7 Linear Filters

Pictures of zebras and of dalmatians have black and white pixels, and in about the same number, too. The differences between the two have to do with the characteristic appearance of small groups of pixels, rather than individual pixel values. In this chapter, we introduce methods for obtaining descriptions of the appearance of a small group of pixels.

Our main strategy is to use weighted sums of pixel values using different patterns of weights to find different image patterns. Despite its simplicity, this process is extremely useful. It allows us to smooth noise in images, and to find edges and other image patterns.

7.1 LINEAR FILTERS AND CONVOLUTION

Many important effects can be modeled with a simple model. Construct a new array, the same size as the image. Fill each location of this new array with a weighted sum of the pixel values from the locations surrounding the corresponding location in the image *using the same set of weights each time*. Different sets of weights could be used to represent different processes. One example is computing a local average taken over a fixed region. We could average all pixels within a $2k + 1 \times 2k + 1$ block of the pixel of interest. For an input image $\mathcal{F}$, this gives an output

$$\mathcal{R}_{ij} = \frac{1}{(2k+1)^2} \sum_{u=i-k}^{u=i+k} \sum_{v=j-k}^{v=j+k} \mathcal{F}_{uv}.$$

The weights in this example are simple (each pixel is weighted by the same constant), but we could use a more interesting set of weights. For example, we could use a set of weights that was large at the center and fell off sharply as the distance from the center increased to model the kind of smoothing that occurs in a defocused lens system.

Whatever the weights chosen, the output of this procedure is *shift-invariant*—meaning that the value of the output depends on the pattern in an image neighborhood, rather than the position of the neighborhood—and *linear*—meaning that the output for the sum of two images is the same as the sum of the outputs obtained for the images separately. The procedure is known as **linear filtering**.

7.1.1 Convolution

We introduce some notation at this point. The pattern of weights used for a linear filter is usually referred to as the *kernel* of the filter. The process of applying the filter is usually referred to as *convolution*. There is a catch: For reasons that will appear later (Section 7.2.1), it is convenient to write the process in a non-obvious way. In particular, given a filter kernel $\mathcal{H}$, the convolution of the kernel with image $\mathcal{F}$ is an image $\mathcal{R}$. The i, jth component of $\mathcal{R}$ is given by

$$R_{ij} = \sum_{u,v} H_{i-u,j-v} F_{u,v}.$$

This process defines convolution—we say that $\mathcal{H}$ has been convolved with $\mathcal{F}$ to yield $\mathcal{R}$. You should look closely at this expression—the "direction" of the dummy variable u (resp. v) has been reversed compared with correlation. This is important, because if you forget that it is there you compute the wrong answer. The reason for the reversal emerges from the derivation of Section 7.2.1. We carefully avoid inserting the range of the sum; in effect, we assume that the sum is over a large enough range of u and v that all nonzero values are taken into account. Furthermore, we assume that any values that haven't been specified are zero; this means that we can model the kernel as a small block of nonzero values in a sea of zeros. We use this convention, which is common, regularly in what follows.

Example 7.1 Smoothing by Averaging.

Images typically have the property that the value of a pixel is usually similar to that of its neighbor. Assume that the image is affected by noise of a form where we can reasonably expect that this property is preserved. For example, there might be occasional dead pixels, or small random numbers with zero mean might have been added to the pixel values. It is natural to attempt to reduce the effects of this noise by replacing each pixel with a weighted average of its neighbors, a process often referred to as *smoothing* or *blurring*.

Replacing each pixel with an unweighted average computed over some fixed region centered at the pixel is the same as convolution with a kernel that is a block of ones multiplied by a constant. You can (and should) establish this point by close attention to the range of the sum. This process is a poor model of blurring—its output does not look like that of a defocused camera (Figure 7.1). The reason is clear. Assume that we have an image in which every point but the center point was zero, and the center point was one. If we blur this image by forming an unweighted average at each point, the result looks like a small bright box, but this is not what defocused cameras do. We want a blurring process that takes a small bright dot to a circularly symmetric region of blur, brighter at the center than at the edges and fading slowly to darkness. As Figure 7.1 suggests, a set of weights of this form produces a much more convincing defocus model.

Example 7.2 Smoothing with a Gaussian.

