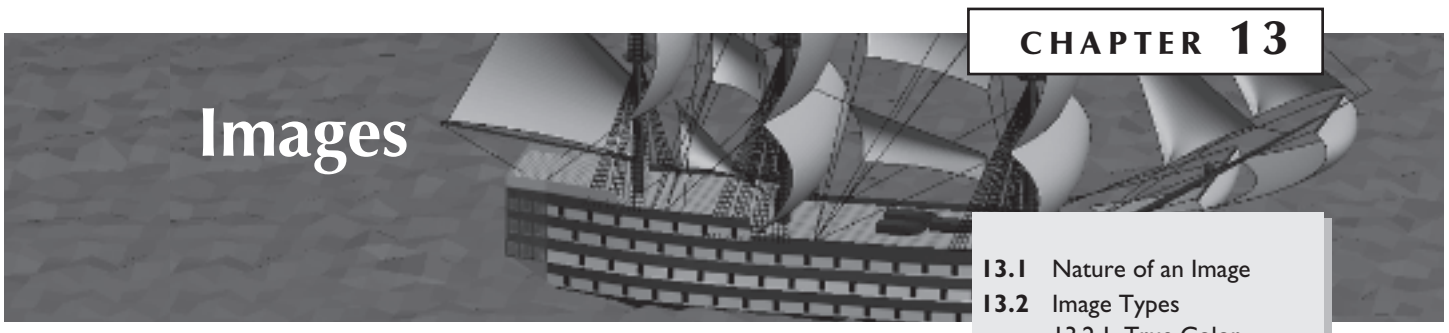
CHAPTER 13

Images



13.1 Nature of an Image
13.2 Image Types
- 13.2.1 True Color Images
- 13.2.2 Gray Scale Images
- 13.2.3 Color Mapped Images
- 13.2.4 Preferred Image Format

13.3 Reading, Displaying, and Writing Images
13.4 Operating on Images
- 13.4.1 Stretching or Shrinking Images
- 13.4.2 Color Masking
- 13.4.3 Creating a Kaleidoscope
- 13.4.4 Images on a Surface

13.5 Engineering Example—Detecting Edges

Chapter Objectives

This chapter covers:

- The basic representation of images
- How to read, display, and write JPEG image files
- Some basic operations on images
- Some advanced image processing techniques

Introduction

The graphical techniques we have seen so far have been 2-D and 3-D plots, whose basic concept is to write in places on the screen where data are required and to leave the rest of the screen blank. These presentations are easily generated when we have a mathematical model of the data and wish to represent it graphically. However, many sensors observing the world do not have that underlying model of the data. Rather, they passively generate 2-D representations that we see as images, leaving the interpretation of those images to a human observer. This kind of presentation is exemplified by a digital photograph but includes images from many other sources like radar or X-ray machines.

This chapter discusses some of the elementary processes that can be applied to images in order to begin to extract meaning from them.

13.1 Nature of an Image

Before we confine ourselves to practical, computational reality, we need to understand the general nature of an image. The easiest answer would be that an image is a 2-D sheet on which the color at any point can have essentially infinite variability. However, since we live in a digital world, we will immediately confine ourselves to the conventional representation of images required for most digital display processors, as shown in Figure 13.1. We can represent any image as a 2-D, M × N array of points usually referred to as picture elements, or **pixels,** where M and N are the number of rows and columns, respectively. Each pixel is "painted" by blending variable amounts of the three primary colors: red, green, and blue. (Notice that this is not the same blending process used in painting with oils or water colors, where the second primary color is yellow and the combination process is reversed—increasing amounts of the primary colors tends toward black, not white.)

The resolution of a picture is measured by the number of pixels per unit of picture width and height. This governs the fuzziness of its appearance in print, and controls the maximum size of good-quality photo printing. The color resolution is measured by the number of bits in the words containing the red, green, and blue (RGB) components. Since one value generally exists for each of the M × N pixels in the array, increasing the number of bits for each pixel color will have a significant effect on the stored size of the image. Typically, 8 bits (values 0–255) are assigned to each color.

The MATLAB language has a data type, `uint8`, which uses 8 bits to store an unsigned integer in the range 0–255. It is unsigned because we are not interested in negative color values, and to specify the sign value would cost a data bit and reduce the resolution of the data to 0–127. By combining the three color values, there are actually 2^{24} different combinations of color available to a true-color image—many more possible combinations than the human eye can distinguish.

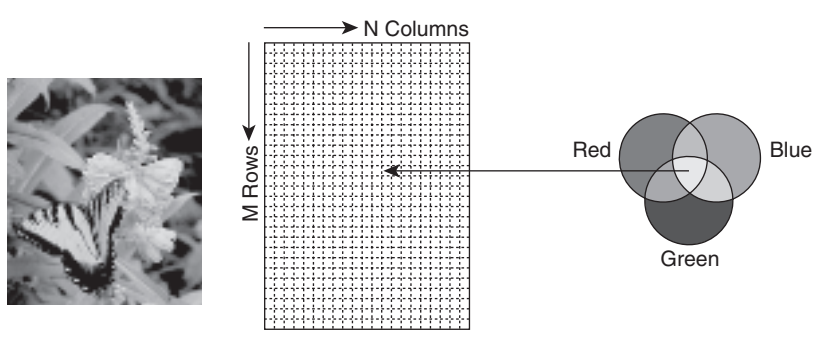


Figure 13.1 *The nature of images*

13.2 Image Types

Our sources for images to process are data files captured by imaging devices such as cameras, scanners, and graphic arts systems, and these image files are provided in a wide variety of formats. According to the MATLAB documentation, it recognizes files in TIFF, PNG, HDF, BMP, JPEG (JPG), GIF, PCX, XWD, CUR, and ICO formats. The various file formats are usually identified by their file extensions. While this seems a bewildering collection of formats, MATLAB provides one image reading function that converts these file formats to one of three internal representations: true color, gray scale, or color mapped images. In the MATLAB implementation, we will confine our interests to two formats: .png files when absolute color fidelity is required and .jpg files that offer better compression ratios to give a smaller file size for a given image.

