CHAPTER

11

Two-Dimensional Signals—Basic Image Analysis

11.1 GOALS OF THIS CHAPTER

The idea that an image is nothing more than a two-dimensional (2-D) signal was introduced in Chapter 1 (Section 1.2.5). In fact, many signal processing operations explained in previous chapters apply directly to image analysis: the Fourier transform, convolution, filtering, and some nonlinear transformations. The images we work with here are digital images; they are the result of sampling, just as digital signals are sampled. As they are 2-D signals, they are stored in arrays or matrices rather than vectors. As with one-dimensional (1-D) signals, the sampling frequency is important, and, as shown in the next section, undersampling an image also results in aliasing. With images, the independent variable is length rather than time, so the image sampling frequency, termed "spatial frequency," is in samples/cm or samples/in.[1]

For images, as noted in Chapter 1, each sample is termed a pixel, or if the sample represents a three-dimensional (3-D) space, a "voxel." Because images are represented as arrays, memory requirements can become quite large. An $8^1/_2$ by 11 in. image that is sampled at a frequency of 300 samples/in. would have 8,415,000 pixels. However, in MATLAB a variable requires 8 bytes, so this image would require 67 Mb of memory. The MATLAB Image Processing Toolbox can represent pixels using only one or two bytes, substantially reducing memory requirements. On the other hand, the type of computations that can be performed on these reduced formats are limited, whereas images represented as standard 8-byte variables are amenable to the full range of MATLAB operations.

For heavy duty image processing, you might want to use the Image Processing Toolbox, or other software specifically designed for images, but the basic and most important concepts

[1]Printer resolution usually is given in terms of sampling frequency but stated as dots/inch where a dot is the same as a sample.

Circuits, Signals, and Systems for Bioengineers
https://doi.org/10.1016/B978-0-12-809395-5.00011-4

of image processing and image analysis can be examined using standard MATLAB. Topics in this chapter include the following:

- Basic image format and display
- Sampling and the 2-D Fourier transform
- 2-D convolution
- Image filtering, including filter construction
- Identifying regions of interest (ROIs): basics of image segmentation, including edge detection and texture analysis.

11.2 IMAGE FORMAT AND DISPLAY

Images are stored in arrays, and when we need to refer to specific image elements we use the coordinate system that MATLAB terms the "pixel coordinate system" shown in Figure 11.1. In this format, the matrix element (1,1) represents the upper left-hand pixel in the image and element (*m*,*n*), the bottom right-hand pixel, where *m* is the number of rows in the matrix and *n* the number of columns.

The matrix values can represent image intensities or they might be pointers to another array known as a colormap. A colormap is a three-by-n matrix where the three columns contain values of red, blue, and green intensity values. In "colormapped" images, the pixel value indicates a row in the colormap that corresponds to the desired mixture of red, blue, and green. This mapping process allows color images to be stored in a single matrix plane. The appropriate colormap must be stored with the image. Neighborhood operations, operations that work on groups of adjoining pixels, may not be appropriate for colormapped images.

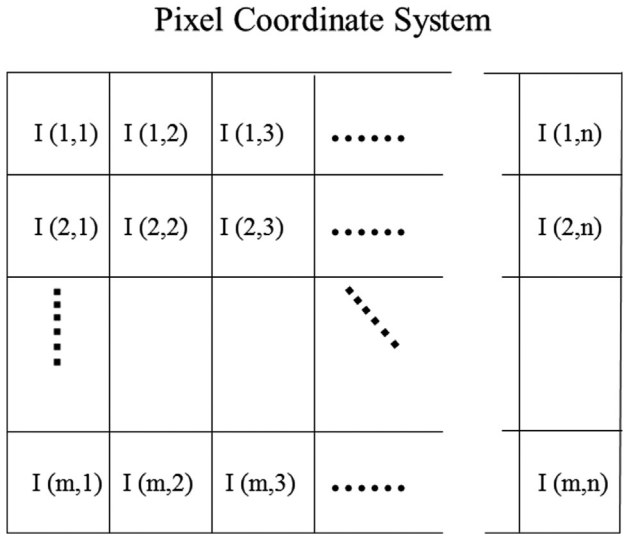

FIGURE 11.1 Indexing format for MATLAB images using the pixel coordinate system. This indexing protocol follows the standard matrix notation.

For all the images used in this chapter, pixel values describe the image intensity at that location. We use two different types of images: black and white (BW) images where the image matrix contains only two intensity values, and grayscale images where the matrix has a range of values. For all of our images, 0.0 is assumed to represent black for both BW and grayscale images. For BW images, 1.0 is usually used to represent white and we follow that convention. For grayscale images, white is 255, but in some conventions, white is 1.0 and intensity values range between 0 and 1.0. The images used here are stored without a colormap.

All images used in this chapter are stored as tiff files. They are loaded into MATLAB workspace using the following routine:

```
I = importdata('filename.tif');  % Load tiff file image in matrix I
```

where `I` is the matrix that contains the image and `filename` is the name of the particular image file. It is common to name variables that contain grayscale images with words beginning with capital `I`. Similarly, it is the convention to name variables that contain BW images with words that begin with the capitals BW. Once loaded, the intensity values in matrix, `I`, are available for all suitable MATLAB operations.

In all the operations in this chapter, we assume that the image is stored as a MATLAB variable double format. This allows us to use all standard MATLAB operations. As many of our tiff file images are stored in `uint8` format,[2] we need to convert them to the double format using:

```
I = double(I);    % Convert to double format.
```

If the image is already in double format, the `double` command has no effect.

To display an image, we could use the approach used in Section 1.2.5 that is based on the MATLAB routine `pcolor`. In this chapter we use the image display routine, `image`. The `image` routine is more flexible in that it can display matrices that have three color planes as well as matrices that are in uint8 and uint16 formats (see Footnote 2). The routine displays images in either levels of gray or color by mapping image intensity values to a colormap. Because our images are stored without a colormap, we need to assign an appropriate colormap to our displayed image. A colormap is assigned to an image by the command `colormap(map_name)`. In textbook presentations, we limit images to either grayscale or BW, so we use `colormap(gray)` or `colormap(bone)` to assign a grayscale to our image. In the problems, you can try some of the more colorful colormaps that produce "pseudocolor" images.

Because the intensity values in our image arrays range between 0 and 255, we need to set the colormap range to these values. This entails using the `image` option `"CDataMapping," "scaled"` in conjunction with the command `caxis([0 255])`, or `caxis([0 1])` for BW images.

[2]The `uint8` format uses a single byte (8 bits) to store a pixel, whereas the MATLAB double format uses 8 bytes. The `uint8` format requires one-eighth the storage and memory space, so it is popular for storing images that usually require large arrays. However, most MATLAB operations require that variables be stored in the double format. Occasionally images are stored in a 2-byte format known as `uint16`. Note that uint8 and uint16 are common terms for 1- and 2-byte variables in other computer languages.

The `caxis` command sets the range of the colormap. When using `image` for image display, two additional MATLAB visualization commands are helpful: (1) the `axis image` command ensures that each pixel is represented as having equal horizontal and vertical dimensions, and (2) the `axis off` command eliminates the numbers on the side of the image. Thus the commands needed to display an image are:

```
image(I,'CDataMapping','scaled'); caxis([0 255]); % Display image I
colormap(bone); axis image; axis off;              % Image options
```

The first example demonstrates these operations.

EXAMPLE 11.1

Load the image of blood cells in the file `blood1.tif` and display. Invert the image so that dark areas are light and vice versa and display.

Solution: Load the image using `importdata` and display using `image` with the appropriate options. To invert the image, subtract the matrix values from 255, the value that represents white.

```
% Example 11.1 Load and display an image.  Invert grayscale and display
%
I = importdata('blood1.tif');          % Load image into matrix I
I = double(I);                         % Convert to double format
subplot(1,2,1);
image(I,'CDataMapping','scaled'); caxis([0 255]);   % Display image
colormap(bone); axis image; axis off;  % Image options
title('Normal Image','FontSize',14);
subplot(1,2,2);
I_invert = 255 - I; % Invert image
image(I_invert,'CDataMapping','scaled')
.....title.....
```

Results: The images produced in this example are shown in Figure 11.2.

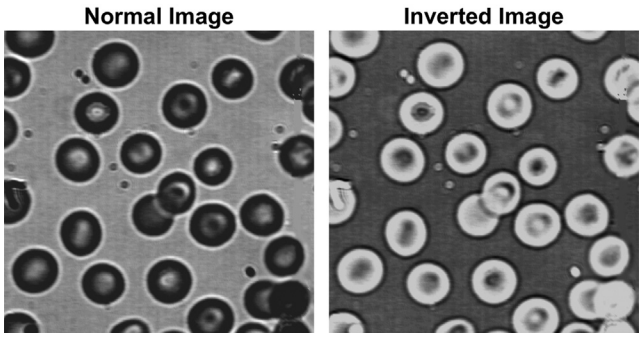

FIGURE 11.2 Image of blood cells displayed normally and with an inverted grayscale.

11.3 THE TWO-DIMENSIONAL FOURIER TRANSFORM

The Fourier transform and the efficient algorithm for computing it, the fast Fourier transform, extend in a straightforward manner to two dimensions. The 2-D version of the Fourier transform can be applied to images providing a spectral analysis of the image content. Of course, the resulting spectrum will be in two dimensions and is more difficult to interpret than a 1-D spectrum. Nonetheless, it can be a useful analysis tool, both for describing the contents of an image and to aid in the construction of imaging filters as described in the next section.

As mentioned, image sampling frequency is in terms of samples/length, such as samples/in. Undersampling an image will lead to aliasing, just as in 1-D signals. In an undersampled image, the spatial frequency content of the original image is greater than $f_s/2$, where f_s now is 1/(pixel size). Figure 11.3 shows an example of aliasing in the frequency domain. The upper left panel shows an image that varies sinusoidally, and the spatial frequency of this sinusoidal variation increases with horizontal position left to right. This is analogous to the 1-D chirp signal and, in fact, was generated by extending a horizontal chirp signal in the vertical direction. The higher-frequency elements on the right side of this image are adequately sampled in the left panel. The same pattern is shown in the upper right panel image except that the sampling frequency has been reduced by a factor of six. The low-frequency variations in intensity are unchanged, but the high-frequency variations have additional frequencies mixed in as aliasing folds in

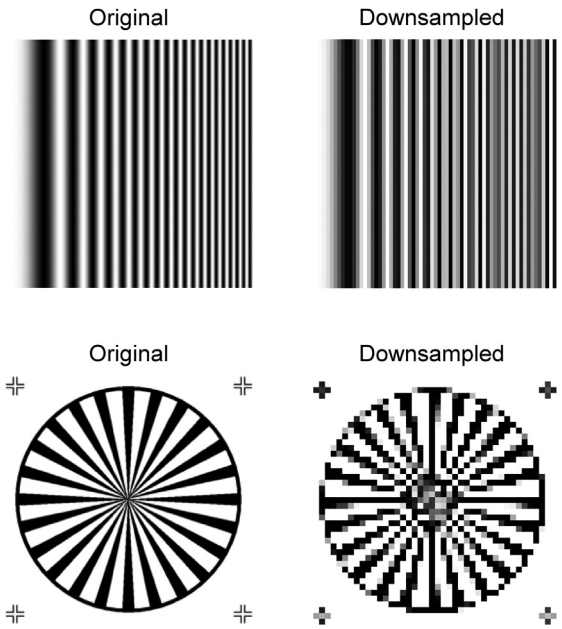

FIGURE 11.3 The influence of aliasing due to undersampling on two images with high spatial frequencies. Aliasing in these images manifests as additional sinusoidal frequencies in the upper right panel and jagged diagonals in the lower right panel.

the frequencies above $f_s/2$ (see Section 4.1.1). The lower panels show the influence of aliasing on a diagonal pattern. The jagged diagonals are characteristic of aliasing, as are the moiré patterns seen in other images. The problem of determining an appropriate sampling size is even more critical in image acquisition because oversampling can quickly lead to excessive memory storage requirements.