A good formal model for this fuzzy blob is the *symmetric Gaussian kernel*

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x^2 + y^2)}{2\sigma^2}\right)$$

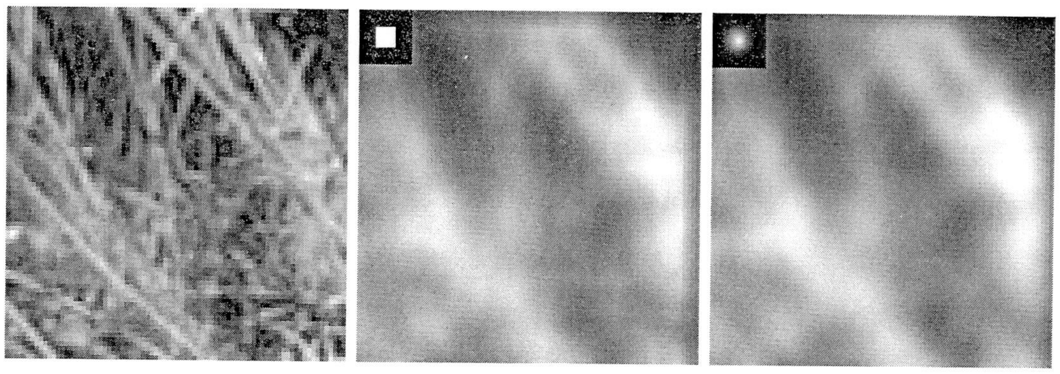

Figure 7.1 Although a uniform local average may seem to give a good blurring model, it generates effects not usually seen in defocusing a lens. The images above compare the effects of a uniform local average with weighted average. The image on the **left** shows a view of grass. In the **center**, the result of blurring this image using a uniform local model and on the **right**, the result of blurring this image using a set of Gaussian weights. The degree of blurring in each case is about the same, but the uniform average produces a set of narrow vertical and horizontal bars—an effect often known as *ringing*. The bottom row shows the weights used to blur the image, themselves rendered as an image; bright points represent large values and dark points represent small values (in this example, the smallest values are zero).

illustrated in Figure 7.2. σ is referred to as the *standard deviation* of the Gaussian (or its "sigma!"); the units are interpixel spaces, usually referred to as *pixels*. The constant term makes the integral over the whole plane equal to one and is often ignored in smoothing applications. The name comes from the fact that this kernel has the form of the probability density for a 2D normal (or Gaussian) random variable with a particular covariance.

This smoothing kernel forms a weighted average that weights pixels at its center much more strongly than at its boundaries. One can justify this approach qualitatively: Smoothing suppresses noise by enforcing the requirement that pixels should look like their neighbors. By downweighting distant neighbors in the average, we can ensure that the requirement that a pixel look like its neighbors is less strongly imposed for distant neighbors. A qualitative analysis gives the following:

- If the standard deviation of the Gaussian is very small—say smaller than one pixel—the smoothing will have little effect because the weights for all pixels off the center will be very small;
- For a larger standard deviation, the neighboring pixels will have larger weights in the weighted average, which means in turn that the average will be strongly biased toward a consensus of the neighbors—this will be a good estimate of a pixel's value, and the noise will largely disappear at the cost of some blurring;
- Finally, a kernel that has a large standard deviation will cause much of the image detail to disappear along with the noise.

Figure 7.3 illustrates these phenomena. You should notice that Gaussian smoothing can be effective at suppressing noise.

In applications, a discrete smoothing kernel is obtained by constructing a $2k + 1 \times 2k + 1$ array whose i, jth value is

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{((i-k-1)^2 + (j-k-1)^2)}{2\sigma^2}\right).$$

