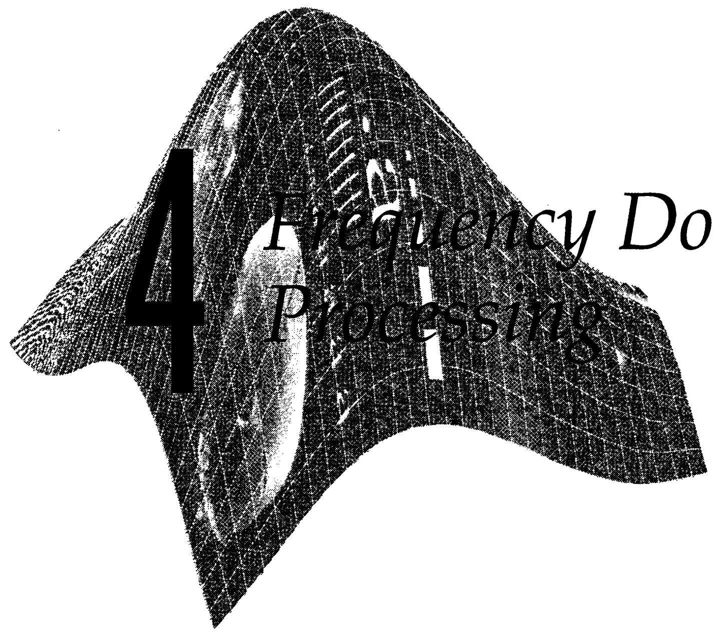


4 Frequency Domain Processing

Preview

For the most part, this chapter parallels the filtering topics discussed in Chapter 3, but with all filtering carried out in the frequency domain via the Fourier transform. In addition to being a cornerstone of linear filtering, the Fourier transform offers considerable flexibility in the design and implementation of filtering solutions in areas such as image enhancement, image restoration, image data compression, and a host of other applications of practical interest. In this chapter, the focus is on the foundation of how to perform frequency domain filtering in MATLAB. As in Chapter 3, we illustrate filtering in the frequency domain with examples of image enhancement, including lowpass filtering, basic highpass filtering, and high-frequency emphasis filtering. We also show briefly how spatial and frequency domain processing can be used in combination to yield results that are superior to using either type of processing alone. The concepts and techniques developed in the following sections are quite general, as is amply illustrated by other applications of this material in Chapters 5, 8, and 11.

4.1 The 2-D Discrete Fourier Transform

Let $f(x, y)$, for $x = 0, 1, 2, \dots, M - 1$ and $y = 0, 1, 2, \dots, N - 1$, denote an $M \times N$ image. The 2-D, *discrete Fourier transform* (DFT) of f, denoted by $F(u, v)$, is given by the equation

$$F(u, v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x, y)e^{-j2\pi(ux/M+vy/N)}$$

for $u = 0, 1, 2, \dots, M - 1$ and $v = 0, 1, 2, \dots, N - 1$. We could expand the exponential into sines and cosines with the variables u and v determining their frequencies (x and y are summed out). The *frequency domain* is simply the coordinate system spanned by $F(u, v)$ with u and v as (frequency) variables. This is analogous to the *spatial domain* studied in the previous chapter, which is the coordinate system spanned by $f(x, y)$, with x and y as (spatial) variables. The $M \times N$ rectangular region defined by $u = 0, 1, 2, \ldots, M - 1$ and $v = 0, 1, 2, \ldots, N - 1$ is often referred to as the *frequency rectangle*. Clearly, the frequency rectangle is of the same size as the input image.

The inverse, discrete Fourier transform is given by

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) e^{j2\pi(ux/M + vy/N)}$$

for $x = 0, 1, 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$. Thus, given $F(u, v)$, we can obtain $f(x, y)$ back by means of the inverse DFT. The values of $F(u, v)$ in this equation sometimes are referred to as the *Fourier coefficients* of the expansion.

In some formulations of the DFT, the $1/MN$ term is placed in front of the transform and in others it is used in front of the inverse. To be consistent with MATLAB's implementation of the Fourier transform, we assume throughout the book that the term is in front of the inverse, as shown in the preceding equation. Because array indices in MATLAB start at 1, rather than 0, `F(1, 1)` and `f(1, 1)` in MATLAB correspond to the mathematical quantities $F(0, 0)$ and $f(0, 0)$ in the transform and its inverse.

The value of the transform at the origin of the frequency domain [i.e., $F(0, 0)$] is called the *dc* component of the Fourier transform. This terminology is from electrical engineering, where "dc" signifies direct current (current of zero frequency). It is not difficult to show that $F(0, 0)$ is equal to MN times the average value of $f(x, y)$.

Even if $f(x, y)$ is real, its transform in general is complex. The principal method of visually analyzing a transform is to compute its *spectrum* [i.e., the magnitude of $F(u, v)$] and display it as an image. Letting $R(u, v)$ and $I(u, v)$ represent the real and imaginary components of $F(u, v)$, the Fourier spectrum is defined as

$$|F(u, v)| = [R^2(u, v) + I^2(u, v)]^{1/2}$$

The *phase angle* of the transform is defined as

$$\phi(u, v) = \tan^{-1}\left[\frac{I(u, v)}{R(u, v)}\right]$$

The preceding two functions can be used to represent $F(u, v)$ in the familiar polar representation of a complex quantity:

$$F(u, v) = |F(u, v)| e^{-j\phi(u, v)}$$

The *power spectrum* is defined as the square of the magnitude:

$$\begin{aligned} P(u, v) &= |F(u, v)|^2 \\ &= R^2(u, v) + I^2(u, v) \end{aligned}$$

For purposes of visualization it typically is immaterial whether we view $|F(u, v)|$ or $P(u, v)$.

If $f(x, y)$ is real, its Fourier transform is conjugate symmetric about the origin; that is,

$$F(u, v) = F^*(-u, -v)$$

which implies that the Fourier spectrum also is symmetric about the origin:

$$|F(u, v)| = |F(-u, -v)|$$

It can be shown by direct substitution into the equation for $F(u, v)$ that

$$F(u, v) = F(u + M, v) = F(u, v + N) = F(u + M, v + N)$$

In other words, the DFT is infinitely periodic in both the u and v directions, with the periodicity determined by M and N. Periodicity is also a property of the inverse DFT:

$$f(x, y) = f(x + M, y) = f(x, y + N) = f(x + M, y + N)$$

That is, an image obtained by taking the inverse Fourier transform is also infinitely periodic. This is a frequent source of confusion because it is not at all intuitive that images resulting from taking the inverse Fourier transform should turn out to be periodic. It helps to remember that this is simply a mathematical property of the DFT and its inverse. Keep in mind also that DFT implementations compute only one period, so we work with arrays of size $M \times N$.

The periodicity issue becomes important when we consider *how* DFT data relate to the periods of the transform. For instance, Fig. 4.1(a) shows the spectrum of a one-dimensional transform, $F(u)$. In this case, the periodicity expression becomes $F(u) = F(u + M)$, from which it follows that $|F(u)| = |F(u + M)|$; also, because of symmetry, $|F(u)| = |F(-u)|$. The periodicity property indicates that $F(u)$ has a period of length M, and the symmetry property indicates that the magnitude of the transform is centered on the origin, as Fig. 4.1(a) shows. This figure and the preceding comments demonstrate that the magnitudes of the trans-

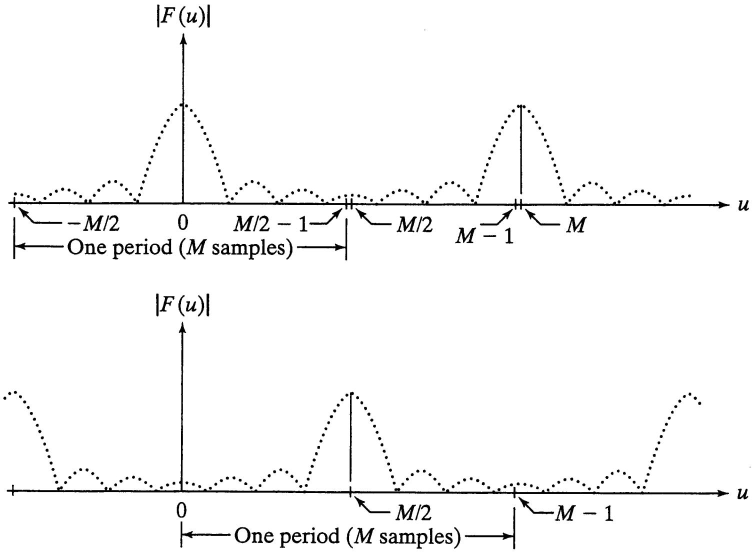


FIGURE 4.1 (a) Fourier spectrum showing back-to-back half periods in the interval $[0, M - 1]$. (b) Centered spectrum in the same interval, obtained by multiplying $f(x)$ by $(-1)^x$ prior to computing the Fourier transform.

form values from $M/2$ to $M - 1$ are repetitions of the values in the half period to the left of the origin. Because the 1-D DFT is implemented for only M points (i.e., for values of u in the interval $[0, M - 1]$), it follows that computing the 1-D transform yields two back-to-back half periods in this interval. We are interested in obtaining one full, *properly ordered* period in the interval $[0, M - 1]$. It is not difficult to show (Gonzalez and Woods [2002]) that the desired period is obtained by multiplying $f(x)$ by $(-1)^x$ prior to computing the transform. Basically, what this does is move the origin *of the transform* to the point $u = M/2$, as Fig. 4.1(b) shows. Now, the value of the spectrum at $u = 0$ in Fig. 4.1(b) corresponds to $|F(-M/2)|$ in Fig. 4.1(a). Similarly, the values at $|F(M/2)|$ and $|F(M - 1)|$ in Fig. 4.1(b) correspond to $|F(0)|$ and $|F(M/2 - 1)|$ in Fig. 4.1(a).

A similar situation exists with two-dimensional functions. Computing the 2-D DFT now yields transform points in the rectangular interval shown in Fig. 4.2(a), where the shaded area indicates values of $F(u, v)$ obtained by implementing the 2-D Fourier transform equation defined at the beginning of this section. The dashed rectangles are periodic repetitions, as in Fig. 4.1(a). The shaded region shows that the values of $F(u, v)$ now encompass four back-to-back quarter periods that meet at the point shown in Fig. 4.2(a). Visual analysis of the spectrum is simplified by moving the values at the origin of the transform to the center of the frequency rectangle. This can be accomplished by multiplying $f(x, y)$ by $(-1)^{x+y}$ prior to computing the 2-D Fourier transform. The periods then would align as shown in Fig. 4.2(b). As in the previous discussion for 1-D functions, the value of the spectrum at $(M/2, N/2)$ in Fig. 4.2(b) is the same as its value at $(0, 0)$ in Fig. 4.2(a), and the value at $(0, 0)$ in Fig. 4.2(b) is the same as the value at $(-M/2, -N/2)$ in Fig. 4.2(a). Similarly, the value at $(M - 1, N - 1)$ in Fig. 4.2(b) is the same as the value at $(M/2 - 1, N/2 - 1)$ in Fig. 4.2(a).