13.2.1 True Color Images

True color images are stored according to the scheme shown in Figure 13.2 as an M × N × 3 array where every pixel is directly stored as `uint8` values in three layers of the 3-D array. The first layer contains the red value, the second layer the green value, and the third layer the blue value. The advantage of this approach, as the name suggests, is that every pixel can be represented as its true color value without compromise. The only disadvantage is the size of the image in memory because there are three color values for every pixel.

13.2.2 Gray Scale Images

Gray scale images are also directly stored, but save the black-to-white intensity value for each pixel as a single `uint8` value rather than three values.

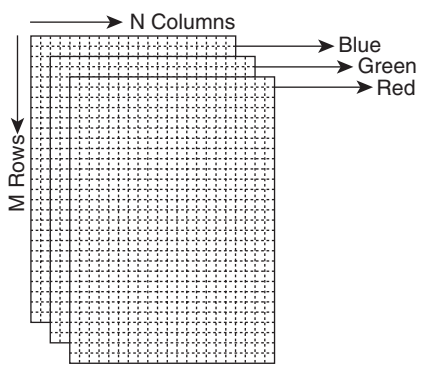


Figure 13.2 *A true color image*

13.2.3 Color Mapped Images

Color mapped, or indexed, images keep a separate color map either 256 items long (for maximum economy of memory) or up to 32,768 items long. Each item in the color map contains the red, blue, and green values of a color, respectively. As illustrated in Figure 13.3, the image itself is stored as an M × N array of indices into the color map. So, for example, a certain pixel index might contain the value 143. The color to be shown at that pixel location would be the 143rd color set (RGB) on the color map.

If the color map is restricted to 256 colors, each pixel can be drawn at the same color resolution as a true color image, as three 8-bit values, but the choice of colors is very restricted, and normal pictures of scenery—sky, for instance—take on a "layered color" appearance. Color mapped images can be used effectively, however, to store "cartoon pictures" economically where limited color choices are not a problem. Using a larger color map provides a larger, but still sometimes restrictive, range of color choices; but since the indices in the picture array must be 16-bit values and the color map is larger, the memory size advantages of this method of storage are diminished. Computationally, it is possible to convert a color mapped image to true color, but true color or black-and-white images cannot normally be converted to color mapped format without loss of fidelity in the color representation.

13.2.4 Preferred Image Format

In order to avoid confusion in the format of images, we will confine our discussions to one specific image file format that is prevalent at the time of writing and that provides a nice compromise between economy of storage as an image file and accessibility within MATLAB. We will discuss files compressed according to a standard algorithm originally proposed by the Joint Photographic Experts Group (JPEG). When MATLAB reads JPEG images, they are decoded as true color images; when MATLAB writes them, they are again encoded in compressed form. The file size for a typical JPEG file is 30 times less than the size you would need to store the M × N × 3

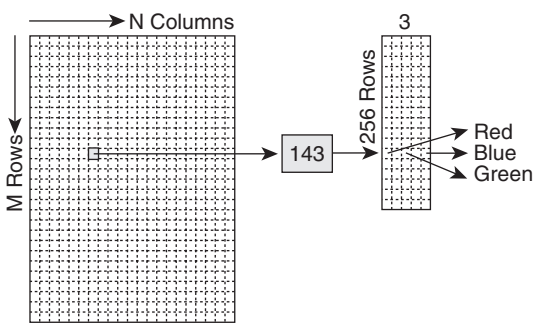


Figure 13.3 *A color mapped image*

bytes of the image. As we will see later, however, this compression does not come without cost.

13.3 Reading, Displaying, and Writing Images

MATLAB uses one image reading function, `imread(...)`, for all image file types. To read a file named `myPicture.jpg`, we use the following command:

```
>> pic = imread('myPicture.jpg', 'jpg')
```

where the result, `pic`, is an $M \times N \times 3$ `uint8` array of pixel color values, and the second parameter, `'jpg'`, provides the format of the file explicitly. This parameter is optional; MATLAB usually infers the file format correctly from the file contents.

Once the picture has been read, you can display it in a figure window with fixed size and axes visible by using the following command:

```
>> image(pic)
```

This actually stretches or shrinks the image to fit the size of the normal plot figure, a behavior you normally desire; however, occasionally, you want the plot figure to match the actual image size (or at least, preserving its aspect ratio). Releases of MATLAB after R20008a provide the `imshow(...)` function, which presents the image without stretching, shrinking, or axes (unless the figure window is too small).

Similarly, there is one function for writing files: `imwrite(...)`, which can be used to write most common file formats. If we have made some changes to `pic`, the internal representation of the image, we could write a new version to the disk by using the following:

```
>> imwrite( pic, 'newPicture.jpg', 'jpg')
```

where the third parameter, `'jpg'`, is required to specify the output format of the file.

13.4 Operating on Images

Since images are stored as arrays, it is not surprising that we can employ the normal operations of creation, manipulation, slicing, and concatenation. We will note one particular matrix operation that will be of great value before examining some applications of array manipulation related to image processing.