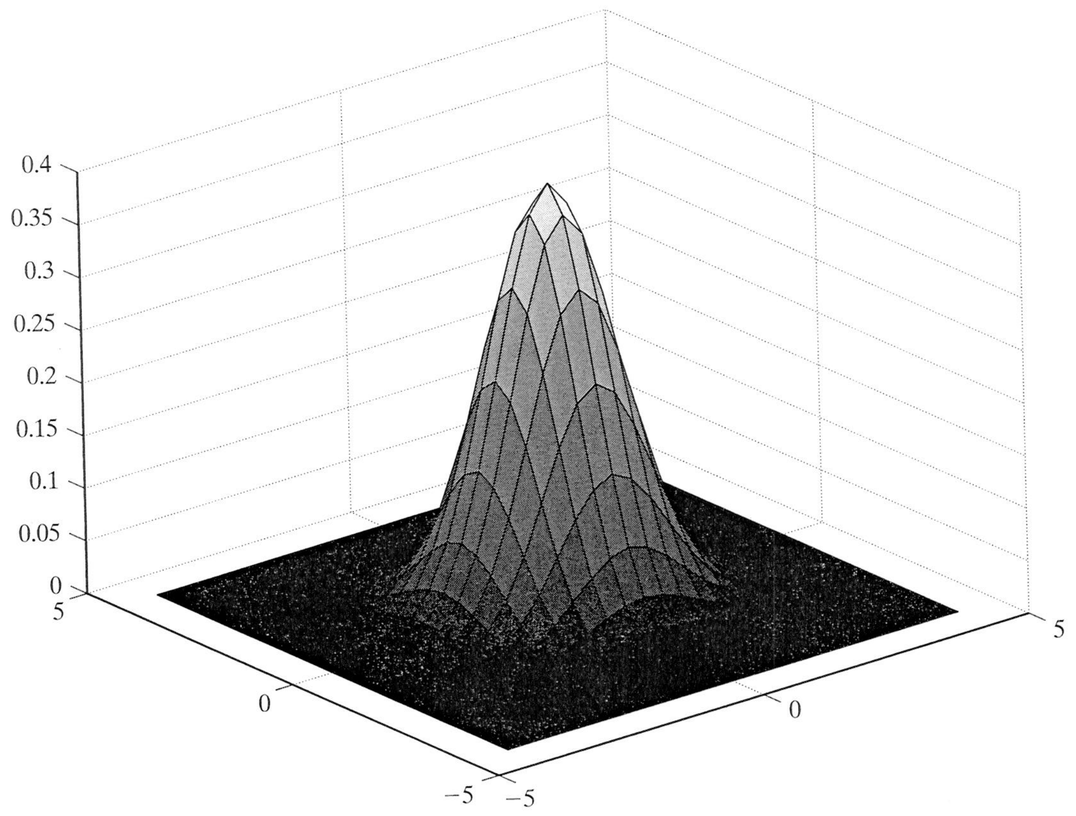

Figure 7.2 The symmetric Gaussian kernel in 2D. This view shows a kernel scaled so that its sum is equal to one; this scaling is quite often omitted. The kernel shown has $\sigma = 1$. Convolution with this kernel forms a weighted average that stresses the point at the center of the convolution window and incorporates little contribution from those at the boundary. Notice how the Gaussian is qualitatively similar to our description of the point spread function of image blur; it is circularly symmetric, has strongest response in the center, and dies away near the boundaries.

Notice that some care must be exercised with σ; if σ is too small, then only one element of the array will have a nonzero value. If σ is large, then k must be large, too, otherwise we are ignoring contributions from pixels that should contribute with substantial weight.

Example 7.3 Derivatives and Finite Differences.

Image derivatives can be approximated using another example of a convolution process. Because

$$\frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon, y) - f(x, y)}{\epsilon},$$

we might estimate a partial derivative as a symmetric *finite difference*:

$$\frac{\partial h}{\partial x} \approx h_{i+1,j} - h_{i-1,j}.$$

This is the same as a convolution, where the convolution kernel is

$$\mathcal{H} = \left\{ \begin{array}{ccc} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{array} \right\}.$$