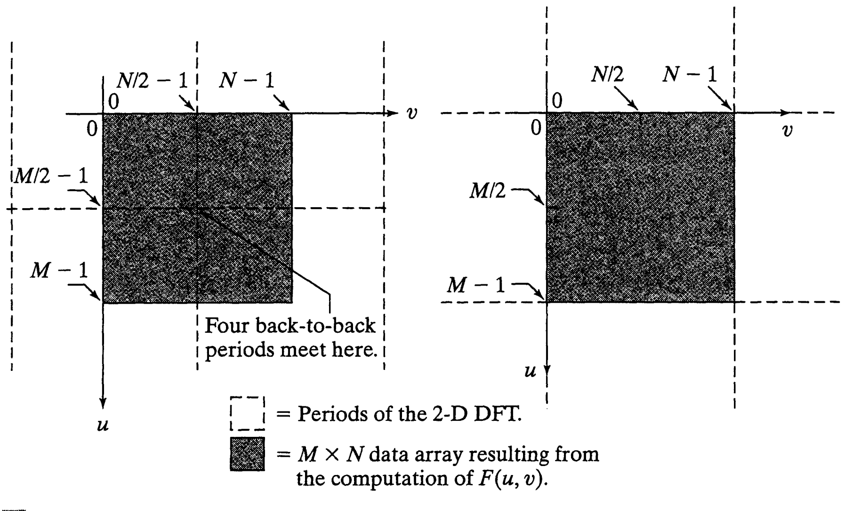

$N/2 - 1$
$N - 1$
0
v
0
$M/2 - 1$
$M - 1$
Four back-to-back periods meet here.
u
$N/2$
$N - 1$
0
0
v
$M/2$
$M - 1$
u
= Periods of the 2-D DFT.
= $M \times N$ data array resulting from the computation of $F(u, v)$.

FIGURE 4.2 (a) $M \times N$ Fourier spectrum (shaded), showing four back-to-back quarter periods contained in the spectrum data. (b) Spectrum obtained by multiplying $f(x, y)$ by $(-1)^{x+y}$ prior to computing the Fourier transform. Only one period is shown shaded because this is the data that would be obtained by an implementation of the equation for $F(u, v)$.

The preceding discussion for centering the transform by multiplying $f(x, y)$ by $(-1)^{x+y}$ is an important concept that is included here for completeness. When working in MATLAB, the approach is to compute the transform without multiplication by $(-1)^{x+y}$ and then to rearrange the data afterwards using function `fftshift`. This function and its use are discussed in the following section.

4.2 Computing and Visualizing the 2-D DFT in MATLAB

The DFT and its inverse are obtained in practice using a fast Fourier transform (FFT) algorithm. The FFT of an $M \times N$ image array `f` is obtained in the toolbox with function `fft2`, which has the simple syntax:

fft2

```
F = fft2(f)
```

This function returns a Fourier transform that is also of size $M \times N$, with the data arranged in the form shown in Fig. 4.2(a); that is, with the origin of the data at the top left, and with four quarter periods meeting at the center of the frequency rectangle.

As explained in Section 4.3.1, it is necessary to pad the input image with zeros when the Fourier transform is used for filtering. In this case, the syntax becomes

```
F = fft2(f, P, Q)
```

With this syntax, `fft2` pads the input with the required number of zeros so that the resulting function is of size $P \times Q$.

The Fourier spectrum is obtained by using function `abs`:

abs

```
S = abs(F)
```

which computes the magnitude (square root of the sum of the squares of the real and imaginary parts) of each element of the array.

Visual analysis of the spectrum by displaying it as an image is an important aspect of working in the frequency domain. As an illustration, consider the simple image, `f`, in Fig. 4.3(a). We compute its Fourier transform and display the spectrum using the following sequence of steps:

```
>> F = fft2(f);
>> S = abs(F);
>> imshow(S, [ ])
```

Figure 4.3(b) shows the result. The four bright spots in the corners of the image are due to the periodicity property mentioned in the previous section.

IPT function `fftshift` can be used to move the origin of the transform to the center of the frequency rectangle. The syntax is

fftshift

```
Fc = fftshift(F)
```

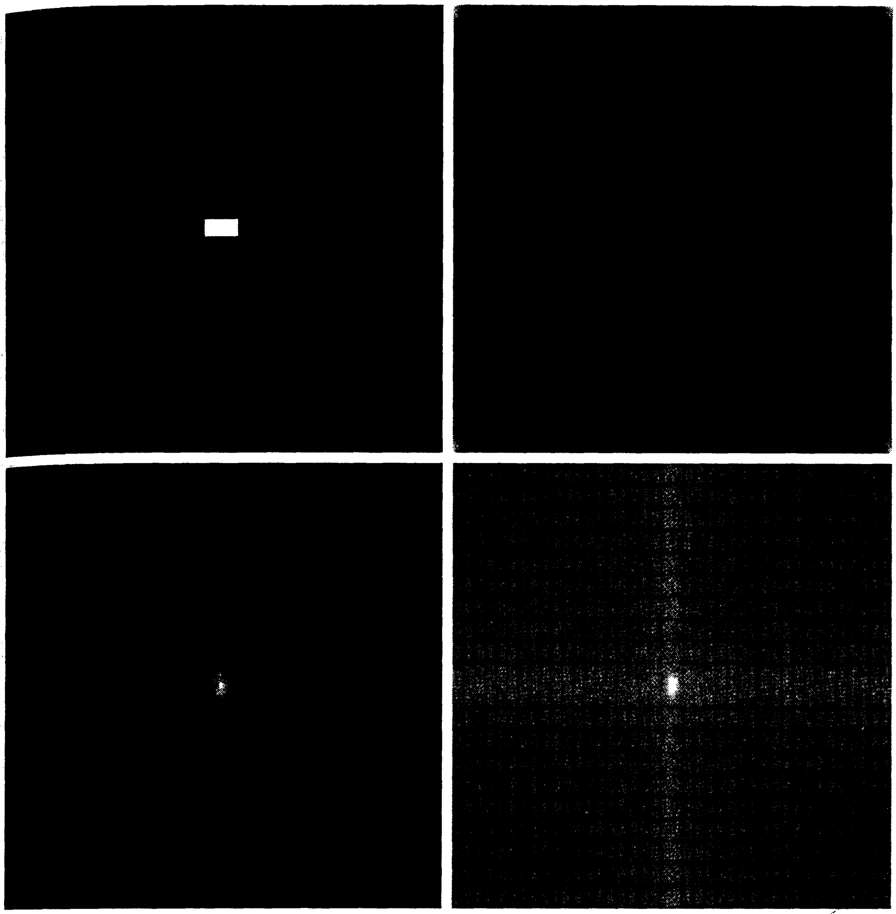


FIGURE 4.3 (a) A simple image. (b) Fourier spectrum. (c) Centered spectrum. (d) Spectrum visually enhanced by a log transformation.

where `F` is the transform computed using `fft2` and `Fc` is the centered transform. Function `fftshift` operates by swapping quadrants of `F`. For example, if `a = [1 2; 3 4]`, `fftshift(a) = [4 3; 2 1]`. When applied to a transform after it has been computed, the net result of using `fftshift` is the same as if the input image had been multiplied by $(-1)^{x+y}$ prior to computing the transform. Note, however, that the two processes are *not* interchangeable. That is, letting $\Im[\cdot]$ denote the Fourier transform of the argument, we have that $\Im[(-1)^{x+y}f(x, y)]$ is equal to `fftshift(fft2(f))`, but this quantity is not equal to `fft2(fftshift(f))`.

In the present example, typing

```
>> Fc = fftshift(F);
>> imshow(abs(Fc), [ ])
```

yielded the image in Fig. 4.3(c). The result of centering is evident in this image.

Although the shift was accomplished as expected, the dynamic range of the values in this spectrum is so large (0 to 204000) compared to the 8 bits of the display that the bright values in the center dominate the result. As discussed in Section 3.2.2, this difficulty is handled via a log transformation. Thus, the commands

```
>> S2 = log(1 + abs(Fc));
>> imshow(S2, [ ])
```

resulted in Fig. 4.3(d). The increase in visual detail is evident in this image.

Function `ifftshift` reverses the centering. Its syntax is

ifftshift

```
F = ifftshift(Fc)
```

This function can be used also to convert a function that is initially centered on a rectangle to a function whose center is at the top, left corner of the rectangle. We make use of this property in Section 4.4.

While on the subject of centering, keep in mind that the center of the frequency rectangle is at $(M/2, N/2)$ if the variables u and v run from 0 to $M - 1$ and $N - 1$, respectively. For example, the center of an 8×8 frequency square is at point $(4,4)$, which is the 5th point along each axis, counting up from $(0,0)$. If, as in MATLAB, the variables run from 1 to M and 1 to N, respectively, then the center of the square is at $[(M/2) + 1, (N/2) + 1]$. In the case of our 8×8 example, the center would be at point $(5,5)$, counting up from $(1,1)$. Obviously, the two centers are the same point, but this can be a source of confusion when deciding how to specify the location of DFT centers in MATLAB computations.

`B = floor(A)` *rounds each element of* `A` *to the nearest integer less than or equal to its value. Function* `ceil` *rounds to the nearest integer greater than or equal to the value of each element of* `A`.

If M and N are odd, the center for MATLAB computations is obtained by rounding $M/2$ and $N/2$ down to the closest integer. The rest of the analysis is as in the previous paragraph. For example, the center of a 7×7 region is at $(3,3)$ if we count up from $(0,0)$ and at $(4,4)$ if we count up from $(1,1)$. In either case, the center is the fourth point from the origin. If only one of the dimensions is odd, the center along that dimension is similarly obtained by rounding down in the manner just explained. Using MATLAB's function `floor`, and keeping in mind that the origin is at $(1,1)$, the center of the frequency rectangle for MATLAB computations is at

floor
ceil

```
[floor(M/2) + 1, floor(N/2) + 1]
```

The center given by this expression is valid both for odd and even values of M and N.

Finally, we point out that the inverse Fourier transform is computed using function `ifft2`, which has the basic syntax

ifft2

```
f = ifft2(F)
```

where `F` is the Fourier transform and `f` is the resulting image. If the input used to compute `F` is real, the inverse in theory should be real. In practice, however, the output of ifft2 often has very small imaginary components resulting from round-off errors that are characteristic of floating point computations. Thus, it is good practice to extract the real part of the result after computing the inverse to obtain an image consisting only of real values. The two operations can be combined:

```
>> f = real(ifft2(F));
```

As in the forward case, this function has the alternate format ifft2(F, P, Q), which pads F with zeros so that its size is $P \times Q$ before computing the inverse. This option is not used in the book.

real(arg) *and* imag(arg) *extract the real and imaginary parts of* arg*, respectively.*

4.3 Filtering in the Frequency Domain

Filtering in the frequency domain is quite simple conceptually. In this section we give a brief overview of the concepts involved in frequency domain filtering and its implementation in MATLAB.