13.4.1 Stretching or Shrinking Images

In earlier chapters we have seen the basic ability to use index vectors to extract rows and columns from an array. Now we extend these ideas to

understand how to uniformly shrink or stretch an array to match an exact size. Consider, for example, `A`, a `rows` × `cols` array. Assume for a moment that the vertical size is good, but we want to stretch or shrink the image horizontally to `newRows`—a number that might be larger or smaller than `rows`. We use `linspace(...)` to create an index vector as follows:

```
>> rowVector = linspace(1, rows, newRows)
```

where the third parameter is the desired size of the new array. In general, this index vector will contain fractional values, but MATLAB will truncate the index values. We can round the results as follows:

```
>> rowVector = round(rowVector)
```

Then we can use this vector to shrink or stretch the picture `pic` as follows:

```
>> newPic = pic(rowVector, cols, :)
```

Clearly, this can be applied to both dimensions simultaneously, as shown in Exercise 13.1.

In this exercise, first we read an image and determine its size. Note that with 3-D images, you must give to the `size(...)` function three variables. Then we illustrate the "normal" slicing operations by reducing the image to the even rows, and every third column. Next, we generalize this image slicing by stretching the number of rows by a factor 1.43 and shrinking the number of columns by a factor 0.75. This is accomplished by building a row index vector, `rowVec`, and a column index vector, `colVec`, according to the algorithm above. The stretching is achieved by repeating selected values in the index vector, and shrinking is achieved by omitting some.

13.4.2 Color Masking

As an example of image manipulation, consider the image shown in Figure 13.4. This is a 2400 × 1600 JPEG image that can be taken with any good digital camera. However, the appearance of the Vienna garden is somewhat

Exercise 13.1 Working with image stretching

```
>> pic = imread(<your favorite image>);
>> [rows cols clrs] = size(pic)
>> imshow( pic(2:2:end, 3:3:end, :);
>> RFactor = 1.43; CFactor = 0.75; % shrink / stretch factors
>> rowVec = round(linspace(1, rows, Rfactor*rows));
>> colVec = round(linspace(1, cols, Cfactor*cols));
>> imshow(pic(rowVec, colVec,:)); % shrunk / stretched image
>> imshow(pic(:, :, [2 3 1])); % re-ordering the color layers
```

Figure 13.4 *A garden in Vienna*

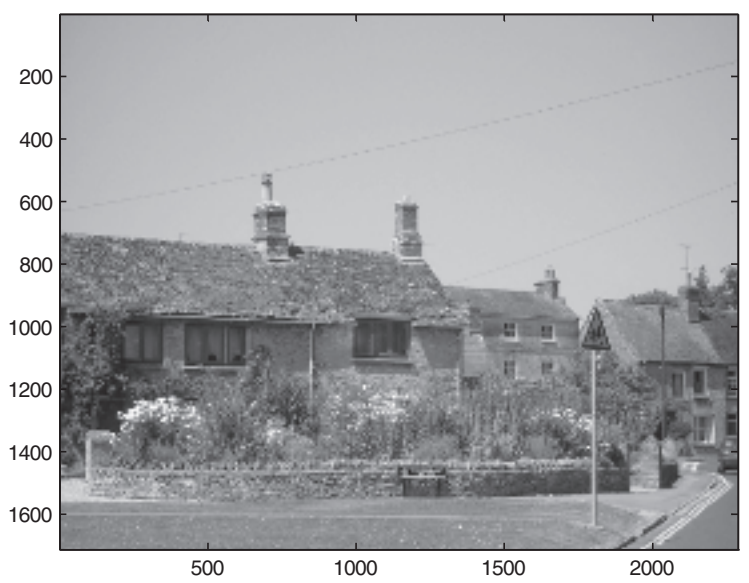


Figure 13.5 *A cottage in Oxfordshire*

marred by the fact that the sky is gray, not blue. Fortunately, we have a picture of a cottage, as shown in Figure 13.5, with a nice, clear blue sky. So our goal is to replace the gray sky in the Vienna garden with the blue sky from the cottage picture.

Initial Exploration Before we can do this, however, we need to explore the Vienna picture to determine how to distinguish the gray sky from the rest of the picture. In particular, there are patches of sky visible between the tree

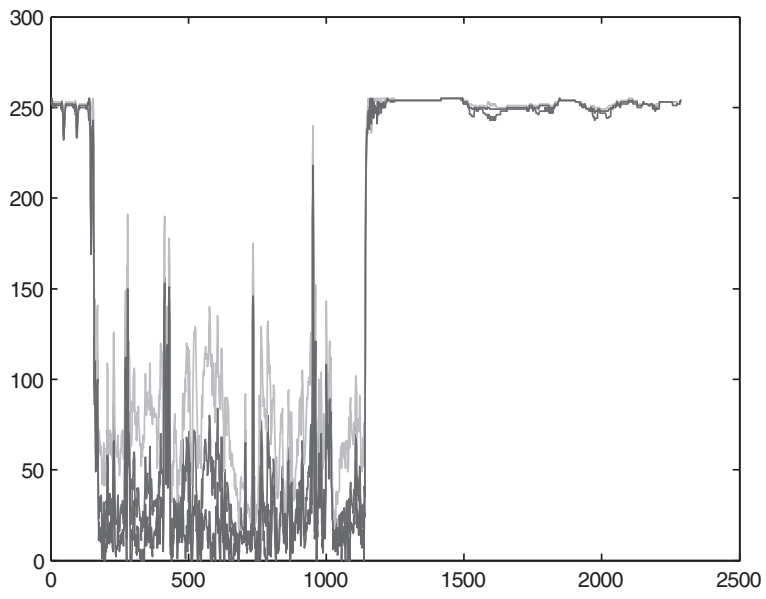


Figure 13.6 *Plot of the color values on one row of the Vienna image*

branches that must be changed as well as the open sky. Listing 13.1 illustrates a good way to accomplish this. Here we display the image in one figure; choose a representative row in the image that includes some sky showing through the tree (we chose row 350); and then plot the red, blue, and green values of the pixels across that row. Figure 13.6 shows the resulting plot.

