Chapter 6
Create and Proccess Images[1]

MATLAB is a powerful tool for image processing. You can use MATLAB to create visual stimuli either by importing/exporting images or by drawing them from scratch in a quite simple manner. In this chapter, we give an introduction to image drawing and image manipulation.

Images Basics

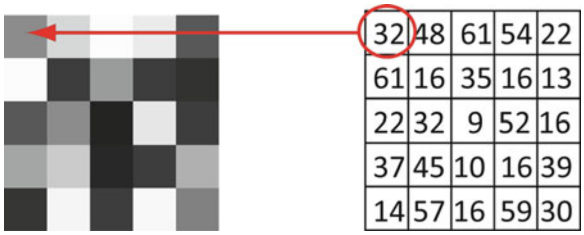
A digital image may be defined as a two-dimensional function *f(x,y)*. Here *x* and *y* are spatial coordinates and f(x,y) is the intensity of the image at the particular coordinates. For example, in a grayscale image, f(x,y) is the intensity of the gray at the particular x and y position.

Digital images differ from analog images (e.g., analog photos) because *x, y* coordinates and the *f(x,y)* intensity values are discrete instead of continuous. In digital images, a single (x,y) point is called a *pixel.* The intensity *f(x,y)* depends on the number of bits used to represent it. Usually the intensity is represented with 8 bits, yielding 2^8 values. As an example, a gray-scale image has intensity values within the 0–255 range, or in other words, the gray can assume 256 different levels from black (0) to white (255). Often, the range is normalized within the 0–1 range (i.e., the intensity values are divided by 255) (Fig. 6.1).

To get a color image, we need to superpose different colors, for example *Red*, *Green*, and *Blue* in the *RGB* system. In color images, each coordinate has n intensity values, one for each of color of the system in use. For example, each pixel of an RGB image is a triplet of intensity values, one for red, one for green, and one for blue. By default, MATLAB represents these triplets with 8 bits, for a total of 24 bits, yielding 2^{24} colors. This type of image is usually called *TrueColor*.

[1] Note that, although the book figures are black and white, the commands reported in the current chapter generate color figures.

DOI 10.1007/978-1-4614-2197-9_6, © Springer Science+Business Media, LLC 2012



32	48	61	54	22
61	16	35	16	13
22	32	9	52	16
37	45	10	16	39
14	57	16	59	30

Fig. 6.1 An example of a 5×5 gray-scale image. The first pixel in position (1, 1) has intensity equal to 32

There is also another method of treating color images: *indexing images*. Each pixel has a value that represents not a color but the index in a *color map* matrix. The *color map* (or *color palette*) is a list of all the colors used in that image. Such indexing images occupy less memory than RGB images, so they are a good option for saving space. The indexing concept is graphically illustrated in Fig. 6.2.[2] Each image has a zoomed area of 17×17 pixels. Each zoomed area shows the intensity for the gray-scale image and the RGB values for the color image. The indexed image is obtained from the RGB image using a palette with the 60 colors included in the RGB image.

MATLAB uses indexing images by default, and has a default color map from which it gets the colors for plotting figures such as histograms, pies charts, and 3-D graphs. Let's see the first five rows of the default color map by typing the following statement:

```
>> colormap('default');
DefColMap = colormap;
>> whos DefColMap
       Name          Size           Bytes  Class    Attributes
       DefColMap     64x3           1536   double
>> DefColMap(1:5,:)
ans =
       0             0      0.5625
       0             0      0.6250
       0             0      0.6875
       0             0      0.7500
       0             0      0.8125
```

The `colormap(cmap)` function can be used to set the color map to the matrix cmap. In our case, `colormap` sets the matrix color map to the default one. If we write the statement `colormap` as is, MATLAB returns the current color-map matrix. Here we saved the current color map into the `DefColMap` variable. As can be seen, the default color map has 64 colors (number of rows) and three columns: the first column corresponds to the color red, the second to green, and the third to blue. For example, in the first row of `DefColMap` there is 0% red, 0% green, and 56.25% blue (the values of the default color map are normalized), so the first five rows correspond to a variation of the blue color only.

[2] The image is copyrighted by the artist Mirta Caccaro.

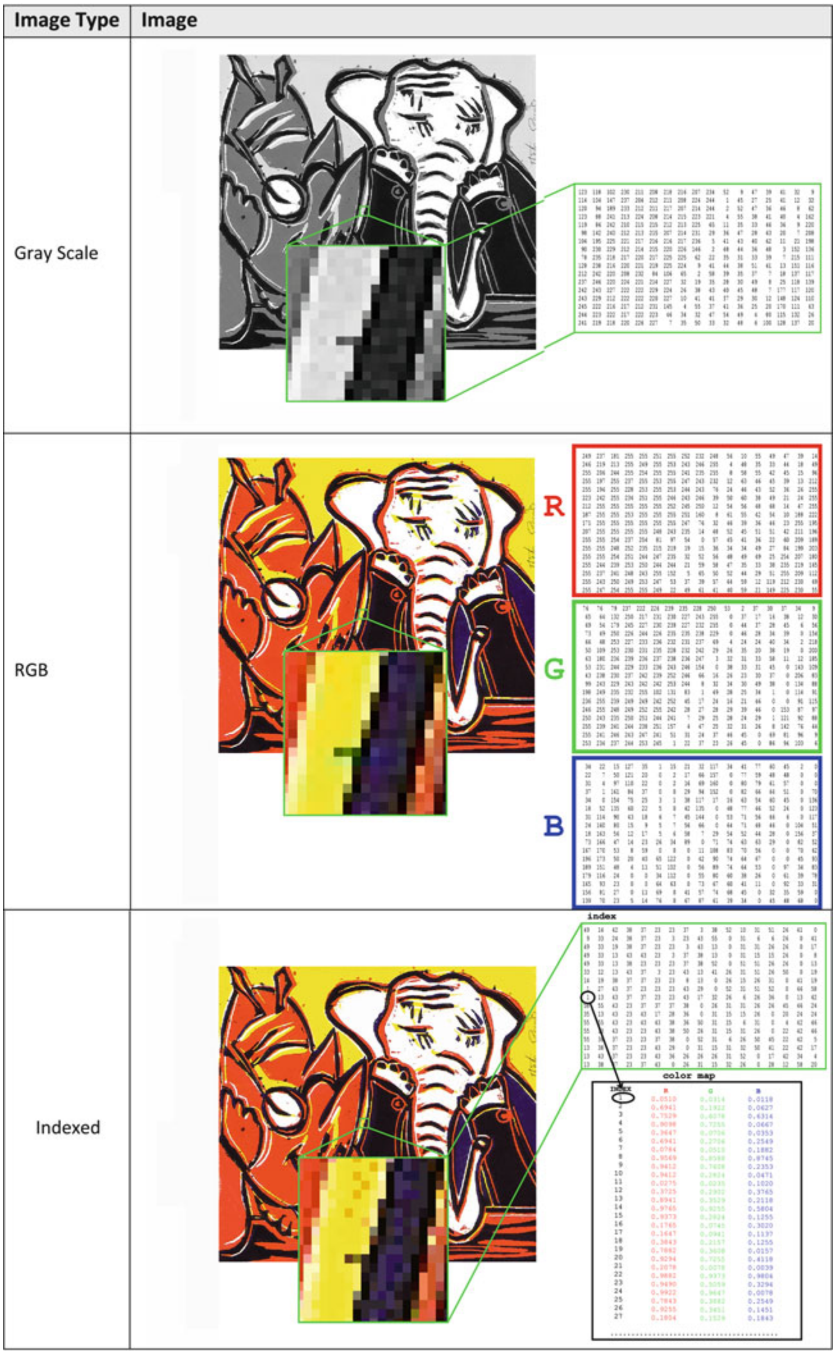
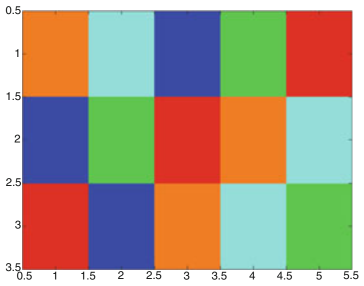