The 2-D Fourier transform in continuous form is a direct extension of the equation given in Chapter 3:

$$F(\omega_1\omega_2) = \int_{m=-\infty}^{\infty} \int_{n=-\infty}^{\infty} f(m,n)e^{-j\omega_1 m}e^{-j\omega_2 n}dm\,dn \tag{11.1}$$

The variables ω_1 and ω_2 are still frequency variables, although their units are in radians per sample size. As with the time domain spectrum, the image spectrum, $F(\omega_1,\omega_2)$, is a complex-valued function that is infinitely periodic in both ω_1 and ω_2. Usually only a single period of the spectral function is displayed as with the time domain analog.

The inverse 2-D Fourier transform is defined as:

$$f(x,y) = \frac{1}{4p^2}\int_{\omega_1=-p}^{p} \int_{\omega_2=-p}^{p} F(\omega_1\omega_2)e^{-j\omega_1 x}e^{-j\omega_2 y}d\omega_1 d\omega_2 \tag{11.2}$$

This statement is a 2-D extension of the 1-D equivalent; any image can be represented by a series (possibly infinite) of sinusoids, but now the sinusoids extend over two dimensions.

The discrete forms of Equations 11.1 and 11.2 are again similar to their time domain analogs. For an image size of M by N, the discrete Fourier transform becomes:

$$\begin{aligned} F(p,q) &= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)e^{-j(2pm/M)}e^{-j(2qn/N)} \\ p &= 0,1,\ldots,M-1; \quad q = 0,1,\ldots,N-1 \end{aligned} \tag{11.3}$$

The values $F(p,q)$ are the Fourier transform coefficients of $f(m,n)$. For images, $f(m,n)$ is just the image matrix. The discrete form of the inverse Fourier transform becomes:

$$\begin{aligned} F(m,n) &= \frac{1}{MN}\sum_{p=0}^{M-1}\sum_{q=0}^{N-1} F(p,q)e^{-(2pm/M)}e^{-j(2qn/N)} \\ m &= 0,1,\ldots,M-1; \quad n = 0,1,\ldots,N-1 \end{aligned} \tag{11.4}$$

11.3.1 MATLAB Implementation

Both the Fourier transform and the inverse Fourier transform are supported in two dimensions by MATLAB functions. The 2-D Fourier transform is invoked as:

```
F = fft2(x,M,N);    % Two dimensional Fourier transform
```

where `F` is the output matrix and `x` is the input matrix. `M` and `N` are optional arguments that specify padding for the vertical and horizontal dimensions, respectively, just as in

the 1-D fast Fourier transform (`fft`). In the time domain, the frequency spectrum of simple waveforms can often be anticipated and the spectra of even relatively complicated waveforms can usually be understood. With two dimensions, it becomes more difficult to visualize the expected Fourier transform even for fairly simple images. In Example 11.2, a simple image is constructed consisting of a thin rectangular bar. The Fourier transform of the object is determined, and the resultant spatial frequency function is plotted as a 3-D function.

EXAMPLE 11.2

Determine and display the 2-D Fourier transform of a thin rectangular object. The object should be three-by-nine pixels in size and solid white against a black background. Display the Fourier transform as a 3-D function using MATLAB's `mesh` routine.

Solution: We construct the image by first setting up a black background that is a 22-by-30 matrix of zeros. We do not need a large image because we use the zero-padding option in the Fourier transform routine to extend the matrix to 128-by-128 samples. To complete the image, we insert a narrow vertical center strip of white by assigning the value 1.0 to matrix columns 10 through 12 and rows 11 through 19. The image is plotted using `image` but with the scaling changed to range between 0 and 1 as is common for BW images.

The 2-D Fourier transform is constructed using `fft2` with padding as noted earlier. The `fft2` routine places the zero frequency (DC) component in the upper left corner. This approach to plotting the 2-D Fourier transform is logical but even more difficult to interpret. Interpreting the spectrum can be a little easier if it is shifted so that the zero frequency component falls in the center of the plot. The MATLAB routine `fftshift` swaps the first and third quadrants and the second and fourth quadrants of the spectrum so that the DC component falls in the center position. This shifted spectrum is then plotted using routine `mesh`, which displays the spectrum as a 3-D surface.

Example 11.2 Fourier transform of a simple rectangular image

```
%
I = zeros(22,30);            % Original figure can be any size since it
I(11:19,10:12) = 1;          % will be padded
F = fft2(I,128,128);         % Take Fourier transform padded to 128
F = abs(fftshift(F));        % Shift center; get magnitude
%
image(I,'CDataMapping','scaled');                    % Display image
colormap(bone); caxis([0 1]); axis image; axis off;  % Image options
.....label and new figure.....
mesh(F); colormap(bone);     % Plot Fourier transform as function
.....labels, axis, and new figure.....
```

Result: The image and its spectrum are shown in Figure 11.4. The spectrum has been shifted and displayed as a 3-D surface. With a little thought, the spectrum can be interpreted as the combined spectra resulting from two pulse signals. The image is effectively a wide vertical pulse change in intensity combined with a narrow horizontal pulse change in intensity. Recall the inverse

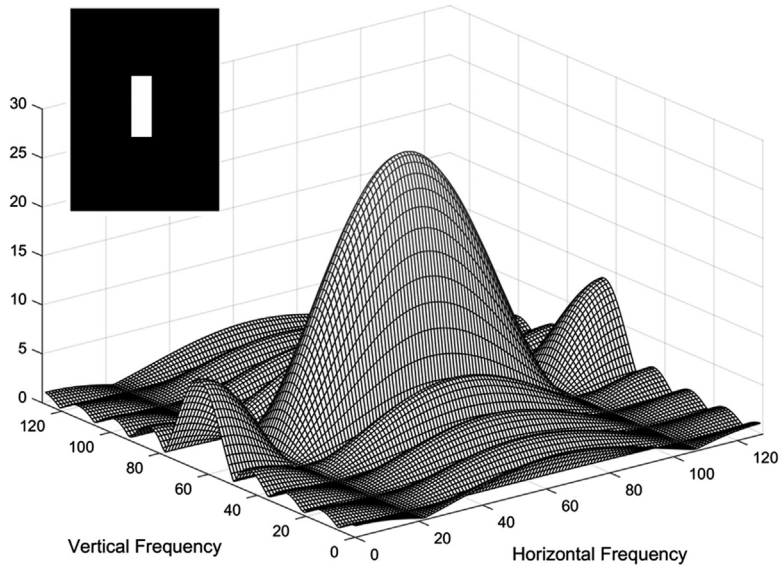

FIGURE 11.4 Upper left: the image of a rectangular object three-by-nine pixels used in Example 11.2. Lower main: the magnitude Fourier transform of this image. More energy is seen, particularly at the higher frequencies, along the horizontal axis because the image's vertical cross-section appears as a narrower pulse change in intensity. The broader vertical cross-section produces frequency characteristics that fall off more rapidly at higher spatial frequencies. This is analogous to what is found in 1-D pulse spectra (see Example 3.3).

relationship between pulse width and spectral width: the wider the pulse, the narrower its spectral peaks (see Example 3.3, and particularly Figure 3.12, which shows the spectra of a narrow and wide pulse). Extrapolating this to our 2-D image, we would expect our spectrum to have a narrower vertical peak (because the strip is wider in the vertical dimension) and a broader horizontal peak. For both directions, the spectrum should have the general shape of $|\sin(x)/x|$ as is seen in Figure 11.4.

Another easy-to-interpret 2-D spectrum is shown in Figure 11.5 and is generated from the image in the upper left panel of Figure 11.3. This image consists of a horizontal chirp signal that has been extended in the vertical direction. The fact that this image changes in only one direction, the horizontal direction, is reflected in the Fourier transform. The linear increase in spatial frequency in the horizontal direction produces an approximately constant spectral value over the horizontal spatial frequency. The constant image values in the vertical direction produce zero values over all vertical spatial frequencies other than the zero frequency value. Again, most images produce complex spectra that do not lend themselves to easy interpretation. However, the most useful application of the 2-D Fourier transform is not in image analysis but in the design and evaluation of linear imaging filters. Image filtering is the featured topic of the next section.

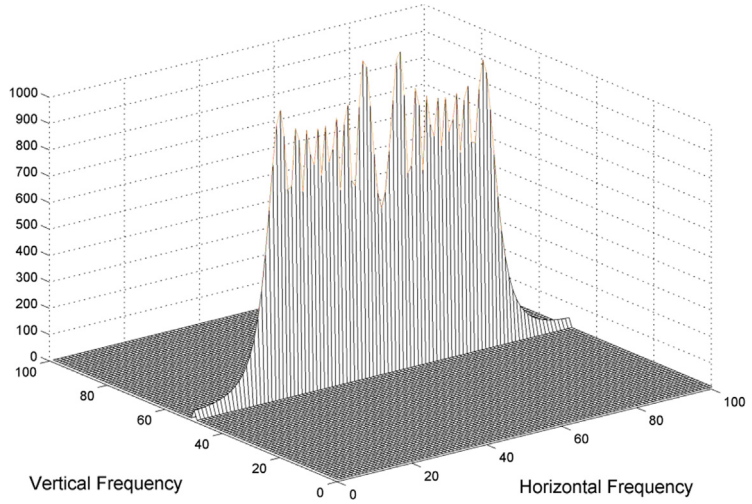

FIGURE 11.5 Magnitude spectrum of the horizontal chirp signal image shown in Figure 11.3, upper left panel. The spatial frequency characteristics of this image are zero for all vertical frequencies (except at zero frequency) because the image has constant intensities in the vertical direction. The linear increase in spatial frequency in the horizontal direction is reflected in the approximately constant amplitude of the spectrum over the horizontal spatial frequencies.

11.4 LINEAR FILTERING

The techniques of linear filtering described in Chapter 8 can be directly extended to two dimensions and applied to images. In image processing, finite impulse response (FIR) filters are used because of their linear phase characteristics. A sliding neighborhood operation is used to apply a 1-D FIR filter to a signal (see Section 8.5). A new sample of the filtered signal is calculated as the summation of a sequence of samples in the original signal scaled by the filter weights. This operation is often applied symmetrically so that the new sample lies at the center point of the filter coefficients, Figure 11.6. The filter coefficients then shift one position along the original signal and a new filtered sample is calculated. This operation is implemented through convolution.

Filtering an image is also a neighborhood operation, but now the neighborhood extends in two directions around a given pixel. In image filtering, the value of a filtered pixel is determined from the surrounding pixels, Figure 11.7. Pixels in a region of the original image are scaled by a matrix of filter coefficients. A summation of scaled pixel values is used as the new pixel and is placed in a position corresponding to the center of the filter matrix. The matrix of filter coefficients is then shifted across the original image both horizontally and then vertically. This operation is equivalent to 2-D convolution.