4.3.1 Fundamental Concepts

The foundation for linear filtering in both the spatial and frequency domains is the convolution theorem, which may be written as[†]

$$f(x, y) * h(h, y) \Leftrightarrow H(u, v)F(u, v)$$

and, conversely,

$$f(x, y)h(h, y) \Leftrightarrow H(u, v) * G(u, v)$$

Here, the symbol "*" indicates convolution of the two functions, and the expressions on the sides of the double arrow constitute a Fourier transform pair. For example, the first expression indicates that convolution of two spatial functions can be obtained by computing the inverse Fourier transform of the product of the Fourier transforms of the two functions. Conversely, the forward Fourier transform of the convolution of two spatial functions gives the product of the transforms of the two functions. Similar comments apply to the second expression.

In terms of filtering, we are interested in the first of the two previous expressions. Filtering in the spatial domain consists of convolving an image $f(x, y)$ with a filter mask, $h(x, y)$. Linear spatial convolution is precisely as explained in Section 3.4.1. According to the convolution theorem, we can obtain the same result in the frequency domain by multiplying $F(u, v)$ by $H(u, v)$, the Fourier transform of the spatial filter. It is customary to refer to $H(u, v)$ as the *filter transfer function*.

Basically, the idea in frequency domain filtering is to select a filter transfer function that modifies $F(u, v)$ in a specified manner. For example, the filter in

[†]For digital images, these expressions are strictly valid only when $f(x, y)$ and $h(x, y)$ have been properly padded with zeros, as discussed later in this section.

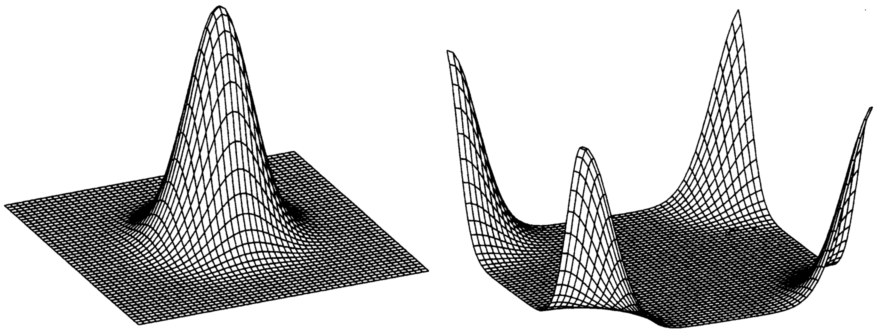


FIGURE 4.4 Transfer functions of (a) a centered lowpass filter, and (b) the format used for DFT filtering. Note that these are frequency domain filters.

Fig. 4.4(a) has a transfer function that, when multiplied by a centered $F(u, v)$, attenuates the high-frequency components of $F(u, v)$, while leaving the low frequencies relatively unchanged. Filters with this characteristic are called *lowpass filters*. As discussed in Section 4.5.2, the net result of lowpass filtering is image blurring (smoothing). Figure 4.4(b) shows the same filter after it was processed with `fftshift`. This is the filter format used most frequently in the book when dealing with frequency domain filtering in which the Fourier transform of the input is not centered.

Based on the convolution theorem, we know that to obtain the corresponding filtered image in the spatial domain we simply compute the inverse Fourier transform of the product $H(u, v)F(u, v)$. It is important to keep in mind that the process just described is identical to what we would obtain by using convolution in the spatial domain, as long as the filter mask, $h(x, y)$, is the inverse Fourier transform of $H(u, v)$. In practice, spatial convolution generally is simplified by using small masks that attempt to capture the salient features of their frequency domain counterparts.

As noted in Section 4.1, images and their transforms are automatically considered periodic if we elect to work with DFTs to implement filtering. It is not difficult to visualize that convolving periodic functions can cause interference between adjacent periods if the periods are close with respect to the duration of the nonzero parts of the functions. This interference, called *wraparound error*, can be avoided by padding the functions with zeros, in the following manner.

Assume that functions $f(x, y)$ and $h(x, y)$ are of size $A \times B$ and $C \times D$,, respectively. We form two *extended* (*padded*) functions, both of size $P \times Q$ by appending zeros to f and g. It can be shown that wraparound error is avoided by choosing

$$P \geq A + C - 1$$

and

$$Q \geq B + D - 1$$

Most of the work in this chapter deals with functions of the same size, $M \times N$, in which case we use the following padding values: $P \geq 2M - 1$ and $Q \geq 2N - 1$.

The following function, called paddedsize, computes the minimum even[†] values of P and Q required to satisfy the preceding equations. It also has an option to pad the inputs to form square images of size equal to the nearest integer power of 2. Execution time of FFT algorithms depends roughly on the number of prime factors in P and Q. These algorithms generally are faster when P and Q are powers of 2 than when P and Q are prime. In practice, it is advisable to work with square images and filters so that filtering is the same in both directions. Function paddedsize provides the flexibility to do this via the choice of the input parameters.

In function paddedsize, the vectors AB, CD, and PQ have elements [A B], [C D], and [P Q], respectively, where these quantities are as defined above.

paddedsize

```
function PQ = paddedsize(AB, CD, PARAM)
%PADDEDSIZE Computes padded sizes useful for FFT-based filtering.
%   PQ = PADDEDSIZE(AB), where AB is a two-element size vector,
%   computes the two-element size vector PQ = 2*AB.
%
%   PQ = PADDEDSIZE(AB, 'PWR2') computes the vector PQ such that
%   PQ(1) = PQ(2) = 2^nextpow2(2*m), where m is MAX(AB).
%
%   PQ = PADDEDSIZE(AB, CD), where AB and CD are two-element size
%   vectors, computes the two-element size vector PQ. The elements
%   of PQ are the smallest even integers greater than or equal to
%   AB + CD - 1.
%
%   PQ = PADDEDSIZE(AB, CD, 'PWR2') computes the vector PQ such that
%   PQ(1) = PQ(2) = 2^nextpow2(2*m), where m is MAX([AB CD]).

if nargin == 1
   PQ = 2*AB;
elseif nargin == 2 & ~ischar(CD)
   PQ = AB + CD - 1;
   PQ = 2 * ceil(PQ / 2);
elseif nargin == 2
   m = max(AB); % Maximum dimension.

   % Find power-of-2 at least twice m.
   P = 2^nextpow2(2*m);
   PQ = [P, P];
elseif nargin == 3
   m = max([AB CD]); % Maximum dimension.
   P = 2^nextpow2(2*m);
   PQ = [P, P];
else
   error('Wrong number of inputs.')
end
```

nextpow2

p = nextpow2(n) *returns the smallest integer power of 2 that is greater than or equal to the absolute value of* n.

[†]It is customary to work with arrays of even dimensions to speed-up FFT computations.

With PQ thus computed using function paddedsize, we use the following syntax for fft2 to compute the FFT using zero padding:

```
F = fft2(f, PQ(1), PQ(2))
```

This syntax simply appends enough zeros to f such that the resulting image is of size PQ(1) × PQ(2), and then computes the FFT as previously described. Note that when using padding the filter function in the frequency domain must be of size PQ(1) × PQ(2) also.

EXAMPLE 4.1: Effects of filtering with and without padding.

■ The image, f, in Fig. 4.5(a) is used in this example to illustrate the difference between filtering with and without padding. In the following discussion we use function lpfilter to generate a Gaussian lowpass filters [similar to Fig. 4.4(b)] with a specified value of sigma (sig). This function is discussed in detail in Section 4.5.2, but the syntax is straightforward, so we use it here and defer further explanation of lpfilter to that section.

The following commands perform filtering without padding:

```
>> [M, N] = size(f);
>> F = fft2(f);
>> sig = 10;
>> H = lpfilter('gaussian', M, N, sig);
>> G = H.*F;
>> g = real(ifft2(G));
>> imshow(g, [ ])
```

Figure 4.5(b) shows image g. As expected, the image is blurred, but note that the vertical edges are not. The reason can be explained with the aid of Fig. 4.6(a), which shows graphically the implied periodicity in DFT computa-

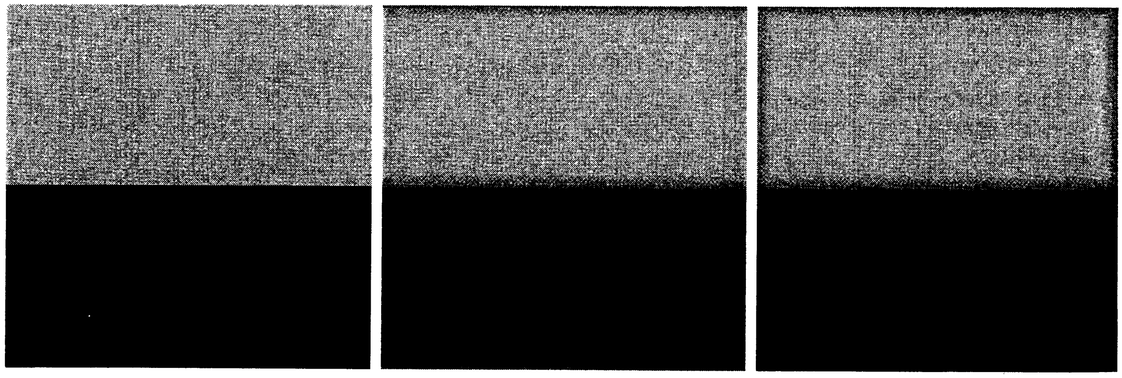
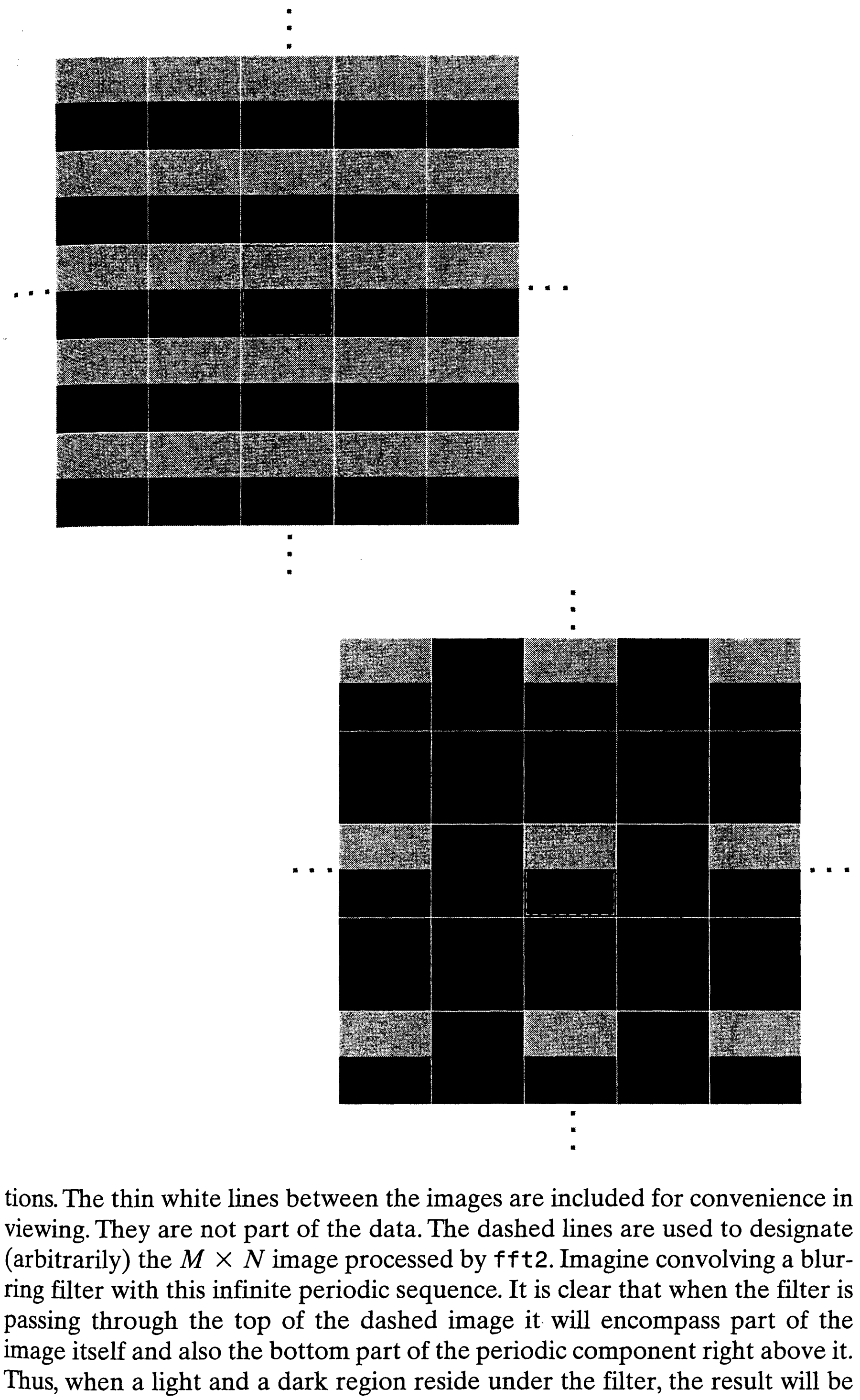