Fig. 6.2 Three different digital representations of the same image

Let’s acquire a better understanding of the color map by creating a custom color map. Let’s suppose we want to create a color map whose first color is red, the second is green, the third is blue, the fourth corresponds to a light azure, and the fifth corresponds to orange. Then we create a 3×5 indexed image with 3×5=15 pixels using the colors of the color map. In the following example we write code that does the job:

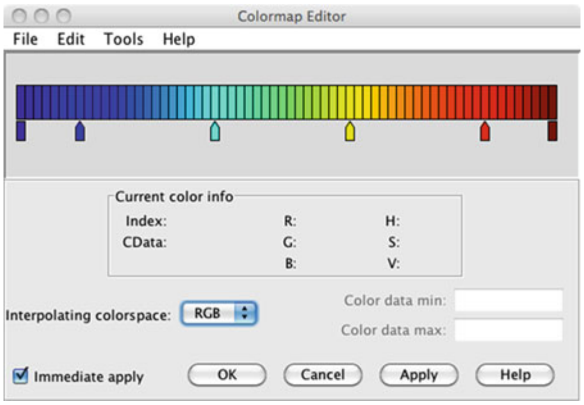
Example	Graphical result

`>> Mycolormap = [1 0 0; 0 1 0; 0 0 1;… 0 1 1; 1 0.5 0];`
`>> im = [5 4 3 2 1; 3 2 1 5 4; 1 3 5 4 2]`
`>> image(im)`
`>> colormap(Mycolormap)`
	

As you can see, the top leftmost pixel is orange, which corresponds to the index (row) 5 in the color map, or equivalently to the combination of 100% red, 50% green, and 0% blue.

You can change the color map just created using the command `colormapeditor;` it displays the current figure’s color map as a strip of rectangular cells in the color-map editor. Node pointers are colored cells below the color-map strip that indicate points in the color map where the rate of the variation of R, G, and B values changes. Please refer to MATLAB help for detailed information.

Example	Graphical result

`>> colormap('default')`
`>> colormapeditor;`
	 Colormap Editor
	File Edit Tools Help
	Current color info
	Index:
	CData:
	R:
	G:
	B:
	H:
	S:
	V:
	Interpolating colorspace: RGB
	Color data min:
	Color data max:
	Immediate apply
	OK
	Cancel
	Apply
	Help

Importing and Exporting Images

The function that enables one to load images into the MATLAB workspace is `imread`. `A=imread(filename, fmt)` reads a gray-scale or color image from the file specified by the string *filename* and stores the result in the matrix `A`. The text string `fmt` specifies the format of the file by its standard file extension. However, it is not necessary to write the extension if the filename already has the standard extension. If the image is an RGB image, the matrix `A` is a cube, i.e., an *Nrows* × *Ncolumns* × *3* matrix. For indexed images, the function `imread` returns the specific color map of the image.

If you need to write an image to a file, the function `imwrite.` `imwrite(A,filename,fmt)` writes the image `A` into a file with the specified filename and in the format specified by the string `fmt`. For indexed images, such as *gif* figures, for example, the function is `imwrite(X,map,filename,fmt)`. The function accepts other input parameters as well, such as the quality of *jpeg* images or the transparency matrix for *png* images. For further information, refer to the online MATLAB help.

An additional way to export your images is through the `print` function. The `print` function can be used as described in Chapter 3, once the image is displayed (see next section). Alternatively, select Copy Figure from the figure window's Edit menu. This action copies the image to the clipboard. Then you can paste the figure wherever you like.

MATLAB handles different image formats, the most common of which are presented in the following table:

File format	Extention	Description	Function use
TIFF	TIFF image	Color, gray-scale, or indexed image(s). The tiff format was originally created in the 1980s to support data output from scanners. This format can contain information about colorimetry calibration, etc.; examples occur with remote sensing	`Tim=imread(filename, 'tiff');` `[Tim, TColMap]=imread(filename,'tiff');` `imwrite(Tim,filename,'tiff');`
PNG	PNG image	True color, gray-scale, and indexed image(s). Very efficient lossless compression, supporting variable transparencies	`Pim=imread(filename,'png');` `[Pim, PColMap]=imread(filename,'png');` `imwrite(Pim,filename,'png');`
BMP	BMP image	True color or indexed image native format for Microsoft Windows. Can support up to 24-bit color. Originally uncompressed	`Bim=imread(filename,'bmp');` `[Bim, BColMap]=imread(filename,'bmp');` `imwrite(Bim,filename,'bpm');`

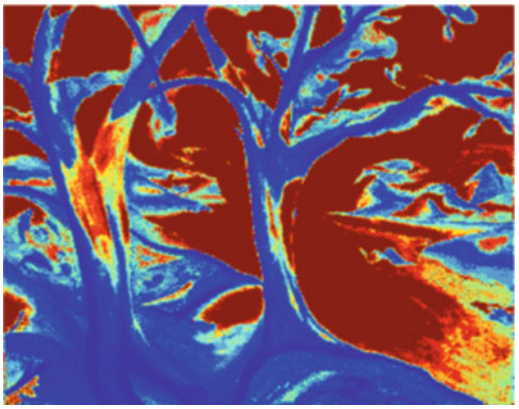
(continued)

(continued)