As described in Chapter 8, FIR filters are uniquely defined by their filter coefficients, $b[k]$. Filter design is a matter of finding those coefficients that produce the desired

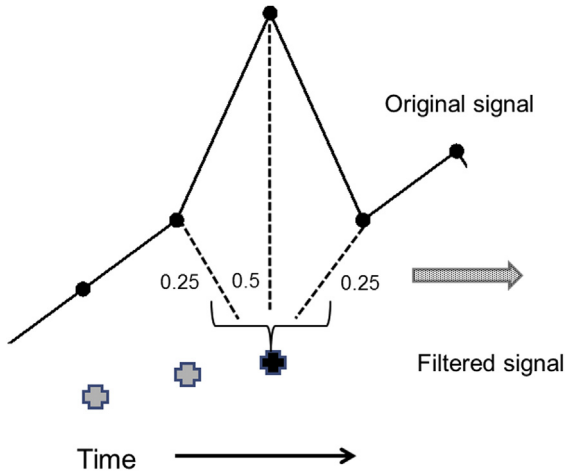

FIGURE 11.6 A finite impulse response filter is implemented as a weighted sum of filter coefficients applied to the original signal. This summation is often implemented symmetrically so that the new filter point is placed on the new signal at a point corresponding to the center of the filter coefficients. The coefficients slide over the original signal sample by sample to produce the filtered signal. This is illustrated for a three-weight filter.

modification of signal spectrum. An image filter is also uniquely defined by filter coefficients, but now these coefficients form a matrix in two dimensions, $b[k_1,k_2]$. These 2-D filter coefficients are applied to the image using 2-D convolution in an analogous approach to 1-D filtering described in Chapter 8.

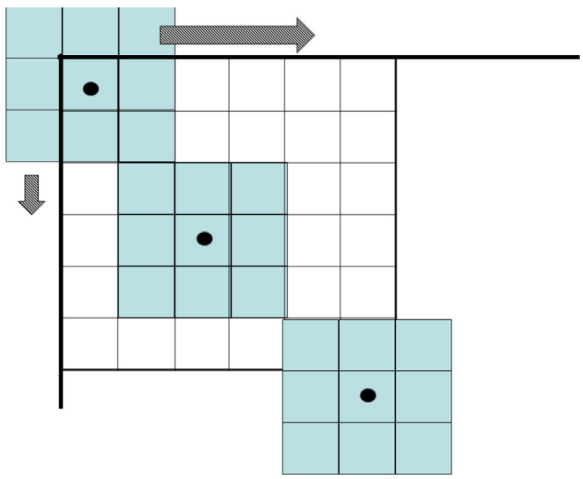

FIGURE 11.7 Implementation of a 2-D filter consisting of a three-by-three set of filter weights (*shaded boxes*). The filtered image is constructed by scaling the original image pixels by the filter coefficients. The weighted summation becomes a filtered image pixel that corresponds in position to the center of the filter coefficient matrix. The filter coefficient matrix slides over the image pixel by pixel in two dimensions. This is the 2-D extension of convolution.

11.4.1 Convolution in Two Dimensions

Using convolution to perform image filtering parallels its use in signal processing: the image array is convolved with a set of filter coefficients. However, in image analysis the filter coefficients are defined in two dimensions, $b[k_1,k_2]$ as a matrix. The equation for convolution in one dimension was given in Chapter 5 (Equation 5.3) and is repeated here:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]b[n-k] \tag{11.5}$$

where $x[k]$ is the input signal and $b[k]$ are the filter coefficients, and n slides the coefficients across the signal. Two-dimensional convolution is a straightforward extension of this equation:

$$y[m,n] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1,k_2]b[m-k_1,n-k_2] \tag{11.6}$$

where $x[k_1,k_2]$ is the image, $b[m,n]$ are the 2-D filter coefficients, and k_1 and k_2 slide these coefficients across the image in the horizontal (k_1) and vertical (k_2) directions. Although this equation would not be difficult to implement in MATLAB, there is, naturally, a MATLAB function that implements 2-D convolution directly, `conv2`:

```
I2 = conv2(I1,b,shape);   % Two-dimensional convolution
```

where `I1` and `b` are image and filter matrices (or more generally, simply two matrices) to be convolved and `shape` is an optional argument that controls the size of the output image. If `shape` is `"full,"` the default, then the size of the output matrix follows the same rules as in 1-D convolution: each dimension of the output is the sum of the two matrix lengths along that dimension minus one. Hence, if the two matrices have sizes `I1(M1,N1)` and `b(M2,N2)`, the output size is `M1+M2-1` by `N1+N2-1`. If `shape` is `"valid,"` then every pixel evaluation that requires image padding is ignored, leading to an output image size of `M1-M2+1` by `N1-N2 +1`. For our applications, the most useful option for image filtering is `"same."` Using this option results in an output matrix that is of the same size as the input image, `I1`, that is, `M1`-by-`N1` and is similar to 1-D convolution using that option. Using the `"same"` option eliminates the need for dealing with the additional, or missing, pixels generated by convolution.

When convolution is used to apply a series of filter weights to either an image or a signal, the weights are applied to the data in reverse order as indicated by the negative sign in the 1-D and 2-D convolution equations (Equations 11.5 and 11.6). This can be a source of mild confusion in 1-D applications and becomes even more confusing in 2-D applications. It becomes difficult to conceptualize how a given filter matrix alters an image after it has been reversed in two directions. One way around this is to apply the filter matrix using correlation rather than convolution. The correlation equation is very similar to the convolution equation except that the negative signs are now positive:

$$y[m,n] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1,k_2]b[m+k_1,n+k_2] \tag{11.7}$$

When correlation is used, the set of filter coefficients is termed the "correlation kernel" to distinguish it from the standard filter coefficients. In fact, the operations of correlation and

convolution both involve weighted sums of neighboring pixels, and the only difference between correlation kernels and convolution kernels is a 180-degree rotation of the coefficient matrix. The MATLAB routine `filter2` uses correlation to implement an image filtering. The call is very similar to that of `conv2`:

```
I2 = filter2(I1,b,shape);   % Image filter using correlation
```

where again `I1` is the original image, `b` is the filter matrix, and shape determines the size of the output in the same manner as `conv2`. For image filtering, we use the shape option `'same'`, which is the default option in `filter2`. This produces a filtered image that is of the same size as the original image. The filtering routine in the Image Processing Toolbox also uses correlation kernels, again because their application is easier to conceptualize.

11.4.2 Linear Image Filters

Image filters are designed differently than 1-D filters. Filters produced using 1-D design strategies, such as the spectral window approach (Section 8.5), do not work well on images. Many image filters were designed heuristically, that is to say, they stem from an idea about what ought to work and then were found that they did work well. In this section we describe five popular filters: one that performs spatial low-pass filtering for image smoothing, a high-pass filter for image sharpening, a high-pass filter that takes a spatial second derivative, and two similar filters that enhance edges for edge detection. Some filters have variably sized coefficients, but many linear filters, including most described here, have coefficients arranged as a three-by-three matrix.

One of the earliest linear filters was a filter designed for edge detection by Irwin Sobel. This "Sobel" filter performs a vertical spatial derivative operation for enhancement of horizontal edges. To detect vertical edges, the coefficient matrix can be rotated by 90 degrees using transposition.

$$b[m,n]_{\text{Sobel}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{11.8}$$

An examination of the Sobel matrix reveals its operation. The correlation between the image and filter coefficient is greatest when the underlying image has a dark (i.e., 0's) region overlapping the lowest row of the filter matrix and a white (i.e., 1's) region overlapping the upper row. If there is no horizontal boundary and the upper and lower regions are the same, the correlation will be 0 (because the lower negative rows cancel the upper positive rows). An application of a Sobel filter to an image is found in the next section.

The upper and lower rows of the Sobel filter have a larger number in the center of the outer rows, which creates a smoothing effect on the resulting image. A "Prewitt" edge detector filter is a modification of the Sobel filter that does not have this smoothing feature.

$$b[m,n]_{\text{Prewitt}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

The Gaussian image filter is a common low-pass filter and, as with the 1-D low-pass filter, is useful in eliminating high-frequency noise. The equation for a Gaussian filter is similar to the equation for the Gaussian distribution:

$$b[m,n] = e^{-(d/2\sigma)^2} \quad \text{where} \quad d = \sqrt{(m^2+n^2)} \tag{11.9}$$

where σ adjusts the attenuation slope of the low-pass filter. A common value for σ of 0.5 produces a modest slope, whereas larger values provide steeper slopes. The filter matrix is square but can be of any size. Changing the dimension of the coefficient matrix alters the number of pixels over which the filter operates. This filter has particularly desirable properties when applied to an image; it provides an optimal compromise between smoothness and filter sharpness. For this reason, it is popular as a general purpose image low-pass filter. The next example constructs a Gaussian filter and determines its spectral characteristics.

EXAMPLE 11.3

Construct a Gaussian filter having a five-by-five coefficients matrix. Determine the magnitude frequency spectrum of this filter for two values of σ: 0.5 and 1.9.

Solution: To create the Gaussian filter, we write a function that follows Equation 11.9. As in 1-D filtering, the filter coefficients are also the 2-D impulse response of the filter. So the filter's spectrum can be obtained from the 2-D Fourier transform of the filter coefficient matrix. The magnitude spectra are plotted for both values of σ using `mesh` as in the previous example.

To generate the filter coefficients, we construct a routine called `gaussian`. In this routine, the input variables are matrix dimension and σ, whereas the output variable is the filter coefficient matrix. The value of $2\sigma^2$ can be determined first because it is a constant in the equation. The variable d in Equation 11.9 represents the distance of a given matrix coefficient from the center of the matrix. To determine this value we need to calculate the indices of the center location: x_{center}, y_{center}. The distance then becomes $d = \sqrt{(x_{\text{index}} - x_{\text{center}})^2 + (y_{\text{index}} - y_{\text{center}})^2}$. In the `gaussian` routine, this distance variable is determined for every matrix position using a double loop. Because Equation 11.9 calls for d^2 in the exponent, the square root is not taken. The coefficients are then found by taking the exponential of d^2 divided by $2\sigma^2$. It is common to normalize the coefficients so they sum to 1.0, so the output image has the same overall intensities as the input image.

```
function b = gaussian(N,sigma)
% Function to construct a Gaussian lowpass filter
%
sig_sq = 2*(sigma^2);      % Calculate 2 sigma squared
center = (N+1)/2;          % Find center point
for k1 = 1:N
  for k2 = 1:N
    d_sq(k1,k2) = (k1-center)^2 + (k2-center)^2; % d squared
  end
end
b = exp(-d_sq/sig_sq);     % Calculate filter coefficients
b = b/sum(b(:));           % Normalize filter coefficients
```

The main program simply calls the `gaussian` routine and plots the magnitude of Fourier transforms for both values of σ.

```
% Example 11.3 Program to plot the magnitude spectrum of a Gaussian filter
% for two values of sigma
%
N = 5;                          % Filter dimension
sigma = [0.5 1.0];              % Values of sigma
for k = 1: length(sigma)
  b = gaussian(N,sigma(k));  % Determine filter coefficients
  F = fft2(b,128,128);       % Calculate magnitude spectrum
  F = abs(fftshift(F));      % Shift spectrum
subplot(1,2,k);
mesh(F);                        % Plot spectrum
.....labels, title, color axis, colormap.....
end
```

Results: The two spectra are shown in Figure 11.8. As expected, the filter with the higher value of σ shows a sharper attenuation. The effect of the attenuation slope on a filtered image is evaluated in the next section.