In Listing 13.1:

Line 1: Reads the image.

Line 2: Displays the image.

Listing 13.1 Exploring the sky situation

```
1. v = imread('Vienna.jpg');
2. image(v)
3. figure
4. row = 400;
5. red = v(row, :, 1);
6. gr = v(row, :, 2);
7. bl = v(row, :, 3);
8. plot(red, 'r');
9. hold on
10. plot(gr,  'g');
11. plot(bl, 'b');
```

Line 3: Creates a new figure window for the next plots.
Line 4: Determines a suitable row (350 is a good choice).
Lines 5–7: Extract the three color layers for the chosen row.
Lines 8–11: Plot the three colors. Since we omitted one of the axis values, we make the assumption that the `x` values are the integers `1:length(y)`, which give us the horizontal pixel number across the row.

Analysis As we examine Figure 13.6, we see that the red, green, and blue values for the open sky are all around 250 because the sky is almost white. However, the color "spikes" that correspond to the color values of the sky elements that show through the tree are actually lower. We could decide, for example, to define the sky as all those pixels where the red, blue, and green values are all above a chosen threshold, and we could comfortably set that threshold at 160.

There is one more important consideration. It would be unfortunate to turn the hair of the lady (the author's wife) blue, and there are fountains and walkways that might also logically appear to be "sky." We can prevent this embarrassment by limiting the color replacement to the upper portion of the picture above row 700.

Final Computation So we are ready to create the code that will replace the gray sky with blue. The code in Listing 13.2 accomplishes this, and Figure 13.7 shows the resulting image.

Figure 13.7 *The Vienna garden with a blue sky*

Listing 13.2 Replacing the gray sky

```
 1. v = imread('Vienna.jpg');
 2. w = imread('Witney.jpg');
 3. image(w)
 4. figure
 5. thres = 160;
 6. layer = (v(:,:,1) > thres) ...
 7.       & (v(:,:,2) > thres) ...
 8.       & (v(:,:,3) > thres);
 9. mask(:,:,1) = layer;
10. mask(:,:,2) = layer;
11. mask(:,:,3) = layer;
12. mask(700:end,:,:) = false;
13. nv = v;
14. nv(mask) = w(mask);
15. image(nv);
16. imwrite(nv, 'newVienna.jpg', 'jpg')
```

In Listing 13.2:

Lines 1 and 2: Read the two images.

Line 3: Draws the cottage picture.

Line 4: Makes a new figure window.

Line 5: Sets the arbitrary threshold.

Lines 6–8: Define a 2-D layer containing logic that separates the Vienna sky from the rest of the picture.

Lines 9–11: Build a logical mask to replace the appropriate pixels from the cottage picture into the Vienna picture by populating each color layer of the mask with that layer.

Line 12: Refuses to replace any pixels below row 700.

Line 13: Copies the original image.

Line 14: Replaces the sky.

Line 15: Shows the image.

Line 16: Saves the JPEG result.

Post-operative Analysis We realize that this is not quite the end of the story, because a wire has suddenly become evident in the picture. Furthermore, if we take a close look at the wire (Figure 13.8), we see a number of disturbing things:

- The sky is by no means uniform in color—justifying the assertion that color mapped images do not have enough different colors to draw a true sky effectively
- The color of the wire is not far removed from the color of some parts of the blue sky—so replacing slightly darker blue would be problematic

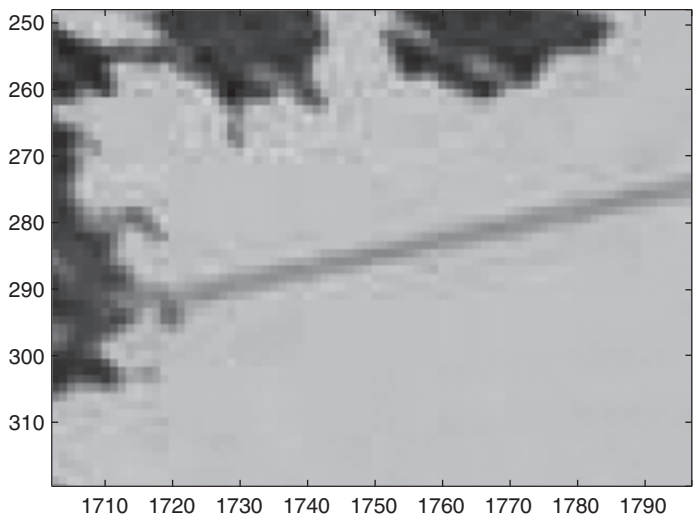


Figure 13.8 *Magnified image of the wire*

- There is a light colored "halo" around the wire that is actually a result of the original JPEG compression of the image so that even if we did replace the darker colors, the "ghost" of the wire would still be visible

Common Pitfalls 13.1

Be careful requesting the size of 3-D (and more) arrays. If you leave off variables—as here, you might be tempted not to ask for the number of colors because you know it's three—the `size(...)` function multiplies together the remaining dimension sizes. So if `img` is sized 1200 * 1600, `[r,c] = size(img)` would return `r = 1200` and `c = 4800`! If you provide to only one variable, it returns a vector of the sizes of each dimension of the array. So `v = size(img)` returns `[1200 1600 3]`.

