5 Image Restoration

Preview

The objective of restoration is to improve a given image in some predefined sense. Although there are areas of overlap between image enhancement and image restoration, the former is largely a subjective process, while image restoration is for the most part an objective process. Restoration attempts to reconstruct or recover an image that has been degraded by using a priori knowledge of the degradation phenomenon. Thus, restoration techniques are oriented toward modeling the degradation and applying the inverse process in order to recover the original image.

This approach usually involves formulating a criterion of goodness that yields an optimal estimate of the desired result. By contrast, enhancement techniques basically are heuristic procedures designed to manipulate an image in order to take advantage of the psychophysical aspects of the human visual system. For example, contrast stretching is considered an enhancement technique because it is based primarily on the pleasing aspects it might present to the viewer, whereas removal of image blur by applying a deblurring function is considered a restoration technique.

In this chapter we explore how to use MATLAB and IPT capabilities to model degradation phenomena and to formulate restoration solutions. As in Chapters 3 and 4, some restoration techniques are best formulated in the spatial domain, while others are better suited for the frequency domain. Both methods are investigated in the sections that follow.

5.1 A Model of the Image Degradation/Restoration Process

As Fig. 5.1 shows, the degradation process is modeled in this chapter as a degradation function that, together with an additive noise term, operates on an input image $f(x, y)$ to produce a degraded image $g(x, y)$:

$$g(x, y) = H[f(x, y)] + \eta(x, y)$$

Given $g(x, y)$, some knowledge about the degradation function $\mathcal{H}$, and some knowledge about the additive noise term $\eta(x, y)$, the objective of restoration is to obtain an estimate, $\hat{f}(x, y)$, of the original image. We want the estimate to be as close as possible to the original input image. In general, the more we know about H and η, the closer $\hat{f}(x, y)$ will be to $f(x, y)$.

If H is a *linear, spatially invariant* process, it can be shown that the degraded image is given in the *spatial domain* by

$$g(x, y) = h(x, y) * f(x, y) + \eta(x, y)$$

where $h(x, y)$ is the spatial representation of the degradation function and, as in Chapter 4, the symbol "$*$" indicates convolution. We know from the discussion in Section 4.3.1 that convolution in the spatial domain and multiplication in the frequency domain constitute a Fourier transform pair, so we may write the preceding model in an equivalent *frequency domain* representation:

Following convention, we use an in-line asterisk in equations to denote convolution and a superscript asterisk to denote the complex conjugate. As required, we also use an asterisk in MATLAB expressions to denote multiplication. Care should be taken not to confuse these unrelated uses of the same symbol.

$$G(u, v) = H(u, v)F(u, v) + N(u, v)$$

where the terms in capital letters are the Fourier transforms of the corresponding terms in the convolution equation. The degradation function $H(u, v)$ sometimes is called the *optical transfer function* (OTF), a term derived from the Fourier analysis of optical systems. In the spatial domain, $h(x, y)$ is referred to as the *point spread function* (PSF), a term that arises from letting $h(x, y)$ operate on a point of light to obtain the characteristics of the degradation for any type of input. The OTF and PSF are a Fourier transform pair, and the toolbox provides two functions, `otf2psf` and `psf2otf`, for converting between them.

`otf2psf`
`psf2otf`

Because the degradation due to a linear, space-invariant degradation function, H, can be modeled as convolution, sometimes the degradation process is referred to as "convolving the image with a PSF or OTF." Similarly, the restoration process is sometimes referred to as *deconvolution.*

In the following three sections, we assume that H is the identity operator, and we deal only with degradation due to noise. Beginning in Section 5.6 we look at several methods for image restoration in the presence of both H and η.

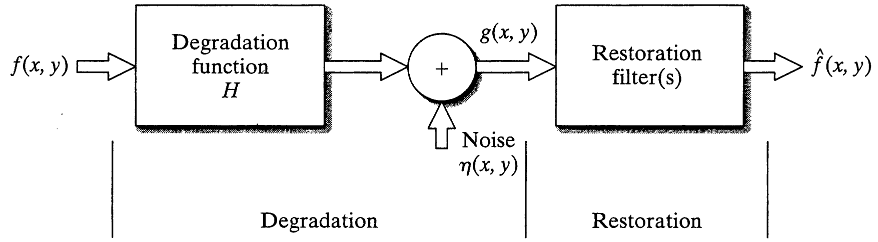


FIGURE 5.1 A model of the image degradation/restoration process.

5.2 Noise Models

The ability to simulate the behavior and effects of noise is central to image restoration. In this chapter, we are interested in two basic types of noise models: noise in the spatial domain (described by the noise probability density function), and noise in the frequency domain, described by various Fourier properties of the noise. With the exception of the material in Section 5.2.3, we assume in this chapter that noise is independent of image coordinates.

5.2.1 Adding Noise with Function imnoise

The toolbox uses function imnoise to corrupt an image with noise. This function has the basic syntax

```
g = imnoise(f, type, parameters)
```

imnoise

where f is the input image, and type and parameters are as explained later. Function imnoise converts the input image to class double in the range [0, 1] before adding noise to it. This must be taken into account when specifying noise parameters. For example, to add Gaussian noise of mean 64 and variance 400 to an uint8 image, we scale the mean to 64/255 and the variance to $400/(255)^2$ for input into imnoise. The syntax forms for this function are:

- g = imnoise(f, 'gaussian', m, var) adds Gaussian noise of mean m and variance var to image f. The default is zero mean noise with 0.01 variance.
- g = imnoise(f, 'localvar', V) adds zero-mean, Gaussian noise of local variance, V, to image f, where V is an array of the same size as f containing the desired variance values at each point.
- g = imnoise(f, 'localvar', image_intensity, var) adds zero-mean, Gaussian noise to image f, where the local variance of the noise, var, is a function of the image intensity values in f. The image_intensity and var arguments are vectors of the same size, and plot(image_intensity, var) plots the functional relationship between noise variance and image intensity. The image_intensity vector must contain normalized intensity values in the range [0, 1].
- g = imnoise(f, 'salt & pepper', d) corrupts image f with salt and pepper noise, where d is the noise density (i.e., the percent of the image area containing noise values). Thus, approximately d*numel(f) pixels are affected. The default is 0.05 noise density.
- g = imnoise(f, 'speckle', var) adds multiplicative noise to image f, using the equation g = f + n*f, where n is uniformly distributed random noise with mean 0 and variance var. The default value of var is 0.04.
- g = imnoise(f, 'poisson') generates Poisson noise from the data instead of adding artificial noise to the data. In order to comply with Poisson statistics, the intensities of uint8 and uint16 images must correspond to the number of photons (or any other quanta of information). Double-precision images are used when the number of photons per pixel is larger than 65535

(but less than 10^{12}). The intensity values vary between 0 and 1 and correspond to the number of photons divided by 10^{12}.

Several illustrations of `imnoise` are given in the following sections.

5.2.2 Generating Spatial Random Noise with a Specified Distribution

Often, it is necessary to be able to generate noise of types and parameters beyond those available in function `imnoise`. Spatial noise values are random numbers, characterized by a probability density function (PDF) or, equivalently, by the corresponding cumulative distribution function (CDF). Random number generation for the types of distributions in which we are interested follow some fairly simple rules from probability theory.

Numerous random number generators are based on expressing the generation problem in terms of random numbers with a uniform CDF in the interval $(0, 1)$. In some instances, the base random number generator of choice is a generator of Gaussian random numbers with zero mean and unit variance. Although we can generate these two types of noise using `imnoise`, it is more meaningful in the present context to use MATLAB function `rand` for uniform random numbers and `randn` for normal (Gaussian) random numbers. These functions are explained later in this section.

The foundation of the approach described in this section is a well-known result from probability (Peebles [1993]) which states that if w is a uniformly distributed random variable in the interval $(0, 1)$, then we can obtain a random variable z with a specified CDF, F_z, by solving the equation

$$z = F_z^{-1}(w)$$

This simple, yet powerful, result can be stated equivalently as finding a solution to the equation $F_z(z) = w$.

EXAMPLE 5.1: Using uniform random numbers to generate random numbers with a specified distribution.

■ Assume that we have a generator of uniform random numbers, w, in the interval $(0, 1)$, and suppose that we want to use it to generate random numbers, z, with a Rayleigh CDF, which has the form

$$F_z(z) = \begin{cases} 1 - e^{-(z-a)^2/b} & \text{for } z \geq a \\ 0 & \text{for } z < a \end{cases}$$

To find z we solve the equation

$$1 - e^{-(z-a)^2/b} = w$$

or

$$z = a + \sqrt{b \ln(1 - w)}$$

Because the square root term is nonnegative, we are assured that no values of z less than a are generated. This is as required by the definition of the Rayleigh CDF. Thus, a uniform random number w from our generator can be used in the previous equation to generate a random variable z having a Rayleigh distribution with parameters a and b.

In MATLAB this result is easily generalized to an $M \times N$ array, R, of random numbers by using the expression

```
>> R = a + sqrt(b*log(1 – rand(M, N)));
```

where, as discussed in Section 3.2.2, log is the natural logarithm, and, as mentioned earlier, rand generates uniformly distributed random numbers in the interval (0, 1). If we let M = N = 1, then the preceding MATLAB command line yields a single value from a random variable with a Rayleigh distribution characterized by parameters a and b. ■

The expression $z = a + \sqrt{b \ln(1 - w)}$ sometimes is called a *random number generator equation* because it establishes how to generate the desired random numbers. In this particular case, we were able to find a closed-form solution. As will be shown shortly, this is not always possible and the problem then becomes one of finding an applicable random number generator equation whose outputs will approximate random numbers with the specified CDF.

Table 5.1 lists the random variables of interest in the present discussion, along with their PDFs, CDFs, and random number generator equations. In some cases, as with the Rayleigh and exponential variables, it is possible to find a closed-form solution for the CDF and its inverse. This allows us to write an expression for the random number generator in terms of uniform random numbers, as illustrated in Example 5.1. In others, as in the case of the Gaussian and lognormal densities, closed-form solutions for the CDF do not exist, and it becomes necessary to find alternate ways to generate the desired random numbers. In the lognormal case, for instance, we make use of the knowledge that a lognormal random variable, z, is such that $\ln(z)$ has a Gaussian distribution and write the expression shown in Table 5.1 in terms of Gaussian random variables with zero mean and unit variance. Yet in other cases, it is advantageous to reformulate the problem to obtain an easier solution. For example, it can be shown that Erlang random numbers with parameters a and b can be obtained by adding b exponentially distributed random numbers that have parameter a (Leon-Garcia [1994]).

The random number generators available in imnoise and those shown in Table 5.1 play an important role in modeling the behavior of random noise in image-processing applications. We already saw the usefulness of the uniform distribution for generating random numbers with various CDFs. Gaussian noise is used as an approximation in cases such as imaging sensors operating at low light levels. Salt-and-pepper noise arises in faulty switching devices. The size of silver particles in a photographic emulsion is a random variable described by a lognormal distribution. Rayleigh noise arises in range imaging, while exponential and Erlang noise are useful in describing noise in laser imaging.

M-function imnoise2, listed later in this section, generates random numbers having the CDFs in Table 5.1. This function makes use of MATLAB function rand, which, for the purposes of this chapter, has the syntax

```
A = rand(M, N)
```

rand

TABLE 5.1 Generation of random variables.

Name	PDF	Mean and Variance	CDF	Generator[†]
Uniform	$p_z(z) = \begin{cases} \dfrac{1}{b-a} & \text{if } a \leq z \leq b \\ 0 & \text{otherwise} \end{cases}$	$m = \dfrac{a+b}{2}, \quad \sigma^2 = \dfrac{(b-a)^2}{12}$	$F_z(z) = \begin{cases} 0 & z < a \\ \dfrac{z-a}{b-a} & a \leq z \leq b \\ 1 & z > b \end{cases}$	MATLAB function `rand`
Gaussian	$p_z(z) = \dfrac{1}{\sqrt{2\pi}b} e^{-(z-a)^2/2b^2}$ $-\infty < z < \infty$	$m = a, \quad \sigma^2 = b^2$	$F_z(z) = \int_{-\infty}^{z} p_z(v)\, dv$	MATLAB function `randn`
Salt & Pepper	$p_z(z) = \begin{cases} P_a & \text{for } z = a \\ P_b & \text{for } z = b \\ 0 & \text{otherwise} \end{cases}$ $b > a$	$m = aP_a + bP_b$ $\sigma^2 = (a-m)^2 P_a + (b-m)^2 P_b$	$F_z(z) = \begin{cases} 0 & \text{for } z < a \\ P_a & \text{for } a \leq z < b \\ P_a + P_b & \text{for } b \leq z \end{cases}$	MATLAB function `rand` with some additional logic
Lognormal	$p_z(z) = \dfrac{1}{\sqrt{2\pi}bz} e^{-[\ln(z)-a]^2/2b^2}$ $z > 0$	$m = e^{a+(b^2/2)}, \sigma^2 = [e^{b^2} - 1]e^{2a+b^2}$	$F_z(z) = \int_0^z p_z(v)\, dv$	$z = ae^{bN(0,1)}$
Rayleigh	$p_z(z) = \begin{cases} \dfrac{2}{b}(z-a)e^{-(z-a)^2/b} & z \geq a \\ 0 & z < a \end{cases}$	$m = a + \sqrt{\pi b/4}, \sigma^2 = \dfrac{b(4-\pi)}{4}$	$F_z(z) = \begin{cases} 1 - e^{-(z-a)^2/b} & z \geq a \\ 0 & z < a \end{cases}$	$z = a + \sqrt{b \ln[1 - U(0,1)]}$
Exponential	$p_z(z) = \begin{cases} ae^{-az} & z \geq 0 \\ 0 & z < 0 \end{cases}$	$m = \dfrac{1}{a}, \quad \sigma^2 = \dfrac{1}{a^2}$	$F_z(z) = \begin{cases} 1 - e^{-az} & z \geq 0 \\ 0 & z < 0 \end{cases}$	$z = -\dfrac{1}{a}\ln[1 - U(0,1)]$
Erlang	$p_z(z) = \dfrac{a^b z^{b-1}}{(b-1)!} e^{-az}$ $z \geq 0$	$m = \dfrac{b}{a}, \quad \sigma^2 = \dfrac{b}{a^2}$	$F_z(z) = \left[1 - e^{-az} \sum_{n=0}^{b-1} \dfrac{(az)^n}{n!}\right]$ $z \geq 0$	$z = E_1 + E_2 + \cdots + E_b$ (The E's are exponential random numbers with parameter a.)

[†] $N(0,1)$ denotes normal (Gaussian) random numbers with mean 0 and a variance of 1. $U(0,1)$ denotes uniform random numbers in the range $(0,1)$.

This function generates an array of size M × N whose entries are uniformly distributed numbers with values in the interval (0, 1). If N is omitted it defaults to M. If called without an argument, rand generates a single random number that changes each time the function is called. Similarly, the function

```
A = randn(M, N)
```

randn

generates an M × N array whose elements are normal (Gaussian) numbers with zero mean and unit variance. If N is omitted it defaults to M. When called without an argument, randn generates a single random number.

Function imnoise2 also uses MATLAB function find, which has the following syntax forms:

```
      I = find(A)
   [r, c] = find(A)
[r, c, v] = find(A)
```

find

The first form returns in I all the indices of array A that point to *nonzero* elements. If none is found, find returns an empty matrix. The second form returns the row and column indices of the nonzero entries in the matrix A. In addition to returning the row and column indices, the third form also returns the nonzero values of A as a column vector, v.

The first form treats the array A in the format A(:), so I is a column vector. This form is quite useful in image processing. For example, to find and set to 0 all pixels in an image whose values are less than 128 we write

```
>> I = find(A < 128);
>> A(I) = 0;
```

Recall that the logical statement A < 128 returns a 1 for the elements of A that satisfy the logical condition and 0 for those that do not. To set to 128 all pixels in the closed interval [64, 192] we write

```
>> I = find(A >= 64 & A <= 192);
>> A(I) = 128;
```

The first two forms of function find are used frequently in the remaining chapters of the book.

Unlike imnoise, the following M-function generates an M × N noise array, R, that is not scaled in any way. Another major difference is that imnoise outputs a noisy image, while imnoise2 produces the noise pattern itself. The user specifies the desired values for the noise parameters directly. Note that the noise array resulting from salt-and-pepper noise has three values: 0 corresponding to pepper noise, 1 corresponding to salt noise, and 0.5 corresponding to no noise.