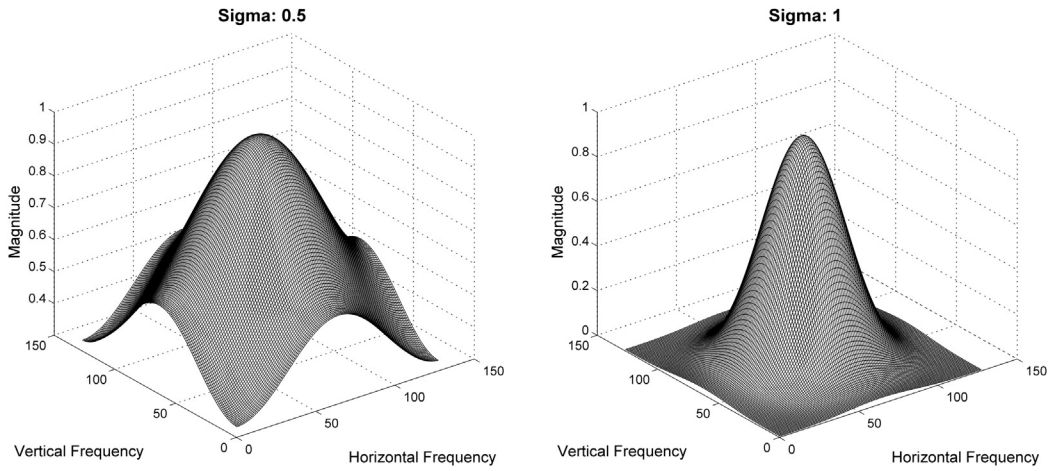

FIGURE 11.8 The magnitude spectra of a Gaussian low-pass filter with two values of σ. The higher value of σ produces a sharper cutoff. The effect of σ, as well as filter dimensions, is examined in the next section and in the problems.

The high-pass filter termed the "Laplacian of Gaussian" filter approximates a spatial second derivative operation: $\partial^2 I/\partial x^2$ and $\partial^2 I/\partial y^2$. A three-by-three version of this filter is shown in Equation 11.10.

$$b[m \cdot n]_{\text{Laplacian}} = \begin{bmatrix} \frac{\alpha}{\alpha+1} & \frac{1-\alpha}{\alpha+1} & \frac{\alpha}{\alpha+1} \\ \frac{1-\alpha}{\alpha+1} & \frac{4}{\alpha+1} & \frac{1-\alpha}{\alpha+1} \\ \frac{\alpha}{\alpha+1} & \frac{1-\alpha}{\alpha+1} & \frac{\alpha}{\alpha+1} \end{bmatrix} \tag{11.10}$$

where the variable α ranges between zero and one and adjusts the steepness of the spectral curve. You might guess that because this filter functions as a high-pass filter, its shape would be roughly the inverse of the low-pass filters shown in Figure 11.8. In the next example, we show that this is indeed the case.

EXAMPLE 11.4

Plot the magnitude of the Laplacian of Gaussian filter for two values of α: 0.2 and 0.8.

Solution: Again, we write a special routine, `lgauss`, to construct the filter coefficient matrix based on Equation 11.10. We then use this routine as in Example 11.3 to determine and plot the magnitude spectra.

```
function h_l = lgauss(alpha)
% Function to design a 3-by-3 Laplacian of Gaussian filter
%
h1 = alpha/(alpha+1);        % Set up end filter coefficients
h2 = (1-alpha)/(alpha+1);    % Center end coefficients
h_lap = [h1 h2 h1; h2 -4/(alpha+1) h2; h1 h2 h1];    % Define coeff. matrix
```

The main routine is as follows:

```
Example 11.4 Plot the magnitude spectrum of Laplacian of Gaussian filter
% for two values of alpha: 0.2 and 0.8.
%
alpha = [0.2 0.8];           % Define values of alpha
for k = 1:length(alpha)
  b = lgauss(alpha(k));      % Get filter coefficients
  F = fft2(b,128,128);       % Calculate magnitude spectrum
  F = abs(fftshift(F));      % Shift spectrum
  subplot(1,2,k);
  mesh(F);
  ......labels, titles, z axis scaling, caxis scaling, colormap
end
```

Results: The spectra produced in this example are shown in Figure 11.9, and the general form is inverted with respect to the low-pass filter spectra shown in Figure 11.8. Again the effect of α on the slope of the spectra is apparent. For this filter, increasing α decreases the spectrum's slope.

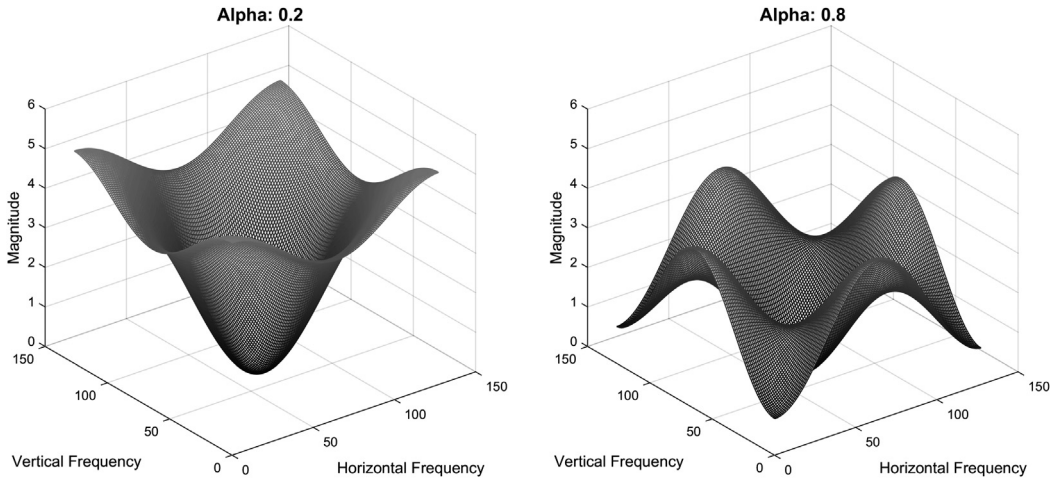

FIGURE 11.9 The magnitude spectra of the Laplacian of Gaussian filter for two different values of the constant α. This filter approximates a second derivative operation on an image. The magnitude spectrum shows that increasing α decreases the upward slope of the spectrum.

The other high-pass filter is the "unsharp" filter, which produces contrast enhancement. The unsharp filter gets its name from an abbreviation of the term "unsharp masking," a double negative that indicates that the unsharp or low spatial frequencies are suppressed (i.e., masked). Suppressing low frequencies is the same as enhancing high frequencies, so this is a high-pass filter. In fact, as shown in one of the problems, the magnitude spectrum of the unsharp filter is almost identical to that of the Laplacian of Gaussian filter, but the phase characteristics are very different. Except for the center coefficient, the unsharp filter coefficients are the negative of the Laplacian of Gaussian filter coefficients. The center coefficient is one minus the center term of the Laplacian of Gaussian filter:

$$b[m \cdot n]_{\text{Laplacian}} = \begin{bmatrix} \dfrac{-\alpha}{\alpha+1} & -\left(\dfrac{1-\alpha}{\alpha+1}\right) & \dfrac{-\alpha}{\alpha+1} \\ -\left(\dfrac{1-\alpha}{\alpha+1}\right) & 1-\left(\dfrac{4}{\alpha+1}\right) & -\left(\dfrac{1-\alpha}{\alpha+1}\right) \\ \dfrac{-\alpha}{\alpha+1} & -\left(\dfrac{1-\alpha}{\alpha+1}\right) & \dfrac{-\alpha}{\alpha+1} \end{bmatrix} \tag{11.11}$$

It is easy to generate this filter in MATLAB from the Laplacian of Gaussian filter:

```
h_unsharp = [0 0 0;0 1 0;0 0 0] - h_laplacian;    % Generate the unsharp filter
```

This code has been added to routine `lgauss`, and this routine outputs the unsharp coefficients as a second argument. The influence of these two filters on images is explored in the next section and in the problems.

11.4.3 Linear Filter Application

Now it is time to apply some of these filters to images, which we do in the next two examples.

EXAMPLE 11.5

Load the magnetic resonance (MR) image of the brain found in `brain1.tif`. Sharpen the image with the unsharp masking filter described earlier. Use an alpha of 0.2.

Solution: Load the image using `importdata` as in Example 11.1. The unsharp masking filter coefficients are determined from the Laplacian of Gaussian filter as described earlier and are found as the second output argument of `lgauss`. Apply these coefficients to the image using `filter2`. Display the image using `image` with the necessary options as in Example 11.1.

```
% Example 11.5 Sharpen an MR image with the unsharp masking filter
%
alpha = 0.2;                        % Unsharp filter alpha
I = importdata('brain1.tif');       % Load image
I = double(I);                      % Convert to double format
[~,b_unsharp] = lgauss(alpha);      % Unsharp filter coefficients (second output)
%
I_unsharp = filter2(b_unsharp,I);  % Filter image
subplot(1,2,1);
  image(I,'CDataMapping','scaled'); caxis([0 255]);          % Display
  colormap(gray); axis off; axis image;                      % Necessary options
  ......title.......
subplot(1,2,2);
  image(I_unsharp,'CDataMapping','scaled'); caxis([0 255]);  % Display
  colormap(gray); axis off; axis image;                      % Necessary options
  title('Unsharp Filter','FontSize',14);
```

Results: The filtered and unfiltered images are displayed in Figure 11.10. The filtered image shows more detail and has sharper, better defined boundaries, but the background neural tissue shows some noise. The unsharp masking filter acts as a high-pass filter, hence it performs a kind of spatial differentiation. We know that 1-D differentiation enhances noise in a signal (see Section 8.5.1), so

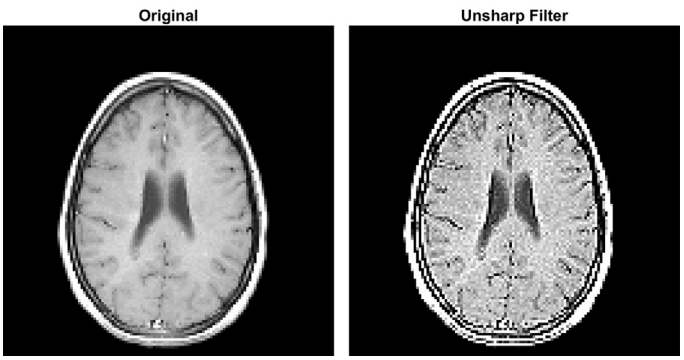

FIGURE 11.10 Left: MR image of the brain. Right: the original image is now sharper after application of the unsharp masking filter in Example 11.5.

it is not surprising that making an image sharper using spatial differentiation enhances noise in the image. This is a traditional engineering compromise. If we wanted to reduce noise, we could low-pass filter the image, but this would lead to some blurring. This is demonstrated in the next example.

EXAMPLE 11.6

Load the image stored as variable `I_noise` in the file `brain_noise.mat`. (Note, `I_noise` is already in `double` format.) This image contains what is known as "salt and pepper" noise. This is the type of noise generated when individual pixels drop out and become either white or black speckles. Apply a low-pass Gaussian filter to reduce the noise. Use a three-by-three coefficient matrix and a sigma of 2.0 (just because it seems to work well). Display the noisy and filtered images side by side.

Solution: Load the image as in Example 11.5 and use routine `gaussian` to get the low-pass filter coefficients. Apply the coefficient to the noisy image and display as in Example 11.5.

```
% Example 11.6 Filter noisy MR image of the brain using a lowpass Gaussian filter
%
N = 3;                              % Filter dimension
sigma = 0.7;                        % Gaussian sigma
load('brain_noise.tif');            % Load image
h_gauss = gaussian(N,sigma);        % Construct Gaussian lowpass filter.
I_lowpass = filter2(h_gauss,I);     % Apply filter

.....display images following same code as in  Example 11.5.....
```

Results: The results are shown in Figure 11.11. The salt and pepper noise in the original figure appear as BW speckles. Although some dark speckles can still be seen in the filtered image, the noise is substantially reduced.