File format	Extention	Description	Function use
JPEG	JPEG image	True color or gray-scale image. 24-bit (true color) support. Created to support the photographic industry with various levels of compression. Compression can result in noticeable loss of image quality in some images	`Jim=imread(filename,'jpg');` `imwrite(Bim,filename,'jpg');`
GIF	GIF image	Indexed image, Very common and used extensively in the Internet. It works well for illustrations or clip-art that have large areas of flat colors. Limited to 256 colors	`[Gim, GColMap]=imread(filename,'gif');` `imwrite(Gim,filename,'jpg','Quality',75);`

Display Images

If you need to display an image, use the function `image`. Now let's try to load an image and display it as in the following example:

Example	Graphical result
`>> [Trees,mapTrees] = imread ('trees.tif');` `>> image(Trees);` `>> axis off;` `>> size(mapTrees)` `ans =` `256 3`	

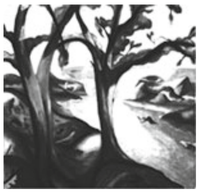
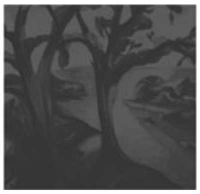
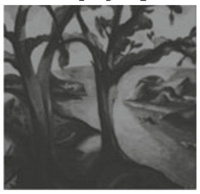
The image you see seems to have the wrong colors. The reason is the following. The image was indexed with its own custom color map. However, MATLAB does not load the image color map but uses the default color map instead. To show the correct image, you need to change the color map as follows:

```
>> colormap(mapTrees)
```

Note that `mapTrees` is the color-map matrix obtained using the command `imread`.

If you have the MATLAB image-processing toolbox, there is another function for displaying images: `imshow(X,map)`, where `X` is the image and `map` is the color map. Use only one argument in case of true-color or gray-scale images.

If you need to obtain the gray-scale version of the previous image, you need to change the color map. You can do it in either of two ways, by editing a new custom color map or by using the function `gray(M)`. The function `gray(M)` returns an M-by-3 matrix containing a linear gray-scale color map. Use `gray` as in the following example:

Example 1	Example 2	Example 3
`colormap(gray(100));` 	`colormap(gray(450));` 	`>> colormap(gray(256));` 

The effect of using a color map with fewer (or more) colors than those we are starting with (= 256 in this case) can give unexpected results. The color map can have up to a maximum of 256 entries (= rows). If we create a color map with only 100 rows, all the indices with values greater than 100 will not know which color refer to. MATLAB automatically sets all the indexes greater than 100 to point at the last row, i.e., the 100th row. This is why the image in Example 1 appears lighter. In contrast, Example 2 shows a darker image. This is because only the first 256 colors are used (which correspond to darker grays).

Note that for true color images, the image data will be read as a three-dimensional array. In such a case, `image` will ignore the current color map, and assign colors to the display based on the values in the array.

Basic Manipulation of Images

In MATLAB, images are treated as numbers embedded in matrices; therefore, they can be manipulated like any other array. Each intensity value is related to a pixel of the images and can be changed with a simple transformation. Such single-pixel transformations are generally called *point operations*. A different approach is to consider not only a single pixel but also a set of neighboring pixels. There is usually a strong correlation between the intensity values of a set of pixels that are close to each other. For example, we can change the gray level of a given pixel according to the values of the gray levels in a small neighborhood of pixels surrounding the given pixel. These transformations are called *neighborhood processing*. The current section shows some simple processing functions.

Point Operations

Intensity Transformation

Within the point operations, the intensity transformation is the simplest form of processing. Let's suppose we have an indexed gray-scale image. If the gray scale is linear, the index of a pixel is equivalent to its intensity. Such a value (the intensity) can be added/subtracted or multiplied/divided by a constant value. If we refer to indices, it is important to round the result (to obtain an integer where necessary) of the operation and to "clip" the values when they are greater than the maximum or lower than the minimum.

Let's load an image (The file `mandrill` contains the image `X` variable and the color-map `map` variable) and add a constant equal to 128 to each pixel's intensity value:

```
>> load mandrill
>> Y = X +128;
>> Y(9,1)
ans =
   270
```

as you can see, the pixel intensity value in position (9,1) is greater than 255 (= 2^8). In this case we need to "clip" the value and set it to 255. The operation can be done efficiently by selecting the minimum between the actual value and 255.

```
>> Y(9,1)=min( Y(9,1) , 255);
>> Y(9,1)
ans =
    255
```

We now use the function `floor` to round the result (if necessary) to obtain an integer. The `floor` function returns the greatest integer less than or equal to the input argument.

In the same way, if we need to be sure the values of an intensity matrix are greater than 0, we should type:

```
>> Y=floor(max(Y,0));
```

We show here the code to obtain the aforementioned intensity transformation with the MATLAB image called mandrill (Fig. 6.3).