FIGURE 4.5 (a) A simple image of size 256 × 256. (b) Image lowpass-filtered in the frequency domain without padding. (c) Image lowpass-filtered in the frequency domain with padding. Compare the light portion of the vertical edges in (b) and (c).



a
b

FIGURE 4.6
(a) Implied, infinite periodic sequence of the image in Fig. 4.5(a). The dashed region represents the data processed by `fft2`. (b) The same periodic sequence after padding with 0s. The thin white lines in both images are shown for convenience in viewing; they are not part of the data.

tions. The thin white lines between the images are included for convenience in viewing. They are not part of the data. The dashed lines are used to designate (arbitrarily) the $M \times N$ image processed by `fft2`. Imagine convolving a blurring filter with this infinite periodic sequence. It is clear that when the filter is passing through the top of the dashed image it will encompass part of the image itself and also the bottom part of the periodic component right above it. Thus, when a light and a dark region reside under the filter, the result will be a mid-gray, blurred output. This is precisely what the top of the image in

Fig. 4.5(b) shows. On the other hand, when the filter is on the light sides of the dashed image, it will encounter an identical region on the periodic component. Since the average of a constant region is the same constant, there is no blurring in this part of the result. Other parts of the image in Fig. 4.5(b) are explained in a similar manner.

Consider now filtering with padding:

```
>> PQ = paddedsize(size(f));
>> Fp = fft2(f, PQ(1), PQ(2)); % Compute the FFT with padding.
>> Hp = lpfilter('gaussian', PQ(1), PQ(2), 2*sig);
>> Gp = Hp.*Fp;
>> gp = real(ifft2(Gp));
>> gpc = gp(1:size(f,1), 1:size(f,2));
>> imshow(gp, [ ])
```

where we used `2*sig` because the filter size is now twice the size of the filter used without padding.

Figure 4.7 shows the full, padded result, `gp`. The final result in Fig. 4.5(c) was obtained by cropping Fig. 4.7 to the original image size (see the next-to-last command above). This result can be explained with the aid of Fig. 4.6(b), which shows the dashed image padded with zeros as it would be set up internally in `fft2(f, PQ(1), PQ(2))` prior to computing the transform. The implied periodicity is as explained earlier. The image now has a uniform black border all around it, so convolving a smoothing filter with this infinite sequence would show a gray blur in the light edges of the images. A similar result would be obtained by performing the following spatial filtering,

```
>> h = fspecial('gaussian', 15, 7);
>> gs = imfilter(f, h);
```

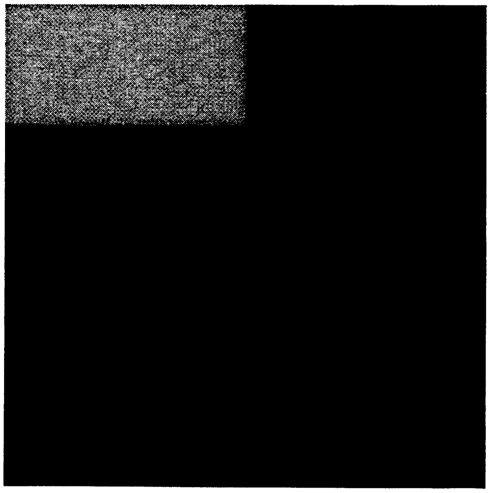


FIGURE 4.7 Full padded image resulting from `ifft2` after filtering. This image is of size 512 × 512 pixels.

Recall from Section 3.4.1 that this call to function `imfilter` pads the border of the image with 0s by default. ■

4.3.2 Basic Steps in DFT Filtering

The discussion in the previous section can be summarized in the following step-by-step procedure involving MATLAB functions, where `f` is the image to be filtered, `g` is the result, and it is assumed that the filter function $H(u, v)$ is of the same size as the padded image:

1. Obtain the padding parameters using function `paddedsize`:
   ```
   PQ = paddedsize(size(f));
   ```
2. Obtain the Fourier transform with padding:
   ```
   F = fft2(f, PQ(1), PQ(2));
   ```
3. Generate a filter function, `H`, of size `PQ(1)` × `PQ(2)` using any of the methods discussed in the remainder of this chapter. The filter must be in the format shown in Fig. 4.4(b). If it is centered instead, as in Fig. 4.4(a), let `H = fftshift(H)` before using the filter.
4. Multiply the transform by the filter:
   ```
   G = H.*F;
   ```
5. Obtain the real part of the inverse FFT of `G`:
   ```
   g = real(ifft2(G));
   ```
6. Crop the top, left rectangle to the original size:
   ```
   g = g(1:size(f, 1), 1:size(f, 2));
   ```

This filtering procedure is summarized in Fig. 4.8. The preprocessing stage might encompass procedures such as determining image size, obtaining the padding parameters, and generating a filter. Postprocessing entails computing the real part of the result, cropping the image, and converting it to class `uint8` or `uint16` for storage.

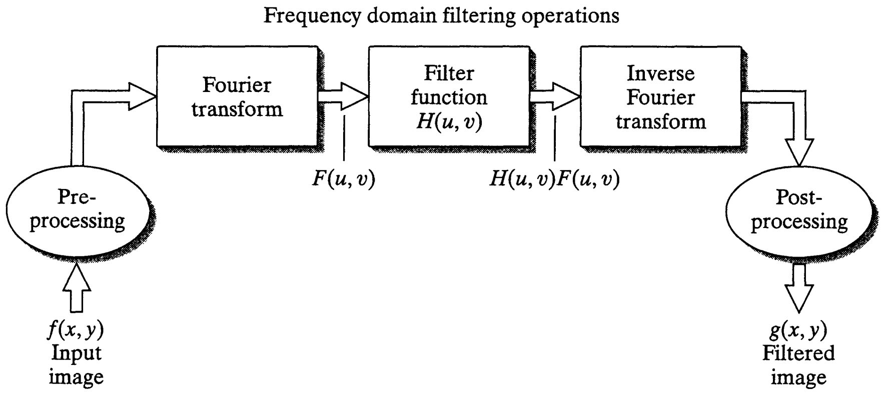


FIGURE 4.8 Basic steps for filtering in the frequency domain.

The filter function $H(u, v)$ in Fig. 4.8 multiplies both the real and imaginary parts of $F(u, v)$. If $H(u, v)$ is real, then the phase of the result is not changed, a fact that can be seen in the phase equation (Section 4.1) by noting that, if the multipliers of the real and imaginary parts are equal, they cancel out, leaving the phase angle unchanged. Filters that operate in this manner are called *zero-phase-shift filters*. These are the only types of linear filters considered in this chapter.

It is well known from linear system theory that, under certain mild conditions, inputting an impulse into a linear system completely characterizes the system. When working with finite, discrete data as we do in this book, the response of a linear system, including the response to an impulse, also is finite. If the linear system is just a spatial filter, then we can completely determine the filter simply by observing its response to an impulse. A filter determined in this manner is called a *finite-impulse-response* (FIR) *filter*. All the linear spatial filters in this book are FIR filters.

4.3.3 An M-function for Filtering in the Frequency Domain

The sequence of filtering steps described in the previous section is used throughout this chapter and parts of the next, so it will be convenient to have available an M-function that accepts as inputs an image and a filter function, handles all the filtering details, and outputs the filtered, cropped image. The following function does this.

dftfilt

```
function g = dftfilt(f, H)
%DFTFILT Performs frequency domain filtering.
%   G = DFTFILT(F, H) filters F in the frequency domain using the
%   filter transfer function H. The output, G, is the filtered
%   image, which has the same size as F. DFTFILT automatically pads
%   F to be the same size as H. Function PADDEDSIZE can be used
%   to determine an appropriate size for H.
%
%   DFTFILT assumes that F is real and that H is a real, uncentered,
%   circularly-symmetric filter function.

% Obtain the FFT of the padded input.
F = fft2(f, size(H, 1), size(H, 2));

% Perform filtering.
g = real(ifft2(H.*F));

% Crop to original size.
g = g(1:size(f, 1), 1:size(f, 2));
```

Techniques for generating frequency-domain filters are discussed in the following three sections.

4.4 Obtaining Frequency Domain Filters from Spatial Filters

In general, filtering in the spatial domain is more efficient computationally than frequency domain filtering when the filters are small. The definition of *small* is a complex question whose answer depends on such factors as the machine and algorithms used and on issues such the sizes of buffers, how well complex data are handled, and a host of other factors beyond the scope of this discussion. A comparison by Brigham [1988] using 1-D functions shows that filtering using an FFT algorithm can be faster than a spatial implementation when the functions have on the order of 32 points, so the numbers in question are not large. Thus, it is useful to know how to convert a spatial filter into an equivalent frequency domain filter in order to obtain meaningful comparisons between the two approaches.

One obvious approach for generating a frequency domain filter, `H`, that corresponds to a given spatial filter, `h`, is to let `H = fft2(h, PQ(1), PQ(2))`, where the values of vector `PQ` depend on the size of the image we want to filter, as discussed in the last section. However, we are interested in this section on two major topics: (1) how to convert spatial filters into equivalent frequency domain filters; and (2) how to compare the results between spatial domain filtering using function `imfilter`, and frequency domain filtering using the techniques discussed in the previous section. Because, as explained in detail in Section 3.4.1, `imfilter` uses correlation and the origin of the filter is considered at its center, a certain amount of data preprocessing is required to make the two approaches equivalent. The toolbox provides a function, `freqz2`, that does precisely this and outputs the corresponding filter in the frequency domain.