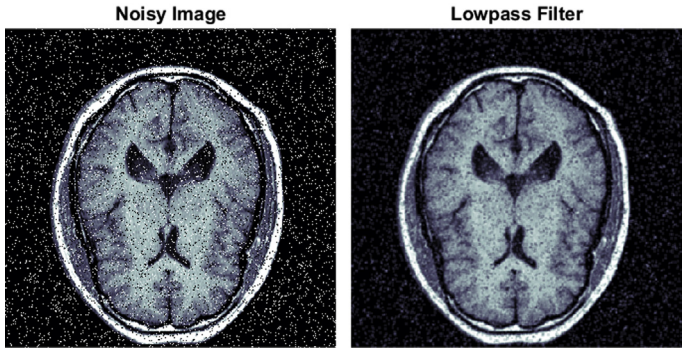

FIGURE 11.11 Left: an MR image of the brain with salt and pepper noise, noise that occurs when individual pixels drop out. Right: the noisy image after filtering with a low-pass Gaussian filter.

Linear filters are also used to aid in identifying physiological structures in medical images. Identifying and isolating regions of particular medical interest, such as bone, organs, or soft tissue structures, is perhaps the greatest challenge in biomedical image analysis. Identifying such ROIs is known as image "segmentation." It is a rich and deep area of biomedical image processing with much ongoing research. In the next section, we explore some basics of image segmentation.

11.5 IMAGE SEGMENTATION

The problems associated with segmentation have been well studied and a large number of approaches have been developed, many specific to particular image features. General approaches to segmentation can be grouped into four classes: pixel-based, edge-based, regional, and morphological methods. Pixel-based methods are the easiest to understand and to implement but are also the least powerful. Because they operate on one element at time, they are also susceptible to noise. Continuity-based and edge-based methods approach the segmentation problem from opposing sides: edge-based methods search for differences, whereas continuity-based methods search for similarities. Morphological methods use information about the shape, properties, and/or mechanical properties of a particular organ or tissue to aid the image segmentation. As these methods tend to be specific to a specific organ, they are not covered in this brief overview.

11.5.1 Pixel-Based Methods

The most straightforward and common of the pixel-based methods is "thresholding" in which all pixels having intensity values above or below some level are classified as part of the segment. Thresholding is also used to convert a grayscale image to a BW (or binary) image. Thresholds can be used in conjunction with other methods to ultimately produce a segmentation mask, a BW template that identifies the region of interest.

Thresholding is usually quite fast and can be done in real time, allowing for interactive adjustment of the threshold. The basic concept of thresholding can be extended to include both upper and lower boundaries, an operation termed "slicing," because it isolates a specific range of pixel intensities. Slicing can be generalized to include any number of different upper and lower boundaries having different intensities.

A technique that can aid in all image analyses, but is particularly useful in pixel-based methods, is intensity remapping. In this global procedure, pixel values are rescaled so as to extend over different maximum and minimum values. Usually the rescaling is linear, so each point is adjusted proportionally with a possible offset. Rescaling operations are easily done in MATLAB using basic arithmetic.

11.5.1.1 Threshold Level Adjustment

The essential problem in pixel-based methods is setting the threshold(s) or slicing level(s) appropriately. Usually these levels are set by the program, although they can be set interactively by the user. Finding an appropriate threshold level can be aided by a plot of

distributions of pixel intensity over the image. Such a plot is termed the "intensity histogram." A histogram was used in Example 1.4 and the MATLAB routine `hist` is described in that example.

Although intensity histograms contain no information on position, they can still be useful for segmentation, particularly for estimating threshold(s) from the histogram. If the intensity histogram is, or is assumed to be, bimodal (or multimodal), a common strategy is to search for low points or minima in the histogram. This histogram-based strategy is used in the next example to aid in setting a threshold.

EXAMPLE 11.7

Load the x-ray image of the spine in the file `spine.tiff`. Plot the intensity histogram and determine a threshold that best separates the spine from the black background. Superimpose the threshold value on the histogram and then display the original image and the thresholded image.

Solution: First we need a routine to threshold the image. Because the thresholded image is a BW image, its intensity values are either 0 or 1.0 as is the convention for BW images. Our `thresh_image` routine takes the image and threshold as input arguments and outputs a BW thresholded image. To accomplish this, the routine first sets up a matrix of zeros the same size as the input image. MATLAB's `find` routine is used to locate input image pixels > `threshold` and set these values to 1.0 in the output matrix.

```
function BW = thresh_image(I,thresh)
% Function to threshold an input image and create a new BW output image
%
BW = zeros(size(I));          % Set up output matrix
BW(find(I > thresh)) = 1;     % Set appropriate locations to 1.0
```

In the main routine, we load the image and use `histogram`[3] to get the intensity histogram. From the histogram, we select and input (manually) a value that seems to divide the black background from the image of the spine. We display the selected value as a line superimposed on the histogram plot. We threshold the image using `thresh_image` and then display the original and thresholded images as in previous examples.

```
% Example 11.7 Example of segmentation using thresholding.
%
I = importdata('spine.tif');                % Load image
I = double(I);                              % Convert to double format
histogram(I);[3] xlim([0 255]); hold on;    % Plot histogram
threshold = input('Input threshold: ');     % Input threshold manually
plot([threshold threshold],[0 40000],'k','LineWidth',2);  % Display threshold
BW = thresh_image(I,threshold);             % Threshold image
.....display images as in previous examples
```

Results: The histogram is shown in Figure 11.12, upper plot. As seen the distribution is bimodal with a large number of 0.0 values originating from the black background. The lowest point between the two distributions occurs around an intensity value of 30–40. A value of 40 was selected and is plotted as a thick line superimposed on the histogram, Figure 11.12, upper plot. The original spine x-ray is shown in the lower left image of Figure 11.12, and the threshold image is shown as the

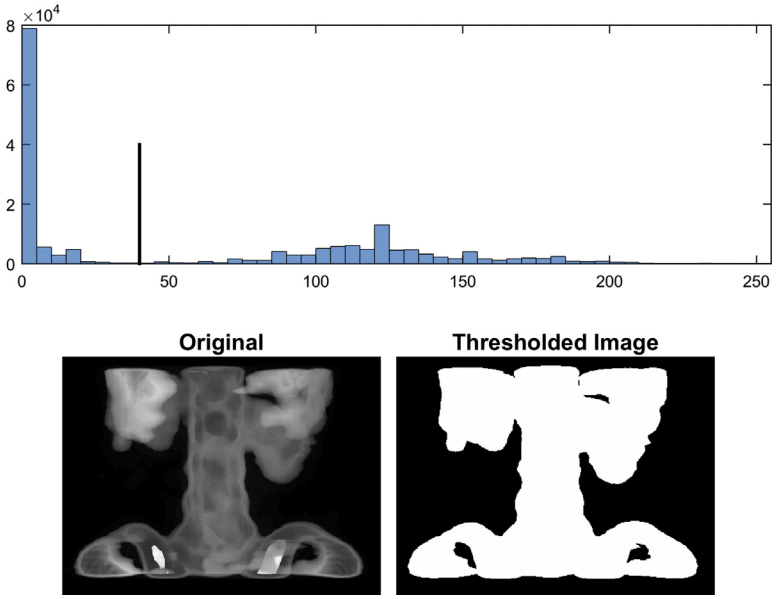

FIGURE 11.12 Upper plot: intensity histogram of the x-ray image of a spine showing a clear bimodal distribution. The lowest point between the two distributions occurs around an intensity value of 30–40, and a value of 40 was selected manually as a threshold value. Lower left: original x-ray image of the spine. Lower right: black and white image after thresholding. *Original image courtesy of MATLAB.*

lower right image. The BW threshold image isolates the spine image and can be used as a "mask" to distinguish this region from the black background.

[3]Older versions of MATLAB may not have the `histogram` function. If `histogram` is not available to you, use `histogrm(I)` found in the associated files instead. This routine first aligns the image into a single row and then calls MATLAB's `hist` routine. As with `histogram`, the second argument is optional and specifies the number of bins. The default value of 60 seems appropriate for most images.

Pixel-based approaches can lead to serious errors, even when the average intensities of the various segments are clearly different, because of noise-induced intensity variations within the structure. Such variations could be acquired during image acquisition but could also be inherent in the structure itself. For example, the left image in Figure 11.13 shows two regions with quite different average intensities. Even with optimal threshold selection, many inappropriate pixels are found in both segments because of intensity variations within the segments, Figure 11.13, right image. Techniques for improving separation in such images are explored in the sections on continuity-based approaches.

11.5.2 Edge Detection

Edge detection methods can be more powerful than pixel-based methods because they can involve multiple criteria. For example, edges might be required to be continuous so small

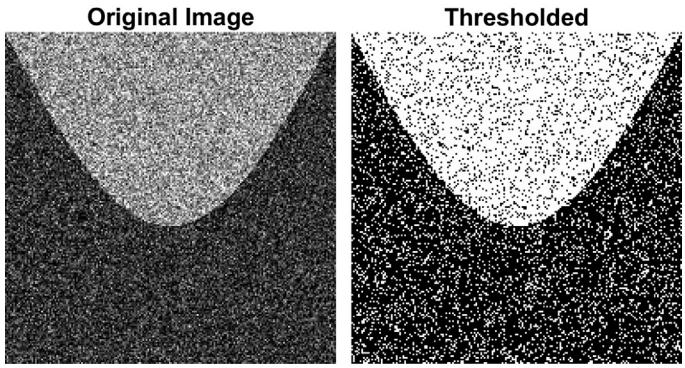

FIGURE 11.13 An image with two regions that have different average gray levels. The two regions are clearly distinguishable, but it is not possible to accurately segment the two regions using thresholding alone because of noise. In the section on continuity-based methods, we separate these two regions.

gaps between edges can be identified and filled. Shape criteria can also be imposed on edges based on the underlying physiology. Edges might be forced to be convex or, in other circumstances, concave. Edges might be constrained to form closed boundaries. For example, the boundary of a red blood cell should be both convex and closed.

Often advanced edge detection routines begin with a set of candidate edges found using more fundamental methods, such as taking the spatial derivatives. Spatial derivatives enhance changes in intensity usually associated with boundaries. In this section, we use the Sobel derivative filter (Equation 11.8) to take the spatial derivative followed by thresholding to isolate the boundary. Figure 11.14, from the next example, shows the Sobel filter applied to an image of blood cells.

The Sobel filter takes the spatial derivative in only one orientation, so it enhances boundaries going from light to dark but only in the vertical direction and only from top to bottom. To enhance all sides of the cell boundaries would require rotating the filter to the three other

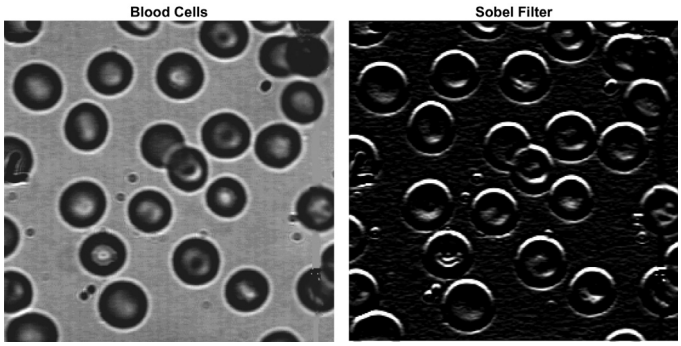

FIGURE 11.14 The right-hand image was generated by applying the Sobel filter defined in Equation 11.8 to the image of blood cells on the left. The boundaries going from light to dark and from top to bottom are enhanced by this filter. To enhance boundaries in other directions would require applying rotated versions of this filter to the image. This is done in Example 11.8. *Image courtesy of MATLAB.*

possible alignments and applying the rotated coefficients to the filter. After thresholding, the result would be four BW images highlighting the boundaries in the four directions. Because BW images are essentially binary images (they consist only of 0's and 1's) we could use the 'or' operation to combine them to get a binary image of the total cell boundary. This is the goal of the next example.

EXAMPLE 11.8

Load the image of blood cells in the file blood.tiff. Apply the Sobel filter to enhance the cell boundaries and determine a threshold that extracts the boundary as a BW, or binary, image. Rotate the Sobel coefficient matrix so that the boundaries in all four directions are highlighted and thresholded. Combine the thresholded images using the 'or' operation to display the combined image.