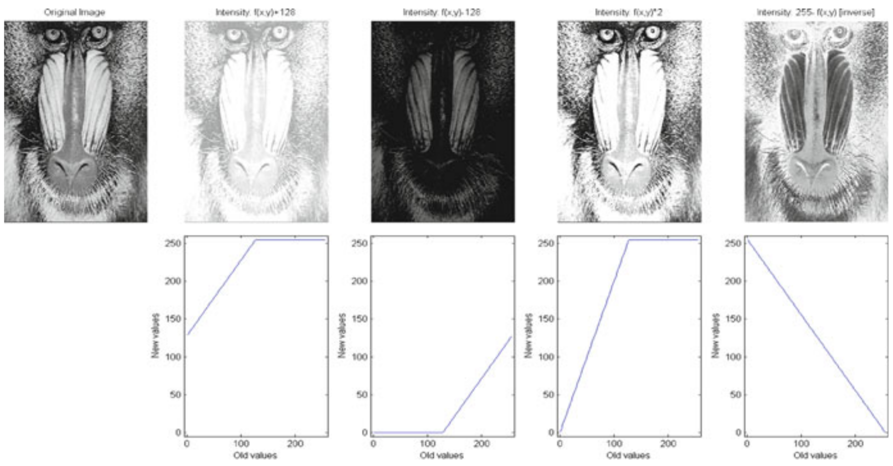


Fig. 6.3 Intensity variations applied to the same image

Listing 6.1

```
% Test the intensity transformation
% AUTHOR:   Borgo-Soranzo-Grassi  2012

load mandrill               % load a MATLAB image. X is the image.
                            % map is the colormap
ColMap=gray(255);           % Calculate a colormap of 255 values.
X1=floor(min(X+128,255)); % Increase the intensity and clip
X2=floor(max(X-128,0));    % Decrease the intensity and clip
X3=floor(min(X*2,255));    % Multiply the intensity by 2 and clip
X4=256-X;                   % invert the intensity

figure;
subplot(2,5,1); image(X); axis off;
subplot(2,5,2); image(X1); axis off;
subplot(2,5,3); image(X2); axis off;
subplot(2,5,4); image(X3); axis off;
subplot(2,5,5); image(X4); axis off;
colormap(ColMap);

Oldval=[1:255];
subplot(2,5,7)
plot(Oldval,floor(min(Oldval+128,255))); % Show the new Intensity value
axis([-5 260 -5 260]);                   % vs. the old ones
xlabel('Old values'); ylabel('New values');

Oldval=[1:255];
subplot(2,5,8)
plot(Oldval,floor(max(Oldval-128,0))); % Show the new Intensity value
axis([-5 260 -5 260]);                   % vs. the old ones
xlabel('Old values'); ylabel('New values');

Oldval=[1:255];
subplot(2,5,9)
plot(Oldval,floor(min(Oldval*2,255)));   % Show the new Intensity value
axis([-5 260 -5 260]);                   % vs. the old ones
xlabel('Old values'); ylabel('New values');

Oldval=[1:255];
subplot(2,5,10)
plot(Oldval,256-Oldval);                 % Show the new Intensity value
axis([-5 260 -5 260]);                   % vs. the old ones
xlabel('Old values'); ylabel('New values');
```

Note that we are providing an example with a gray-scale image. For color images, all the intensity matrices (e.g., the red, green, and blue matrix for RGB images) should be changed with the same function, or equivalently, by changing the color map. In any case, the most useful intensity transformations are brightening and contrasting. There are two built-in functions that operate within the color map and do these jobs: the `brighten` and the `contrast` functions. They are present in the following table.

Function	Description

`Brighten(beta)`	`Brighten increases or decreases the color intensities in the current color map. The modified color map is:`
	 `- brighter if 0 < beta < 1`
	 `- darker if -1 < beta < 0.`

`cmap=contrast(X)`	`The contrast function enhances the contrast of an image. It creates a new gray color map, cmap, that has an approximately an equal intensity distribution. All three elements in each row are identical`

The MATLAB Image Toolbox gives a simple graphical interface to explore, display, and perform common image-processing tasks. The Image Tool provides access to several other tools. For example, you can get information about single pixels and distances, and you can adjust the contrast of an image or crop a portion of it. Type `imtool(filename)` at the MATLAB prompt to use these tools. You can try out these tools with the image of the trees by typing `imtool('trees.tif')`.

Windowing

The concept of windowing is the multiplication of an image matrix by a matrix of the same size having values within the range from 0 to 1. The "window" can be thought of as another image. It is often used to smooth edges or to highlight certain parts of the image. Here we show an example:

Listing 6.2

```
% M-Script to test the windowing concept
%
% Authors: Borgo, Soranzo, Grassi
% Date: 2009

load mandrill
WCentx=size(X,1)/2;      % calculate the center of image, x axis
WCenty=size(X,2)/2;      % calculate the center of image, y axis
WSize=100;               % WSize is HALF the length of the square window

% Create the Window
SqWindow=zeros(size(X));
SqWindow([-WSize:WSize]+WCentx,[-WSize:WSize]+WCentx)=1;

[Xax,Yax]=meshgrid([1:size(X,2)],[1:size(X,1)]);
StanDev = 100;                              % Standard deviation
GaWindow=(1/sqrt(2*pi*StanDev)).*exp(-0.5*(((Xax-WCentx)/StanDev).^2+...
         ((Yax-WCenty)/StanDev).^2));
GaWindow=GaWindow./max(max(GaWindow));      % Normalization

SqWindowedImage=X.*(SqWindow);  % Windowing of the original image;
GaWindowedImage=X.*(GaWindow);  % Windowing of the original image;

figure;                  % Display the window
colormap(gray(255));
subplot(1,2,1);
imagesc(SqWindow);       % imagesc is the same as image
                         % but scale the data to the full
                         % range of the current colormap
axis off;title('Square Window');
subplot(1,2,2); imagesc(GaWindow*254+1);
axis off;title('Gaussian Window');

figure;                  % Display the windowed image
colormap(gray(255));
subplot(1,2,1);imagesc(SqWindowedImage);
axis off; title('Squared Windowed Image');
subplot(1,2,2);imagesc(GaWindowedImage);
axis off;title('Gaussian Windowed Image');
```

The matrix `SqWindow` is a matrix of ones and zeros. If you multiply it by the original image, the resulting image in unchanged only where the window is equal to one. However, the resulting image will change in those pixels where the value of the window is less than one (i.e., zero in this case). The result of this windowing is shown in Fig. 6.4. This type of windowing works best with gray-scale images. Keep in mind that in the case of or RGB images, the windowing has to be applied to each color. Such a windowing type is useful for creating gabor patches.

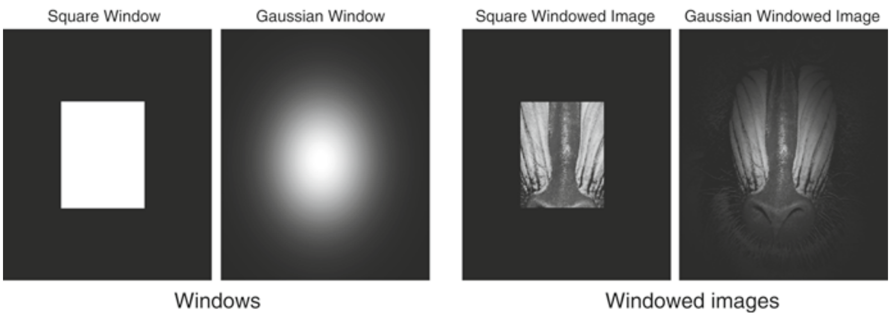


Fig. 6.4 Windowing concept. Two different windows are applied to the same figure

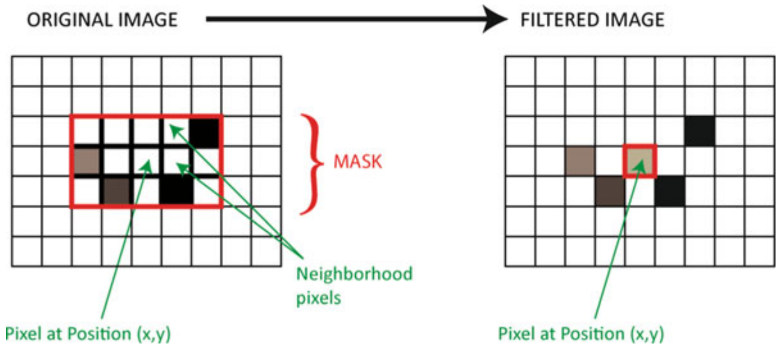


Fig. 6.5 Neighborhood processing concept

Neighborhood Processing

In the previous section we have seen how to modify images by applying a transformation of the intensity to each pixel. In this section we extend such an approach by including as well a neighborhood of each pixel. Overall, the neighboring pixels belong to a mask centered on the pixel where we want to obtain the new intensity value. The new intensity is calculated by combining all the intensities of the mask. Such an operation is called space *filtering*. In Fig. 6.5, the concept is illustrated graphically.