Function `freqz2` computes the frequency response of FIR filters, which, as mentioned at the end of Section 4.3.2, are the only linear filters considered in this book. The result is the desired filter in the frequency domain. The syntax of interest in the present discussion is

```
H = freqz2(h, R, C)
```

freqz2

where `h` is a 2-D spatial filter and `H` is the corresponding 2-D frequency domain filter. Here, `R` is the number of rows, and `C` the number of columns that we wish filter `H` to have. Generally, we let `R = PQ(1)` and `C = PQ(2)`, as explained in Section 4.3.1. If `freqz2` is written without an output argument, the absolute value of `H` is displayed on the MATLAB desktop as a 3-D perspective plot. The mechanics involved in using function `freqz2` are easily explained by an example.

EXAMPLE 4.2: A comparison of filtering in the spatial and frequency domains.

■ Consider the image, `f`, of size 600 × 600 pixels shown in Fig. 4.9(a). In what follows, we generate the frequency domain filter, `H`, corresponding to the Sobel spatial filter that enhances vertical edges (see Table 3.4). We then compare the result of filtering `f` in the spatial domain with the Sobel mask (using `imfilter`) against the result obtained by performing the equivalent process in the frequency domain. In practice, filtering with a small filter like a Sobel mask would be implemented directly in the spatial domain, as mentioned earlier. However, we selected this filter for demonstration purposes because its coefficients are simple and because the results of filtering are intuitive and straightforward to compare. Larger spatial filters are handled in exactly the same manner.

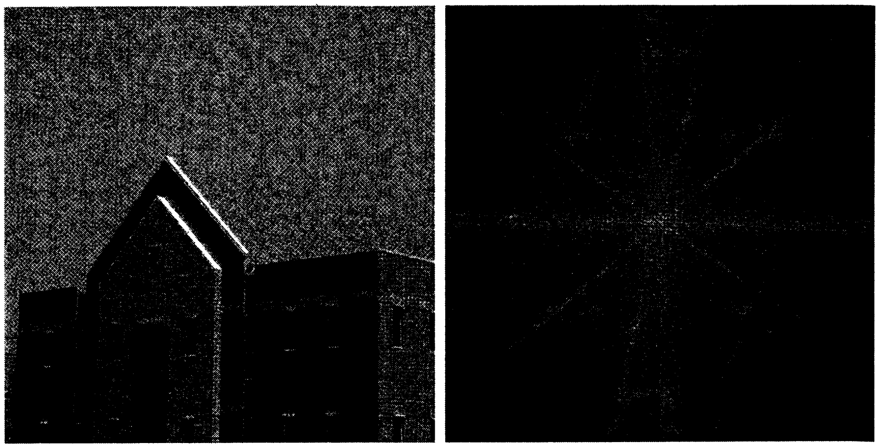


FIGURE 4.9
(a) A gray-scale image. (b) Its Fourier spectrum.

Figure 4.9(b) is an image of the Fourier spectrum of `f`, obtained as follows:

```
>> F = fft2(f);
>> S = fftshift(log(1 + abs(F)));
>> S = gscale(S);
>> imshow(S)
```

Next, we generate the spatial filter using function `fspecial`:

```
h = fspecial('sobel)'
h =
    1   0   -1
    2   0   -2
    1   0   -1
```

To view a plot of the corresponding frequency domain filter we type

```
>> freqz2(h)
```

Figure 4.10(a) shows the result, with the axes suppressed (techniques for obtaining perspective plots are discussed in Section 4.5.3). The filter itself was obtained using the commands:

```
>> PQ = paddedsize(size(f));
>> H = freqz2(h, PQ(1), PQ(2));
>> H1 = ifftshift(H);
```

where, as noted earlier, `ifftshift` is needed to rearrange the data so that the origin is at the top, left of the frequency rectangle. Figure 4.10(b) shows a plot of `abs(H1)`. Figures 4.10(c) and (d) show the absolute values of `H` and `H1` in image form, displayed with the commands

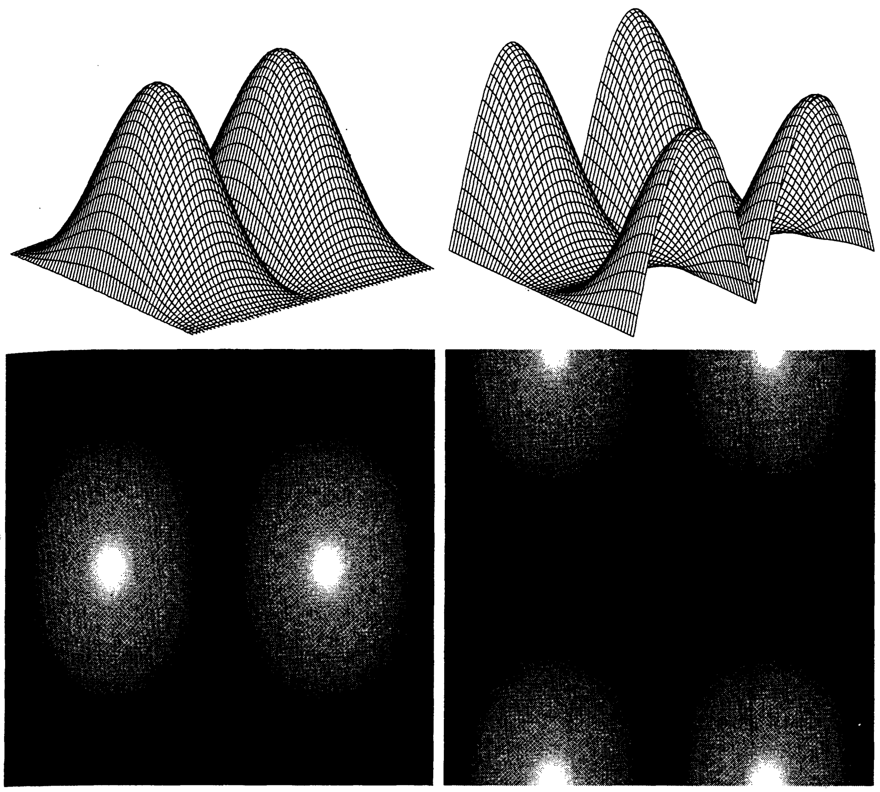


FIGURE 4.10 (a) Absolute value of the frequency domain filter corresponding to a vertical Sobel mask. (b) The same filter after processing with function `fftshift`. Figures (c) and (d) are the filters in (a) and (b) shown as images.

```
>> imshow(abs(H), [ ])
>> figure, imshow(abs(H1), [ ])
```

Next, we generate the filtered images. In the spatial domain we use

```
>> gs = imfilter(double(f), h);
```

We use `double(f)` *here so that* `imfilter` *will produce an output of class* `double`, *as explained in Section 3.4.1. The* `double` *format is required for some of the operations that follow.*

which pads the border of the image with 0s by default. The filtered image obtained by frequency domain processing is given by

```
gf = dftfilt(f, H1);
```

Figures 4.11(a) and (b) show the result of the commands:

```
>> imshow(gs, [ ])
>> figure, imshow(gf, [ ])
```

The gray tonality in the images is due to the fact that both `gs` and `gf` have negative values, which causes the average value of the images to be increased by the scaled `imshow` command. As discussed in Sections 6.6.1 and 10.1.3, the

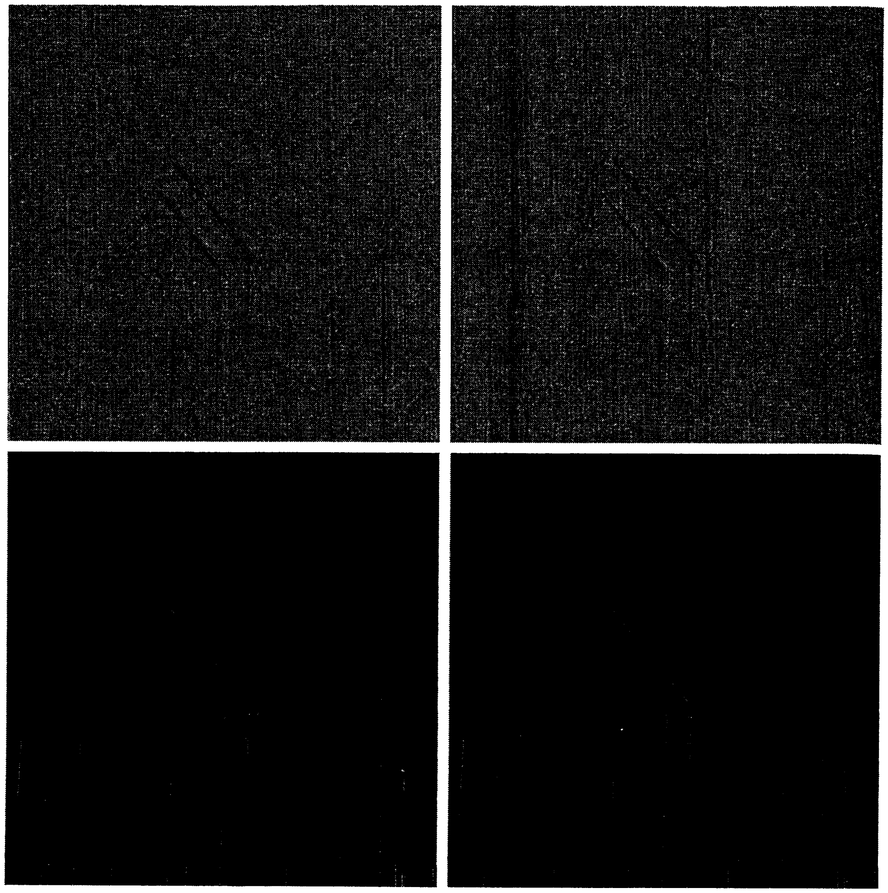


FIGURE 4.11 (a) Result of filtering Fig. 4.9(a) in the spatial domain with a vertical Sobel mask. (b) Result obtained in the frequency domain using the filter shown in Fig. 4.10(b). Figures (c) and (d) are the absolute values of (a) and (b), respectively.