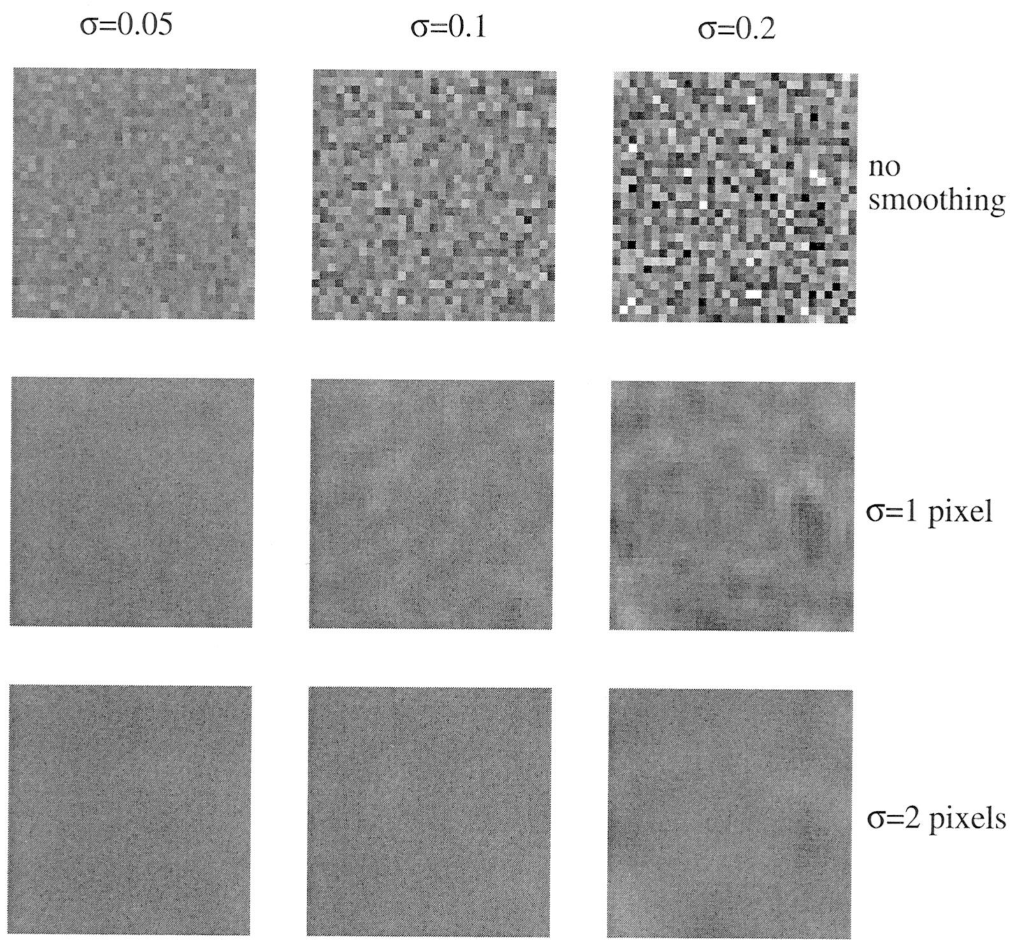

Figure 7.3 The **top row** shows images of a constant mid-gray level corrupted by additive Gaussian noise. In this noise model, each pixel has a zero-mean normal random variable added to it. The range of pixel values is from zero to one, so that the standard deviation of the noise in the first column is about 1/20 of full range. The **center row** shows the effect of smoothing the corresponding image in the top row with a Gaussian filter of σ one pixel. Notice the annoying overloading of notation here; there is Gaussian noise and Gaussian filters, and both have σ's. One uses context to keep these two straight, although this is not always as helpful as it could be because Gaussian filters are particularly good at suppressing Gaussian noise. This is because the noise values at each pixel are independent, meaning that the expected value of their average is going to be the noise mean. The **bottom row** shows the effect of smoothing the corresponding image in the top row with a Gaussian filter of σ two pixels.

Notice that this kernel could be interpreted as a template: It gives a large positive response to an image configuration that is positive on one side and negative on the other, and a large negative response to the mirror image.

As Figure 7.4 suggests, finite differences give a most unsatisfactory estimate of the derivative. This is because finite differences respond strongly (i.e., have an output with large magnitude) at fast changes, and fast changes are characteristic of noise. Roughly, this is because image pixels tend to look like one another. For example, if we had bought a discount camera with some pixels that were stuck at either black or white, the output of the finite difference process would be large at those pixels