This array needs to be processed further to make it useful. For example, to corrupt an image with this array, we find (using function find) all the coordinates in R that have value 0 and set the corresponding coordinates in the image to the smallest possible gray-level value (usually 0). Similarly, we find all the coordinates in R that have value 1 and set all the corresponding coordinates in the image to the highest possible value (usually 255 for an 8-bit image). This process simulates how salt-and-pepper noise affects an image in practice.

imnoise2

```
function R = imnoise2(type, M, N, a, b)
%IMNOISE2 Generates an array of random numbers with specified PDF.
%   R = IMNOISE2(TYPE, M, N, A, B) generates an array, R, of size
%   M-by-N, whose elements are random numbers of the specified TYPE
%   with parameters A and B. If only TYPE is included in the
%   input argument list, a single random number of the specified
%   TYPE and default parameters shown below is generated. If only
%   TYPE, M, and N are provided, the default parameters shown below
%   are used. If M = N = 1, IMNOISE2 generates a single random
%   number of the specified TYPE and parameters A and B.
%
%   Valid values for TYPE and parameters A and B are:
%
%   'uniform'       Uniform random numbers in the interval (A, B).
%                   The default values are (0, 1).
%   'gaussian'      Gaussian random numbers with mean A and standard
%                   deviation B. The default values are A = 0, B = 1.
%   'salt & pepper' Salt and pepper numbers of amplitude 0 with
%                   probability Pa = A, and amplitude 1 with
%                   probability Pb = B. The default values are Pa =
%                   Pb = A = B = 0.05. Note that the noise has
%                   values 0 (with probability Pa = A) and 1 (with
%                   probability Pb = B), so scaling is necessary if
%                   values other than 0 and 1 are required. The noise
%                   matrix R is assigned three values. If R(x, y) =
%                   0, the noise at (x, y) is pepper (black). If
%                   R(x, y) = 1, the noise at (x, y) is salt
%                   (white). If R(x, y) = 0.5, there is no noise
%                   assigned to coordinates (x, y).
%   'lognormal'     Lognormal numbers with offset A and shape
%                   parameter B. The defaults are A = 1 and B =
%                   0.25.
%   'rayleigh'      Rayleigh noise with parameters A and B. The
%                   default values are A = 0 and B = 1.
%   'exponential'   Exponential random numbers with parameter A. The
%                   default is A = 1.
%   'erlang'        Erlang (gamma) random numbers with parameters A
%                   and B. B must be a positive integer. The
%                   defaults are A = 2 and B = 5. Erlang random
%                   numbers are approximated as the sum of B
%                   exponential random numbers.
% Set default values.
if nargin == 1
   a = 0; b = 1;
   M = 1; N = 1;
elseif nargin == 3
   a = 0; b = 1;
end

% Begin processing. Use lower(type) to protect against input
% being capitalized.
switch lower(type)
case 'uniform'
   R = a + (b - a)*rand(M, N);
case 'gaussian'
   R = a + b*randn(M, N);
case 'salt & pepper'
   if nargin <= 3
      a = 0.05; b = 0.05;
   end
   % Check to make sure that Pa + Pb is not > 1.
   if (a + b) > 1
      error('The sum Pa + Pb must not exceed 1.')
   end
   R(1:M, 1:N) = 0.5;
   % Generate an M-by-N array of uniformly-distributed random numbers
   % in the range (0, 1). Then, Pa*(M*N) of them will have values <=
   % a. The coordinates of these points we call 0 (pepper
   % noise). Similarly, Pb*(M*N) points will have values in the range
   % > a & <= (a + b). These we call 1 (salt noise).
   X = rand(M, N);
   c = find(X <= a);
   R(c) = 0;
   u = a + b;
   c = find(X > a & X <= u);
   R(c) = 1;
case 'lognormal'
   if nargin <= 3
      a = 1; b = 0.25;
   end
   R = a*exp(b*randn(M, N));
case 'rayleigh'
   R = a + (-b*log(1 - rand(M, N))).^0.5;
case 'exponential'
   if nargin <= 3
      a = 1;
   end
   if a <= 0
      error('Parameter a must be positive for exponential type.')
   end
   k = -1/a;
   R = k*log(1 - rand(M, N));
```

```
case 'erlang'
   if nargin <= 3
      a = 2; b = 5;
   end
   if (b ~= round(b) | b <= 0)
      error('Param b must be a positive integer for Erlang.')
   end
   k = -1/a;
   R = zeros(M, N);
   for j = 1:b
      R = R + k*log(1 - rand(M, N));
   end
otherwise
   error('Unknown distribution type.')
end
```

EXAMPLE 5.2: Histograms of data generated using the function imnoise2.

Figure 5.2 shows histograms of all the random number types in Table 5.1. The data for each plot were generated using function imnoise2. For example, the data for Fig. 5.2(a) were generated by the following command:

```
>> r = imnoise2('gaussian', 100000, 1, 0, 1);
```

This statement generated a column vector, r, with 100000 elements, each being a random number from a Gaussian distribution with mean 0 and standard deviation of 1. The histogram was then obtained using function hist, which has the syntax

hist

```
p = hist(r, bins)
```

where bins is the number of bins. We used bins = 50 to generate the histograms in Fig. 5.2. The other histograms were generated in a similar manner. In each case, the parameters chosen were the default values listed in the explanation of function imnoise2.

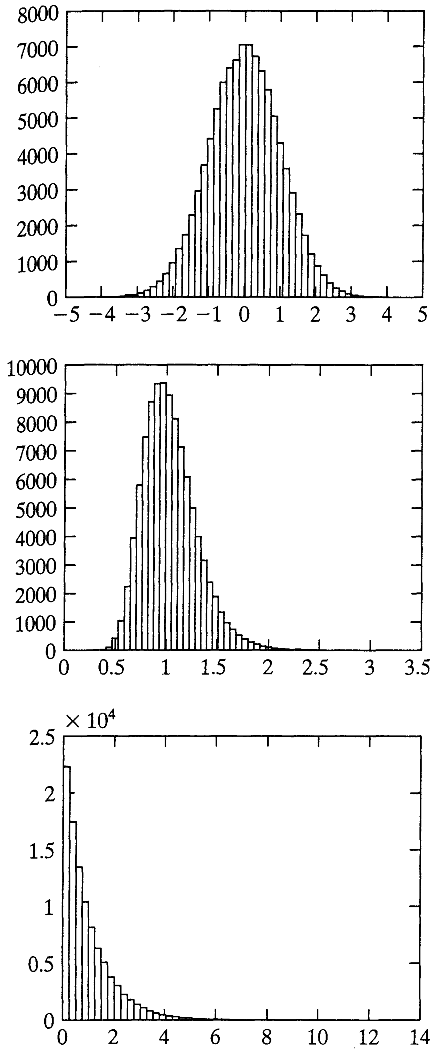
5.2.3 Periodic Noise

Periodic noise in an image arises typically from electrical and/or electromechanical interference during image acquisition. This is the only type of spatially dependent noise that will be considered in this chapter. As discussed in Section 5.4, periodic noise is typically handled in an image by filtering in the frequency domain. Our model of periodic noise is a 2-D sinusoid with equation

$$r(x, y) = A\sin[2\pi u_0(x + B_x)/M + 2\pi v_0(y + B_y)/N]$$

where A is the amplitude, u_0 and v_0 determine the sinusoidal frequencies with respect to the x- and y-axis, respectively, and B_x and B_y are phase displacements with respect to the origin. The $M \times N$ DFT of this equation is

$$R(u, v) = j\frac{A}{2}[(e^{j2\pi u_0 B_x/M})\delta(u + u_0, v + v_0) - (e^{j2\pi v_0 B_y/N})\delta(u - u_0, v - v_0)]$$



a b
c d
e f

FIGURE 5.2 Histograms of random numbers: (a) Gaussian, (b) uniform, (c) lognormal, (d) Rayleigh, (e) exponential, and (f) Erlang. In each case the default parameters listed in the explanation of function `imnoise2` were used.

which we see is a pair of complex conjugate impulses located at $(u + u_0, v + v_0)$ and $(u - u_0, v - v_0)$, respectively.

The following M-function accepts an arbitrary number of impulse locations (frequency coordinates), each with its own amplitude, frequencies, and phase displacement parameters, and computes $r(x, y)$ as the sum of sinusoids of the form described in the previous paragraph. The function also outputs the Fourier transform of the sum of sinusoids, $R(u, v)$, and the spectrum of $R(u, v)$. The sine waves are generated from the given impulse location information via the inverse DFT. This makes it more intuitive and simplifies visualization of frequency content in the spatial noise pattern. Only one pair of coordinates is required to define the location of an impulse. The program generates the conjugate symmetric impulses. (Note in the code the use of function ifftshift to convert the centered R into the proper data arrangement for the ifft2 operation, as discussed in Section 4.2.)

imnoise3

```
function [r, R, S] = imnoise3(M, N, C, A, B)
%IMNOISE3 Generates periodic noise.
%   [r, R, S] = IMNOISE3(M, N, C, A, B), generates a spatial
%   sinusoidal noise pattern, r, of size M-by-N, its Fourier
%   transform, R, and spectrum, S. The remaining parameters are as
%   follows:
%
%   C is a K-by-2 matrix containing K pairs of frequency domain
%   coordinates (u, v) indicating the locations of impulses in the
%   frequency domain. These locations are with respect to the
%   frequency rectangle center at (M/2 + 1, N/2 + 1). Only one pair
%   of coordinates is required for each impulse. The program
%   automatically generates the locations of the conjugate symmetric
%   impulses. These impulse pairs determine the frequency content
%   of r.
%
%   A is a 1-by-K vector that contains the amplitude of each of the
%   K impulse pairs. If A is not included in the argument, the
%   default used is A = ONES(1, K). B is then automatically set to
%   its default values (see next paragraph). The value specified
%   for A(j) is associated with the coordinates in C(j, 1:2).
%
%   B is a K-by-2 matrix containing the Bx and By phase components
%   for each impulse pair. The default values for B are B(1:K, 1:2)
%   = 0.

% Process input parameters.
[K, n] = size(C);
if nargin == 3
   A(1:K) = 1.0;
   B(1:K, 1:2) = 0;
elseif nargin == 4
   B(1:K, 1:2) = 0;
end

% Generate R.
R = zeros(M, N);
for j = 1:K
   u1 = M/2 + 1 + C(j, 1); v1 = N/2 + 1 + C(j, 2);
   R(u1, v1) = i * (A(j)/2) * exp(i*2*pi*C(j, 1) * B(j, 1)/M);
   % Complex conjugate.
   u2 = M/2 + 1 - C(j, 1); v2 = N/2 + 1 - C(j, 2);
   R(u2, v2) = -i * (A(j)/2) * exp(i*2*pi*C(j, 2) * B(j, 2)/N);
end

% Compute spectrum and spatial sinusoidal pattern.
S = abs(R);
r = real(ifft2(ifftshift(R)));
```

EXAMPLE 5.3: Using function `imnoise3`.

■ Figures 5.3(a) and (b) show the spectrum and spatial sine noise pattern generated using the following commands:

```
>> C = [0 64; 0 128; 32 32; 64 0; 128 0; -32 32];
>> [r, R, S] = imnoise3(512, 512, C);
>> imshow(S, [ ])
>> figure, imshow(r, [ ])
```

Recall that the order of the coordinates is (u, v). These two values are specified with reference to the center of the frequency rectangle (see Section 4.2 for a definition of the coordinates of this center point). Figures 5.3(c) and (d) show the result obtained by repeating the previous commands, but with

```
>> C = [0 32; 0 64; 16 16; 32 0; 64 0; -16 16];
```

Similarly, Fig. 5.3(e) was obtained with

```
>> C = [6 32; -2 2];
```

Figure 5.3(f) was generated with the same `C`, but using a nondefault amplitude vector:

```
>> A = [1 5];
>> [r, R, S] = imnoise3(512, 512, C, A);
```

As Fig. 5.3(f) shows, the lower-frequency sine wave dominates the image. This is as expected because its amplitude is five times the amplitude of the higher-frequency component. ■

5.2.4 Estimating Noise Parameters

The parameters of periodic noise typically are estimated by analyzing the Fourier spectrum of the image. Periodic noise tends to produce frequency spikes that often can be detected even by visual inspection. Automated analysis is possible in situations in which the noise spikes are sufficiently pronounced, or when some knowledge about the frequency of the interference is available.

In the case of noise in the spatial domain, the parameters of the PDF may be known partially from sensor specifications, but it is often necessary to estimate them from sample images. The relationships between the mean, m, and variance, σ^2, of the noise, and the parameters a and b required to completely specify the noise PDFs of interest in this chapter are listed in Table 5.1. Thus, the problem becomes one of estimating the mean and variance from the sample image(s) and then using these estimates to solve for a and b.

Let z_i be a discrete random variable that denotes intensity levels in an image, and let $p(z_i)$, $i = 0, 1, 2, \ldots, L - 1$, be the corresponding normalized

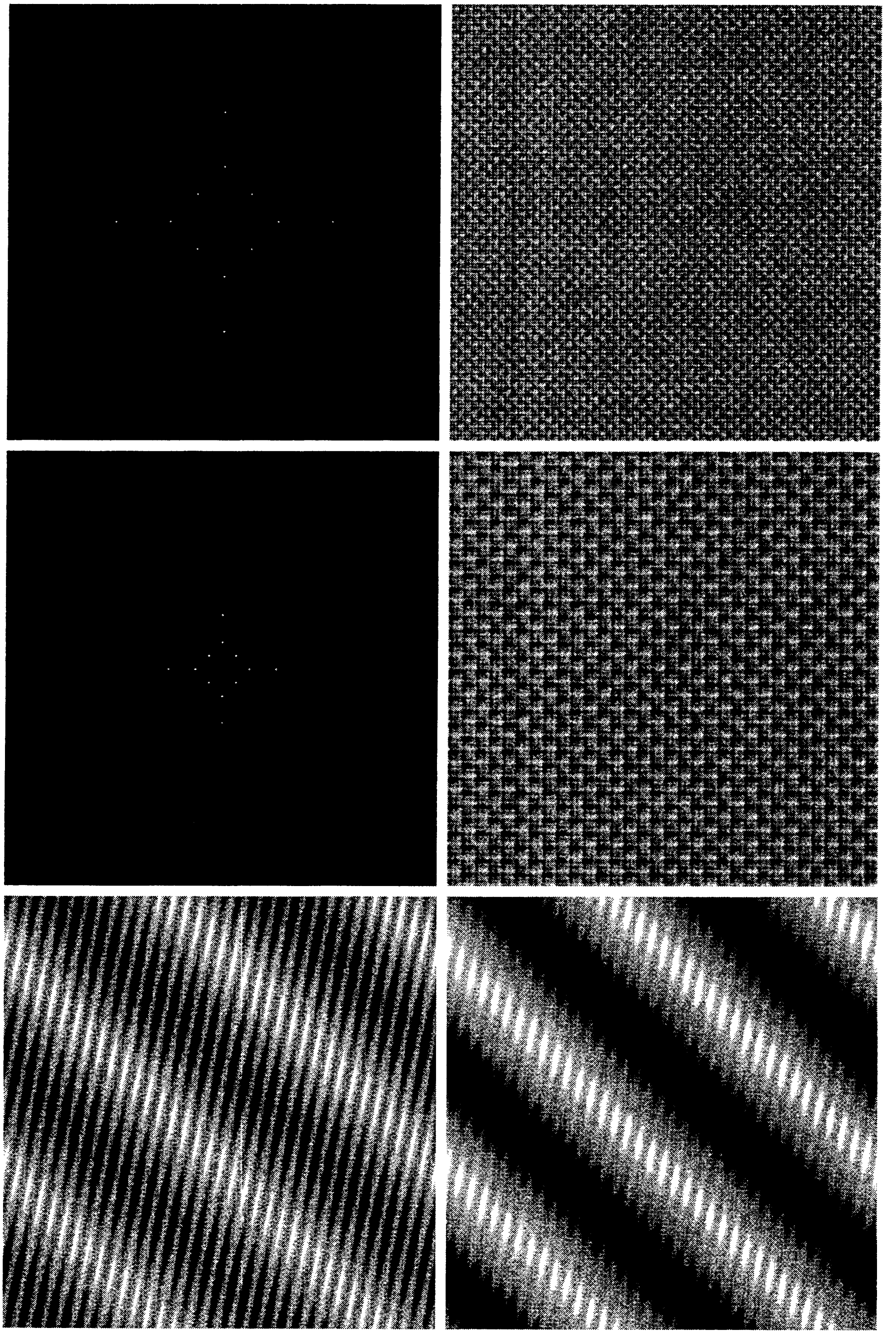


FIGURE 5.3 (a) Spectrum of specified impulses. (b) Corresponding sine noise pattern. (c) and (d) A similar sequence. (e) and (f) Two other noise patterns. The dots in (a) and (c) were enlarged to make them easier to see.

histogram, where L is the number of possible intensity values. A histogram component, $p(z_j)$, is an estimate of the probability of occurrence of intensity value z_j, and the histogram may be viewed as an approximation of the intensity PDF.