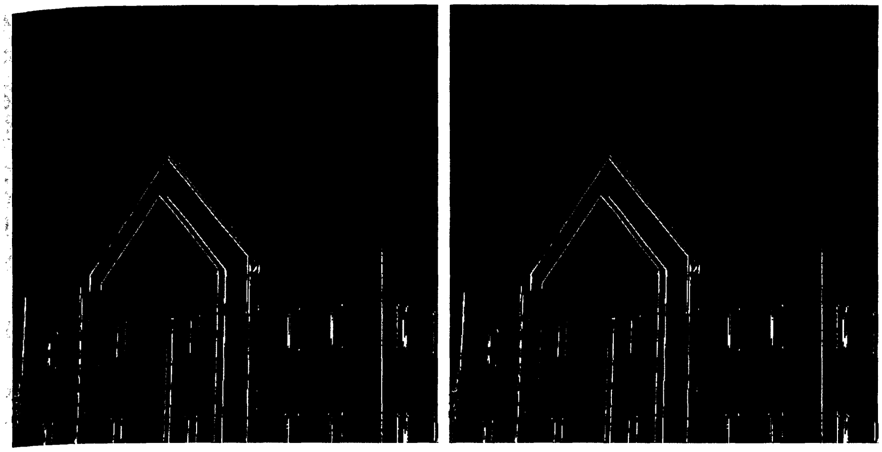
Sobel mask, h, generated above is used to detect vertical edges in an image using the absolute value of the response. Thus, it is more relevant to show the absolute values of the images just computed. Figures 4.11(c) and (d) show the images obtained using the commands

```
>> figure, imshow(abs(gs), [ ])
>> figure, imshow(abs(gf), [ ])
```

The edges can be seen more clearly by creating a thresholded binary image:

```
>> figure, imshow(abs(gs) > 0.2*abs(max(gs(:))))
>> figure, imshow(abs(gf) > 0.2*abs(max(gf(:))))
```

where the 0.2 multiplier was selected (arbitrarily) to show only the edges with strength greater than 20% of the maximum values of gs and gf. Figures 4.12(a) and (b) show the results.



a b

FIGURE 4.12 Thresholded versions of Figs. 4.11(c) and (d), respectively, to show the principal edges more clearly.

The images obtained using spatial and frequency domain filtering are for all practical purposes identical, a fact that we confirm by computing their difference:

```
>> d = abs(gs - gf);
```

The maximum and minimum differences are

```
>> max(d(:))
ans =
    5.4015e-012
>> min(d(:))
ans =
    0
```

The approach just explained can be used to implement in the frequency domain the spatial filtering approach discussed in Sections 3.4.1 and 3.5.1, as well as any other FIR spatial filter of arbitrary size. ■

4.5 Generating Filters Directly in the Frequency Domain

In this section, we illustrate how to implement filter functions directly in the frequency domain. We focus on circularly symmetric filters that are specified as various functions of distance from the origin of the transform. The M-functions developed to implement these filters are a foundation that is easily extendable to other functions within the same framework. We begin by implementing several well-known smoothing (lowpass) filters. Then, we show how to use several of MATLAB's wireframe and surface plotting capabilities that aid in filter visualization. We conclude the section with a brief discussion of sharpening (highpass) filters.

4.5.1 Creating Meshgrid Arrays for Use in Implementing Filters in the Frequency Domain

Central to the M-functions in the following discussion is the need to compute distance functions from any point to a specified point in the frequency rectangle. Because FFT computations in MATLAB assume that the origin of the transform is at the top, left of the frequency rectangle, our distance computations are with respect to that point. The data can be rearranged for visualization purposes (so that the value at the origin is translated to the center of the frequency rectangle) by using function `fftshift`.

The following M-function, which we call `dftuv`, provides the necessary meshgrid array for use in distance computations and other similar applications. (See Section 2.10.4 for an explanation of function `meshgrid` used in the following code.). The meshgrid arrays generated by `dftuv` are in the order required for processing with `fft2` or `ifft2`, so no rearranging of the data is required.

`dftuv`

```
function [U, V] = dftuv(M, N)
%DFTUV Computes meshgrid frequency matrices.
%   [U, V] = DFTUV(M, N) computes meshgrid frequency matrices U and
%   V.  U and V are useful for computing frequency-domain filter
%   functions that can be used with DFTFILT.  U and V are both
%   M-by-N.

% Set up range of variables.
u = 0:(M - 1);
v = 0:(N - 1);

% Compute the indices for use in meshgrid.
idx = find(u > M/2);
u(idx) = u(idx) - M;
idy = find(v > N/2);
v(idy) = v(idy) - N;

% Compute the meshgrid arrays.
[V, U] = meshgrid(v, u);
```

Function `find` *is discussed in Section 5.2.2.*

EXAMPLE 4.3: Using function `dftuv`.

■ As an illustration, the following commands compute the distance squared from every point in a rectangle of size 8 × 5 to the origin of the rectangle:

```
>> [U, V] = dftuv(8, 5);
>> D = U.^2 + V.^2
D =

      0     1     4     4     1
      1     2     5     5     2
      4     5     8     8     5
      9    10    13    13    10
     16    17    20    20    17
      9    10    13    13    10
      4     5     8     8     5
      1     2     5     5     2
```

Note that the distance is 0 at the top, left, and the larger distances are in the center of the frequency rectangle, following the basic format explained in Fig. 4.2(a). We can use function `fftshift` to obtain the distances with respect to the center of the frequency rectangle,

```
>> fftshift(D)

ans =

    20    17    16    17    20
    13    10     9    10    13
     8     5     4     5     8
     5     2     1     2     5
     4     1     0     1     4
     5     2     1     2     5
     8     5     4     5     8
    13    10     9    10    13
```

The distance is now 0 at coordinates (5, 3), and the array is symmetric about this point. ■

4.5.2 Lowpass Frequency Domain Filters

An *ideal lowpass filter* (ILPF) has the transfer function

$$H(u, v) = \begin{cases} 1 & \text{if } D(u, v) \le D_0 \\ 0 & \text{if } D(u, v) > D_0 \end{cases}$$

where D_0 is a specified nonnegative number and $D(u, v)$ is the distance from point (u, v) to the center of the filter. The locus of points for which $D(u, v) = D_0$ is a circle. Keeping in mind that filter H multiplies the Fourier transform of an image, we see that an ideal filter "cuts off" (multiplies by 0) all components of F outside the circle and leaves unchanged (multiplies by 1) all components on, or inside, the circle. Although this filter is not realizable in analog form using electronic components, it certainly can be simulated in a computer using the preceding transfer function. The properties of ideal filters often are useful in explaining phenomena such as wraparound error.

A *Butterworth lowpass filter* (BLPF) of order n, with a cutoff frequency at a distance D_0 from the origin, has the transfer function

$$H(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2n}}$$

Unlike the ILPF, the BLPF transfer function does not have a sharp discontinuity at D_0. For filters with smooth transfer functions, it is customary to define a cutoff frequency locus at points for which $H(u, v)$ is down to a specified fraction of its maximum value. In the preceding equation, $H(u, v) = 0.5$ (down 50% from its maximum value of 1) when $D(u, v) = D_0$.

The transfer function of a *Gaussian lowpass filter* (GLPF) is given by

$$H(u, v) = e^{-D^2(u, v)/2\sigma^2}$$

where σ is the standard deviation. By letting $\sigma = D_0$, we obtain the following expression in terms of the cutoff parameter D_0:

$$H(u, v) = e^{-D^2(u,v)/2D_0^2}$$

When $D(u, v) = D_0$ the filter is down to 0.607 of its maximum value of 1.

EXAMPLE 4.4: Lowpass filtering.

■ As an illustration, we apply a Gaussian lowpass filter to the 500 × 500-pixel image, `f`, in Fig. 4.13(a). We use a value of D_0 equal to 5% of the padded image width. With reference to the filtering steps discussed in Section 4.3.2 we have

```
>> PQ = paddedsize(size(f));
>> [U, V] = dftuv(PQ(1), PQ(2));
>> DO = 0.05*PQ(2);
>> F = fft2(f, PQ(1), PQ(2));
>> H = exp(-(U.^2 + V.^2)/(2*(DO^2)));
>> g = dftfilt(f, H);
```

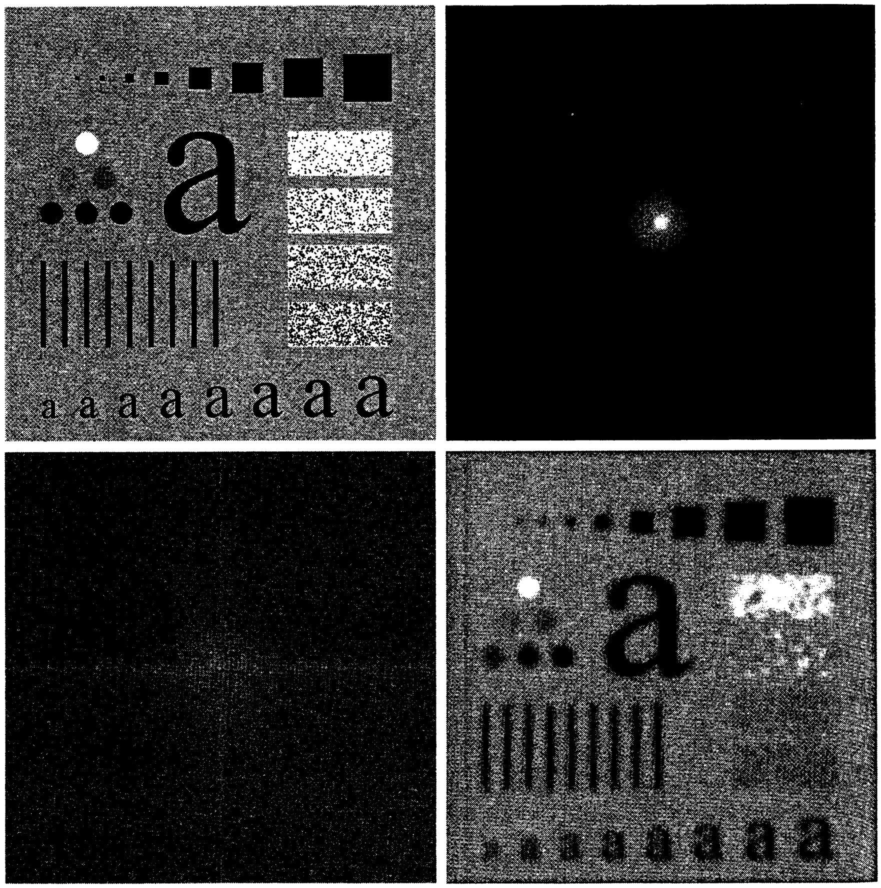


FIGURE 4.13 Lowpass filtering. (a) Original image. (b) Gaussian lowpass filter shown as an image. (c) Spectrum of (a). (d) Processed image.

We can view the filter as an image [Fig. 4.13(b)] by typing

```
>> figure, imshow(fftshift(H), [ ])
```

Similarly, the spectrum can be displayed as an image [Fig. 4.13(c)] by typing

```
>> figure, imshow(log(1 + abs(fftshift(F))), [ ])
```

Finally, Fig. 4.13(d) shows the output image, displayed using the command

```
>> figure, imshow(g, [ ])
```

As expected, this image is a blurred version of the original. ■

The following function generates the transfer functions of all the lowpass filters discussed in this section.

lpfilter

```
function [H, D] = lpfilter(type, M, N, D0, n)
%LPFILTER Computes frequency domain lowpass filters.
%   H = LPFILTER(TYPE, M, N, D0, n) creates the transfer function of
%   a lowpass filter, H, of the specified TYPE and size (M-by-N). To
%   view the filter as an image or mesh plot, it should be centered
%   using H = fftshift(H).
%
%   Valid values for TYPE, D0, and n are:
%
%   'ideal'    Ideal lowpass filter with cutoff frequency D0. n need
%              not be supplied. D0 must be positive.
%
%   'btw'      Butterworth lowpass filter of order n, and cutoff
%              D0. The default value for n is 1.0. D0 must be
%              positive.
%
%   'gaussian' Gaussian lowpass filter with cutoff (standard
%              deviation) D0. n need not be supplied. D0 must be
%              positive.

% Use function dftuv to set up the meshgrid arrays needed for
% computing the required distances.
[U, V] = dftuv(M, N);

% Compute the distances D(U, V).
D = sqrt(U.^2 + V.^2);

% Begin filter computations.
switch type
case 'ideal'
   H = double(D <= D0);
case 'btw'
   if nargin == 4
      n = 1;
   end
   H = 1./(1 + (D./D0).^(2*n));
case 'gaussian'
   H = exp(-(D.^2)./(2*(D0^2)));
otherwise
   error('Unknown filter type.')
end
```

Function `lpfilter` is used again in Section 4.6 as the basis for generating highpass filters.