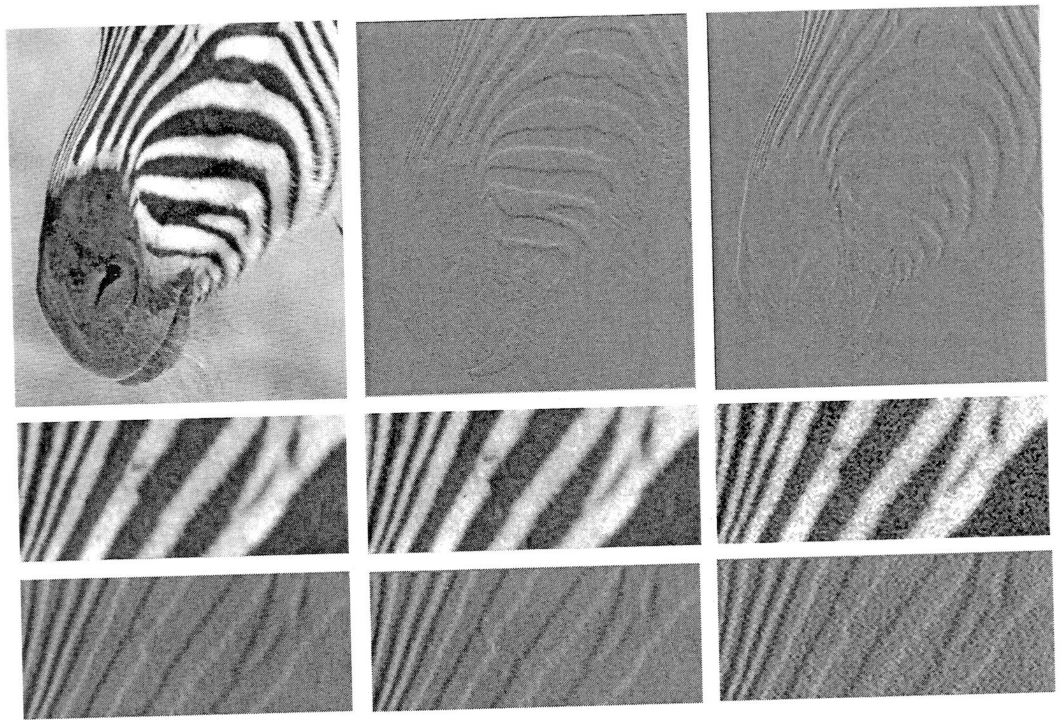

Figure 7.4 The **top row** shows estimates of derivatives obtained by finite differences. The image at the **left** shows a detail from a picture of a zebra. The **center** image shows the partial derivative in the y-direction— which responds strongly to horizontal stripes and weakly to vertical stripes—and the **right** image shows the partial derivative in the x-direction— which responds strongly to vertical stripes and weakly to horizontal stripes. However, finite differences respond strongly to noise. The image at **center left** shows a detail from a picture of a zebra; the next image in the row is obtained by adding a random number with zero mean and normal distribution ($\sigma = 0.03$—the darkest value in the image is 0, and the lightest 1) to each pixel; and the third image is obtained by adding a random number with zero mean and normal distribution ($\sigma = 0.09$) to each pixel. The **bottom row** shows the partial derivative in the x-direction of the image at the head of the row. Notice how strongly the differentiation process emphasizes image noise—the derivative figures look increasingly grainy. In the derivative figures, a mid-gray level is a zero value, a dark gray level is a negative value, and a light gray level is a positive value.

because they are, in general, substantially different from their neighbors. All this suggests that some form of smoothing is appropriate before differentiation; the details appear in Sections 8.1 and 8.2.

7.2 SHIFT INVARIANT LINEAR SYSTEMS

Convolution represents the effect of a large class of system. In particular, most imaging systems have, to a good approximation, three significant properties:

- **Superposition:** We expect that

$$R(f+g) = R(f) + R(g);$$

that is, the response to the sum of stimuli is the sum of the individual responses.

- **Scaling:** The response to a zero input is zero. Taken with superposition, we have that the response to a scaled stimulus is a scaled version of the response to the original stimulus—that is,

$$R(kf) = kR(f).$$

A device that exhibits superposition and scaling is *linear*.

- **Shift invariance:** In a *shift invariant* system, the response to a translated stimulus is just a translation of the response to the stimulus. This means that, for example, if a view of a small light aimed at the center of the camera is a small bright blob, then if the light is moved to the periphery, we should see the same small bright blob, only translated.