So pixel replacement will probably not solve our wire problem. We will take a different approach to solve this problem in Chapter 15.

13.4.3 Creating a Kaleidoscope

Originally, a kaleidoscope was a cardboard tube in which a number of mirrors were arranged in such a manner that one image—usually, a collection of colored beads—was reflected to produce a symmetrical collection of images. We will replicate that general idea using MATLAB. Figure 13.9 illustrates the geometric manipulation necessary to create one particular kaleidoscope picture. We start with an arbitrary image and use shrinking or stretching to generate a square picture—the 'F' in the figure. We then mirror it horizontally and concatenate it horizontally with the original image. We then mirror these two images vertically and concatenate them vertically. Finally, we take that compound image and repeat the process to produce the 4×4 image on the right side.

Figure 13.10 shows the original image and the results. The overall logic flow of the solution matches that shown in Figure 13.9.

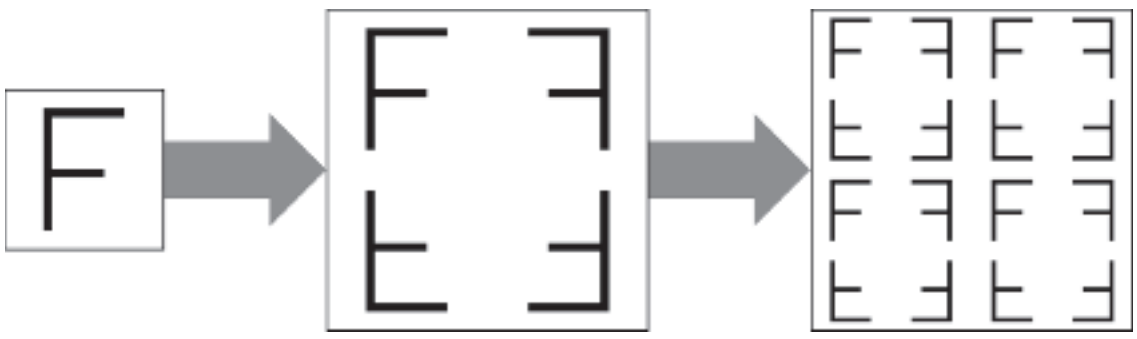


Figure 13.9 *Logic for the kaleidoscope*

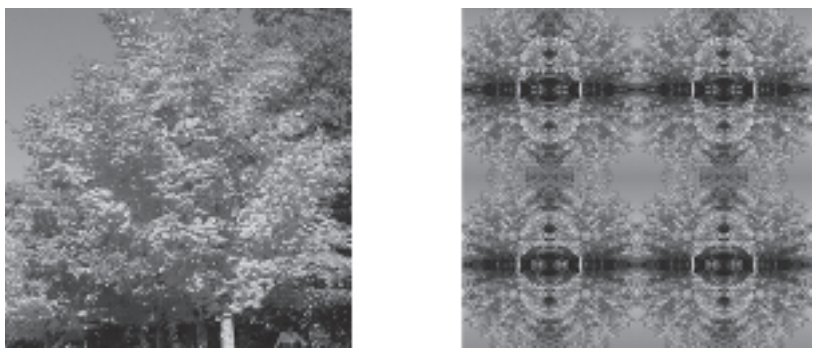


Figure 13.10 *The kaleidoscope*

Listing 13.3 shows the code that makes the kaleidoscope.

In Listing 13.3:

Line 2: Reads the original image.
Lines 2–3: Draw it on the left subplot.

Listing 13.3 Making a kaleidoscope

```
1. function kaleidoscope(name)
   % Making a kaleidoscope
   % usage: kaleidoscope(file_name)
      %read the image
2.    picture = imread(name);
3.    subplot(1,2,1); imshow(picture(ceil(1:1.5:end),:,:))
      % resize it to 128*128
4.    [rows cols ~] = size(picture);
5.    n = 128;
6.    rndx = ceil(linspace(1,rows, n));
7.    cndx = ceil(linspace(1,cols, n));
```

```
 8.     pic = picture(rndx, cndx, :);
        % build the kaleidoscope
 9.     img = buildIt(buildIt(pic));
10.     subplot(1,2,2); imshow(img)
11. end
12. function img = buildIt(img)
    % helper function to do the manipulations
    %          top left          top right
    %          bottom left       bottom right
13.     img = [img               img(:,end:-1:1,:)
14.            img(end:-1:1,:,:) img(end:-1:1,end:-1:1,:)];
15. end
```

Lines 4–8: Make it square.

Line 9: Calls the helper function to build the first set of 4, and then immediately call it again to build the 4×4 compound image.

Line 10: Draws it on the right panel.

Lines 12–15: Helper function to build four mirrored images from the original.

13.4.4 Images on a Surface

In Chapter 11 we saw how to create a surface representing solid objects and, in particular, how to create a spherical image that rotates with lighting.

Spectacular effects can be created by "pasting" images onto these surfaces, as will be illustrated in this last example. Here, we are given an image of the surface of the earth using Mercator projection, shown in Figure 13.11.[1] It is important to use the Mercator projection, named for the sixteenth-century Flemish cartographer Gerardus Mercator, because this projection keeps the lines of latitude and longitude on a rectangular grid. This allows a correct representation of the map as it is pasted onto the spherical surface. However, it also presents a challenge because in this projection, the north and south poles would be stretched to infinite length across the top and bottom of the map. This map, therefore, leaves off the region near the poles, and we have to replace those regions.