4.5.3 Wireframe and Surface Plotting

Plots of functions of one variable were introduced in Section 3.3.1. In the following discussion we introduce 3-D wireframe and surface plots, which are useful for visualizing the transfer functions of 2-D filters. The easiest way to draw a wireframe plot of a given 2-D function, `H`, is to use function `mesh`, which has the basic syntax

mesh

```
mesh(H)
```

This function draws a wireframe for `x = 1:M` and `y = 1:N`, where `[M, N] = size(H)`. Wireframe plots typically are unacceptably dense if `M` and `N` are large, in which case we plot every `k`th point using the syntax

```
mesh(H(1:k:end, 1:k:end))
```

As a rule of thumb, 40 to 60 subdivisions along each axis usually provide a good balance between resolution and appearance.

MATLAB plots mesh figures in color, by default. The command

colormap

```
colormap([0 0 0])
```

sets the wireframe to black (we discuss function `colormap` in Chapter 6). MATLAB also superimposes a grid and axes on a mesh plot. These can be turned off using the commands

grid

```
grid off
axis off
```

They can be turned back on by replacing `off` with `on` in these two statements. Finally, the viewing point (location of the observer) is controlled by function `view`, which has the syntax

view

```
view(az, el)
```

As Fig. 4.14 shows, `az` and `el` represent azimuth and elevation angles (in degrees), respectively. The arrows indicate positive direction. The default values

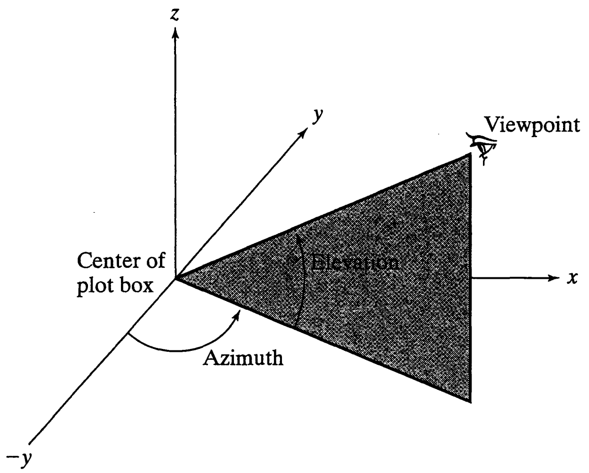


FIGURE 4.14 Geometry for function `view`.

are `az` = −`37.5` and `el` = `30`, which place the viewer in the quadrant defined by the $-x$ and $-y$ axes, and looking into the quadrant defined by the positive x and y axes in Fig. 4.14.

To determine the current viewing geometry, we type

```
>> [az, el] = view;
```

To set the viewpoint to the default values, we type

```
>> view(3)
```

The viewpoint can be modified interactively by clicking on the **Rotate 3D** button in the figure window's toolbar and then clicking and dragging in the figure window.

As discussed in Chapter 6, it is possible to specify the viewer location in Cartesian coordinates, (x, y, z), which is ideal when working with RGB data. However, for general plot-viewing purposes, the method just discussed involves only two parameters and is more intuitive.

EXAMPLE 4.5: Wireframe plotting.

■ Consider a Gaussian lowpass filter similar to the one used in Example 4.4:

```
>> H = fftshift(lpfilter('gaussian', 500, 500, 50));
```

Figure 4.15(a) shows the wireframe plot produced by the commands

```
>> mesh(H(1:10:500, 1:10:500))
>> axis([0 50 0 50 0 1])
```

where the `axis` command is as described in Section 3.3.1, except that it contains a third range for the z axis.

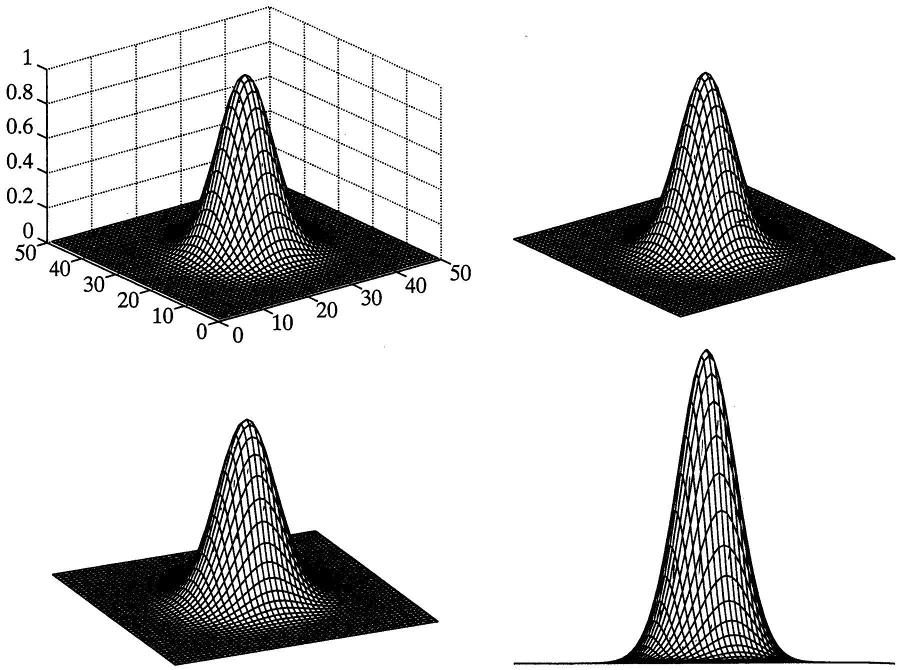


FIGURE 4.15 (a) A plot obtained using function `mesh`. (b) Axes and grid removed. (c) A different perspective view obtained using function `view`. (d) Another view obtained using the same function.

As noted earlier in this section, the wireframe is in color by default, transitioning from blue at the base to red at the top. We convert the plot lines to black and eliminate the axes and grid by typing

```
>> colormap([0 0 0])
>> axis off
>> grid off
```

Figure 4.15(b) shows the result. Figure 4.15(c) shows the result of the command

```
>> view(-25, 30)
```

which moved the observer slightly to the right, while leaving the elevation constant. Finally, Fig. 4.15(d) shows the result of leaving the azimuth at –25 and setting the elevation to 0:

```
>> view(-25, 0)
```

This example shows the significant plotting power of the simple function `mesh`. ■

Sometimes it is desirable to plot a function as a surface instead of as a wireframe. Function `surf` does this. Its basic syntax is

surf

```
surf(H)
```

This function produces a plot identical to mesh, with the exception that the quadrilaterals in the mesh are filled with colors (this is called *faceted shading*). To convert the colors to gray, we use the command

```
colormap(gray)
```

The axis, grid, and view functions work in the same way as described earlier for mesh. For example, Fig. 4.16(a) is the result of the following sequence of commands:

```
>> H = fftshift(lpfilter('gaussian', 500, 500, 50));
>> surf(H(1:10:500, 1:10:500))
>> axis([0 50 0 50 0 1])
>> colormap(gray)
>> grid off; axis off
```

The faceted shading can be smoothed and the mesh lines eliminated by interpolation using the command

```
shading interp
```

shading interp

Typing this command at the prompt produced Fig. 4.16(b).

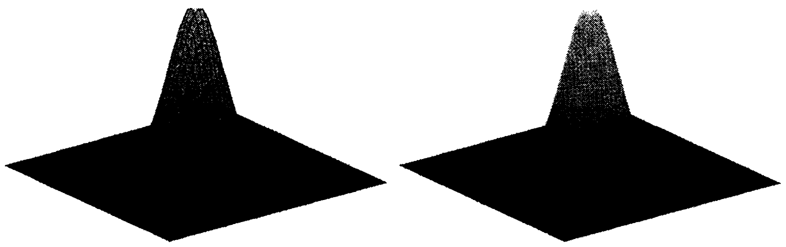
When the objective is to plot an analytic function of two variables, we use meshgrid to generate the coordinate values and from these we generate the discrete (sampled) matrix to use in mesh or surf. For example, to plot the function

$$f(x, y) = xe^{(-x^2-y^2)}$$

from -2 to 2 in increments of 0.1 for both x and y, we write

```
>> [Y, X] = meshgrid(-2:0.1:2, -2:0.1:2);
>> Z = X.*exp(-X.^2 - Y.^2);
```

and then use mesh(Z) or surf(Z) as before. Recall from the discussion in Section 2.10.4 that that columns (Y) are listed first and rows (X) second in function meshgrid.



a b

FIGURE 4.16 (a) Plot obtained using function surf. (b) Result of using the command shading interp.

4.6 Sharpening Frequency Domain Filters

Just as lowpass filtering blurs an image, the opposite process, *highpass filtering*, sharpens the image by attenuating the low frequencies and leaving the high frequencies of the Fourier transform relatively unchanged. In this section we consider several approaches to highpass filtering.

4.6.1 Basic Highpass Filtering

Given the transfer function $H_{\text{lp}}(u, v)$ of a lowpass filter, we obtain the transfer function of the corresponding highpass filter by using the simple relation

$$H_{\text{hp}}(u, v) = 1 - H_{\text{lp}}(u, v).$$

Thus, function `lpfilter` developed in the previous section can be used as the basis for a highpass filter generator, as follows:

hpfilter

```
function H = hpfilter(type, M, N, D0, n)
%HPFILTER Computes frequency domain highpass filters.
%   H = HPFILTER(TYPE, M, N, D0, n) creates the transfer function of
%   a highpass filter, H, of the specified TYPE and size (M-by-N).
%   Valid values for TYPE, D0, and n are:
%
%   'ideal'     Ideal highpass filter with cutoff frequency D0. n
%               need not be supplied. D0 must be positive.
%
%   'btw'       Butterworth highpass filter of order n, and cutoff
%               D0. The default value for n is 1.0. D0 must be
%               positive.
%
%   'gaussian'  Gaussian highpass filter with cutoff (standard
%               deviation) D0. n need not be supplied. D0 must be
%               positive.

% The transfer function Hhp of a highpass filter is 1 - Hlp,
% where Hlp is the transfer function of the corresponding lowpass
% filter. Thus, we can use function lpfilter to generate highpass
% filters.

if nargin == 4
   n = 1; % Default value of n.
end

% Generate highpass filter.
Hlp = lpfilter(type, M, N, D0, n);
H = 1 - Hlp;
```

EXAMPLE 4.6: Highpass filters.