Solution: Load the blood cell image and display as in previous examples. Generate the Sobel and apply it to the image using `filter2`. The intensity histogram is not so useful in this application, so find the threshold empirically. Examining the right-hand image in Figure 11.14 indicates that the enhanced boundaries are quite light suggesting a high value for the threshold. In fact a threshold of 240, near the maximum value of 255, works well.

To enhance the boundaries in the four directions, it is necessary to rotate the Sobel matrix, and then reapply it to the image and take the threshold of that new filtered image. Rotation can be done by transposing the matrix, flipping the matrix, or a combination of the two as we do here. The four binary boundary images are combined through the 'or' operation and the resultant image displayed.

```
% Example 11.8 Blood cell boundary identification
%
I = importdata('blood1.tif');                % Load image
I = double(I);                               % Convert data
h_sobel = [1 2 1; 0 0 0; -1 -2 -1];          % Sobel filter coefficient
I_sobel = filter2(h_sobel,I);                % Apply Sobel filter
.......display original and filter images using image.m.......
%
I_sobel1 = filter2(h_sobel',I);              % Rotate 90 deg counterclockwise
I_sobel2 = filter2(flipud(h_sobel),I);       % Rotate 180
I_sobel3 = filter2(flipud(h_sobel)',I);      % Rotate 90 clockwise
%
% Threshold all images
threshold = 240;                             % Threshold. Determined empirically
I_sobel = thresh_image(I_sobel,threshold);   % Threshold the filtered images
I_sobel1 = thresh_image(I_sobel1,threshold);
I_sobel2 = thresh_image(I_sobel2,threshold);
I_sobel3 = thresh_image(I_sobel3,threshold);
......display as in previous examples but make caxis([0 1].......
%
BW = I_sobel | I_sobel1 | I_sobel2 | I_sobel3; % Combine the images
......display binary image.......
```

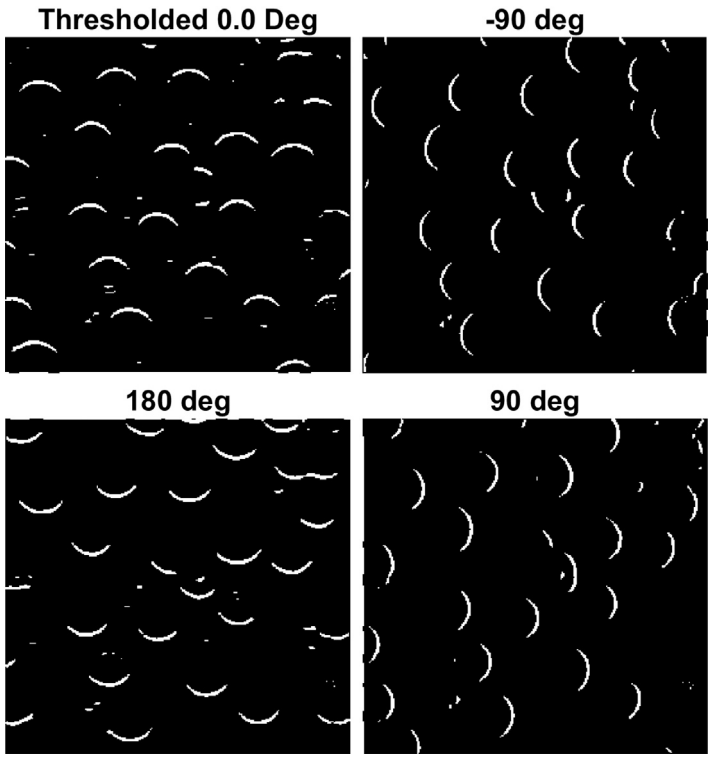

FIGURE 11.15 Four Sobel filtered images were thresholded to produce the BW (binary) images shown here. The filtered images were generated by applying Sobel filters to the original filters in four different orientations. Each orientation enhances the cell boundaries in a specific direction.

Results: The original and one filtered image have already been shown in Figure 11.14. The four thresholded images in Figure 11.15 highlight the cell boundaries in their respective directions. Combining these four images using the 'or' operation produces the image shown in Figure 11.16. The cell boundaries are well delineated. To segment (i.e., isolate) these cells, we need an algorithm

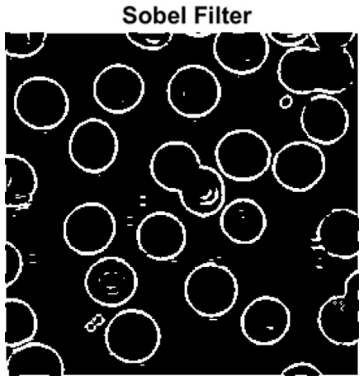

FIGURE 11.16 Edges of the blood cells shown in Figure 11.14, left side. This BW image was made by combining the four images in Figure 11.15 using the logical 'or.'

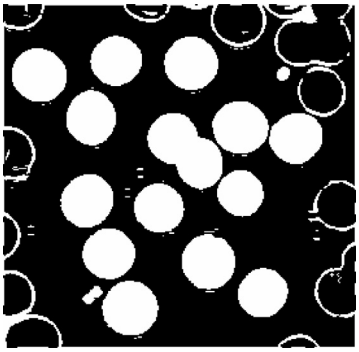

FIGURE 11.17 A binary image constructed by filling the closed blood cell boundaries shown in Figure 11.16. A routine from MATLAB's Image Processing Toolbox, `imfill`, was used to generate this binary image from the boundary outlines. Only boundaries that are complete and unbroken are identified and filled by this routine.

that detects, and marks, the interiors of closed boundaries. MATLAB's Image Processing Toolbox has such a routine called `imfill`. When it is applied to the image in Figure 11.16, it produces the binary image in Figure 11.17 that fills the interior of cells, producing a mask that can be used to isolate the cells. Some cells are not identified by this routine because they have broken or incomplete borders and, thus, are not closed boundaries.

The next section looks at methods based on image similarities. It is possible, and common, to combine different methods to isolate medical features.

11.5.3 Continuity-Based Methods

Continuity-based approaches look for similarity or consistency in the search for feature elements. These approaches can be effective in segmentation tasks, but they all tend to reduce edge definition. This is because they are based on neighborhood operations that operate on a local area and blur the distinction between edge and feature regions. The larger the neighborhood used, the more poorly the edges will be defined. Increasing neighborhood size usually improves the power of any given continuity-based operation, setting up a compromise between identification ability and edge definition.

One simple continuity-based technique is low-pass filtering. Because a low-pass filter is a sliding neighborhood operation that takes a weighted average over a region, it enhances similarities and consistent features. In the next example we use a Gaussian low-pass filter to segment the image in Figure 11.13.

EXAMPLE 11.9

Use a low-pass filter to enhance the similarities of the two regions in Figure 11.13. The pattern is found in the file `texture1.tif`. Plot the histograms of the original and filtered signal, and select a

boundary from the histogram of the filtered signal. Threshold the signal, and display the filtered and thresholded images.

Solution: The image is loaded in the usual way. To differentiate the two regions, we need to filter over a large area, so we use a 10-by-10 Gaussian filter with a σ of 1.5. From the filtered image histogram, we select a threshold that is at a low point between the two distributions. A BW thresholded image is generated using `thresh_image` and displayed.

```
% Example 11.9 Demonstrates the effectiveness of simple linear filtering
% in separating two segments containing noise.
%
I = importdata('texture1.tif');              % Load image
I = double(I);                               % Convert data
b = gaussian(10, 1.5);                       % Gaussian lowpass filter
I_lowpass = filter2(b,I);                    % Apply Gaussian filter
% Plot histograms
subplot(2,1,1);
  histogram(I);
  ......title and labels.......
subplot(2,1,2);
  histogram(I_lowpass);
  ......title and labels.......
threshold = input('Input threshold: ');      % Input threshold (approx.. 110)
BW = thresh_image(I_lowpass,threshold);      % Apply threshold
......new figure; display images; add titles.......
```

Results: The histograms are shown in Figure 11.18. The effect of low-pass filtering is evident from the histogram. The histogram of the original signal has a unimodal distribution, whereas the histogram of the filtered signal is bimodal, reflecting the two regions. The low point between the two distributions occurs at an intensity of around 110. Thresholding the filtered image at this value results in perfect separation of the two regions. The filtered image and the BW image separating the two regions are shown in Figure 11.19. This example shows the power of simple low-pass filtering to identify ROIs based on similarities.

11.5.3.1 Texture Analysis and Nonlinear Filtering

Image features related to "texture" can be particularly useful in segmentation. Figure 11.20 shows three regions that have approximately the same average intensity values but are distinguished visually because of differences in texture. Several neighborhood-based operations can be used to distinguish textures: the small segment Fourier transform, local variance (or standard deviation), the "Laplacian" operator, the "range" operator (the difference between maximum and minimum pixel values in the neighborhood), the "Hurst" operator (maximum difference as a function of pixel separation), and the "Haralick" operator (a measure of distance moment). These operations are neighborhood operations, but, unlike linear filtering, they involve nonlinear processes. The Image Processing Toolbox has a routine, `nlfilter`, that can implement some of these operations, but it is not too difficult to code some of these features in standard MATLAB.

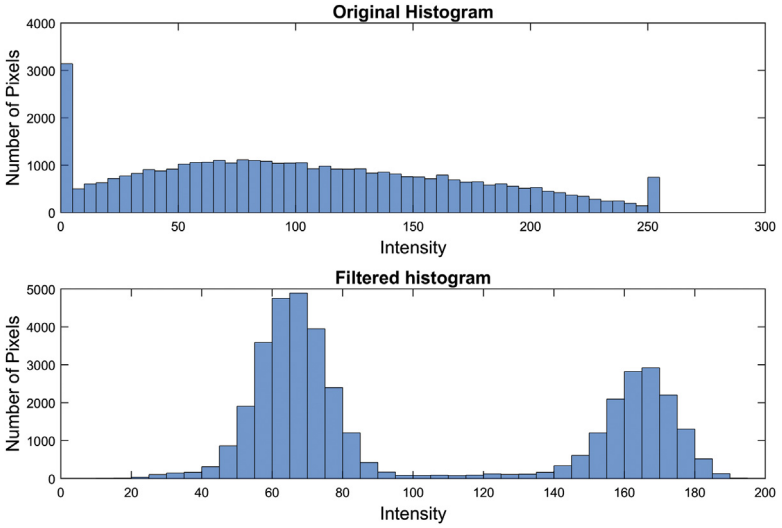

FIGURE 11.18 Histogram of the image shown in Figure 11.13 before (upper) and after (lower) low-pass filtering. Before filtering, the two regions overlap to such an extent that they cannot be identified in the histogram. After low-pass filtering, the two regions are evident in the bimodal distribution. An intensity value around 110 is a low point between the two distributions. Thresholding the filtered image at this value results in perfect separation as shown in Figure 11.19.

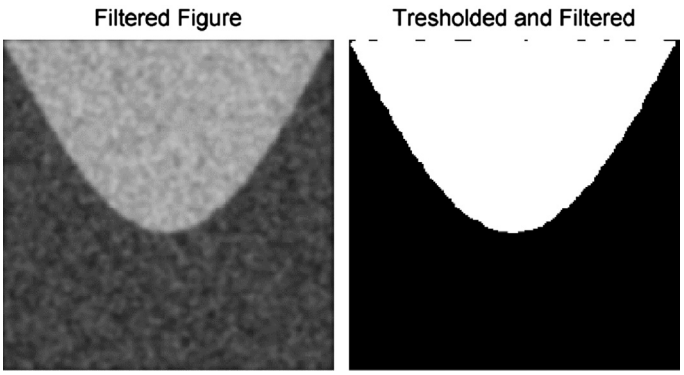

FIGURE 11.19 Left: the same image as in Figure 11.13 after low-pass filtering. Right: the two regions can now be separated perfectly by thresholding.