One of the principal approaches for describing the shape of a histogram is via its *central moments* (also called *moments about the mean*), which are defined as

$$\mu_n = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i)$$

where n is the moment *order*, and m is the mean:

$$m = \sum_{i=0}^{L-1} z_i p(z_i)$$

Because the histogram is assumed to be normalized, the sum of all its components is 1, so, from the preceding equations, we see that $\mu_0 = 1$ and $\mu_1 = 0$. The second moment,

$$\mu_2 = \sum_{i=0}^{L-1} (z_i - m)^2 p(z_i)$$

is the variance. In this chapter, we are interested only in the mean and variance. Higher-order moments are discussed in Chapter 11.

Function `statmoments` computes the mean and central moments up to order n, and returns them in row vector `v`. Because the moment of order 0 is always 1, and the moment of order 1 is always 0, `statmoments` ignores these two moments and instead lets `v(1)` $= m$ and `v(k)` $= \mu_k$ for $k = 2, 3, \ldots, n$. The syntax is as follows (see Appendix C for the code):

statmoments

```
[v, unv] = statmoments(p, n)
```

where `p` is the histogram vector and `n` is the number of moments to compute. It is required that the number of components of `p` be equal to 2^8 for class `uint8` images, 2^{16} for class `uint16` images, and 2^8 or 2^{16} for images of class `double`. Output vector `v` contains the normalized moments based on values of the random variable that have been scaled to the range [0, 1], so all the moments are in this range also. Vector `unv` contains the same moments as `v`, but computed with the data in its original range of values. For example, if `length(p) = 256`, and `v(1) = 0.5`, then `unv(1)` would have the value `127.5`, which is half of the range `[0, 255]`.

Often, noise parameters must be estimated directly from a given noisy image or set of images. In this case, the approach is to select a region in an image with as featureless a background as possible, so that the variability of intensity values in the region will be due primarily to noise. To select a region of interest (ROI) in MATLAB we use function roipoly, which generates a polygonal ROI. This function has the basic syntax

roipoly

```
B = roipoly(f, c, r)
```

where f is the image of interest, and c and r are vectors of corresponding (sequential) column and row coordinates of the vertices of the polygon (note that columns are specified first). The output, B, is a binary image the same size as f with 0's outside the region of interest and 1's inside. Image B is used as a mask to limit operations to within the region of interest.

To specify a polygonal ROI interactively, we use the syntax

```
B = roipoly(f)
```

which displays the image f on the screen and lets the user specify the polygon using the mouse. If f is omitted, roipoly operates on the last image displayed. Using normal button clicks adds vertices to the polygon. Pressing **Backspace** or **Delete** removes the previously selected vertex. A shift-click, right-click, or double-click adds a final vertex to the selection and starts the fill of the polygonal region with 1s. Pressing **Return** finishes the selection without adding a vertex.

To obtain the binary image and a list of the polygon vertices, we use the construct

```
[B, c, r] = roipoly(. . .)
```

where roipoly(. . .) indicates any valid syntax for this function and, as before, c and r are the column and row coordinates of the vertices. This format is particularly useful when the ROI is specified interactively because it gives the coordinates of the polygon vertices for use in other programs or for later duplication of the same ROI.

The following function computes the histogram of an image within a polygonal region whose vertices are specified by vectors c and r, as in the preceding discussion. Note the use within the program of function roipoly to duplicate the polygonal region defined by c and r.

histroi

```
function [p, npix] = histroi(f, c, r)
%HISTROI Computes the histogram of an ROI in an image.
%   [P, NPIX] = HISTROI(F, C, R) computes the histogram, P, of a
%   polygonal region of interest (ROI) in image F. The polygonal
%   region is defined by the column and row coordinates of its
%   vertices, which are specified (sequentially) in vectors C and R,
%   respectively. All pixels of F must be >= 0. Parameter NPIX is the
%   number of pixels in the polygonal region.

% Generate the binary mask image.
B = roipoly(f, c, r);
```

```
% Compute the histogram of the pixels in the ROI.
p = imhist(f(B));

% Obtain the number of pixels in the ROI if requested in the output.
if nargout > 1
   npix = sum(B(:));
end
```

EXAMPLE 5.4: Estimating noise parameters.

■ Figure 5.4(a) shows a noisy image, denoted by f in the following discussion. The objective of this example is to estimate the noise type and its parameters using the techniques and tools developed thus far. Figure 5.4(b) shows the mask, B, generated interactively using the command:

```
>> [B, c, r] = roipoly(f);
```

Figure 5.4(c) was generated using the commands

```
>> [p, npix] = histroi(f, c, r);
>> figure, bar(p, 1)
```

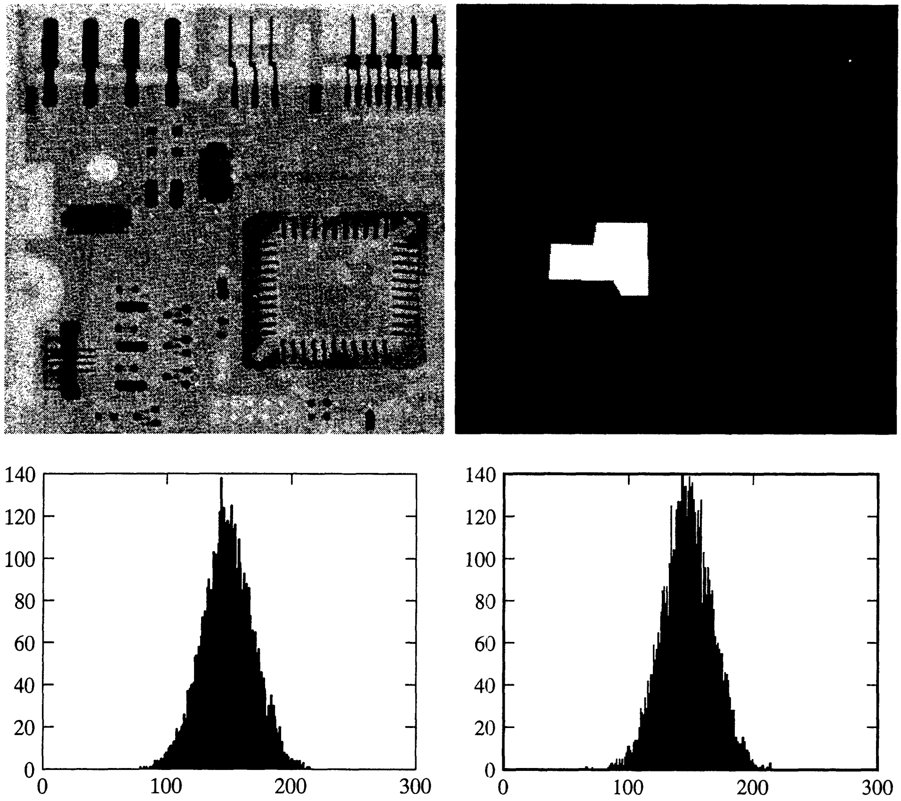


FIGURE 5.4 (a) Noisy image. (b) ROI generated interactively. (c) Histogram of ROI. (d) Histogram of Gaussian data generated using function imnoise2. (Original image courtesy of Lixi, Inc.)

The mean and variance of the region masked by B were obtained as follows:

```
>> [v, unv] = statmoments(h, 2);
>> v

v =

   0.5794     0.0063

>> unv
   147.7430   410.9313
```

It is evident from Fig. 5.4(c) that the noise is approximately Gaussian. In general, it is not possible to know the exact mean and variance of the noise because it is added to the gray levels of the image in region B. However, by selecting an area of nearly constant background level (as we did here), and because the noise appears Gaussian, we can estimate that the average gray level of the area B is reasonably close to the average gray level of the image without noise, indicating that the noise has zero mean. Also, the fact that the area has a nearly constant gray level tells us that the variability in the region defined by B is due primarily to the variance of the noise. (When feasible, another way to estimate the mean and variance of the noise is by imaging a target of constant, known gray level.) Figure 5.4(d) shows the histogram of a set of npix (this number is returned by histroi) Gaussian random variables with mean 147 and variance 400, obtained with the following commands:

```
>> X = imnoise2('gaussian', npix, 1, 147, 20);
>> figure, hist(X, 130)
>> axis([0 300 0 140])
```

where the number of bins in hist was selected so that the result would be compatible with the plot in Fig. 5.4(c). The histogram in this figure was obtained within function histroi using imhist (see the preceding code), which employs a different scaling than hist. We chose a set of npix random variables to generate X, so that the number of samples was the same in both histograms. The similarity between Figs. 5.4(c) and (d) clearly indicates that the noise is indeed well-approximated by a Gaussian distribution with parameters that are close to the estimates v(1) and v(2). ■

5.3 Restoration in the Presence of Noise Only—Spatial Filtering

When the only degradation present is noise, then it follows from the model in Section 5.1 that

$$g(x, y) = f(x, y) + \eta(x, y)$$

The method of choice for reduction of noise in this case is spatial filtering, using techniques similar to those discussed in Sections 3.4 and 3.5. In this section we summarize and implement several spatial filters for noise reduction. Additional details on the characteristics of these filters are discussed by Gonzalez and Woods [2002].

5.3.1 Spatial Noise Filters

Table 5.2 lists the spatial filters of interest in this section, where S_{xy} denotes an $m \times n$ subimage (region) of the input noisy image, g. The subscripts on S indicate that the subimage is centered at coordinates (x, y), and $\hat{f}(x, y)$ (an estimate of f) denotes the filter response at those coordinates. The linear filters are implemented using function `imfilter` discussed in Section 3.4. The median, max, and min filters are nonlinear, order-statistic filters. The median filter can be implemented directly using IPT function `medfilt2`. The max and min filters are implemented using the more general order-filter function `ordfilt2` discussed in Section 3.5.2.

The following function, which we call `spfilt`, performs filtering in the spatial domain with any of the filters listed in Table 5.2. Note the use of function `imlincomb` (mentioned in Table 2.5) to compute the linear combination of the inputs. The syntax for this function is

```
B = imlincomb(c1, A1, c2, A2, . . ., ck, Ak)
```

imlincomb

which implements the equation

```
B = c1*A1 + c2*A2 + · · · + ck*Ak
```

where the c's are real, `double` scalars, and the A's are numeric arrays of the same class and size. Note also in subfunction `gmean` how function `warning` can be turned on and off. In this case, we are suppressing a warning that would be issued by MATLAB if the argument of the `log` function becomes 0. In general, `warning` can be used in any program. The basic syntax is

```
warning('message')
```

warning

This function behaves exactly like function `disp`, except that it can be turned on and off with the commands `warning on` and `warning off`.

spfilt

```
function f = spfilt(g, type, m, n, parameter)
%SPFILT Performs linear and nonlinear spatial filtering.
%   F = SPFILT(G, TYPE, M, N, PARAMETER) performs spatial filtering
%   of image G using a TYPE filter of size M-by-N. Valid calls to
%   SPFILT are as follows:
%
%      F = SPFILT(G, 'amean', M, N)       Arithmetic mean filtering.
%      F = SPFILT(G, 'gmean', M, N)       Geometric mean filtering.
%      F = SPFILT(G, 'hmean', M, N)       Harmonic mean filtering.
%      F = SPFILT(G, 'chmean', M, N, Q)   Contraharmonic mean
%                                         filtering of order Q. The
%                                         default is Q = 1.5.
%      F = SPFILT(G, 'median', M, N)      Median filtering.
%      F = SPFILT(G, 'max', M, N)         Max filtering.
%      F = SPFILT(G, 'min', M, N)         Min filtering.
```

TABLE 5.2 Spatial filters. The variables m and n denote respectively the number of rows and columns of the filter neighborhood.

Filter Name	Equation	Comments
Arithmetic mean	$\hat{f}(x,y) = \dfrac{1}{mn}\sum_{(s,t)\in S_{xy}} g(s,t)$	Implemented using IPT functions `w = fspecial('average', [m, n])` and `f = imfilter(g, w)`.
Geometric mean	$\hat{f}(x,y) = \left[\prod_{(s,t)\in S_{xy}} g(s,t)\right]^{\frac{1}{mn}}$	This nonlinear filter is implemented using function `gmean` (see custom function `spfilt` in this section).
Harmonic mean	$\hat{f}(x,y) = \dfrac{mn}{\sum_{(s,t)\in S_{xy}} \dfrac{1}{g(s,t)}}$	This nonlinear filter is implemented using function `harmean` (see custom function `spfilt` in this section).
Contraharmonic mean	$\hat{f}(x,y) = \dfrac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}}$	This nonlinear filter is implemented using function `charmean` (see custom function `spfilt` in this section).
Median	$\hat{f}(x,y) = \underset{(s,t)\in S_{xy}}{\text{median}}\{g(s,t)\}$	Implemented using IPT function `medfilt2`: `f = medfilt2(g, [m n])`.
Max	$\hat{f}(x,y) = \max_{(s,t)\in S_{xy}}\{g(s,t)\}$	Implemented using IPT function `ordfilt2`: `f = ordfilt2(g, m*n, ones(m, n))`.
Min	$\hat{f}(x,y) = \min_{(s,t)\in S_{xy}}\{g(s,t)\}$	Implemented using IPT function `ordfilt2`: `f = ordfilt2(g, 1, ones(m, n))`.
Midpoint	$\hat{f}(x,y) = \dfrac{1}{2}\left[\max_{(s,t)\in S_{xy}}\{g(s,t)\} + \min_{(s,t)\in S_{xy}}\{g(s,t)\}\right]$	Implemented as 0.5 times the sum of the max and min filtering operations.
Alpha-trimmed mean	$\hat{f}(x,y) = \dfrac{1}{mn-d}\sum_{(s,t)\in S_{xy}} g_r(s,t)$	The $d/2$ lowest and $d/2$ highest intensity levels of $g(s,t)$ in S_{xy} are deleted, $g_r(s,t)$ denotes the remaining $mn - d$ pixels in the neighborhood. Implemented using function `alphatrim` (see custom function `spfilt` in this section).

```
%     F = SPFILT(G, 'midpoint', M, N)    Midpoint filtering.
%     F = SPFILT(G, 'atrimmed', M, N, D) Alpha-trimmed mean filtering.
%                                        Parameter D must be a nonnegative
%                                        even integer; its default
%                                        value is D = 2.
%
%  The default values when only G and TYPE are input are M = N = 3,
%  Q = 1.5, and D = 2.

% Process inputs.
if nargin == 2
   m = 3; n = 3; Q = 1.5; d = 2;
elseif nargin == 5
   Q = parameter; d = parameter;
elseif nargin == 4
   Q = 1.5; d = 2;
else
   error('Wrong number of inputs.');
end

% Do the filtering.
switch type
case 'amean'
   w = fspecial('average', [m n]);
   f = imfilter(g, w, 'replicate');
case 'gmean'
   f = gmean(g, m, n);
case 'hmean'
   f = harmean(g, m, n);
case 'chmean'
   f = charmean(g, m, n, Q);
case 'median'
   f = medfilt2(g, [m n], 'symmetric');
case 'max'
   f = ordfilt2(g, m*n, ones(m, n), 'symmetric');
case 'min'
   f = ordfilt2(g, 1, ones(m, n), 'symmetric');
case 'midpoint'
   f1 = ordfilt2(g, 1, ones(m, n), 'symmetric');
   f2 = ordfilt2(g, m*n, ones(m, n), 'symmetric');
   f = imlincomb(0.5, f1, 0.5, f2);
case 'atrimmed'
   if (d < 0) | (d/2 ~= round(d/2))
      error('d must be a nonnegative, even integer.')
   end
   f = alphatrim(g, m, n, d);
otherwise
   error('Unknown filter type.')
end

%-------------------------------------------------------------------%
```