The objective of this exercise is to paste this image onto a rotating globe. The trick to accomplishing this is to use a feature of the `surf(...)` function, whereby the image is supplied in a specific form as the fourth parameter, as follows:

```
surf(xx, yy, zz, img)
```

[1]The file *earth_s.jpg* is provided as part of the MATLAB system.

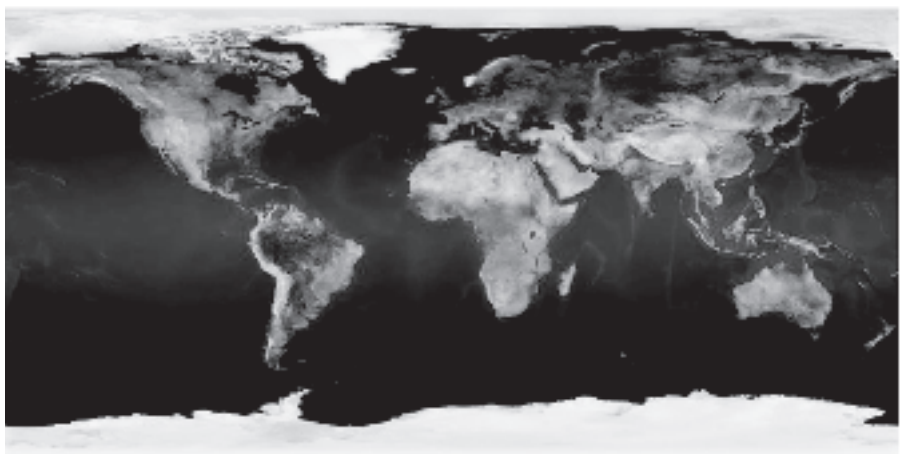


Figure 13.11 *Map projection*

It will replace the normal coloring scheme of the surface with the image under the following conditions:

- The rows and columns of the image match the rows and columns of the `xx`, `yy`, `zz` plaid
- The image supplies the red, green, and blue layers in the same form as true color images
- The color values, however, must be of type `double` in the range `0..1`

In the following code, rather than stretching the image to the size of the plaid, we choose to size the plaid to the image, thereby preserving all the image resolution. Clearly, in different circumstances where the size of the plaid is specified, the image can be stretched to suit those dimensions. The code to accomplish all this is shown in Listing 13.4.

In Listing 13.4:

Line 1: Reads the JPEG image.

Line 2: Enables good closure at the image edge by copying the first column of the map beyond the last column.

Line 3: Computes the mean image intensity of the snow on the top edge of the image. This will be used to fill the circles at the north and south poles.

Line 4: Fetches the size of the map.

Line 5: To calculate the size of the circles at the poles, we assume that the map takes us to ±85° of latitude, so we need the equivalent of 5° at the top and bottom of the map. This line calculates how many rows represent 1° of latitude.

Listing 13.4 Rotating a globe

```
 1. WM = imread('earthmap_s.jpg');
 2. WM(:,end+1,:) = WM(:,1,:);
 3. snow = mean( mean(WM(1,:,:)));
 4. [WMr, WMc, clr] = size(WM);
 5. rowsperdeglat = WMr/170
 6. add = floor(rowsperdeglat * 5)
 7. addlayer = uint8(ones(add, WMc) * snow);
 8. toAdd(:,:,1) = addlayer;
 9. toAdd(:,:,2) = addlayer;
10. toAdd(:,:,3) = addlayer;
11. worldMap = [toAdd; WM; toAdd];
12. [nlat nlong clr] = size(worldMap)
13. lat = double(0:nlat-1) * pi / nlat;
14. long = double(0:nlong-1) * 2 * pi / (nlong-1);
15. [th phi] = meshgrid(long, lat);
16. radius = 10;
17. zz = radius * cos(phi);
18. xx = radius * sin(phi) .* cos(th);
19. yy = radius * sin(phi) .* sin(th);
20. wM = double(worldMap) / 256;
21. surf(xx, yy, zz, wM);
22. shading interp
23. axis equal, axis off, axis tight
24. material dull
25. th = 0;
26. handle = light('Color',[int,int,int]); % a custom light source
27. while true
28.     th = th - 1;
29.     view([th 20]);
30.     lightangle(handle, th+50, 20)
31.     pause(.001)
32. end
```

Line 6: Shows the number of rows to add to the map.

Line 7: Computes the values of a single color layer by making an array with `ones(...)` using the number of rows to add and the number of map columns, and multiplying by the snow intensity.

Lines 8–10: Build the strips to add to the globe map by copying this layer to the red, green, and blue layers of a new image array.

Line 11: Prepares the complete map by concatenating this image to the top and bottom of the map.

Line 12: Retrieves the size of this map.

Lines 13 and 14: Prepare the vectors defining the plaid by spreading the map dimensions across π radians in latitude and 2π radians in longitude.

Lines 15–19: Prepare the sphere.

Figure 13.12 *Globe*

Line 20: Scales the image to double values between 0 and 1 as required by `surf(...)`.

Lines 21–23: Draw the surface as usual, using the image as the color distribution.

Lines 24–26: Special preparation of the surface luminosity and light characteristics to prevent glare spots.

Lines 27–32: The perpetual rotation with the angle `th` moving backward one degree at a time.

Line 30: This keeps the light in the same position relative to the observer.

Line 31: The usual `pause` to allow the drawing to take place for each iteration.