Spatial filtering requires two steps:

1. Place the mask over the current pixel,
2. Calculate the intensity combination of all the pixel intensities within the mask.

Here we give a simple example: a filter that gives the average of the nearest pixel. The mask is a matrix of 3×3 pixels. The operation to obtain the new pixel intensity is simple: multiply each intensity in the mask by 1/9 (9 is the total number of pixels in a mask) and sum them. The operation is performed for each image pixel using the function `filter2`.

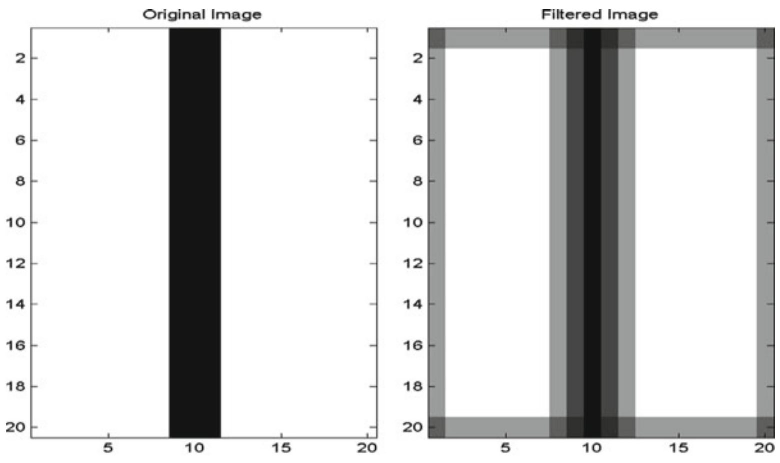


Fig. 6.6 Image filtering example

Listing 6.3

```
1   % M-Script to test the filtering concept
2   %
3   % Authors: Borgo, Soranzo, Grassi
4   % Date: 2009
5
6   Filter = ones(3,3)*1/9;      % generate the filter
7   ImageTest=ones(20,20)*255;   % generate a white image
8   ImageTest(:,9:11)=1;     % Plot a three pixel width vertical line
9
10  Filtered=filter2(Filter, ImageTest, 'same'); % do the filtering
11
12  figure;                  % show the original and filtered image
13  subplot(1,2,1)
14  image(ImageTest);
15  title('Original Image');
16  subplot(1,2,2)
17  image(Filtered);
18  title('Filtered Image');
19  colormap(gray(255));
```

The result is shown in Fig. 6.6.

At line 10 we used the function `filter2`, which is a function that filters the data in the second argument with the FIR filter (the mask and the values of such a mask) in the first argument. The third argument of the function controls how the edges are treated. You may have noticed that the filtered image has artifacts on the edges. These artifacts are explained in the following section.

If you have the MATLAB image toolbox, use `imfilter()` instead of `filter2()`.

There are many types of filters, e.g. low-pass, high-pass. The filtering action is always the same, but the difference lies in the filter's design. Here we do not want to explore the world of filter design. However, we would like to give you just another example: the Gaussian filter.

Listing 6.4

```
% M-Script to test the Gaussian Filter
%
% Authors: Borgo, Soranzo, Grassi
% Date: 2009

[MoonPic,Moonmap] = imread('moon.tif');
[Xax,Yax]=meshgrid([1:21],[1:21]);
StanDev=4;
FilterG = (1/sqrt(2*pi*StanDev)).*exp(-0.5*(((Xax-11)/StanDev).^2+...
((Yax-11)/StanDev).^2));
FilterG = FilterG/sum(sum(FilterG));

Filtered1=filter2(FilterG, MoonPic, 'same');

figure;
subplot(1,2,1)
image(MoonPic);
title('Original Image');
axis image;
subplot(1,2,2)
image(Filtered1);
title('Filtered Image');
axis image;
colormap(gray(255));
figure;
mesh(Xax,Yax,FilterG);
title('Filter Values');
```

The result of Listing 6.4 is shown in Fig. 6.7. On the right we show the filter values.

As we mentioned before, filter design is not simple. However, the MATLAB image toolbox has a function called `fspecial` that helps you to create 2-D filters. The function `h=fspecial(type)` creates a two-dimensional filter h of the specified type, which is the appropriate form to use with `imfilter`. Here `type` is a string having one of the following values: `'average'`, `'disk'`, `'gaussian'`, `'laplacian'`, `'log'`, `'motion'`, `'prewitt'`, `'sobel'` and `'unsharp'`. Each type needs some other specific values (i.e., mask dimension and other parameters). For example, in order to create a Gaussian filter similar to the one we have used in the previous example, type the following:

```
>> FilterGSpecial = fspecial('gaussian', 21, 4);
```

`FilterGSpecial` is a rotationally symmetric Gaussian filter of size 21 pixels with standard deviation of 4 pixels. For further information please refer to the MATLAB help.