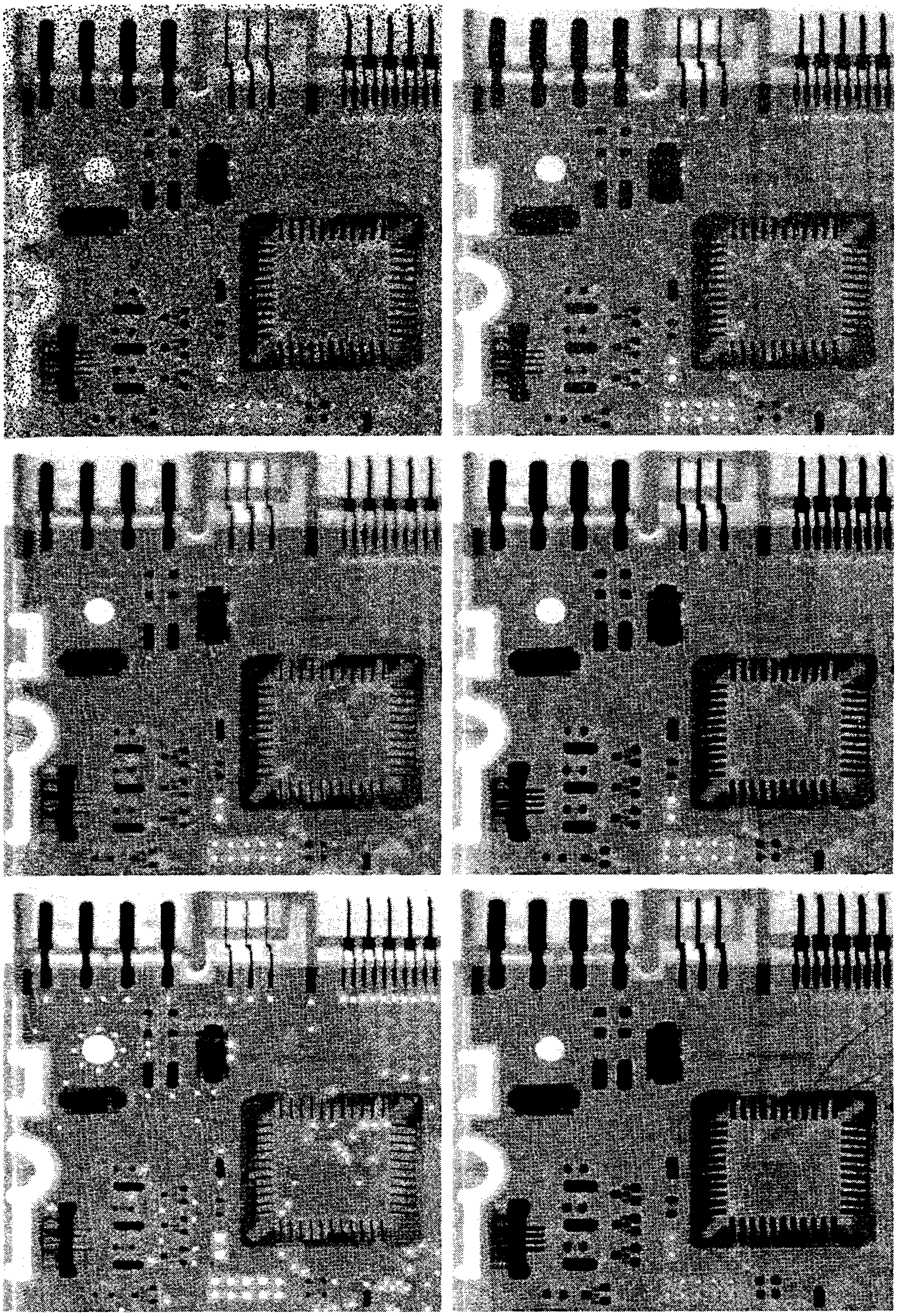
```
function f = gmean(g, m, n)
%  Implements a geometric mean filter.
inclass = class(g);
g = im2double(g);
% Disable log(0) warning.
warning off;
f = exp(imfilter(log(g), ones(m, n), 'replicate')).^(1 / m / n);
warning on;
f = changeclass(inclass, f);

%-------------------------------------------------------------------%
function f = harmean(g, m, n)
%  Implements a harmonic mean filter.
inclass = class(g);
g = im2double(g);
f = m * n ./ imfilter(1./(g + eps), ones(m, n), 'replicate');
f = changeclass(inclass, f);

%-------------------------------------------------------------------%
function f = charmean(g, m, n, q)
%  Implements a contraharmonic mean filter.
inclass = class(g);
g = im2double(g);
f = imfilter(g.^(q+1), ones(m, n), 'replicate');
f = f ./ (imfilter(g.^q, ones(m, n), 'replicate') + eps);
f = changeclass(inclass, f);

%-------------------------------------------------------------------%
function f = alphatrim(g, m, n, d)
%  Implements an alpha-trimmed mean filter.
inclass = class(g);
g = im2double(g);
f = imfilter(g, ones(m, n), 'symmetric');
for k = 1:d/2
   f = imsubtract(f, ordfilt2(g, k, ones(m, n), 'symmetric'));
end
for k = (m*n - (d/2) + 1):m*n
   f = imsubtract(f, ordfilt2(g, k, ones(m, n), 'symmetric'));
end
f = f / (m*n - d);
f = changeclass(inclass, f);
```

EXAMPLE 5.5: Using function spfilt.

■ The image in Fig. 5.5(a) is an uint8 image corrupted by pepper noise only with probability 0.1. This image was generated using the following commands [f denotes the original image, which is Fig. 3.18(a)]:

```
>> [M, N] = size(f);
>> R = imnoise2('salt & pepper', M, N, 0.1, 0);
>> c = find(R == 0);
>> gp = f;
>> gp(c) = 0;
```

The image in Fig. 5.5(b), corrupted by salt noise only, was generated using the statements



a b
c d
e f

FIGURE 5.5 (a) Image corrupted by pepper noise with probability 0.1. (b) Image corrupted by salt noise with the same probability. (c) Result of filtering (a) with a 3 × 3 contraharmonic filter of order $Q = 1.5$. (d) Result of filtering (b) with $Q = -1.5$. (e) Result of filtering (a) with a 3 × 3 max filter. (f) Result of filtering (b) with a 3 × 3 min filter.

```
>> R = imnoise2('salt & pepper', M, N, 0, 0.1);
>> c = find(R == 1);
>> gs = f;
>> gs(c) = 255;
```

A good approach for filtering pepper noise is to use a contraharmonic filter with a positive value of Q. Figure 5.5(c) was generated using the statement

```
>> fp = spfilt(gp, 'chmean', 3, 3, 1.5);
```

Similarly, salt noise can be filtered using a contraharmonic filter with a negative value of Q:

```
>> fs = spfilt(gs, 'chmean', 3, 3, -1.5);
```

Figure 5.5(d) shows the result. Similar results can be obtained using max and min filters. For example, the images in Figs. 5.5(e) and (f) were generated from Figs. 5.5(a) and (b), respectively, with the following commands:

```
>> fpmax = spfilt(gp, 'max', 3, 3);
>> fsmin = spfilt(gs, 'min', 3, 3);
```

Other solutions using `spfilt` are implemented in a similar manner. ■

5.3.2 Adaptive Spatial Filters

The filters discussed in the previous section are applied to an image without regard for how image characteristics vary from one location to another. In some applications, results can be improved by using filters capable of adapting their behavior depending on the characteristics of the image in the area being filtered. As an illustration of how to implement adaptive spatial filters in MATLAB, we consider in this section an adaptive median filter. As before, S_{xy} denotes a subimage centered at location (x, y) in the image being processed. The algorithm, which is explained in detail in Gonzalez and Woods [2002], is as follows: Let

$$\begin{aligned} z_{\min} &= \text{minimum intensity value in } S_{xy} \\ z_{\max} &= \text{maximum intensity value in } S_{xy} \\ z_{\text{med}} &= \text{median of the intensity values in } S_{xy} \\ z_{xy} &= \text{intensity value at coordinates } (x, y) \end{aligned}$$

The adaptive median filtering algorithm works in two levels, denoted level A and level B:

```
Level A:    If $z_{\min} < z_{\text{med}} < z_{\max}$, go to level B
            Else increase the window size
            If window size $\leq S_{\max}$, repeat level A
            Else output $z_{\text{med}}$

Level B:    If $z_{\min} < z_{xy} < z_{\max}$, output $z_{xy}$
            Else output $z_{\text{med}}$
```

where $S_{\max}$ denotes the maximum allowed size of the adaptive filter window. Another option in the last step in Level A is to output z_{xy} instead of the median. This produces a slightly less blurred result but can fail to detect salt (pepper)

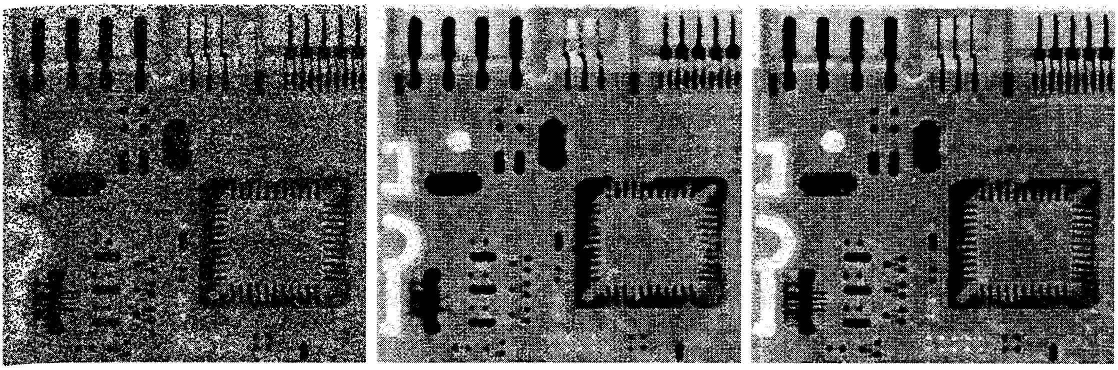


FIGURE 5.6 (a) Image corrupted by salt-and-pepper noise with density 0.25. (b) Result obtained using a median filter of size 7 × 7. (c) Result obtained using adaptive median filtering with $S_{\max} = 7$.

noise embedded in a constant background having the same value as pepper (salt) noise.

An M-function that implements this algorithm, which we call adpmedian, is included in Appendix C. The syntax is

```
f = adpmedian(g, Smax)
```

adpmedian

where g is the image to be filtered, and, as defined above, Smax is the maximum allowed size of the adaptive filter window.

EXAMPLE 5.6: Adaptive median filtering.

■ Figure 5.6(a) shows the circuit board image, f, corrupted by salt-and-pepper noise, generated using the command

```
>> g = imnoise(f, 'salt & pepper', .25);
```

and Fig. 5.6(b) shows the result obtained using the command (see Section 3.5.2 regarding the use of medfilt2):

```
>> f1 = medfilt2(g, [7 7], 'symmetric');
```

This image is reasonably free of noise, but it is quite blurred and distorted (e.g., see the connector fingers in the top middle of the image). On the other hand, the command

```
>> f2 = adpmedian(g, 7);
```

yielded the image in Fig. 5.6(c), which is also reasonably free of noise, but is considerably less blurred and distorted than Fig. 5.6(b). ■

5.4 Periodic Noise Reduction by Frequency Domain Filtering

As noted in Section 5.2.3, periodic noise manifests itself as impulse-like bursts that often are visible in the Fourier spectrum. The principal approach for filtering these components is via notch filtering. The transfer function of a Butterworth notch filter of order n is given by

$$H(u, v) = \frac{1}{1 + \left[\dfrac{D_0^2}{D_1(u, v)D_2(u, v)}\right]^n}$$

where

$$D_1(u, v) = \left[(u - M/2 - u_0)^2 + (v - N/2 - v_0)^2\right]^{1/2}$$

and

$$D_2(u, v) = \left[(u - M/2 + u_0)^2 + (v - N/2 + v_0)^2\right]^{1/2}$$

where (u_0, v_0) (and by symmetry) $(-u_0, -v_0)$ are the locations of the "notches," and D_0 is a measure of their radius. Note that the filter is specified with respect to the center of the frequency rectangle, so it must be preprocessed with function `fftshift` prior to its use, as explained in Sections 4.2 and 4.3.

Writing an M-function for notch filtering follows the same principles used in Section 4.5. It is good practice to write the function so that multiple notches can be input, as in the approach used in Section 5.2.3 to generate multiple sinusoidal noise patterns. Once H has been obtained, filtering is done using function `dftfilt` explained in Section 4.3.3.

5.5 Modeling the Degradation Function

When equipment similar to the equipment that generated a degraded image is available, it is generally possible to determine the nature of the degradation by experimenting with various equipment settings. However, relevant imaging equipment availability is the exception, rather than the rule, in the solution of image restoration problems, and a typical approach is to experiment by generating PSFs and testing the results with various restoration algorithms. Another approach is to attempt to model the PSF mathematically. This approach is outside the mainstream of our discussion here; for an introduction to this topic see Gonzalez and Woods [2002]. Finally, when no information is available about the PSF, we can resort to "blind deconvolution" for inferring the PSF. This approach is discussed in Section 5.10. The focus of the remainder of the present section is on various techniques for modeling PSFs by using functions `imfilter` and `fspecial`, introduced in Sections 3.4 and 3.5, respectively, and the various noise-generating functions discussed earlier in this chapter.

One of the principal degradations encountered in image restoration problems is image blur. Blur that occurs with the scene and sensor at rest with respect to each other can be modeled by spatial or frequency domain lowpass filters. Another important degradation model is image blur due to uniform linear motion between the sensor and scene during image acquisition. Image blur can be modeled using IPT function `fspecial`:

```
PSF = fspecial('motion', len, theta)
```

This call to `fspecial` returns a PSF that approximates the effects of linear motion of a camera by `len` pixels. Parameter `theta` is in degrees, measured with respect to the positive horizontal axis in a counter-clockwise direction. The default value of `len` is 9 and the default `theta` is 0, which corresponds to motion of 9 pixels in the horizontal direction.

We use function `imfilter` to create a degraded image with a `PSF` that is either known or is computed by using the method just described:

```
>> g = imfilter(f, PSF, 'circular');
```

where `'circular'` (Table 3.2) is used to reduce border effects. We then complete the degraded image model by adding noise, as appropriate:

```
>> g = g + noise;
```

where `noise` is a random noise image of the same size as `g`, generated using one of the methods discussed in Section 5.2.

When comparing in a given situation the suitability of the various approaches discussed in this and the following sections, it is useful to use the same image or test pattern so that comparisons are meaningful. The test pattern generated by function `checkerboard` is particularly useful for this purpose because its size can be scaled without affecting its principal features. The syntax is

```
C = checkerboard(NP, M, N)
```

`checkerboard`

where `NP` is the number of pixels on the side of each square, `M` is the number of rows, and `N` is the number of columns. If `N` is omitted, it defaults to `M`. If both `M` and `N` are omitted, a square checkerboard with 8 squares on the side is generated. If, in addition, `NP` is omitted, it defaults to 10 pixels. The light squares on the left half of the checkerboard are white. The light squares on the right half of the checkerboard are gray. To generate a checkerboard in which all light squares are white we use the command

```
>> K = im2double(checkerboard(NP, M, N)) > 0.5;
```

Using the > operator produces a `logical` *result;* `im2double` *is used to produce an image of class* `double`*, which is consistent with the output format of function* `checkerboard`.

The images generated by function `checkerboard` are of class `double` with values in the range [0, 1].

Because some restoration algorithms are slow for large images, a good approach is to experiment with small images to reduce computation time and thus improve interactivity. In this case, it is useful for display purposes to be able to zoom an image by pixel replication. The following function does this (see Appendix C for the code):

pixeldup

```
B = pixeldup(A, m, n)
```

This function duplicates every pixel in A a total of m times in the vertical direction and n times in the horizontal direction. If n is omitted, it defaults to m.

EXAMPLE 5.7: Modeling a blurred, noisy image.