■ Figure 4.17 shows plots and images of ideal, Butterworth, and Gaussian highpass filters. The plot in Fig. 4.17(a) was generated using the commands

```
>> H = fftshift(hpfilter('ideal', 500, 500, 50));
>> mesh(H(1:10:500, 1:10:500));
>> axis([0 50 0 50 0 1])
```

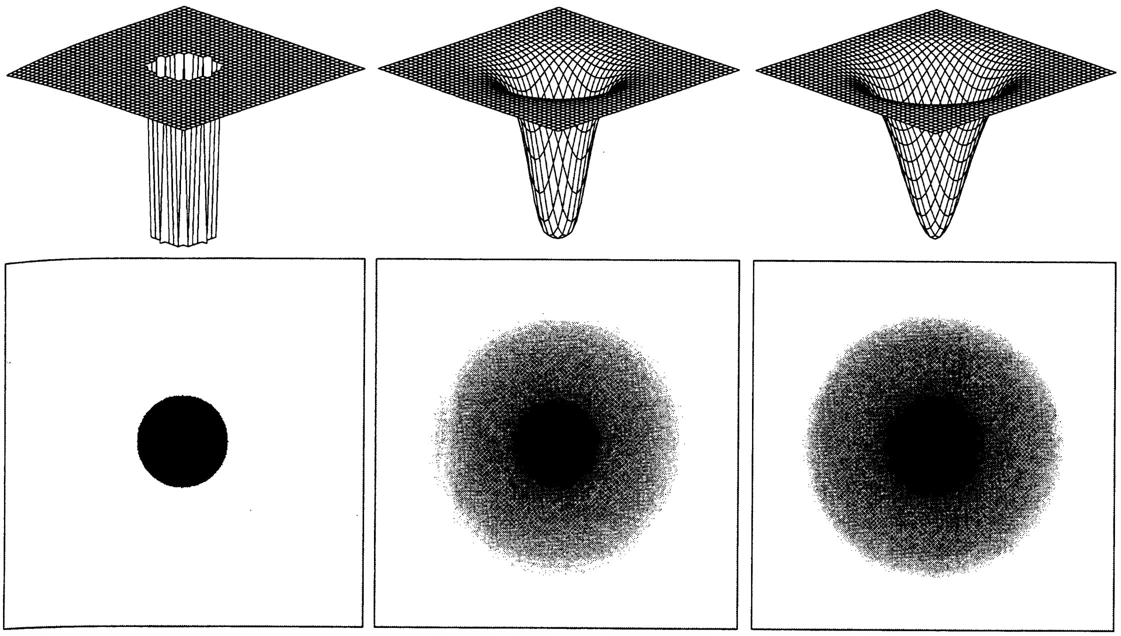


FIGURE 4.17 Top row: Perspective plots of ideal, Butterworth, and Gaussian highpass filters. Bottom row: Corresponding images.

```
>> colormap([0 0 0])
>> axis off
>> grid off
```

The corresponding image in Fig. 4.17(d) was generated using the command

```
>> figure, imshow(H, [ ])
```

where the thin black border is superimposed on the image to delineate its boundary. Similar commands yielded the rest of Fig. 4.17 (the Butterworth filter is of order 2). ■

EXAMPLE 4.7: Highpass filtering.

■ Figure 4.18(a) is the same test pattern, `f`, shown in Fig. 4.13(a). Figure 4.18(b), obtained using the following commands, shows the result of applying a Gaussian highpass filter to `f`:

```
>> PQ = paddedsize(size(f));
>> D0 = 0.05*PQ(1);
>> H = hpfilter('gaussian', PQ(1), PQ(2), D0);
>> g = dftfilt(f, H);
>> figure, imshow(g, [ ])
```

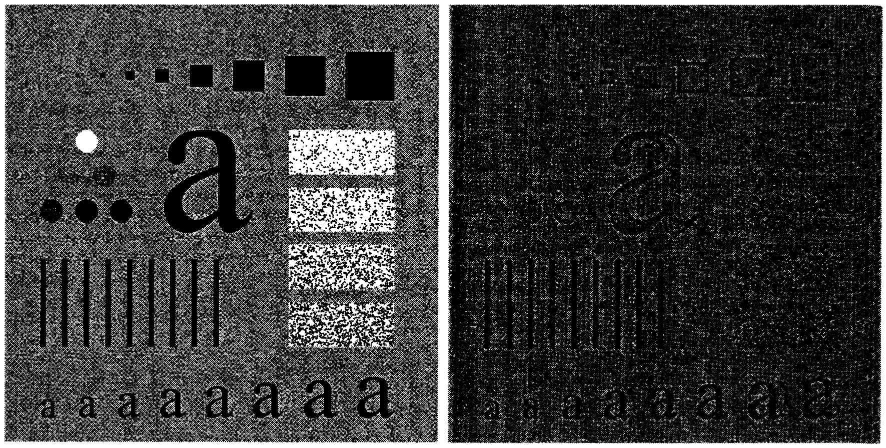


FIGURE 4.18 (a) Original image. (b) Result of Gaussian highpass filtering.

As Fig. 4.18(b) shows, edges and other sharp intensity transitions in the image were enhanced. However, because the average value of an image is given by $F(0,0)$, and the highpass filters discussed thus far zero-out the origin of the Fourier transform, the image has lost most of the background tonality present in the original. This problem is addressed in the following section. ■

4.6.2 High-Frequency Emphasis Filtering

As mentioned in Example 4.7, highpass filters zero out the *dc* term, thus reducing the average value of an image to 0. An approach to compensate for this is to add an offset to a highpass filter. When an offset is combined with multiplying the filter by a constant greater than 1, the approach is called *high-frequency emphasis* filtering because the constant multiplier highlights the high frequencies. The multiplier increases the amplitude of the low frequencies also, but the low-frequency effects on enhancement are less than those due to high frequencies, as long as the offset is small compared to the multiplier. High-frequency emphasis has the transfer function

$$H_{\text{hfe}}(u,v) = a + bH_{\text{hp}}(u,v)$$

where a is the offset, b is the multiplier, and $H_{\text{hp}}(u,v)$ is the transfer function of a highpass filter.

EXAMPLE 4.8: Combining high-frequency emphasis and histogram equalization.

■ Figure 4.19(a) shows a chest X-ray image, `f`. X-ray imagers cannot be focused in the same manner as optical lenses, so the resulting images generally tend to be slightly blurred. The objective of this example is to sharpen Fig. 4.19(a). Because the gray levels in this particular image are biased toward the dark end of the gray scale, we also take this opportunity to give an example of how spatial domain processing can be used to complement frequency domain filtering.

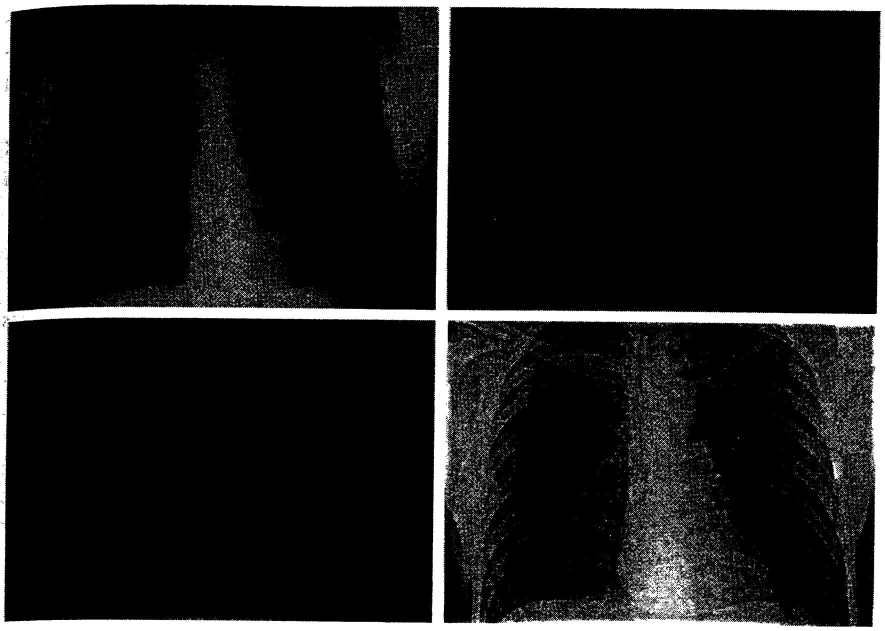


FIGURE 4.19 High-frequency emphasis filtering. (a) Original image. (b) Highpass filtering result. (c) High-frequency emphasis result. (d) Image (c) after histogram equalization. (Original image courtesy of Dr. Thomas R. Gest, Division of Anatomical Sciences, University of Michigan Medical School.)

Figure 4.19(b) shows the result of filtering Fig. 4.19(a) with a Butterworth highpass filter of order 2, and a value of D_0 equal to 5% of the vertical dimension of the padded image. Highpass filtering is not overly sensitive to the value of D_0, as long as the radius of the filter is not so small that frequencies near the origin of the transform are passed. As expected, the filtered result is rather featureless, but it shows faintly the principal edges in the image. The advantage of high-emphasis filtering (with $a = 0.5$ and $b = 2.0$ in this case) is shown in the image of Fig. 4.19(c), in which the gray-level tonality due to the low-frequency components was retained. The following sequence of commands was used to generate the processed images in Fig. 4.19, where f denotes the input image [the last command generated Fig. 4.19(d)]:

```
>> PQ = paddedsize(size(f));
>> D0 = 0.05*PQ(1);
>> HBW = hpfilter('btw', PQ(1), PQ(2), D0, 2);
>> H = 0.5 + 2*HBW;
>> gbw = dftfilt(f, HBW);
>> gbw = gscale(gbw);
>> ghf = dftfilt(f, H);
>> ghf = gscale(ghf);
>> ghe = histeq(ghf, 256);
```

As indicated in Section 3.3.2, an image characterized by gray levels in a narrow range of the gray scale is an ideal candidate for histogram equalization. As Fig. 4.19(d) shows, this indeed was an appropriate method to further enhance the image in this example. Note the clarity of the bone structure and other details that simply are not visible in any of the other three images. The final enhanced image appears a little noisy, but this is typical of X-ray images when their gray scale is expanded. The result obtained using a combination of high-frequency emphasis and histogram equalization is superior to the result that would be obtained by using either method alone. ■

Summary

In addition to the image enhancement applications that we used as illustrations in this and the preceding chapter, the concepts and techniques developed in these two chapters provide the basis for other areas of image processing addressed in subsequent discussions in the book. Intensity transformations are used frequently for intensity scaling, and spatial filtering is used extensively for image restoration in the next chapter, for color processing in Chapter 6, for image segmentation in Chapter 10, and for extracting descriptors from an image in Chapter 11. The Fourier techniques developed in this chapter are used extensively in the next chapter for image restoration, in Chapter 8 for image compression, and in Chapter 11 for image description.