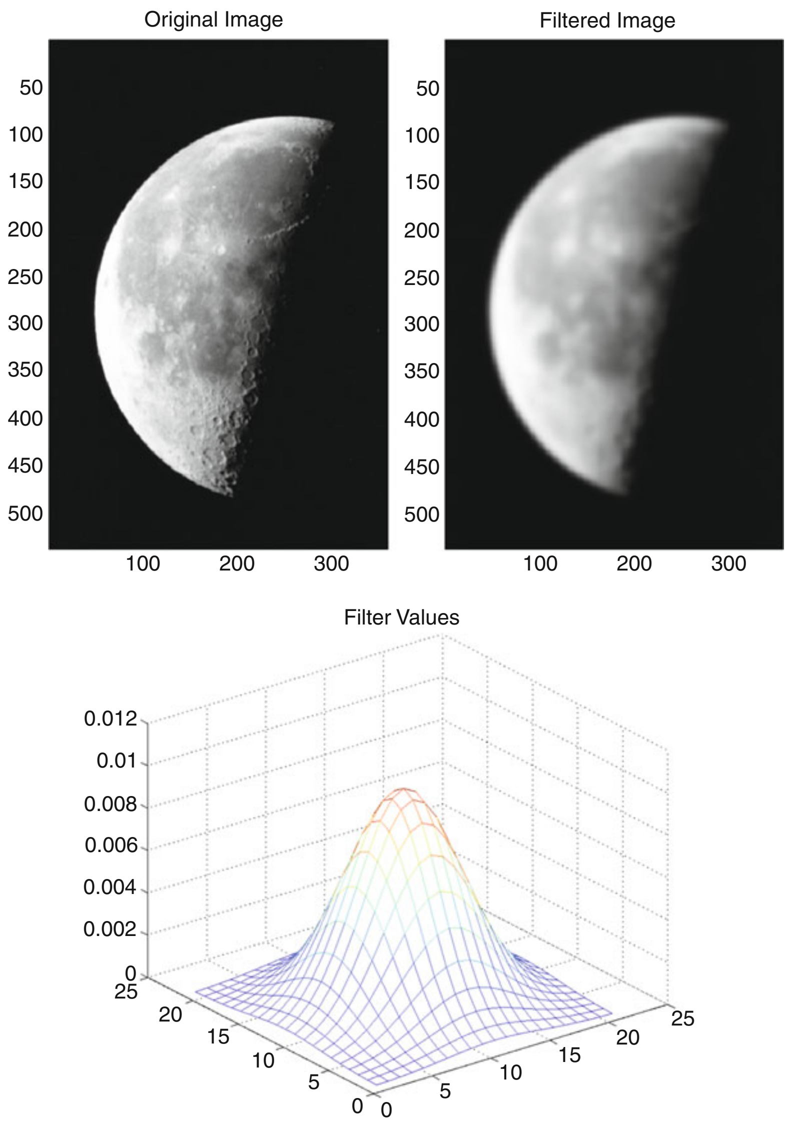


Fig. 6.7 A lunar image filtered with a Gaussian filter. The Gaussian (spatial) filter values are given on the *right*

We conclude this section by reminding you that the resizing procedure is also a form of neighborhood processing. The `imresize` function does image resizing. When you resize an image, you specify the image you want to resize and the magnification factor. To enlarge an image, specify a magnification factor greater than 1.

To reduce an image, specify a magnification factor between 0 and 1. Here there is an example to reduce the image by 50%:

```
>> TreesRes = imresize(Trees,0.5);
>> image(TreeRes);
```

The Edges of the Image

When we filter an image, there is a problem at the edges, where the mask partly falls outside them. There is a number of different approaches to solve this problem:

- *Ignore the edges.* The mask is applied only to those pixels of the image where the mask fully lies within the image. This results in an output image that is smaller than the original. To obtain this result, you should specify `'valid'` as the third argument of the `filter2` function.
- *Pad with zeros*. The missing values in the neighborhood of edge pixels are set to zero. This gives us a complete set of values to work with, and the result will be an output image of the same size as the original, but it may have the effect of introducing unwanted artifacts around the image. To obtain this result, you should specify `'same'` as the third argument of the `filter2` function.

Advanced Image Processing

The aforementioned methods are not straightforward. However, these methods are useful if you need to create and modify images. If you need more complex images processing, perhaps it is simpler to use the MATLAB image toolbox.

There are also devoted software packages for working with images, such as Adobe Photoshop. However, MATLAB can be useful when you need to modify repeatedly a certain number of images in the same way: writing a MATLAB script could be less time-consuming than repeatedly performing the same operation with Photoshop. Moreover, starting from version CS3, MATLAB and Photoshop (using Photoshop Extended) are connected: MATLAB can use Photoshop functions (and vice versa). For further information please read the Photoshop Manual.

Creating Images by Computation

In this section we see how to design and plot simple images. There is a partial overlap between the way to plot images in the current section and the way to plot images using the PsychToolbox as explained in the following chapters. However, it is useful to know both, so that you can use the best method according to your specific needs.

It is quite simple to plot images using the `plot` command. However, MATLAB has other simple functions to plot in 2-D (lines and polygons) and 3-D (spheres, cylinders, etc.). In the following table the main plotting commands are presented:

Function	Description

`line(x,y)`
`line(x,y,z)`
	Plot a multiline to the current figure. `x` and `y` are vectors of the same size specifying the endpoints of the line. `line(X,Y,Z)` creates lines in 3-D coordinates

`fill(x,y,c)`	Fills the 2-D polygon defined by vectors `x` and `y` with the color specified by `c`. `c` can be a string (such as a plot color specification) or an RGB vector. If c is a vector of numbers of the same length as x and y, its elements are used as indices into the current color map to specify colors at the vertices; the color within the polygon is obtained by bilinear interpolation of the vertex colors
`fill3(x,y,z,c)`	This is equivalent to `fill` but in 3-D space

`cylinder(r,n)`
`[x,y,z]=cylinder(r,n)`
	Forms a 3-D unit "cylinder" with `n` equally spaced vertices around the circumference of radius r. If r is a vector, the resulting figure is the connection between successive vertices at different radii, expressed by the vector r. It returns three matrices to be used with the function `surf`

`ellipsoid(xc,yc,zc,xr,yr,zr,n)`
`[X,Y,Z]=ellipsoid(xc,yc,zc,xr,yr,zr,n)`
	Plot an ellipsoid with center at `xc`, `yc`, and `zc` and radii `xr`, `yr`, `zr`. It returns three matrices to be used with the function `surf`. n is the number of surfaces used to form the ellipsoid

Here we provide a simple script to show the simultaneous lightness contrast effect [which is the condition whereby a gray patch on a dark background appears lighter than an identical patch on a light background; see Kingdom (1997) for a historical review of this perceptual phenomenon] and the successive color contrast effect, which is the condition whereby the perception of currently viewed colors is affected by previously viewed ones (see, for example, Helmholtz (1866/1964)), using some of the commands presented previously.

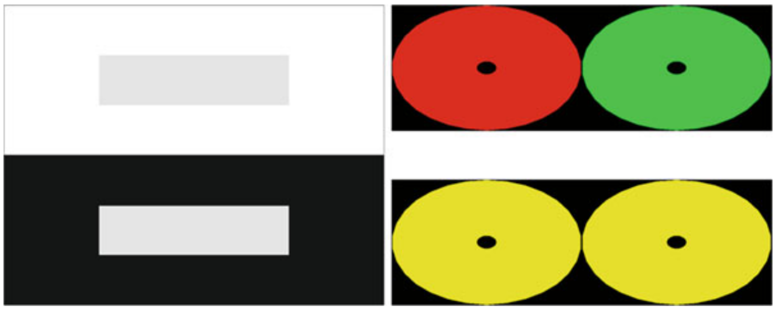


Fig. 6.8 Example of contrast and successive contrast effect images

Listing 6.5

```
% M-Script to test the function to
% generate simple graphics.
% This show a figure with simultaneous
% Contrast and successive contrast
% effect
%
% Authors: Borgo, Soranzo, Grassi
% Date: 2009

figure; hold on;
colormap(gray(255));                % Set Colormap
fill([0,4,4,0],[0,0,3,3],30);       % plot
fill([1,3,3,1],[1,1,2,2],170);      % different
fill([0,4,4,0],[3,3,6,6],200);      % color
H=fill([1,3,3,1],[4,4,5,5],170);% rectangles
set(H,'EdgeAlpha',0);               % Clean last rectangle edge

Npoint=30;                          % Number of points
x=[1:Npoint]./Npoint*2*pi;           % to plot a circumference.

figure;
subplot(2,1,1); hold on;
fill([0,4,4,0],[0,0,2,2],[0 0 0]);    % Plot a black rectangle
fill(sin(x)+1,cos(x)+1,'r');          % plot a red circle centered in
                                      % x=1,y=1
fill(sin(x)*0.1+1,cos(x)*0.1+1,'k'); % plot a little black circle
fill(sin(x)+3,cos(x)+1,'g');          % plot a green circle
fill(sin(x)*0.1+3,cos(x)*0.1+1,'k');
axis off;
subplot(2,1,2); hold on;
fill([0,4,4,0],[0,0,2,2],[0 0 0]);
fill(sin(x)+1,cos(x)+1,'y');
fill(sin(x)*0.1+1,cos(x)*0.1+1,'k');
fill(sin(x)+3,cos(x)+1,'y');
fill(sin(x)*0.1+3,cos(x)*0.1+1,'k');
axis off;
```