■ Figure 5.7(a) shows a checkerboard image generated by the command

```
>> f = checkerboard(8);
```

The degraded image in Fig. 5.7(b) was generated using the commands

```
>> PSF = fspecial('motion', 7, 45);
>> gb = imfilter(f, PSF, 'circular');
```

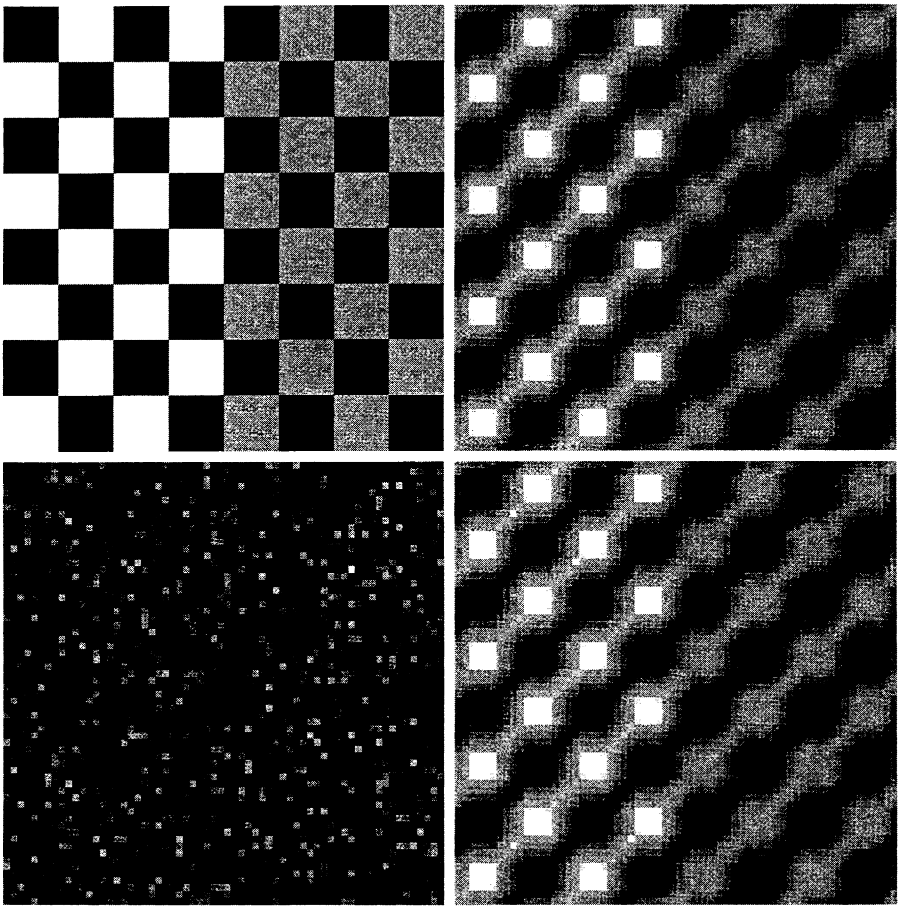


FIGURE 5.7 (a) Original image. (b) Image blurred using fspecial with len = 7, and theta = −45 degrees. (c) Noise image. (d) Sum of (b) and (c).

Note that the PSF is just a spatial filter. Its values are

```
>> PSF
PSF =
         0         0         0         0         0    0.0145         0
         0         0         0         0    0.0376    0.1283    0.0145
         0         0         0    0.0376    0.1283    0.0376         0
         0         0    0.0376    0.1283    0.0376         0         0
         0    0.0376    0.1283    0.0376         0         0         0
    0.0145    0.1283    0.0376         0         0         0         0
         0    0.0145         0         0         0         0         0
```

The noisy pattern in Fig. 5.7(c) was generated using the command

```
>> noise = imnoise(zeros(size(f)), 'gaussian', 0, 0.001);
```

Normally, we would have added noise to gb directly using imnoise(gb, 'gaussian', 0, 0.001). However, the noise image is needed later in this chapter, so we computed it separately here.

The blurred noisy image in Fig. 5.7(d) was generated as

```
>> g = gb + noise;
```

The noise is not easily visible in this image because its maximum value is on the order of 0.15, whereas the maximum value of the image is 1. As shown in Sections 5.7 and 5.8, however, this level of noise is not insignificant when attempting to restore g. Finally, we point out that all images in Fig. 5.7 were zoomed to size 512 × 512 and displayed using a command of the form

```
>> imshow(pixeldup(f, 8), [ ])
```

The image in Fig. 5.7(d) is restored in Examples 5.8 and 5.9. ■

5.6 Direct Inverse Filtering

The simplest approach we can take to restoring a degraded image is to form an estimate of the form

$$\hat{F}(u, v) = \frac{G(u, v)}{H(u, v)}$$

and then obtain the corresponding estimate of the image by taking the inverse Fourier transform of $\hat{F}(u, v)$ [recall that $G(u, v)$ is the Fourier transform of the degraded image]. This approach is appropriately called *inverse filtering*. From the model discussed in Section 5.1, we can express our estimate as

$$\hat{F}(u, v) = F(u, v) + \frac{N(u, v)}{H(u, v)}$$

This deceptively simple expression tells us that, even if we knew $H(u, v)$ exactly, we could not recover $F(u, v)$ [and hence the original, undegraded image $f(x, y)$] because the noise component is a random function whose Fourier transform, $N(u, v)$, is not known. In addition, there usually is a problem in practice with function $H(u, v)$ having numerous zeros. Even if the term $N(u, v)$ were negligible, dividing it by vanishing values of $H(u, v)$ would dominate restoration estimates.

The typical approach when attempting inverse filtering is to form the ratio $\hat{F}(u, v) = G(u, v)/H(u, v)$ and then limit the frequency range for obtaining the inverse, to frequencies "near" the origin. The idea is that zeros in $H(u, v)$ are less likely to occur near the origin because the magnitude of the transform typically is at its highest value in that region. There are numerous variations of this basic theme, in which special treatment is given at values of (u, v) for which H is zero or near zero. This type of approach sometimes is called *pseudoinverse* filtering. In general, approaches based on inverse filtering of this type are seldom practical, as Example 5.8 in the next section shows.

5.7 Wiener Filtering

Wiener filtering (after N. Wiener, who first proposed the method in 1942) is one of the earliest and best known approaches to linear image restoration. A Wiener filter seeks an estimate $\hat{f}$ that minimizes the statistical error function

$$e^2 = E\{(f - \hat{f})^2\}$$

where E is the expected value operator and f is the undegraded image. The solution to this expression in the frequency domain is

$$\hat{F}(u, v) = \left[\frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + S_\eta(u, v)/S_f(u, v)}\right] G(u, v)$$

where

$$\begin{aligned}
H(u, v) &= \text{the degradation function} \\
|H(u, v)|^2 &= H^*(u, v)H(u, v) \\
H^*(u, v) &= \text{the complex conjugate of } H(u, v) \\
S_\eta(u, v) &= |N(u, v)|^2 = \text{the power spectrum of the noise} \\
S_f(u, v) &= |F(u, v)|^2 = \text{the power spectrum of the undegraded image}
\end{aligned}$$

The ratio $S_\eta(u, v)/S_f(u, v)$ is called the *noise-to-signal power ratio*. We see that if the noise power spectrum is zero for all relevant values of u and v, this ratio becomes zero and the Wiener filter reduces to the inverse filter discussed in the previous section.

Two related quantities of interest are the average noise power and the average image power, defined as

$$\eta_A = \frac{1}{MN} \sum_u \sum_v S_\eta(u, v)$$

and

$$f_A = \frac{1}{MN}\sum_u \sum_v S_f(u, v)$$

where M and N denote the vertical and horizontal sizes of the image and noise arrays, respectively. These quantities are scalar constants, and their ratio,

$$R = \frac{\eta_A}{f_A}$$

which is also a scalar, is used sometimes to generate a constant array in place of the function $S_\eta(u, v)/S_f(u, v)$. In this case, even if the actual ratio is not known, it becomes a simple matter to experiment interactively varying the constant and viewing the restored results. This, of course, is a crude approximation that assumes that the functions are constant. Replacing $S_\eta(u, v)/S_f(u, v)$ by a constant array in the preceding filter equation results in the so-called *parametric Wiener filter*. As illustrated in Example 5.8, the simple act of using a constant array can yield significant improvements over direct inverse filtering.

Wiener filtering is implemented in IPT using function deconvwnr, which has three possible syntax forms. In all these forms, g denotes the degraded image and fr is the restored image. The first syntax form,

```
fr = deconvwnr(g, PSF)
```

deconvwnr

assumes that the noise-to-signal ratio is zero. Thus, this form of the Wiener filter is the inverse filter mentioned in Section 5.6. The syntax

```
fr = deconvwnr(g, PSF, NSPR)
```

assumes that the noise-to-signal power ratio is known, either as a constant or as an array; the function accepts either one. This is the syntax used to implement the parametric Wiener filter, in which case NSPR would be an interactive scalar input. Finally, the syntax

```
fr = deconvwnr(g, PSF, NACORR, FACORR)
```

assumes that autocorrelation functions, NACORR and FACORR, of the noise and undegraded image are known. Note that this form of deconvwnr uses the autocorrelation of η and f instead of the power spectrum of these functions. From the correlation theorem we know that

$$|F(u, v)|^2 = \Im[f(x, y) \circ f(x, y)]$$

where " $\circ$ " denotes the correlation operation and $\Im$ denotes the Fourier transform. This expression indicates that we can obtain the autocorrelation function, $f(x, y) \circ f(x, y)$, for use in deconvwnr by computing the inverse Fourier transform of the power spectrum. Similar comments hold for the autocorrelation of the noise.

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it sometimes helps to use function `edgetaper` prior to calling `deconvwnr`. The syntax is

edgetaper

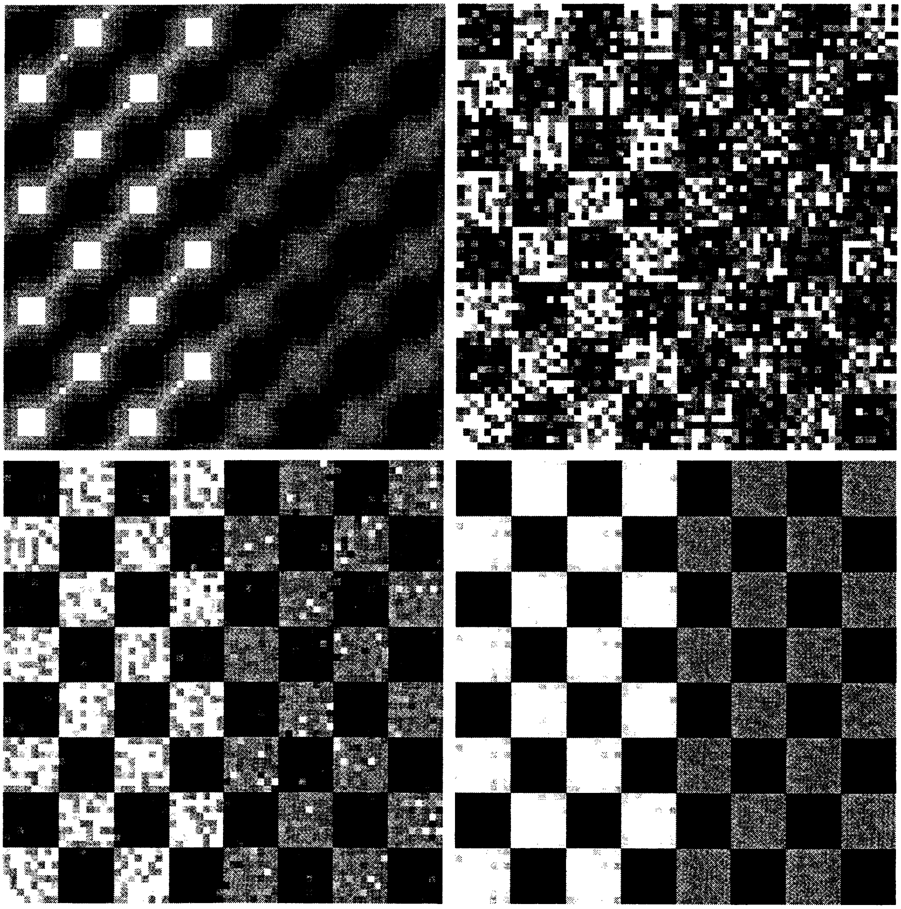
```
J = edgetaper(I, PSF)
```

This function blurs the edges of the input image, `I`, using the point spread function, `PSF`. The output image, `J`, is the weighted sum of `I` and its blurred version. The weighting array, determined by the autocorrelation function of `PSF`, makes `J` equal to `I` in its central region, and equal to the blurred version of `I` near the edges.

EXAMPLE 5.8: Using function `deconvwnr` to restore a blurred, noisy image.

■ Figure 5.8(a) is the same as Fig. 5.7(d), and Fig. 5.8(b) was obtained using the command

```
>> fr1 = deconvwnr(g, PSF);
```



a b
c d

FIGURE 5.8 (a) Blurred, noisy image. (b) Result of inverse filtering. (c) Result of Wiener filtering using a constant ratio. (d) Result of Wiener filtering using autocorrelation functions.

where g is the corrupted image and PSF is the point spread function computed in Example 5.7. As noted earlier in this section, fr1 is the result of direct inverse filtering and, as expected, the result is dominated by the effects of noise. (As in Example 5.7, all displayed images were processed with pixeldup to zoom their size to 512 × 512 pixels.)

The ratio, R, discussed earlier in this section, was obtained using the original and noise images from Example 5.7:

```
>> Sn = abs(fft2(noise)).^2;            % noise power spectrum
>> nA = sum(Sn(:))/prod(size(noise));  % noise average power
>> Sf = abs(fft2(f)).^2;                % image power spectrum
>> fA = sum(Sf(:))/prod(size(f));       % image average power
>> R = nA/fA;
```

To restore the image using this ratio we write

```
>> fr2 = deconvwnr(g, PSF, R);
```

As Fig. 5.8(c) shows, this approach gives a significant improvement over direct inverse filtering.

Finally, we use the autocorrelation functions in the restoration (note the use of fftshift for centering):

```
>> NCORR = fftshift(real(ifft2(Sn)));
>> ICORR = fftshift(real(ifft2(Sf)));
>> fr3 = deconvwnr(g, PSF, NCORR, ICORR);
```

As Fig. 5.8(d) shows, the result is close to the original, although some noise is still evident. Because the original image and noise functions were known, we were able to estimate the correct parameters, and Fig. 5.8(d) is the best that can be accomplished with Wiener deconvolution in this case. The challenge in practice, when one (or more) of these quantities is not known, is the intelligent choice of functions used in experimenting, until an acceptable result is obtained. ■

5.8 Constrained Least Squares (Regularized) Filtering

Another well-established approach to linear restoration is *constrained least squares filtering*, called *regularized filtering* in IPT documentation. The definition of 2-D *discrete convolution* is

$$h(x, y)*f(x, y) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m, n)h(x - m, y - n)$$

Using this equation, we can express the linear degradation model discussed in Section 5.1, $g(x, y) = h(x, y)*f(x, y) + \eta(x, y)$, in vector-matrix form, as

$$\mathbf{g} = \mathbf{Hf} + \boldsymbol{\eta}$$

For example, suppose that $g(x, y)$ is of size $M \times N$. Then we can form the first N elements of the vector $\mathbf{g}$ by using the image elements in the first row of $g(x, y)$, the next N elements from the second row, and so on. The resulting vector will have dimensions $MN \times 1$. These also are the dimensions of $\mathbf{f}$ and $\boldsymbol{\eta}$, as these vectors are formed in the same manner. The matrix $\mathbf{H}$ then has dimensions $MN \times MN$. Its elements are given by the elements of the preceding convolution equation.

It would be reasonable to arrive at the conclusion that the restoration problem can now be reduced to simple matrix manipulations. Unfortunately, this is not the case. For instance, suppose that we are working with images of medium size; say $M = N = 512$. Then the vectors in the preceding matrix equation would be of dimension 262,144 × 1, and matrix $\mathbf{H}$ would be of dimensions 262,144 × 262,144. Manipulating vectors and matrices of these sizes is not a trivial task. The problem is complicated further by the fact that the inverse of $\mathbf{H}$ does not always exist due to zeros in the transfer function (see Section 5.6). However, formulating the restoration problem in matrix form does facilitate derivation of restoration techniques.

Although we do not derive the method of constrained least squares that we are about to present, central to this method is the issue of the sensitivity of the inverse of $\mathbf{H}$ mentioned in the previous paragraph. One way to deal with this issue is to base optimality of restoration on a measure of smoothness, such as the second derivative of an image (e.g., the Laplacian). To be meaningful, the restoration must be constrained by the parameters of the problem at hand. Thus, what is desired is to find the minimum of a criterion function, C, defined as

$$C = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\nabla^2 f(x, y)\right]^2$$

subject to the constraint

$$\|\mathbf{g} - \mathbf{H}\hat{\mathbf{f}}\|^2 = \|\boldsymbol{\eta}\|^2$$

where $\|\mathbf{w}\|^2 \triangleq \mathbf{w}^T\mathbf{w}$ is the Euclidean vector norm,[†] $\hat{\mathbf{f}}$ is the estimate of the undegraded image, and the Laplacian operator ∇^2 is as defined in Section 3.5.1.