A device that is linear and shift invariant is known as a *shift invariant linear system*, or often just as a *system*.

The response of a shift invariant linear system to a stimulus is obtained by convolution. We demonstrate this first for systems that take discrete inputs—say vectors or arrays—and produce discrete outputs. We then use this to describe the behavior of systems that operate on continuous functions of the line or the plane, and from this analysis we obtain some useful facts about convolution.

7.2.1 Discrete Convolution

In the 1D case, we have a shift invariant linear system that takes a vector and responds with a vector. This case is the easiest to handle because there are fewer indices to look after. The 2D case, a system that takes an array and responds with an array, follows easily. In each case, we assume that the input and output are infinite dimensional. This allows us to ignore some minor issues that arise at the boundaries of the input. We deal with these in Section 7.2.3.

Discrete Convolution in One Dimension We have an input vector, $\boldsymbol{f}$. For convenience, we assume that the vector is infinite, and its elements are indexed by the integers (i.e., there is an element with index -1, say). The ith component of this vector is f_i. Now $\boldsymbol{f}$ is a weighted sum of basis elements. A convenient basis is a set of elements that have a one in a single component and zeros elsewhere. We write

$$\boldsymbol{e}_0 = \ldots 0, 0, 0, 1, 0, 0, 0, \ldots$$

This is a data vector that has a 1 in the zeroth place, and zeros elsewhere. Define a shift operation, which takes a vector to a shifted version of that vector. In particular, the vector $\mathtt{Shift}(\boldsymbol{f}, i)$ has, as its jth component, the $j - i$th component of $\boldsymbol{f}$. For example, $\mathtt{Shift}(\boldsymbol{e}_0, 1)$ has a zero in the first component. Now we can write

$$\boldsymbol{f} = \sum_i f_i \,\mathtt{Shift}(\boldsymbol{e}_0, i).$$

We write the response of our system to a vector $\boldsymbol{f}$ as

$$R(\boldsymbol{f}).$$

Now because the system is shift invariant, we have

$$R(\mathtt{Shift}(\boldsymbol{f}, k)) = \mathtt{Shift}(R(\boldsymbol{f}), k).$$

Furthermore, because it is linear, we have

$$R(k\boldsymbol{f}) = kR(\boldsymbol{f}).$$

This means that

$$\begin{aligned} R(f) &= R\left(\sum_i f_i \, \texttt{Shift}(e_0, i)\right) \\ &= \sum_i R(f_i \, \texttt{Shift}(e_0, i)) \\ &= \sum_i f_i R(\texttt{Shift}(e_0, i)) \\ &= \sum_i f_i \, \texttt{Shift}(R(e_0), i)). \end{aligned}$$

This means that, to obtain the system's response to any data vector, we need to know only its response to e_0. This is usually called the system's *impulse response*. Assume that the impulse response can be written as g. We have

$$R(f) = \sum_i f_i \, \texttt{Shift}(g, i) = g * f.$$

This defines an operation—the 1D, discrete version of convolution—which we write with a $*$.

This is all very well, but it doesn't give us a particularly easy expression for the output. If we consider the jth element of $R(f)$, which we write as R_i, we must have

$$R_j = \sum_i g_{j-i} f_i,$$

which conforms to (and explains the origin of) the form used in Section 7.1.1.

Discrete Convolution in Two Dimensions We now use an array of values and write the i, jth element of the array $\mathcal{D}$ as D_{ij}. The appropriate analogy to an impulse response is the response to a stimulus that looks like

$$\mathcal{E}_{00} = \begin{array}{ccccc} \dots & \dots & \dots & \dots & \dots \\ \dots & 0 & 0 & 0 & \dots \\ \dots & 0 & 1 & 0 & \dots \\ \dots & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{array}$$

If $\mathcal{G}$ is the response of the system to this stimulus, the same considerations as for 1D convolution yield a response to a stimulus $\mathcal{F}$—that is,

$$R_{ij} = \sum_{u,v} G_{i-u,j-v} F_{uv},$$

which we write as

$$\mathcal{R} = \mathcal{G} * * \mathcal{H}.$$

7.2.2 Continuous Convolution

There are shift invariant linear systems that produce a continuous response to a continuous input; for example, a camera lens takes a set of radiances and produces another set, and many lenses are approximately shift invariant. A brief study of these systems allows us to study the information lost by approximating a continuous function—the incoming radiance values across an image plane—by a discrete function—the value at each pixel.