If you run the above script, you should see the images in Fig. 6.8.

Not all images can be designed using lines or polygons. Let's suppose that you want a two-dimensional sinusoidal image with a frequency of eight cycles per image, rotated by a certain angle. The following function does the job.

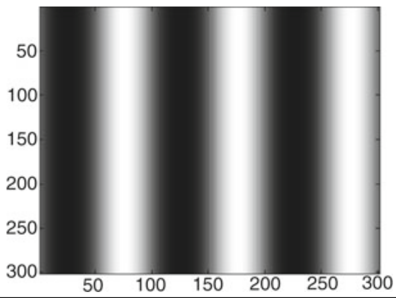
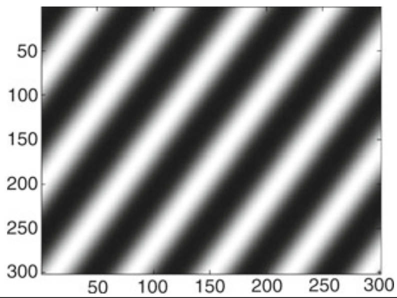
Listing 6.6

```
function [ImOut]=Sinusoid2D(xysize,SAngle,Fcxi,PhGrad)

% function [ImOut]= Sinusoid2D(xysize,SAngle,Fcxi,PhGrad))
%
% the function return a 2-D Spatial Sinusoid
%
% INPUT:    xysize  is the vector containing the x per y dimension
%                   if it is a scalar a square image is created
%           SAngle  is the spatial angle in degree of the sinusoid
%                   0 = vertical, 90 = horizontal
%           Fcxi    is the sinusoid frequency in cycles per image
%           PhGrad  is the phase of the sinusoid (in degree).
% OUTPUT    ImOut   the image of dimension express by xysize.
%
% Authors: Borgo, Soranzo, Grassi
% Date: 2012

% Calculate the image grid.
Boundx=xysize(1);
if length(xysize) == 2
   Boundy=xysize(2);
else
   Boundy=Boundx;
end
[x,y]=meshgrid(1:Boundx, 1:Boundy);

% define the costant for the cosine
wo=SAngle/180*pi;   % conversion from deg to rad
PhRad=PhGrad/180*pi;% conversion from deg to rad
f=Fcxi/xysize(1);   % frequency conversion

ax = f*cos(wo);    % convert the spatial frequency along y
by = f*sin(wo);    % convert the spatial frequency along x

% Calculate the Sinusoid
%(+1 is to obtatin all values being more than 1)
ImOut=sin(2*pi*(ax*x+by*y)+PhRad)+1;
```

Now save it with the name Sinusoid2D and test it with the following parameters.

Example 1	Example 2

`A=Sinusoid2D(301,0,3,0);`
`>> imagesc(A)`
`>> colormap(gray)`
	`A=Sinusoid2D(301,30,5,0);`
	`>> imagesc(A)`
	`>> colormap(gray)`

If we want to obtain a Gabor patch (i.e., a sine-wave grating in a gaussian window) we need to apply a gaussian window to the sinusoidal images we generated previously. The following function does it for you:

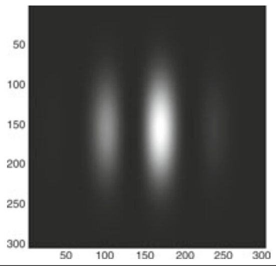
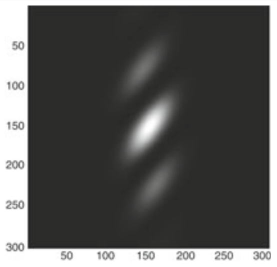
Listing 6.7

```
function [ImOut]=Gabor2D(xysize,SAngle,Fcxi,PhGrad,sigx,sigy)

% function [ImOut]= Gabor2D(xysize,SAngle,Fcxi,PhGrad,sigx,sigy)
%
% The function return a Spatial (2-D) Gabor image(Filter)
% defined in space domain as follow
% g(x,y) = s(x,y) * wr(x,y)
% where s(x,y) is the sinusoid (called also the carrier)
%       wr(x,y) is the gaussian window (called also envelope)
%
% the function return a 2-D Spatial Sinusoid
%
% INPUT:    xysize  is the number of pixel of the image
%           SAngle  is the spatial angle in degree of the sinusoid
%                   0 = vertical, 90 = horizontal
%           Fcxi    is the sinusoid frequency in cycles per image
%           PhGrad  is the phase of the sinusoid (in degree).
%           sigx    is the variance in pixels for the gaussian
%                   window on x axis.
%           sigy    is the variance in pixels for the gaussian
%                   window in y axis.
%                   if not specify, sigy=sigx;
% OUTPUT    ImOut   the image of dimension xysize X xysize.
%
% Authors: Borgo, Soranzo, Grassi
% Date: 2009

if nargin==5
    sigy=sigx;
end

% Calculate the grid of point where calculate the image.
Bound=floor(xysize/2);
[x,y]=meshgrid(-Bound:Bound,-Bound:Bound);

% Calculate the gabor
S=Sinusoid2D(xysize,SAngle,Fcxi, PhGrad);
gaussEnv = exp(-((x/sigx).^2)-((y/sigy).^2));
ImOut=S.*gaussEnv;
```

You can rearrange the scripts and put all the operations in a single code listing. Here we show some examples, using the function `Gabor2D`:

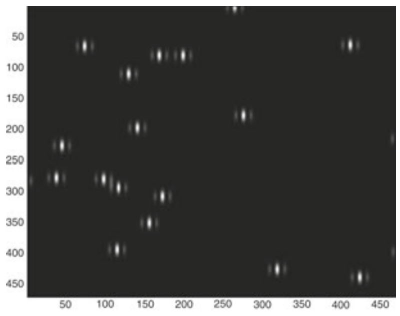
Example 1	Example 2

`>> A=Gabor2D(301,0,4,0,70);`
`>> imagesc(A)`
`>> colormap(gray)`
`>> axis square`
	`>> A=Gabor2D(301,45,5,90,70,30);`
	`>> imagesc(A)`
	`>> colormap(gray)`
	`>> axis square`