The frequency domain solution to this optimization problem is given by the expression

$$\hat{F}(u, v) = \left[\frac{H^*(u, v)}{|H(u, v)|^2 + \gamma|P(u, v)|^2}\right]G(u, v)$$

where γ is a parameter that must be adjusted so that the constraint is satisfied (if γ is zero we have an inverse filter solution), and $P(u, v)$ is the Fourier transform of the function

[†] For a column vector $\mathbf{w}$ with n components, $\mathbf{w}^T\mathbf{w} = \sum_{k=1}^{n} w_k^2$, where w_k, is the kth component of $\mathbf{w}$.

$$p(x, y) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

We recognize this function as the Laplacian operator introduced in Section 3.5.1. The only unknowns in the preceding formulation are γ and $\|\boldsymbol{\eta}\|^2$. However, it can be shown that γ can be found iteratively if $\|\boldsymbol{\eta}\|^2$, which is proportional to the noise power (a scalar), is known.

Constrained least squares filtering is implemented in IPT by function `deconvreg`, which has the syntax

```
fr = deconvreg(g, PSF, NOISEPOWER, RANGE)
```

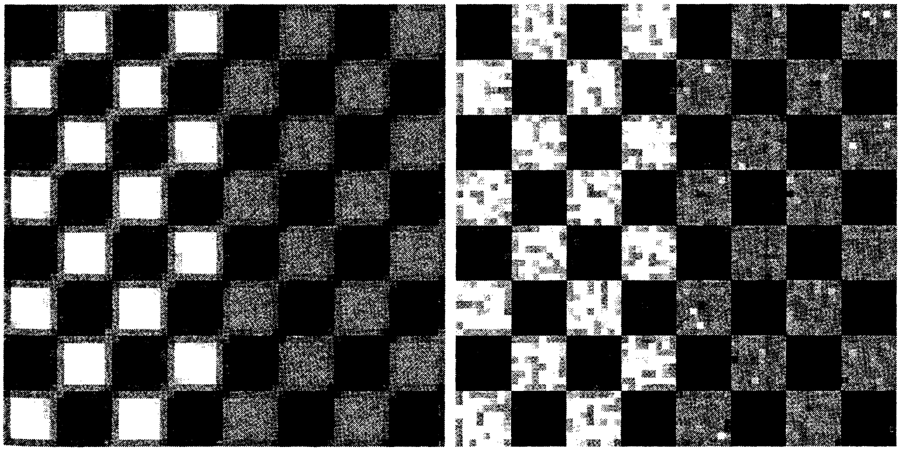
deconvreg

where `g` is the corrupted image, `fr` is the restored image, `NOISEPOWER` is proportional to $\|\boldsymbol{\eta}\|^2$, and `RANGE` is the range of values where the algorithm is limited to look for a solution for γ. The default range is $[10^{-9}, 10^{9}]$ (`[1e–10, 1e10]` in MATLAB notation). If the last two parameters are excluded from the argument, `deconvreg` produces an inverse filter solution. A good starting estimate for `NOISEPOWER` is $MN[\sigma_\eta^2 + m_\eta^2]$, where M and N are the dimensions of the image and the parameters inside the brackets are the noise variance and noise squared mean. This estimate is simply a starting point and, as the next example shows, the final value used can be quite different.

EXAMPLE 5.9: Using function `deconvreg` to restore a blurred, noisy image.

■ We now restore the image in Fig. 5.7(d) using `deconvreg`. The image is of size 64 × 64 and we know from Example 5.7 that the noise has a variance of 0.001 and zero mean. So, our initial estimate of `NOISEPOWER` is $(64)^2[0.001 - 0] \approx 4$. Figure 5.9(a) shows the result of using the command

```
>> fr = deconvreg(g, PSF, 4);
```



a b

FIGURE 5.9 (a) The image in Fig. 5.7(d) restored using a regularized filter with `NOISEPOWER` equal to 4. (b) The same image restored with `NOISEPOWER` equal to 0.4 and a `RANGE` of `[1e–7 1e7]`.

where g and PSF are from Example 5.7. The image was improved somewhat from the original, but obviously this is not a particularly good value for NOISEPOWER. After some experimenting with this parameter and parameter RANGE, we arrived at the result in Fig. 5.9(b), which was obtained using the command

```
>> fr = deconvreg(g, PSF, 0.4, [1e-7 1e7]);
```

Thus we see that we had to go down one order of magnitude on NOISEPOWER, and RANGE was tighter than the default. The Wiener filtering result in Fig. 5.8(d) is much better, but we obtained that result with full knowledge of the noise and image spectra. Without that information, the results obtainable by experimenting with the two filters often are comparable. ■

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it usually helps to use function edgetaper (see Section 5.7) prior to calling deconvreg.

5.9 Iterative Nonlinear Restoration Using the Lucy-Richardson Algorithm

The image restoration methods discussed in the previous three sections are linear. They also are "direct" in the sense that, once the restoration filter is specified, the solution is obtained via one application of the filter. This simplicity of implementation, coupled with modest computational requirements and a well-established theoretical base, have made linear techniques a fundamental tool in image restoration for many years.

During the past two decades, nonlinear iterative techniques have been gaining acceptance as restoration tools that often yield results superior to those obtained with linear methods. The principal objections to nonlinear methods are that their behavior is not always predictable and that they generally require significant computational resources. The first objection often loses importance based on the fact that nonlinear methods have been shown to be superior to linear techniques in a broad spectrum of applications (Jansson [1997]). The second objection has become less of an issue due to the dramatic increase in inexpensive computing power over the last decade. The nonlinear method of choice in the toolbox is a technique developed by Richardson [1972] and by Lucy [1974], working independently. The toolbox refers to this method as the Lucy-Richardson (L-R) algorithm, but we also see it quoted in the literature as the Richardson-Lucy algorithm.

The L-R algorithm arises from a maximum-likelihood formulation (see Section 5.10) in which the image is modeled with Poisson statistics. Maximizing the likelihood function of the model yields an equation that is satisfied when the following iteration converges:

$$\hat{f}_{k+1}(x, y) = \hat{f}_k(x, y)\left[h(-x, -y) * \frac{g(x, y)}{h(x, y) * \hat{f}_k(x, y)}\right]$$

As before, "*" indicates convolution, $\hat{f}$ is the estimate of the undegraded image, and both g and h are as defined in Section 5.1. The iterative nature of the algorithm is evident. Its nonlinear nature arises from the division by $\hat{f}$ on the right side of the equation.

As with most nonlinear methods, the question of when to stop the L-R algorithm is difficult to answer in general. The approach often followed is to observe the output and stop the algorithm when a result acceptable in a given application has been obtained.

The L-R algorithm is implemented in IPT by function `deconvlucy`, which has the basic syntax

```
        fr = deconvlucy(g, PSF, NUMIT, DAMPAR, WEIGHT)
```

deconvlucy

where `fr` is the restored image, `g` is the degraded image, `PSF` is the point spread function, `NUMIT` is the number of iterations (the default is 10), and `DAMPAR` and `WEIGHT` are defined as follows.

`DAMPAR` is a scalar that specifies the threshold deviation of the resulting image from image `g`. Iterations are suppressed for the pixels that deviate within the `DAMPAR` value from their original value. This suppresses noise generation in such pixels, preserving necessary image details. The default is 0 (no damping).

`WEIGHT` is an array of the same size as `g` that assigns a weight to each pixel to reflect its quality. For example, a bad pixel resulting from a defective imaging array can be excluded from the solution by assigning to it a zero weight value. Another useful application of this array is to let it adjust the weights of the pixels according to the amount of flat-field correction that may be necessary based on knowledge of the imaging array. When simulating blurring with a specified PSF (see Example 5.7), `WEIGHT` can be used to eliminate from computation pixels that are on the border of an image and thus are blurred differently by the PSF. If the PSF is of size $n \times n$, the border of zeros used in `WEIGHT` is of width `ceil(n/2)`. The default is a unit array of the same size as input image `g`.

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it sometimes helps to use function `edgetaper` (see Section 5.7) prior to calling `deconvlucy`.

EXAMPLE 5.10: Using function `deconvlucy` to restore a blurred, noisy image.

■ Figure 5.10(a) shows an image generated using the command

```
>> f = checkerboard(8);
```

which produced a square image of size 64 × 64 pixels. As before, the size of the image was increased to size 512 × 512 for display purposes by using function `pixeldup`:

```
>> imshow(pixeldup(f, 8));
```

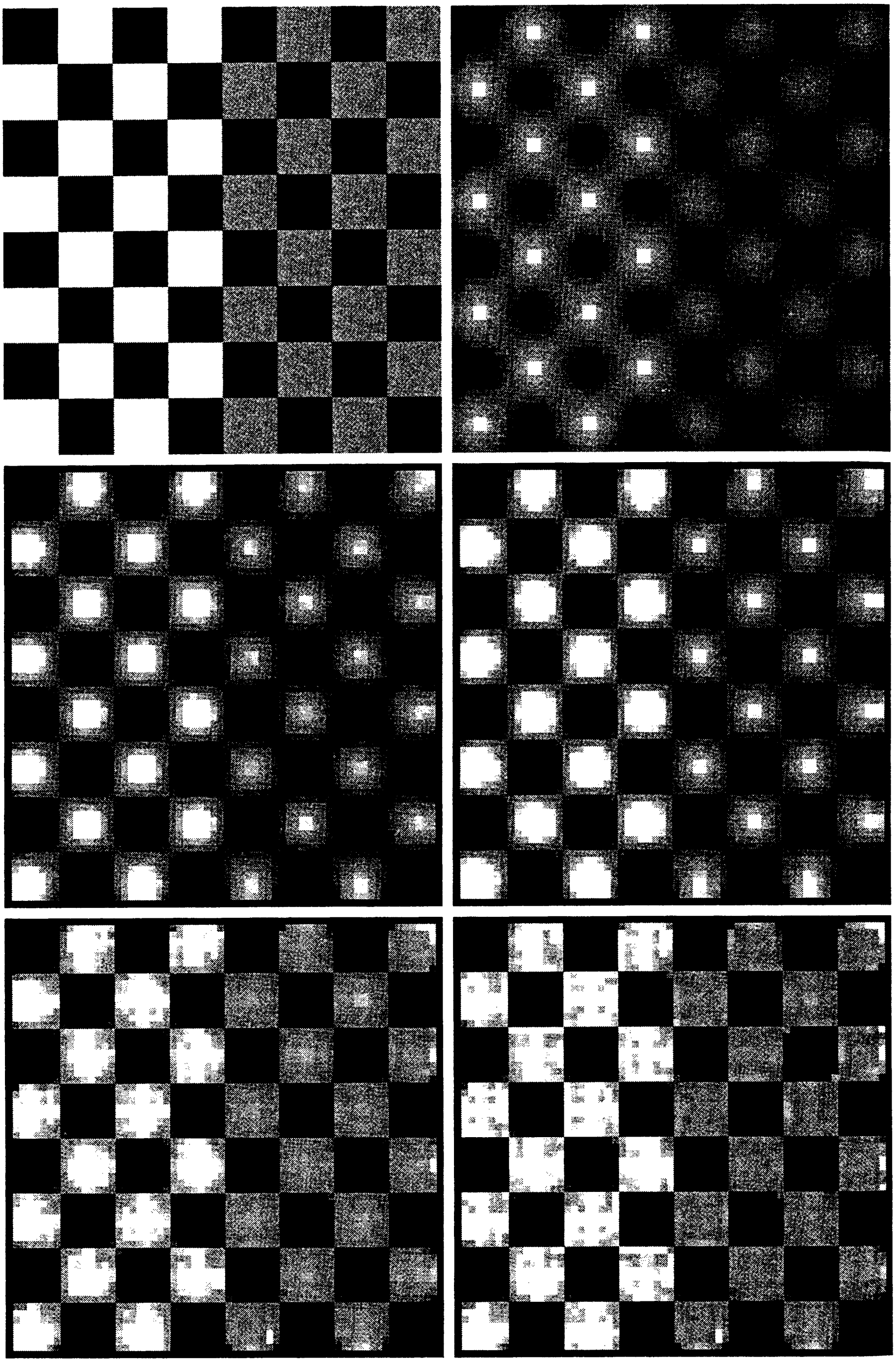


FIGURE 5.10 (a) Original image. (b) Image blurred and corrupted by Gaussian noise. (c) through (f) Image (b) restored using the L-R algorithm with 5, 10, 20, and 100 iterations, respectively.

The following command generated a Gaussian PSF of size 7 × 7 with a standard deviation of 10:

```
>> PSF = fspecial('gaussian', 7, 10);
```

Next, we blurred image `f` using `PDF` and added to it Gaussian noise of zero mean and standard deviation of 0.01:

```
>> SD = 0.01;
>> g = imnoise(imfilter(f, PSF), 'gaussian', 0, SD^2);
```

Figure 5.10(b) shows the result.

The remainder of this example deals with restoring image `g` using function `deconvlucy`. For `DAMPAR` we specified a value equal to 10 times `SD`:

```
>> DAMPAR = 10*SD;
```

Array `WEIGHT` was created using the approach discussed in the preceding explanation of this parameter:

```
>> LIM = ceil(size(PSF, 1)/2);
>> WEIGHT = zeros(size(g));
>> WEIGHT(LIM + 1:end - LIM, LIM + 1:end - LIM) = 1;
```

Array `WEIGHT` is of size 64 × 64 with a border of 0s 4 pixels wide; the rest of the pixels are 1s.

The only variable left is `NUMIT`, the number of iterations. Figure 5.10(c) shows the result obtained using the commands

```
>> NUMIT = 5;
>> fr = deconvlucy(g, PSF, NUMIT, DAMPAR, WEIGHT);
>> imshow(pixeldup(fr, 8))
```

Although the image has improved somewhat, it is still blurry. Figures 5.10(d) and (e) show the results obtained using `NUMIT = 10` and 20. The latter result is a reasonable restoration of the blurred, noisy image. In fact, further increases in the number of iterations did not produce dramatic improvements in the restored result. For example, Fig. 5.10(f) was obtained using 100 iterations. This image is only slightly sharper and brighter than the result obtained using 20 iterations. The thin black border seen in all results was caused by the 0s in array `WEIGHT`. ■

5.10 Blind Deconvolution

One of the most difficult problems in image restoration is obtaining a suitable estimate of the PSF to use in restoration algorithms such as those discussed in the preceding sections. As noted earlier, image restoration methods that are not based on specific knowledge of the PSF are called *blind deconvolution* algorithms.

An approach to blind deconvolution that has received significant attention over the past two decades is based on maximum-likelihood estimation (MLE), an optimization strategy used for obtaining estimates of quantities corrupted by random noise. Briefly, an interpretation of MLE is to think of image data as random quantities having a certain likelihood of being produced from a family of other possible random quantities. The likelihood function is expressed in terms of $g(x, y)$, $f(x, y)$, and $h(x, y)$ (see Section 5.1), and the problem then is to find the maximum of the likelihood function. In blind deconvolution the optimization problem is solved iteratively with specified constraints and, assuming convergence, the specific $f(x, y)$ and $h(x, y)$ that result in a maximum are the restored image *and* the PSF.

A derivation of MLE blind deconvolution is outside the scope of the present discussion, but the reader can gain a solid understanding of this area by consulting the following references: For background on maximum-likelihood estimation, see the classic book by Van Trees [1968]. For a review of some of the original image-processing work in this area see Dempster et al. [1977], and for some of its later extensions see Holmes [1992]. A good general reference book on deconvolution is Jansson [1997]. For detailed examples on the use of deconvolution in microscopy and in astronomy, see Holmes et al. [1995] and Hanisch et al. [1997], respectively.

The toolbox performs blind deconvolution via function deconvblind, which has the basic syntax

deconvblind

```
[fr, PSFe] = deconvblind(g, INITPSF)
```

where g is the degraded image, INITPSF is an initial estimate of the point spread function, PSFe is the final computed estimate of this function, and fr is the image restored using the estimated PSF. The algorithm used to obtain the restored image is the L-R iterative restoration algorithm explained in Section 5.9. The PSF estimation is affected strongly by the size of its initial guess, and less by its values (an array of 1s is a reasonable starting guess).

The number of iterations performed with the preceding syntax is 10 by default. Additional parameters may be included in the function to control the number of iterations and other features of the restoration, as in the following syntax:

```
[fr, PSFe] = deconvblind(g, INITPSF, NUMIT, DAMPAR, WEIGHT)
```

where NUMIT, DAMPAR, and WEIGHT are as described for the L-R algorithm in the previous section.

If the restored image exhibits ringing introduced by the discrete Fourier transform used in the algorithm, it sometimes helps to use function edgetaper (see Section 5.7) prior to calling deconvblind.

EXAMPLE 5.11: Using function deconvblind to estimate a PSF.