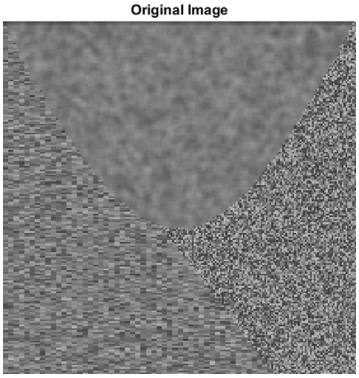

FIGURE 11.20 An image containing three regions having approximately the same intensity but different textures. Although these areas can be distinguished visually, separation based on intensity or edges will surely fail.

EXAMPLE 11.10

Separate out the three segments in Figure 11.20 that differ only in texture. The image is found in the file `texture3.tif`. Use one of the texture operators described above, the "range" operator, and demonstrate the improvement in separability through histogram plots. Determine appropriate threshold levels for the three segments from the histogram plot, and plot the segments.

Solution: First we need a program to code the nonlinear filter. The function `nonlfilter` takes in an image and produces a nonlinearly filtered output image of the same size. The routine first extends the horizontal and vertical extent of the original image with zero padding to take care of edge effects. Then a sliding neighborhood operation is applied to compute the range operation. The program defines an ROI (variable `roi`) in the image corresponding to the filter's dimensions. The range operator value is determined as the difference in maximum and minimum values within this ROI. The ROI is then shifted one pixel position, first horizontally then vertically. At each position, a new range value is computed into the output image matrix.

Because we have done all this work to isolate the ROIs, we might as well throw in some possible other nonlinear computations. Adding the input argument `type`, we can specify a standard deviation operation (`'std'`) and a variance operation (`'var'`) in addition to the range operation (`'range'`). Because the ROI is 2-D, we need to use the MATLAB commands `std2` and `var2` to compute the regional standard deviation and variance.

```
function I1 = nonlfilter(I,type,m,n)
% Nonlinear filtering. m and m specify filter size.
%
[M,N] = size(I);                % Original image size
h_pad =ceil( (n-1)/2);          % Padding dimensions
v_pad = ceil((m-1)/2);          % m and n must be odd
I = [zeros(v_pad,N); I; zeros(v_pad,N)];   % Pad top and bottom
[M1,N1] = size(I);
I = [zeros(M1,h_pad), I, zeros(M1,h_pad)]; % Pad left and right
%
for n1 = 1:N                            % Isolate a region of interest
   for m1 = 1:M
      roi = I(m1:m1+m-1,n1:n1+n-1);     % Define region of interest
      if type(1) == 's'
        I1(m1,n1) = std2(roi);          % Apply operations standard deviation
      elseif type(1) == 'v'
        I1(m1,n1) = var(roi);           % Variance
      elseif type(1) == 'r'
        I1(m1,n1) = max(max(roi)) - min(min(roi)); % Range operator
      end
   end
end
```

Next we use the nonlinear range operator to convert the textural patterns into differences in intensity. The range operator is a sliding neighborhood procedure that sets the center pixel to the difference between the maximum and minimum pixel value found within the neighborhood. We implement this operation in the main program over a seven-by-seven neighborhood. This neighborhood size was found empirically to produce good results.

The three regions are then thresholded using thresholds determined from the histograms. The right-side texture is segmented by an upper threshold. The upper texture is segmented by inverting the BW image and applying the lower threshold to this inverted image. Because these images are binary, inverting can be accomplished using the logical 'not' operation represented in MATLAB by the symbol ~. The remaining texture can be found using a logical combination (the 'and' operation) applied to the two isolated images after inverting.

```
% Example 11.10 Texture analysis using nonlinear filtering
%
M = 7;       % Define filter neighborhood size, horizontal
N = 7;       % and vertical
I = importdata('texture3.tif');      % Load image
I = double(I);                       % Convert data
I_nl = nonlfilter(I,'range',M,N);    % Apply nonlinear filter
% Plot histograms
subplot(2,1,1);
  histogram(I);
.....title and labelsl.......
  histogram(I_nl);
  .....title and labelsl.......
thresh1 = input('Input lower threshold: ');  % Get upper and lower threshold
thresh2 = input('Input upper threshold: ');
BW_upper = ~thresh_image(I_nl,thresh1);      % Isolate upper image (note invert)
BW_right = thresh_image(I_nl,thresh2);       % Isolate right image
BW_left = ~I_upper & ~I_right;               % Isolate left from other two
```

Results: The image produced by the range filter is shown in Figure 11.21, and a clear distinction in intensity level can now be seen between the three regions. This is also demonstrated in the histogram plots of Figure 11.22. The histogram of the original figure (upper plot) shows a single Gaussian-like distribution with no evidence of the three patterns.[4] After nonlinear filtering, the three patterns emerge as three distinct distributions. Using this distribution, two thresholds are chosen where the minimum value occurs between the distributions: 50 and 110.

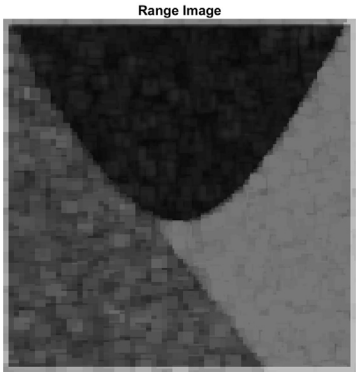

FIGURE 11.21 The texture pattern shown in Figure 11.20 after application of the nonlinear range operation. This operator converts the textural properties in the original figure into a difference in intensities. The three regions are now clearly visible as intensity differences and can be isolated using thresholding.

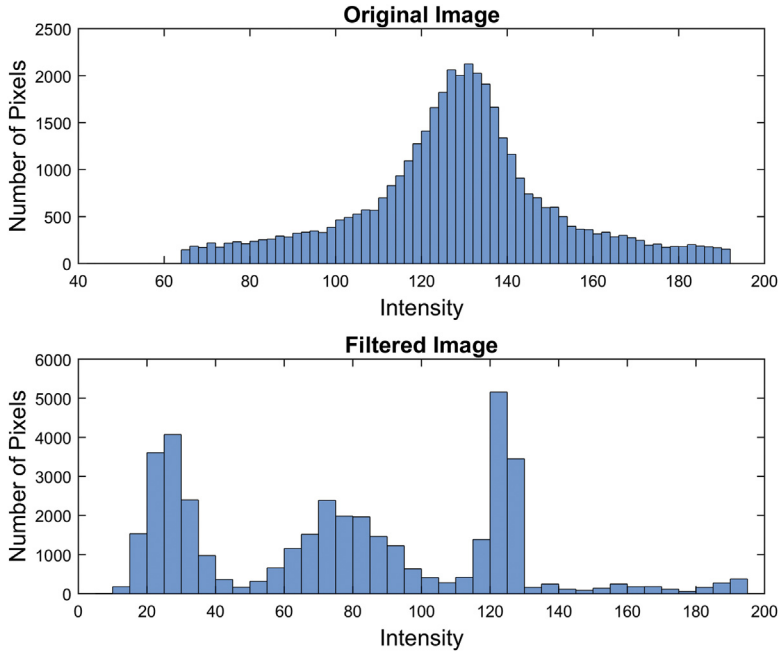

FIGURE 11.22 Histogram of original texture pattern before (upper) and after nonlinear filtering using the *range* operator (lower). After filtering, the three intensity regions are seen as three different distributions. The thresholds used to isolate the three segments are 50 and 110.

FIGURE 11.23 Isolated regions of the texture pattern in Figure 11.20. Although there are some artifacts, the segmentation is quite good considering the original image. Low-pass filtering and other techniques can be used to improve segmentation as shown in the problems.

The three segments are isolated based on these thresholds in conjunction with logical operations. The three fairly well-separated regions are shown in Figure 11.23. A few artifacts remain in the isolated images. The separation can also be improved by applying low-pass filtering to the range image and other minor modifications as demonstrated in the problems.

[4]In fact, the distribution is Gaussian because the image patterns were generated by filtering an array filled with Gaussainly distributed numbers generated by `randn`.

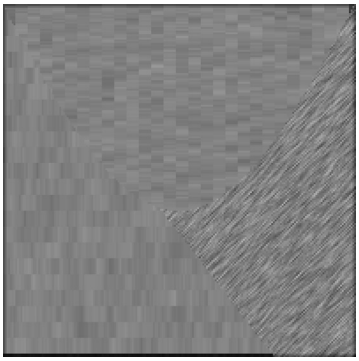

FIGURE 11.24 Textural pattern used in Problem 18. The upper and left diagonal features have similar statistical properties and would not be separable with a standard range operator. However, they do have different orientations. As in Example 11.10, all three features have the same average intensity.

Occasionally, segments have similar intensities and textural properties, except that the texture differs in orientation. Such patterns can be distinguished using a variety of nonlinear operators that have orientation-specific properties. The local Fourier transform can also be used to distinguish orientation. Figure 11.24 shows a pattern with texture regions that are different only in terms of their orientation. This figure is segmented into the three texture-specific features in one of the problems. When textures differ in orientation, a direction-specific operator followed by a low-pass filter is used.

11.6 SUMMARY

An image can be considered as a 2-D signal and many signal processing tools can be extended to images. MATLAB has routines that implement many signal processing operations on 2-D data, including the Fourier transform, convolution, and correlation (filtering). A single image sample, or pixel, can be considered one element of a 2-D array and is referenced by its row and column location in the array. When converted to the frequency domain, an image spectrum is a function of two spatial frequencies. Images usually contain a large number of pixels, so they are often stored using only one or two bytes per pixel, but they can be converted to MATLAB's `double` format and treated as standard MATLAB variables.

Linear image filtering is done using FIR filters where a filter coefficients matrix, $b[k_1,k_2]$, is applied to the image using a modified form of 2-D convolution. Although the filter matrix can be of any size, many image filters use a three-by-three coefficients matrix. Popular filter matrices include the Sobel and Prewitt filters for edge detection, the Gaussian filter for low-pass filtering, the Laplacian of Gaussian filter that approximates a spatial second derivative, and the unsharp (or unsharp masking) filter for high-pass filtering.

Separating out important sections of an image is known as image segmentation and is often used to identify ROIs such as particular organs in a medical image. Image segmentation approaches can be grouped into four classes: pixel-based, edge-based, regional, and

morphological methods. Pixel-based methods operate on one pixel at a time and are easy to implement but are not as powerful as the other methods. A common pixel-based method is thresholding, which works well if the ROI is brighter or darker than the surrounding tissue. Multiple thresholds can be applied if the ROI has a specific range of intensity values. An intensity histogram that shows the distribution of intensity values in an image can be helpful in setting the threshold.

Edge detection methods search for intensity differences, i.e., edges, and usually use multiple criteria to outline the ROI. For example, edges can be forced to be continuous and/or conform to a specific shape or be constrained to form closed loops. Continuity-based methods take the opposite tack: they search for similarities with the ROI. One of the most common continuity-based methods is low-pass filtering. Continuity-based methods also use nonlinear filters that enhance specific textural features in the ROI. Finally, morphological methods use information about the shape, properties, and/or mechanical properties of a particular organ or tissue to aid the image segmentation.