A snapshot of the globe as it is rotating is shown in Figure 13.12.

13.5 Engineering Example—Detecting Edges

While images are powerful methods of delivering information to the human eye, they have limitations when being used by computer programs. Our eyes have an astonishing ability to interpret the content of an image, such as the one shown in Figure 13.13. Even a novice observer would have no difficulty seeing that it is a picture of an aircraft in flight. An experienced observer would be able to identify the type of aircraft as a Lockheed C-130 and perhaps some other characteristics of the aircraft.

Figure 13.13 *C-130 in flight*

Figure 13.14 *Result of edge detection*

While our eyes are excellent at interpreting images, computer programs need a lot of help. One operation commonly performed to reduce the complexity of an image is edge detection, in which the complete image is replaced by a very small number of points that mark the edges of "interesting artifacts." Figure 13.14 shows the results from a simple program attempting to paint the outline of the aircraft in black by putting a black pixel at an identified edge. The key element of the algorithm is the ability to determine unambiguously whether a pixel is part of the object of interest or not. An edge is then defined as a pixel where some of the surrounding pixels are on the object and some are not. The image selected for this exercise makes

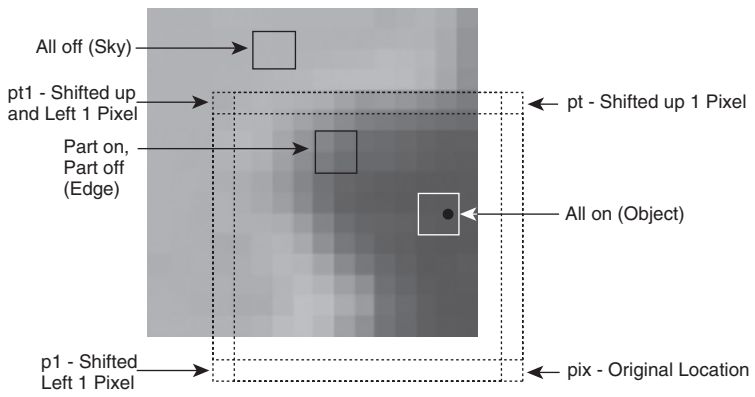


Figure 13.15 *Overlapping picture layers*

edge detection simple since the aircraft is everywhere darker than the surrounding sky.

The script used to generate this picture is shown in Listing 13.5. The basic approach of the algorithm is to use simple array processing tools to detect

Listing 13.5 Edge detection

```
pic = imread('C-130.jpg');
imshow(pic)
figure
[rows, cols, cl] = size(pic);
amps = uint16(pic(:,:,1))...
     + uint16(pic(:,:,2))...
     + uint16(pic(:,:,3));
up = max(max(amps))
dn = min(min(amps))
fact = .5
thresh = uint16(dn + fact * (up - dn))
pix = amps(2:end, 2:end);
ptl = amps(1:end-1, 1:end-1);
 pt = amps(1:end-1, 2:end);
 pl = amps(2:end, 1:end-1);
alloff= and(and((pix > thresh), ( pt  > thresh)),...
            and(( pl > thresh), (ptl  > thresh)));
allon = and(and((pix <= thresh), ( pt <= thresh)),...
            and(( pl <= thresh), (ptl <= thresh)));
edges = and(not(allon), not(alloff));
layer = uint8(ones(rows-1, cols-1) *255);
layer(edges) = 0;
outline(:,:,1) = layer;
outline(:,:,2) = layer;
outline(:,:,3) = layer;
image(outline)
imwrite(outline, 'c-130 edges.jpg', 'jpg')
```

the edges across the whole image at once. To accomplish this, we create four arrays, each one row and one column less than the original image and each offset by one pixel, as illustrated in Figure 13.15. The array `pix` is in the original location, `pt` is one row up from that location, `pl` is one row left, and `ptl` is one row left and up. If we now collapse these arrays on top of each other, we are simultaneously comparing the values of a square of four pixels across the whole image (less one row and one column).

In Listing 13.5:

Lines 1–4: Read the original image, display it, and determine its size.

Lines 5–7: Construct an array of size `rows` × `cols` containing the total color intensity of each pixel. The class `uint16`, using two bytes instead of one, is big enough for the sum of three `unit8`s.

Lines 8–11: Rather than guess an amplitude threshold, we compute a threshold halfway between the maximum and minimum intensities across the picture.

Lines 12–15: Set up the four overlapping arrays offset by a pixel each.

Lines 16–17: The logical array `alloff` will be true wherever all four adjacent pixels have an intensity above the threshold—these are on the sky.

Lines 18–19: The logical array `allon` will be true wherever all four adjacent pixels have an intensity below the threshold—these are on the aircraft.

Line 20: The pixels we are looking for are those where the pixel is neither completely sky nor completely aircraft.

Line 21: Makes a white image the same size as the logical arrays.

Line 22: Sets the edges to black.

Lines 23–27: Put that layer into the RGB layers, show the image, and write it to the disk.

Observation Clearly, while there is much more to be done with this data for it to be useful, the complexity of this image has been reduced from 12 million `uint8` values with no real meaning to a small number of data values that outline an object of interest. Algorithms beyond the scope of this text could be used to convert these outlining points to polynomial shapes. These shapes could then be matched against projections of 3-D models to actually identify the object in the picture.