■ Figure 5.11(a) is the PSF used to generate the degraded image in Fig. 5.10(b):

```
>> PSF = fspecial('gaussian', 7, 10);
>> imshow(pixeldup(PSF, 73), [ ]);
```

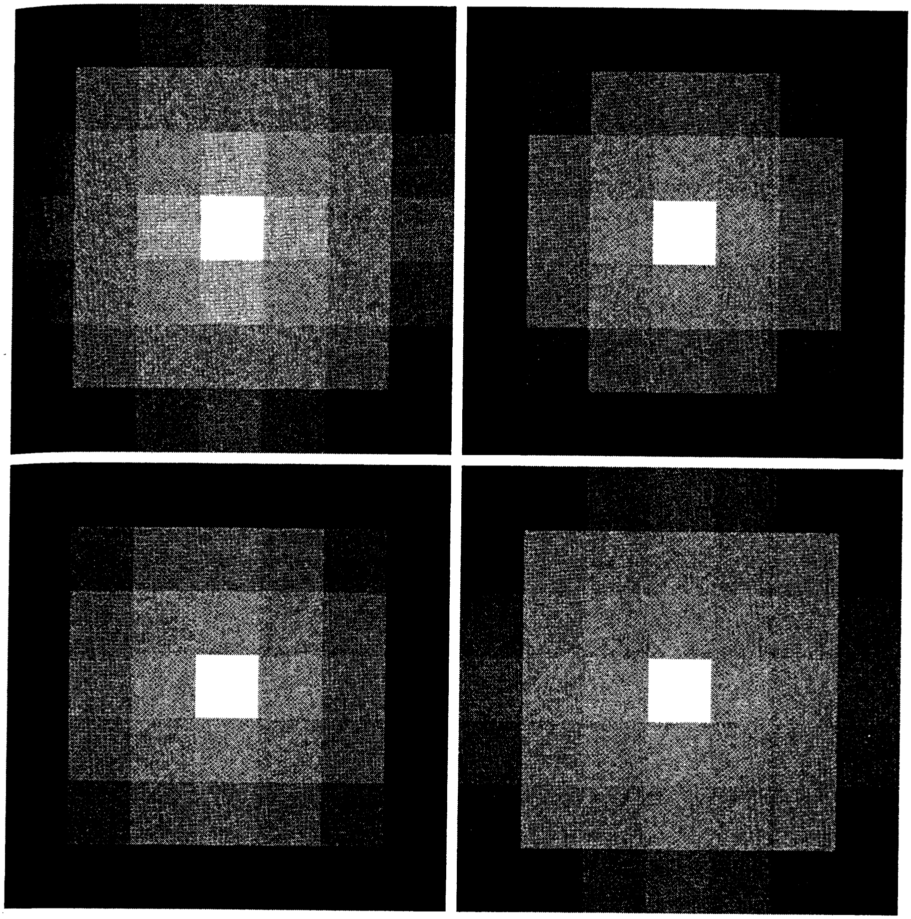


FIGURE 5.11 (a) Original PSF. (b) through (d) Estimates of the PSF using 5, 10, and 20 iterations in function `deconvblind`.

As in Example 5.10, the degraded image in question was obtained with the commands

```
>> SD = 0.01;
>> g = imnoise(imfilter(f, PSF), 'gaussian', 0, SD^2);
```

In the present example we are interested in using function `deconvblind` to obtain an estimate of the PSF, given only the degraded image `g`. Figure 5.11(b) shows the PSF resulting from the following commands:

```
>> INITPSF = ones(size(PSF));
>> NUMIT = 5;
>> [fr, PSFe] = deconvblind(g, INITPSF, NUMIT, DAMPAR, WEIGHT);
>> imshow(pixeldup(PSFe, 73), [ ]);
```

where we used the same values as in Example 5.10 for `DAMPAR` and `WEIGHT`.

Figures 5.11(c) and (d), displayed in the same manner as `PSFe`, show the PSFs obtained with 10, and 20 iterations, respectively. The latter result is close to the true PSF in Fig. 5.11(a). ■

5.11 Geometric Transformations and Image Registration

We conclude this chapter with an introduction to geometric transformations for image restoration. Geometric transformations modify the spatial relationship between pixels in an image. They are often called *rubber-sheet transformations* because they may be viewed as printing an image on a sheet of rubber and then stretching this sheet according to a predefined set of rules.

Geometric transformations are used frequently to perform *image registration*, a process that takes two images of the same scene and aligns them so they can be merged for visualization, or for quantitative comparison. In the following sections, we discuss (1) spatial transformations and how to define and visualize them in MATLAB; (2) how to apply spatial transformations to images; and (3) how to determine spatial transformations for use in image registration.

5.11.1 Geometric Spatial Transformations

Suppose that an image, f, defined over a (w, z) coordinate system, undergoes geometric distortion to produce an image, g, defined over an (x, y) coordinate system. This transformation (of the coordinates) may be expressed as

$$(x, y) = T\{(w, z)\}$$

For example, if $(x, y) = T\{(w, v)\} = (w/2, z/2)$, the "distortion" is simply a shrinking of f by half in both spatial dimensions, as illustrated in Fig. 5.12.

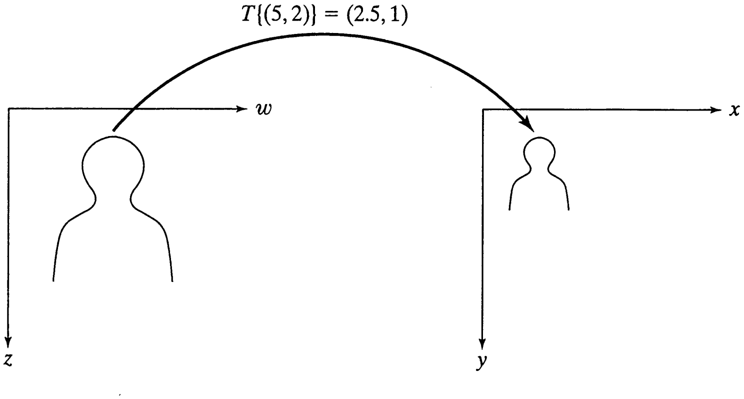


FIGURE 5.12 A simple spatial transformation. (Note that the xy-axes in this figure do not correspond to the image axis coordinate system defined in Section 2.1.1. As mentioned in that section, IPT on occasion uses the so-called spatial coordinate system in which y designates rows and x designates columns. This is the system used throughout this section in order to be consistent with IPT documentation on the topic of geometric transformations.)

One of the most commonly used forms of spatial transformations is the *affine transform* (Wolberg [1990]). The affine transform can be written in matrix form as

$$[x \quad y \quad 1] = [w \quad z \quad 1]\,\mathbf{T} = [w \quad z \quad 1]\begin{bmatrix} t_{11} & t_{12} & 0 \\ t_{21} & t_{22} & 0 \\ t_{31} & t_{32} & 1 \end{bmatrix}$$

This transformation can scale, rotate, translate, or shear a set of points, depending on the values chosen for the elements of **T**. Table 5.3 shows how to choose the values of the elements to achieve different transformations.

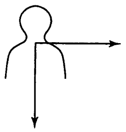
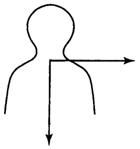
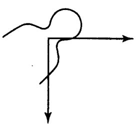
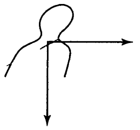
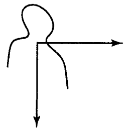
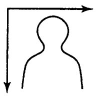
IPT represents spatial transformations using a so-called *tform structure.* One way to create such a structure is by using function `maketform`, whose calling syntax is

See Sections 2.10.6 and 11.1.1 for a discussion of structures.

```
tform = maketform(transform_type, transform_parameters)
```

maketform

TABLE 5.3
Types of affine transformations.

Type	Affine Matrix, T	Coordinate Equations	Diagram
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = z$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = s_x w$ $y = s_y z$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w\cos\theta - z\sin\theta$ $y = w\sin\theta + z\cos\theta$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w + \alpha z$ $y = z$	
Shear (vertical)	$\begin{bmatrix} 1 & \beta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = \beta w + z$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$	$x = w + \delta_x$ $y = z + \delta_y$	

The first input argument, transform_type, is one of these strings: 'affine', 'projective', 'box', 'composite', or 'custom'. These transform types are described in Table 5.4, Section 5.11.3. Additional arguments depend on the transform type and are described in detail in the help page for maketform.

In this section our interest is on affine transforms. For example, one way to create an affine tform is to provide the **T** matrix directly, as in

```
>> T = [2 0 0; 0 3 0; 0 0 1];
>> tform = maketform('affine', T)
tform =
       ndims_in: 2
      ndims_out: 2
    forward_fcn: @fwd_affine
    inverse_fcn: @inv_affine
          tdata: [1 x 1 struct]
```

Although it is not necessary to use the fields of the tform structure directly to be able to apply it, information about **T**, as well as about $\mathbf{T}^{-1}$, is contained in the tdata field:

```
>> tform.tdata

ans =
       T: [3 x 3 double]
    Tinv: [3 x 3 double]

>> tform.tdata.T
ans =
    2   0   0
    0   3   0
    0   0   1

>> tform.tdata.Tinv
ans =
    0.5000         0         0
         0    0.3333         0
         0         0    1.0000
```

tformfwd
tforminv

IPT provides two functions for applying a spatial transformation to points: tformfwd computes the forward transformation, $T\{(w, z)\}$, and tforminv computes the inverse transformation, $T^{-1}\{(x, y)\}$. The calling syntax for tformfwd is XY = tformfwd(WZ, tform). Here, WZ is a $P \times 2$ matrix of points; each row of WZ contains the w and z coordinates of one point. Similarly, XY is a $P \times 2$ matrix of points; each row contains the x and y coordinates of a transformed point. For example, the following commands compute the forward transformation of a pair of points, followed by the inverse transform to verify that we get back the original data:

```
>> WZ = [1 1; 3 2];
>> XY = tformfwd(WZ, tform)
XY =
     2    3
     6    6

>> WZ2 = tforminv(XY, tform)
WZ2 =
     1    1
     3    2
```

To get a better feel for the effects of a particular spatial transformation, it is often useful to see how it transforms a set of points arranged on a grid. The following M-function, vistformfwd, constructs a grid of points, transforms the grid using tformfwd, and then plots the grid and the transformed grid side by side for comparison. Note the combined use of functions meshgrid (Section 2.10.4) and linspace (Section 2.8.1) for creating the grid. The following code also illustrates the use of some of the functions discussed thus far in this section.

vistformfwd

```
function vistformfwd(tform, wdata, zdata, N)
%VISTFORMFWD Visualize forward geometric transform.
%   VISTFORMFWD(TFORM, WRANGE, ZRANGE, N) shows two plots: an N-by-N
%   grid in the W-Z coordinate system, and the spatially transformed
%   grid in the X-Y coordinate system.  WRANGE and ZRANGE are
%   two-element vectors specifying the desired range for the grid. N
%   can be omitted, in which case the default value is 10.

if nargin < 4
   N = 10;
end

% Create the w-z grid and transform it.
[w, z] = meshgrid(linspace(wdata(1), zdata(2), N), ...
                  linspace(wdata(1), zdata(2), N));

wz = [w(:) z(:)];
xy = tformfwd([w(:) z(:)], tform);

% Calculate the minimum and maximum values of w and x,
% as well as z and y. These are used so the two plots can be
% displayed using the same scale.
x = reshape(xy(:, 1), size(w)); % reshape is discussed in Sec. 8.2.2.
y = reshape(xy(:, 2), size(z));
wx = [w(:); x(:)];
wxlimits = [min(wx) max(wx)];
zy = [z(:); y(:)];
zylimits = [min(zy) max(zy)];

% Create the w-z plot.
subplot(1,2,1) % See Section 7.2.1 for a discussion of this function.
plot(w, z, 'b'), axis equal, axis ij
hold on
plot(w', z', 'b')
hold off
xlim(wxlimits)
ylim(zylimits)
```

```
set(gca, 'XAxisLocation', 'top')
xlabel('w'), ylabel('z')

% Create the x-y plot.
subplot(1, 2, 2)
plot(x, y, 'b'), axis equal, axis ij
hold on
plot(x', y', 'b')
hold off
xlim(wxlimits)
ylim(zylimits)
set(gca, 'XAxisLocation', 'top')
xlabel('x'), ylabel('y')
```

EXAMPLE 5.12: Visualizing affine transforms using vistformfwd.

■ In this example we use vistformfwd to visualize the effect of several different affine transforms. We also explore an alternate way to create an affine tform using maketform. We start with an affine transform that scales horizontally by a factor of 3 and vertically by a factor of 2:

```
>> T1 = [3 0 0; 0 2 0; 0 0 1];
>> tform1 = maketform('affine', T1);
>> vistformfwd(tform1, [0 100], [0 100]);
```

Figures 5.13(a) and (b) show the result.

A shearing effect occurs when t_{21} or t_{12} is nonzero in the affine **T** matrix, such as

```
>> T2 = [1 0 0; .2 1 0; 0 0 1];
>> tform2 = maketform('affine', T2);
>> vistformfwd(tform2, [0 100], [0 100]);
```

Figures 5.13(c) and (d) show the effect of the shearing transform on a grid.

An interesting property of affine transforms is that the composition of several affine transforms is also an affine transform. Mathematically, affine transforms can be generated simply by using multiplication of the **T** matrices. The next block of code shows how to generate and visualize an affine transform that is a combination of scaling, rotation, and shear.

```
>> Tscale = [1.5 0 0; 0 2 0; 0 0 1];
>> Trotation = [cos(pi/4) sin(pi/4) 0
               -sin(pi/4) cos(pi/4) 0
                0 0 1];
>> Tshear = [1 0 0; .2 1 0; 0 0 1];
>> T3 = Tscale * Trotation * Tshear;
>> tform3 = maketform('affine', T3);
>> vistformfwd(tform3, [0 100], [0 100])
```

Figures 5.13(e) and (f) show the results. ■

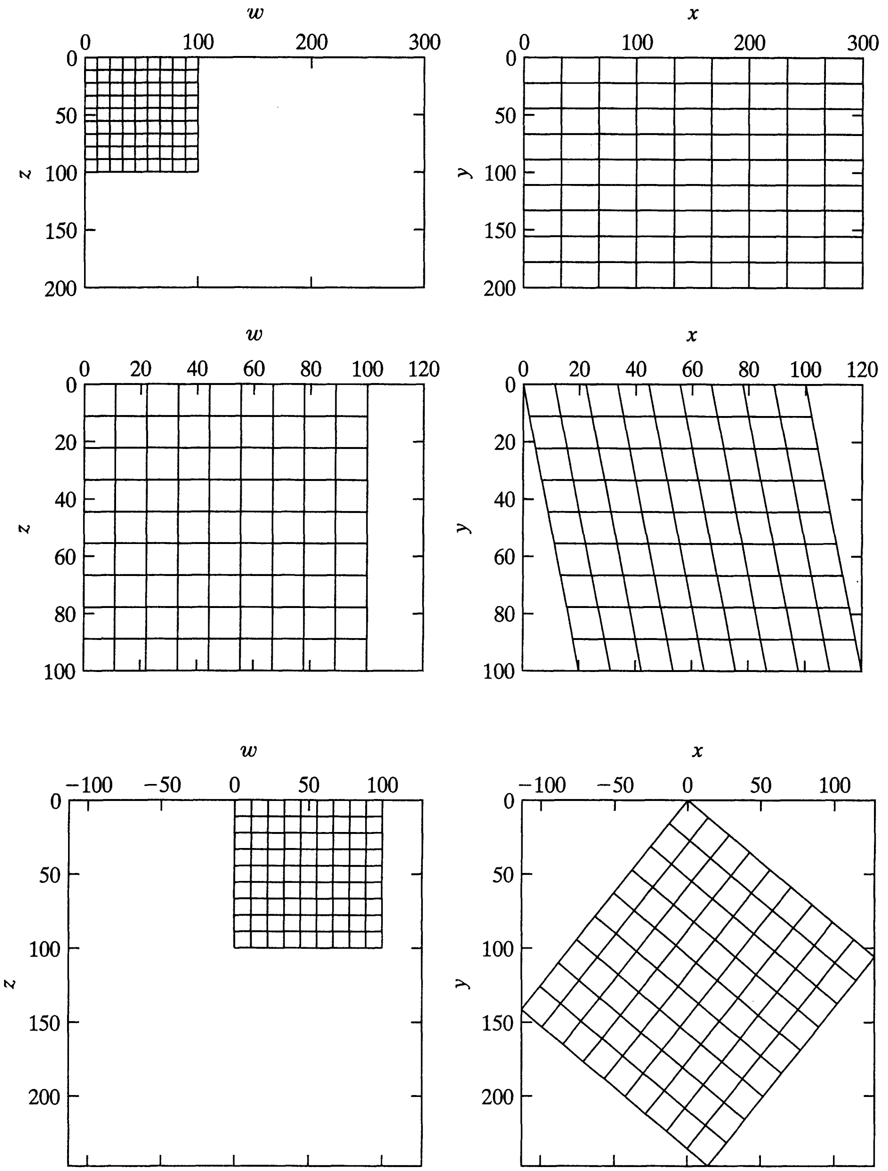


FIGURE 5.13 Visualizing affine transformations using grids. (a) Grid 1. (b) Grid 1 transformed using tform1. (c) Grid 2. (d) Grid 2 transformed using tform2. (e) Grid 3. (f) Grid 3 transformed using tform3.