Now that we have drawn a Gabor Patch, it should be quite simple to realize Gabor patches randomly distributed on the screen. Simply use the Gabor2D function as filter! Let's have a look:

Example	Graphical result

`>> F=zeros(500);`
`>> GaborFilter=Gabor2D(51,0,3,90,10);`
`>> for i=1:20;`
`>> F(floor(rand*500),floor(rand*500))=1;`
`>> end;`
`>> RandIm=filter2(GaborFilter,F,'valid');`
`>> imagesc(RandIm);`
`>> colormap(gray);`
	

Here, each single point is set to 1 at a random position, and it is filtered with a Gabor patch created using Gabor2D function. The result is simply the displacement of the gabor patches of the screen. As you can see, you have a complete tool to work with Gabor patches.

Summary

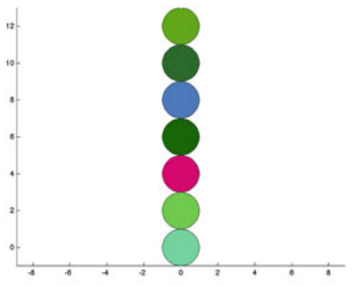
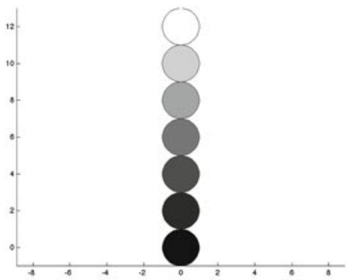
- Digital pictures consist of a discrete number of *pixels.*
- Each pixel is associated with a value:
  - In gray-scale images the value corresponds to a gray level.
  - For color images, the color in the pixel is represented by n values, one for each basic color (i.e., for RGB images, one intensity for red, one intensity for green, and one intensity for blue).
  - For indexed images, the value corresponds to the index of a color-map table.

- The function `colormap` sets and retrieves the current color map used for indexed images.
- The function `imread` reads an image from a file, while the function `imwrite` writes an image file.
- The *point operation* changes each pixel value by modifying its intensity value using direct transformations. The `brighten` and the `contrast` functions are two useful point operations.
- *Windowing* is a point operation whereby there is a multiplication between two matrices: the image matrix and the window matrix. It is often used to create smooth edges, or to highlight certain parts of the image.
- The *neighborhood operation* changes each pixel's intensity value by considering the intensity of a certain number of pixels, generally in a mask around the pixel to be changed. Such an operation is called *space filtering.*
- Space filtering is done using the function `filter2.` Alternatively, you can use the function `imfilter.`
- Using specific commands like `line, cylinder, fill, fill3, ellipsoid,` it is possible to create simple 2-D and 3-D images.
- Gabor patches are created using windowing and filtering techniques.

Exercises

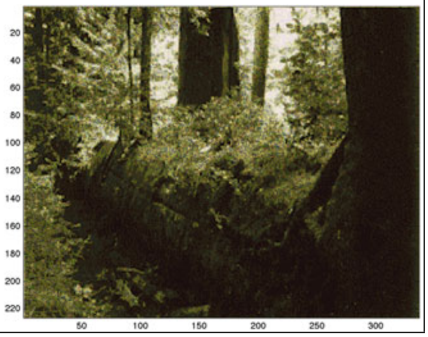
1. Write an M-script to create a color map of seven colors. Create a picture of seven circles. Each circle should be filled with a different color taken from the color map. Display another figure having seven circles but in gray scale (i.e., change the color map)

Solution	Graphical result

`Ncirc = 7;`
 	
`% create a random colormap`
`Mycolormap = rand(Ncirc,3)`
 	
`Npoint=30;`
`x=[1:Npoint]./Npoint*2*pi;`
 	
`figure;`
`hold on;`
`for i = 1:7`
`    fill(sin(x),cos(x)+(i-1)*2,i);`
`end`
`axis equal`
`% Apply the colormap`
`colormap(Mycolormap);`
 	
`figure;`
`hold on;`
`for i = 1:7`
`    fill(sin(x),cos(x)+(i-1)*2,i);`
`end`
`axis equal`
` % Apply the grayscale`
`colormap(gray(7));`
	 

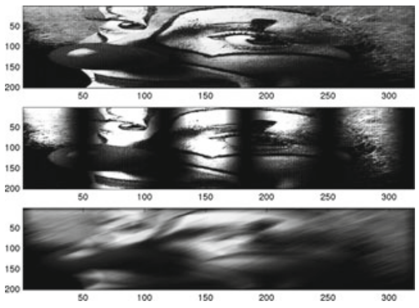
2. Read the image 'forest.tif', display it, and increase its brightness using a beta of 0.2. Resize it to 75% of its original size. Then save it with the new name 'MyForest.tif'.

Solution	Graphical result

`>> [X1,map1]=imread('forest.tif');`
`>> image(X1);`
`>> colormap(map1);`
`>> brighten(0.2);`
`>> X1res = imresize(X1,0.75);`
`>> image(X1res);`
`>> print -dtiff myforest;`
	

3. Load the file clown (i.e., type `load clown`) and use a gray-scale color map. Apply a sinusoidal window with a frequency of three cycles per image. You can create the window using the function `Sinusoid2D` in Listing 6.6. Apply to the original image a filter created with `fspecial` of type 'motion', with length 25 and an angle of 45°. Use the MATLAB help to see how to apply fspecial. Display the three (gray) images.

Solution	Graphical result

`>> load clown;`
`>> SinWin=Sinusoid2D(size(X),0,3,0);`
`>> h = fspecial('motion', 25, 45);`
`>> Xfiltered=filter2(h,X);`
`>> figure;`
`>> subplot(3,1,1);`
`>> image(X);`
`>> subplot(3,1,2);`
`>> image(X.*SinWin);`
`>> subplot(3,1,3);`
`>> image(Xfiltered);`
`>> colormap(gray(length(map)));`
	

References

Helmholtz HV (1866/1964) Helmholtz's treatise on physiological optics. Optical Society of America, New York

Kingdom F (1997) Simultaneous contrast: the legacies of Hering and Helmholtz. Perception 26(6):673–677

Suggested Readings

Some of the concepts illustrated in this chapter can be found, in an extended way, in the following books:

Gonzalez RC, Woods RE, Eddins SL (2009) Digital image processing using MATLAB. Gatesmark Publishing

Poon T-C, Banerjee PP (2001) Contemporary optical image processing with MATLAB. New York: Elsevier Science Ltd