PROBLEMS

1. Load the image of the brain in the file `brain2.tif`. Brighten the image by multiplying it by 1.4. Construct a new image that is the same as the original except that any point between 10 and 120 is white. Finally, construct a new image that is the same as the original except that any point that is less than 120 or greater than 215 is black. This operation tends to highlight the gray matter in the brain. Plot the original and three new images together using `subplot`.
2. Load the MATLAB test pattern image found in `testpat1.png`. Plot the image along with a plot of the magnitude Fourier transform of this image. Use `mesh` to plot the Fourier transform and shift the transform before plotting using `fftshift`. (Note `fftshift` shifts only valid points, i.e., $f_s/2$, so you do not need to restrict the size of the Fourier transform when shifting.) Then eliminate the mean in the Fourier transform by shifting only the second data point to the end, i.e., shift `F(2:end,2:end)`. (Recall, the first point of the Fourier transform contains the mean or average value.) Note the greater detail visible in the spectrum when you eliminate the DC (i.e., zero frequency) component. Nonetheless, the spectrum of this fairly simple image is still difficult to interpret.
3. Load the image in `double_chirp.tif`, which has a chirp going both horizontally and vertically. Plot this image and take the magnitude Fourier transform. As in Problem 2, use `fftshift` to shift the spectrum and `mesh` to plot. The magnitude spectrum is dominated by the DC component and is difficult to interpret. Plot the shifted Fourier transform after removing the DC component from the magnitude spectrum (see Problem 2). Because this is a chirp both horizontally and vertically, we would expect to see a relatively constant spectrum in both spatial frequency directions producing a plateau-like structure, as indeed we do.
4. Plot the magnitude spectrum of the Sobel and Prewitt filters. The spectrum of each filter shows two positive peaks because you are plotting the magnitude spectra. The phase spectra would show that one of the peaks is actually inverted. Although the two spectra are very similar, there are differences at the higher spatial frequencies.

5. Plot the magnitude and phase spectrum of the Laplacian of Gaussian and the unsharp masking filters. You can use routine `lgauss` to get the coefficients of both of these filters. (The first output argument is Laplacian of Gaussian coefficients and the second the unsharp masking coefficients.) Make $\alpha = 0.5$. Note that the two magnitude spectra are almost identical; however, the phase characteristics are very different. Use `subplot` to plot the magnitude and phase together. (The phase plots are easier to read if you change to view to Az = 30, El = 35, i.e., `view([30,35]). Also note the phase wrapping in the phase plots;`).
6. Plot the magnitude and phase spectrum of a three-by-three Gaussian low-pass filter and a three-by-three averaging filter, i.e., nine equal weights of 1/9. Use routine `gaussian` to get the Gaussian filter coefficients, but construct the averaging filter yourself. (Suggestion: Use MATLAB's `ones` operator.) Use a value of sigma of 0.5. Observe the complicated phase spectrum of the averaging filter. Also observe that, although the averaging filter has a sharper cutoff, it has what are known as sidelobes. The next problem compares the operation of the two filters. Use subplot to plot the magnitude and phase together. (The phase plot is easier to read if you change to view to Az = 30, El = 35; i.e., `view([30,35])`; Again note the phase wrapping in the phase plots, particularly for the averaging filter.)
7. Load the image `blood2.tif`, which is an image of blood cells that contains salt and pepper noise. Apply a five-by-five Gaussian low-pass filter and a five-by-five averaging filter. Adjust the Gaussian filter's σ to maximally reduce the noise without making the image of the cell too blurry. Display the original and both filtered images together. Note that, although the ability of both filters to reduce the salt and pepper noise is similar, the Gaussian filter produces a slightly sharper image. (Hint: Use MATLAB's `ones` operator to construct the averaging filter.)
8. Load the image of blood cells found in `blood2.tif`. This image has added salt and pepper noise. Filter this image with two Gaussian low-pass filters of different strengths. A three-by-three filter with $\sigma = 0.5$ and a five-by-five filter with $\sigma = 3.0$. Also plot the magnitude spectra of the two filters. Display the original and both filtered images together.
9. Compare the Laplacian of Gaussian filter with the related unsharp masking filter. Apply both to the MR image of the brain in `brain1.tif`, the image used in Example 11.5. Compare the images from the two filters side by side. The Laplacian of Gaussian filter takes the second derivative so that the output image has both positive and negative values. Hence, when displaying this image set the colormap to range between ± 255 using `caxis([-255 255]);`. The background will be gray, not black. The only features in this image are found where the most severe boundaries occur in the original image. Display the original and two filtered images.
10. Load the image `noise_brain2.tif`, which is an MR image of the brain that contains Gaussian noise. Apply a Gaussian low-pass filter that reduces the noise. Adjust filter dimension and σ to maximally reduce the noise without making the image of the cell too blurry. You should be able to get a clean image. Display the original and filtered images.
11. Modify Example 11.8 to apply the four versions of the Sobel filter to the image of bacteria in the file `bacteria.tif`. Plot the histogram of one of the Sobel filtered images and determine the threshold. (Hint: Look for a notch somewhere below the 50%

maximum intensity level.) Apply `thresh_image` to get an outline of the bacteria cells. Plot the original image and its histogram together, the four Sobel filtered images together, and the final combined image. You should get a good outline of the cells. (Recall that if you do not have MATLAB's `histogram` you can use `histogrm` found in the associated files.)

12. Isolate the bacterial cells in `bacteria.tiff` using a Laplacian of Gaussian filter. Use `lgauss` to generate the filter coefficients and then filter the image with `filter2`. The resulting image has very low spatial contrast, and because this filter calculates the second spatial derivative of the image the mean value of the filtered image is near 0.0. Display this image using a scaled-down colormap to enhance the contrast (suggestion: `caxis([ -4 , 40])`. The histogram is not so useful in determining the threshold from this filtered image. However, you know the threshold will be near zero, so try a small positive threshold (between 4 and 8) to isolate the features in the cell. Display the original and filtered images as well as thresholded BW image.

13. Isolate the cell in `cell.tif` using multiple thresholding. This problem uses multiple thresholds to partially isolate a changeling image. The next problem completes the segmentation task. Again the histogram is not useful in thresholding this image, but from the image you can see that the cells appear to be slightly lighter and darker than the gray background. This suggests two thresholding operations. First, threshold the image slightly above 128 (i.e., 50% of the maximum value) to isolate the lighter sections of the cell. Then threshold a grayscale inverted version of this image somewhat below 128 to isolate the darker sections of the cell. You can combine the two BW images, but you need to invert the second image after thresholding so the areas identified as darker are now white. (You can invert the BW image using the not (~) operation and combine using the or (|) because these can be treated as logical variables, but to invert the original grayscale image you need to subtract it from 255 as in Example 11.1.) The resulting image will have some dark areas remaining inside the cell, but these are removed in the next problem. Plot the original images, the two thresholded images, and the combined images. You should be able to generate an image similar to that stored in `Prob13_data.mat`.

14. The variable BW3 in the file `Prob13_data.mat` is a BW image that was obtained by double thresholding an image of bacterial cells in Problem 13. To remove the remaining material inside the cells, apply a Gaussian low-pass filter and then rethreshold the filtered image. Note that the filtered image is between 0 and 1, so the threshold should be very small (<0.1). Use the weakest filter that completely whitens the cell, i.e., the lowest dimension and smallest σ. Display the original and segmented images.

15. Load the blood cell image in `blood1.tif`. Filter the image with two Gaussian low-pass filters having a dimension of 20. One filter should have a gentle spectral slope (for example, $\sigma = 0.5$) and the other should have a sharper slope ($\sigma = 4.5$). Threshold the two filtered images based on the histogram, but adjust for optimum images, i.e., minimum background spots with relatively unbroken cell walls. (Suggestion: The threshold of the image produced by the weaker filter can be about 10%−30% lower than that used for the image from the stronger filter.) Display the filtered images along with their histograms. Also display the thresholded images side by side. (Hint: Good thresholds can be found around 50% of maximum intensity (128).)

16. Repeat Problem 11.15 for the original image shown in Figure 11.13 and found in the file `Prob11_16data.tif`. The histogram provides guidance on thresholds for both filtered images. Note the difference between the two histograms. Also note that the histogram of the weaker filter indicates that the segments cannot be perfectly separated but that complete segmentation is possible with the image from the stronger filter. The downside is that the border is less precisely defined in the heavily filtered image.
17. Recall the Laplacian of Gaussian filter that calculates the second derivative. It can be useful to highlight texture created by small edges in the image. Load the image of the spine found in `spine.tif`. Filter this image using the Laplacian of Gaussian filter obtained from `lgauss` with $\alpha = 0.5$. Then threshold this image. In this image the edges are located where the second derivative is near zero, so you want to take a low threshold (around 0). Note the extensive tracing of subtle boundaries that this filter produces.
18. In this problem we segment the complicated texture image shown as a teaser in Figure 11.24 and found in file `texture4.tif`. We again apply the range nonlinearity but with a nonsquare region of interest to isolate areas that have different orientations. In particular, we make `m` $= 9$ and `n` $= 1$ in our call to `nonlfilter`: `nonlfilter(I,'range',9,1)`. Also apply an additional Gaussain low-pass filter with a dimension of 9 and $\sigma = 3$. Plot the two filtered images and the histogram for each. The histograms show the improvement in the ability to separate the textures because of low-pass filtering. Although the range and low-pass filtered histogram provides guidance on the placement of thresholds, you may be able to improve separation by adjusting the thresholds. You should be able to achieve almost the same level of separation as in Figure 11.23. Submit the original and segmented images.
19. The last three problems explore methods to improve the separation of the complex textures shown in Figures 11.20 and 11.24. Load the texture orientation image used in Problem 18 and shown in Figure 11.24 that is found in `texture4.tif`. Separate the segments by first preprocessing the image with a Sobel filter that enhances horizontal edges that are brighter on top. Then apply a nonlinear filter using a standard deviation operation determined over a two-by-nine pixel grid. Finally postprocess the image from the nonlinear filter using a strong Gaussian low-pass filter. (Note that you should multiply the output of the nonlinear filter by around 3.5 before applying the Gaussian filter to get it into an appropriate range.) Plot the output image of the nonlinear filter and that of the low-pass-filtered image and the two corresponding histograms. Separate into three segments as in Example 11.10. Use the histogram of the low-pass-filtered image to determine the best boundaries for separating the three segments, although you may need to adjust the thresholds for best separation. Display the three segments as white objects. You should be able to get clean separation for two of the three textures. Submit the original and segmented images.
20. Now we need to improve the separation of the texture image used in Example 11.10 with a low-pass filter. Follow the same procedure used in Example 11.10. Load the image in the file `texture3.tif` and apply the range nonlinearity with a seven-by-seven ROI. Then filter the nonlinear image with a Gaussian low-pass filter. This is the same strategy used in Problem 19 with the textures found in `texture4.tif`. As in Problem 19 you should be able to get perfect isolation of the left and center images.

Set the low-pass filter's dimensions and σ as low as possible yet still with good separation. Submit the original and segmented images.

21. Next we improve the separation of the texture image used in Example 11.10 and Problem 20. Follow the same procedure as in Problem 20 using the same thresholds. Before displaying the three BW images, low-pass filter BW3 using the same Gaussian filter that was applied to the image from the nonlinear filter. Again threshold this low-pass-filtered image to produce an image that affords perfect separation. (Hint: For this last image you need a high threshold, but remember this filtered image ranges between 0 and 1, so >0.85.)

A caveat on this and other segmentation problems in this chapter: Although the relatively simple methods we use here achieve good image separations, they may not be good examples of segmentation. The problem is that when we use low-pass and other filters, the borders we obtain do not precisely align with the edges of the tissue we want to isolate. That is, the segments we have identified in these problems do not exactly overlap the tissue (or texture) of interest. Greater care in identifying boundaries is required and more advanced approaches exist for boundary detection of tissue segment. Image segmentation is a very challenging area of biomedical engineering and, as mentioned previously, an active research area. Nonetheless, these problems do give a good introduction to the tasks and techniques of biomedical image analysis.