5.11.2 Applying Spatial Transformations to Images

Most computational methods for spatially transforming an image fall into one of two categories: methods that use *forward mapping*, and methods that use *inverse mapping*. Methods based on forward mapping scan each input pixel in turn, copying its value into the output image at the location determined by $T\{(w, z)\}$. One problem with the forward mapping procedure is that two or more different pixels in the input image could be transformed into the same pixel in the output image, raising the question of how to combine multiple input pixel values into a single output pixel value. Another potential problem is that some output pixels may not be assigned a value at all. In a more sophisticated form of forward mapping, the four corners of each input pixel are mapped onto quadrilaterals in the output image. Input pixels are distributed among output pixels according to how much each output pixel is covered, relative to the area of each output pixel. Although more accurate, this form of forward mapping is complex and computationally expensive to implement.

IPT function imtransform uses inverse mapping instead. An inverse mapping procedure scans each output pixel in turn, computes the corresponding location in the input image using $T^{-1}\{(x, y)\}$, and interpolates among the nearest input image pixels to determine the output pixel value. Inverse mapping is generally easier to implement than forward mapping.

The basic calling syntax for imtransform is

imtransform

```
g = imtransform(f, tform, interp)
```

where interp is a string that specifies how input image pixels are interpolated to obtain output pixels; interp can be either 'nearest', 'bilinear', or 'bicubic'. The interp input argument can be omitted, in which case it defaults to 'bilinear'. As with the restoration examples given earlier, function checkerboard is useful for generating test images for experimenting with spatial transformations.

EXAMPLE 5.13: Spatially transforming images.

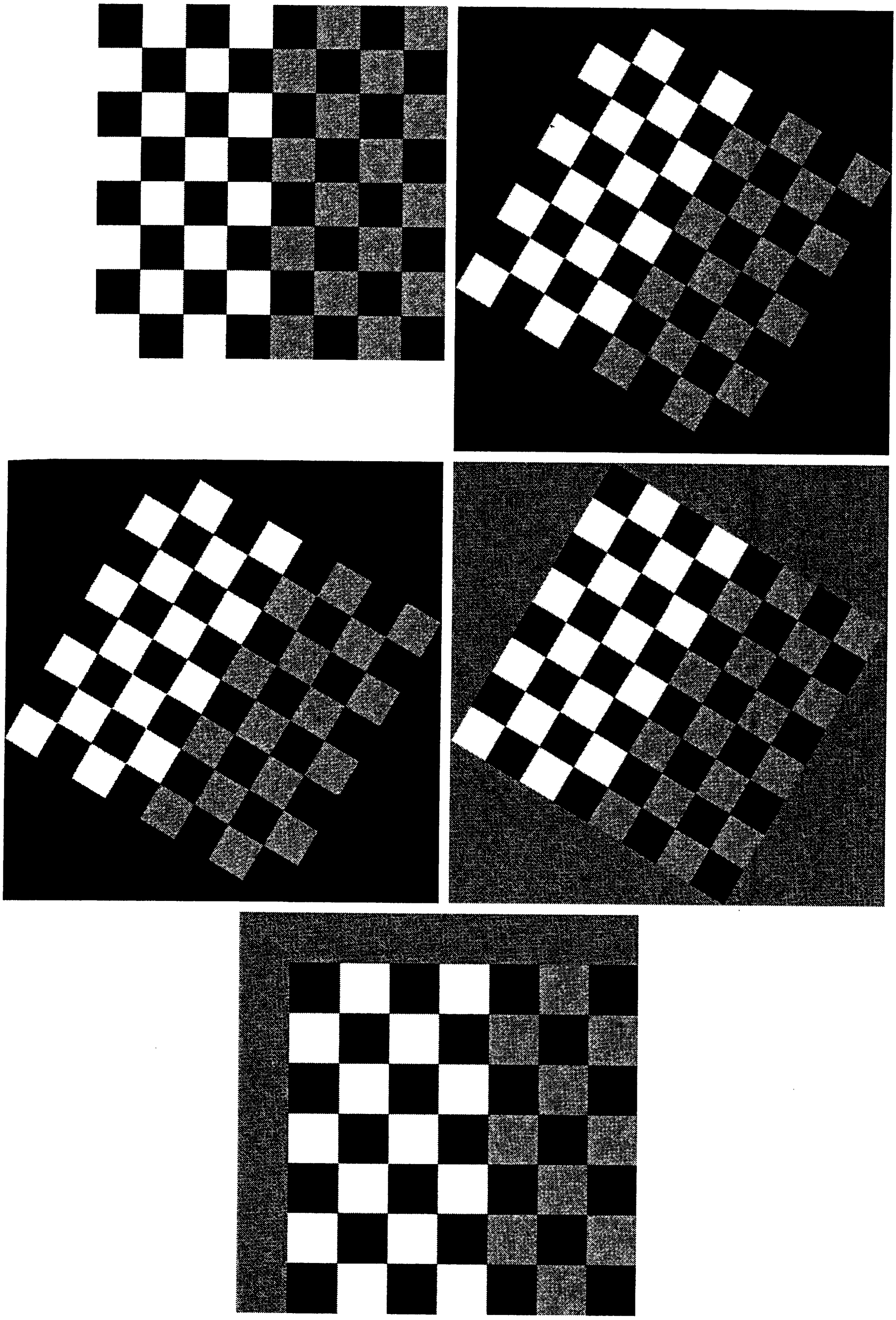
■ In this example we use functions checkerboard and imtransform to explore a number of different aspects of transforming images. A *linear conformal transformation* is a type of affine transformation that preserves shapes and angles. Linear conformal transformations consist of a scale factor, a rotation angle, and a translation. The affine transformation matrix in this case has the form

$$\mathbf{T} = \begin{bmatrix} s\cos\theta & s\sin\theta & 0 \\ -s\sin\theta & s\cos\theta & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$$

The following commands generate a linear conformal transformation and apply it to a test image.

```
>> f = checkerboard(50);
>> s = 0.8;
>> theta = pi/6;
>> T = [s*cos(theta) s*sin(theta) 0
     -s*sin(theta) s*cos(theta) 0
      0   0   1];
>> tform = maketform('affine', T);
>> g = imtransform(f, tform);
```

Figures 5.14(a) and (b) show the original and transformed checkerboard images. The preceding call to imtransform used the default interpolation method,



a b
c d
e

FIGURE 5.14 Affine transformations of the checkerboard image. (a) Original image. (b) Linear conformal transformation using the default interpolation (bilinear). (c) Using nearest neighbor interpolation. (d) Specifying an alternate fill value. (e) Controlling the output space location so that translation is visible.

'bilinear'. As mentioned earlier, we can select a different interpolation method, such as nearest neighbor, by specifying it explicitly in the call to imtransform:

```
>> g2 = imtransform(f, tform, 'nearest');
```

Figure 5.14(c) shows the result. Nearest neighbor interpolation is faster than bilinear interpolation, and it may be more appropriate in some situations, but it generally produces results inferior to those obtained with bilinear interpolation.

Function imtransform has several additional optional parameters that are useful at times. For example, passing it a FillValue parameter controls the color imtransform uses for pixels outside the domain of the input image:

```
>> g3 = imtransform(f, tform, 'FillValue', 0.5);
```

In Fig. 5.14(d) the pixels outside the original image are mid-gray instead of black.

Other extra parameters can help resolve a common source of confusion regarding translating images using imtransform. For example, the following commands perform a pure translation:

```
>> T2 = [1 0 0; 0 1 0; 50 50 1];
>> tform2 = maketform('affine', T2);
>> g4 = imtransform(f, tform2);
```

The result, however, would be identical to the original image in Fig. 5.14(a). This effect is caused by default behavior of imtransform. Specifically, imtransform determines the bounding box (see Section 11.4.1 for a definition of the term *bounding box*) of the output image in the output coordinate system, and by default it only performs inverse mapping over that bounding box. This effectively *undoes* the translation. By specifying the parameters XData and YData, we can tell imtransform exactly where in output space to compute the result. XData is a two-element vector that specifies the location of the left and right columns of the output image; YData is a two-element vector that specifies the location of the top and bottom rows of the output image. The following command computes the output image in the region between $(x, y) = (1, 1)$ and $(x, y) = (400, 400)$.

```
>> g5 = imtransform(f, tform2,'XData', [1 400], 'YData', [1 400], ...
                    'FillValue', 0.5);
```

Figure 5.14(e) shows the result.

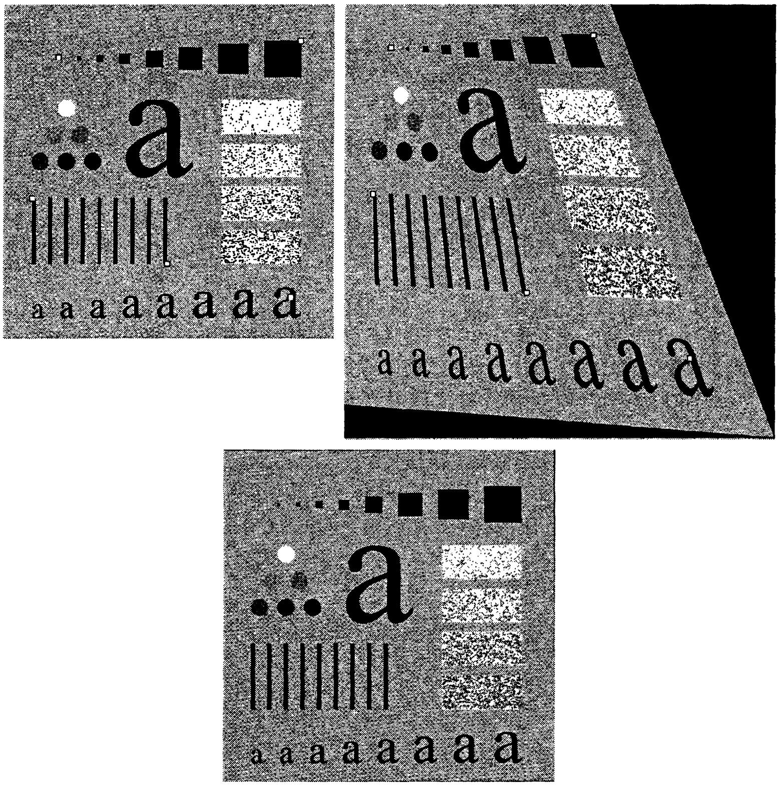
Other settings of imtransform and related IPT functions provide additional control over the result, particularly over how interpolation is performed. Most of the relevant toolbox documentation is in the help pages for functions imtransform and makeresampler. ■

5.11.3 Image Registration

Image registration methods seek to align two images of the same scene. For example, it may be of interest to align two or more images taken at roughly the same time, but using different instruments, such as an MRI (magnetic resonance imaging) scan and a PET (positron emission tomography) scan. Or, perhaps the images were taken at different times using the same instrument, such as satellite images of a given location taken several days, months, or even years apart. In either case, combining the images or performing quantitative analysis and comparisons requires compensating for geometric aberrations caused by differences in camera angle, distance, and orientation; sensor resolution; shift in subject position; and other factors.

The toolbox supports image registration based on the use of *control points*, also known as *tie points*, which are a subset of pixels whose locations in the two images are known or can be selected interactively. Figure 5.15 illustrates the idea of control points using a test pattern and a version of the test pattern that has undergone projective distortion. Once a sufficient number of control points have been chosen, IPT function `cp2tform` can be used to fit a specified type of spatial

`cp2tform`



a b
c

FIGURE 5.15 Image registration based on control points. (a) Original image with control points (the small circles superimposed on the image). (b) Geometrically distorted image with control points. (c) Corrected image using a projective transformation inferred from the control points.

TABLE 5.4 Transformation types supported by cp2tform and maketform.

Transformation Type	Description	Functions
Affine	Combination of scaling, rotation, shearing, and translation. Straight lines remain straight and parallel lines remain parallel.	maketform cp2tform
Box	Independent scaling and translation along each dimension; a subset of affine.	maketform
Composite	A collection of spatial transformations that are applied sequentially.	maketform
Custom	User-defined spatial transform; user provides functions that define T and T^{-1}.	maketform
Linear conformal	Scaling (same in all dimensions), rotation, and translation; a subset of affine.	cp2tform
LWM	Local weighted mean; a locally-varying spatial transformation.	cp2tform
Piecewise linear	Locally varying spatial transformation.	cp2tform
Polynomial	Input spatial coordinates are a polynomial function of output spatial coordinates.	cp2tform
Projective	As with the affine transformation, straight lines remain straight, but parallel lines converge toward vanishing points.	maketform cp2tform

transformation to the control points (using least squares techniques). The spatial transformation types supported by cp2tform are listed in Table 5.4.

For example, let f denote the image in Fig. 5.15(a) and g the image in Fig. 5.15(b). The control point coordinates in f are (83, 81), (450, 56), (43, 293), (249, 392), and (436, 442). The corresponding control point locations in g are (68, 66), (375, 47), (42, 286), (275, 434), and (523, 532). Then the commands needed to align image g to image f are as follows:

```
>> basepoints = [83 81; 450 56; 43 293; 249 392; 436 442];
>> inputpoints = [68 66; 375 47; 42 286; 275 434; 523 532];
>> tform = cp2tform(inputpoints, basepoints, 'projective');
>> gp = imtransform(g, tform, 'XData', [1 502], 'YData', [1 502]);
```

Figure 5.15(c) shows the transformed image.

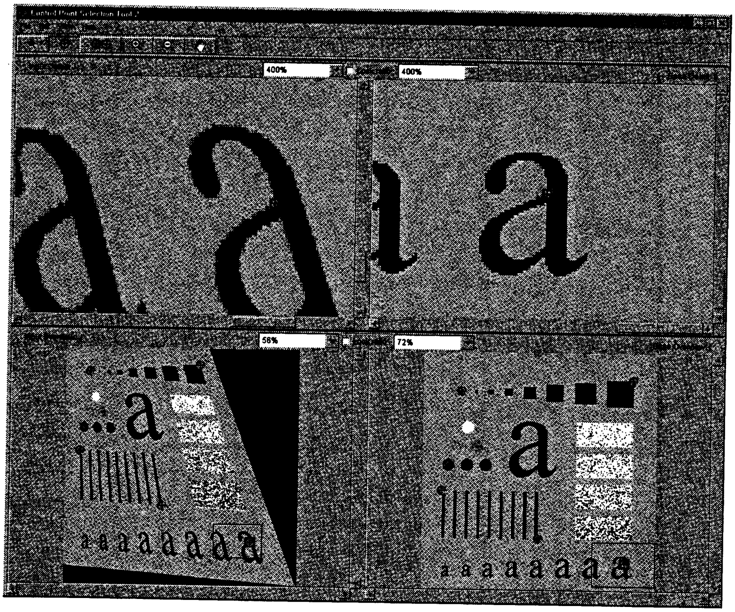


FIGURE 5.16 Interactive tool for choosing control points.

The toolbox includes a graphical user interface designed for the interactive selection of control points on a pair of images. Figure 5.16 shows a screen capture of this tool, which is invoked by the command `cpselect`.

`cpselect`

Summary

The material in this chapter is a good overview of how MATLAB and IPT functions can be used for image restoration, and how they can be used as the basis for generating models that help explain the degradation to which an image has been subjected. The capabilities of IPT for noise generation were enhanced significantly by the development in this chapter of functions `imnoise2` and `imnoise3`. Similarly, the spatial filters available in function `spfilt`, especially the nonlinear filters, are a significant extension of IPT's capabilities in this area. These functions are perfect examples of how relatively simple it is to incorporate MATLAB and IPT functions into new code to create applications that enhance the capabilities of an already large set of existing tools.