Chapter Summary

This chapter covered the following:

- Images represented internally in bit-mapped, gray scale, or true color form

- Image files that come in a large variety of formats; MATLAB provides a single reader function and a single writer function to manipulate all the common image types
- Common operations on images, including cropping, stretching or shrinking, and concatenating and pasting an image onto a surface
- An engineering example showing how edge detection begins the process of extracting meaning from an image

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`image(<picture>)`	Displays an image in a figure of fixed dimensions with axes	13.3
`imread(<file_name>)`	Reads an image file	13.3
`imshow(<picture>)`	Displays an image in a figure of variable dimensions without axes	13.3
`imwrite(data, file, format)`	Writes an image file	13.3
`linspace(from, to, n)`	Defines a linearly spaced vector	13.2.1, 13.4.1
`rot90(A,n)`	Rotates `A` by 90° clockwise `n` times	13.4.4
`tril(A)`	Reduces `A` to its lower triangular half with zeros in the upper triangle	13.4.4
`uint8/16`	Unsigned integer type with the specified number of bits	13.1

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. An image whose color values are all 0 will be all white on the screen.
2. The MATLAB language defines one image reader for all image file types.
3. The normal operations of creation, slicing, and concatenation can be used to manipulate images.
4. `rot90(A)` rotates a 3-D array by 90° clockwise.

5. Edge detection dramatically reduces the amount of data to be processed by image identification software.

Fill in the Blanks

1. Each pixel of a true color image is stored as _____values of type _____ containing values _____.
2. Gray scale images store the black-to-white intensity value for each _____ as a(n) _____.
3. When you read JPEG files, they are _____as _____ images containing _____.
4. Once a picture has been read, you can display it in a(n) _____ with the function _____.
5. The operator _____ mirrors an array about its _____.

Programming Projects

1. As an introduction to image problems, perform the following manipulations:
   a. Find a suitable JPEG image file. Read it, display it, and store the result in `A`.
   b. Create a copy of `A`, flip the image from left to right, and display it in a new figure.
   c. Create a copy of `A`, swap the values for red and blue, and display it in a new figure.
   d. Create a copy of `A`, stretch the image to four times its original size (twice as many rows and twice as many columns), and display it in a new figure.
   e. Create a copy of `A` and then shrink the image to 0.7 its original size in each dimension and display it in a new figure.
2. An image could be scrambled by doing the following in order:
   a. multiple quadrant flips:
      - top left quadrant becomes bottom right quadrant
      - top right quadrant becomes bottom left quadrant
      - bottom right quadrant becomes top left quadrant
      - bottom left quadrant becomes top right quadrant
   b. The image is flipped upside down.
   c. The red values are swapped with the green values.
   d. The blue values are flipped left to right.

   Write a function called `imageScrambler` that takes in an image array and a string. If the string is equal to `'scramble'`, your function should scramble the image according to the above method and

return the modified image in array form. If the string is equal to `'unscramble'`, your function should unscramble the image by reversing the above method and return the modified array. Otherwise, your function should return the array untouched. You may assume that the image array provided will always contain an even number of rows and columns.

Test your solution by writing a script that reads a selected image, `A`, ensures that there is an even number of rows and columns, and tests the scrambling and unscrambling the image.

3. You are provided an image, and your job is to convert the full-sized image to a smaller one. Normally when image processing software is required to resize an image, a complex resizing algorithm is used to accomplish the conversion. We will attempt to duplicate this conversion. Write a function called `resizeMe` that takes in a string as an input corresponding to an image file name. The function should then resize the image to 1.414 times its original size in each dimension and display it. Additionally, your function should use the built-in function `imwrite(...)` to write the new image to a file. The name of the new file will be the original file name preceded by `'LG'`. For example, if the original filename is called `'yellow_bird.jpg'`, the new file should be called `'LGyellow_bird.jpg'`.

4. Write a function called `rotate` that takes in an image array and a number. The number represents the number of times the function will rotate the image clockwise by 90 degrees. A negative number signifies counter-clockwise rotation and a positive one signifies clockwise rotation.

5. We have obtained new intelligence that the Housing Department has plans to renovate all the rooms in the dorms with a new prototype. However, the prototype has been encoded into three separate images to avoid rival students finding out about it and thus seeking refuge here. Each image only contains one layer of color (e.g., `roomScrambledRed.jpg` only contains the Red layer). As a loyal student, it is your job to reconstruct a new image out of these three images.
   a. Create a script called `room`, and read the three layers using `'imread'`. Create the new matrix `ReconImage` with the three layers, and display it using `'imshow'`.
   b. After detailed analysis of the image, you find that it is also scrambled. Using advanced crytography and whizbang mathematical formulas, you have come to the conclusion that the four quadrants of the image have been re-arranged. Manipulate the composite image from part a. and re-arrange the pieces to

form the proper image. Display it using `subplot(...)`, below the first image.

6. For this exercise, you will visit—at least in MATLAB—a place you have always wanted to go.
   a. Find or take a picture of yourself with a plain background such as a green screen, using the JPEG image format. It would be a good idea not to wear the color of the background.
   b. Find a JPEG image of the place you want to go and decide on the rectangle in that scene where your image should appear. Save the width and height of the rectangle and the row and column of its top left corner.
   c. Re-size your image to be the width and height of the rectangle.
   d. Use the color masking technique of section 13.4.2 to copy your image without the green screen into the selected rectangle of your dream scene.

Hint

The trick to this is to move each pixel from its current location (in polar coordinates, $r - \theta$) to a new location on the new image. The new location is found by adding the rotation angle provided to the angular value, θ, of each pixel. Those pixels in the new image not occupied by a pixel will be black.

7. Write a function called `adjustImage` that consumes the name of an image file and an angle in degrees and produces a new image rotated counter-clockwise by that number of degrees about the center of the original image. Your new image will be larger than the original image.