The natural description is in terms of the system's response to a rather unnatural function, the δ-function, which is not a function in formal terms. We do the derivation first in one dimension to make the notation easier.

Convolution in One Dimension We obtain an expression for the response of a continuous shift invariant linear system from our expression for a discrete system. We can take a discrete input and replace each value with a box straddling the value; this gives a continuous input function. We then make the boxes narrower and consider what happens in the limit.

Our system takes a function of one dimension and returns a function of one dimension. Again, we write the response of the system to some input $f(x)$ as $R(f)$; when we need to emphasize that f is a function, we write $R(f(x))$. The response is also a *function*; occasionally, when we need to emphasize this fact, we write $R(f)(u)$. We can express the linearity property in this notation by writing

$$R(kf) = kR(f)$$

(for k some constant) and the shift invariance property by introducing a `Shift` operator, which takes functions to functions:

$$\texttt{Shift}(f, c) = f(u - c).$$

With this `Shift` operator, we can write the shift invariance property as

$$R(\texttt{Shift}(f, c)) = \texttt{Shift}(R(f), c).$$

We define the *box* function as:

$$box_\epsilon(x) = \begin{cases} 0 & abs(x) > \frac{\epsilon}{2} \\ 1 & abs(x) < \frac{\epsilon}{2} \end{cases} .$$

The value of $box_\epsilon(\epsilon/2)$ does not matter for our purposes. The input function is $f(x)$. We construct an even grid of points x_i, where $x_{i+1} - x_i = \epsilon$. We now construct a vector $\mathbf{f}$ whose ith component (written f_i) is $f(x_i)$. This vector can be used to represent the function.

We obtain an approximate representation of f by $\sum_i f_i \, \texttt{Shift}(box_\epsilon, x_i)$. We apply this input to a shift invariant linear system; the response is a weighted sum of shifted responses to box functions. This means that

$$\begin{aligned} R(\sum_i f_i \, \texttt{Shift}(box_\epsilon, x_i)) &= \sum_i R(f_i \, \texttt{Shift}(box_\epsilon, x_i)) \\ &= \sum_i f_i R(\texttt{Shift}(box_\epsilon, x_i)) \\ &= \sum_i f_i \, \texttt{Shift}\left(R\left(\frac{box_\epsilon}{\epsilon}\epsilon\right), x_i\right) \\ &= \sum_i f_i \, \texttt{Shift}\left(R\left(\frac{box_\epsilon}{\epsilon}\right), x_i\right)\epsilon. \end{aligned}$$

So far, everything has followed our derivation for discrete functions. We now have something that looks like an approximate integral if $\epsilon \to 0$.

We introduce a new device, called a δ-function, to deal with the term box_ϵ/ϵ. Define

$$d_\epsilon(x) = \frac{box_\epsilon(x)}{\epsilon}.$$

The δ-function is:

$$\delta(x) = \lim_{\epsilon \to 0} d_\epsilon(x).$$

We don't attempt to evaluate this limit, so we need not discuss the value of $\delta(0)$. One interesting feature of this function is that, for practical shift invariant linear systems, the response of the system to a δ-function exists and has *compact support* (i.e., is zero except on a finite number of intervals of finite length). For example, a good model of a δ-function in 2D is an extremely small, extremely bright light. If we make the light smaller and brighter while ensuring the total energy is constant, we expect to see a small but finite spot due to the defocus of the lens. The δ-function is the natural analogue for $\boldsymbol{e}_0$ in the continuous case.

This means that the expression for the response of the system,

$$\sum_i f_i \, \texttt{Shift}\left(R\left(\frac{box_\epsilon}{\epsilon}\right), x_i\right)\epsilon,$$

turns into an integral as ϵ limits to zero. We obtain

$$\begin{aligned} R(f) &= \int \left\{R(\delta)(u - x')\right\} f(x')\,dx' \\ &= \int g(u - x') f(x')\,dx', \end{aligned}$$

where we have written $R(\delta)$—which is usually called the **impulse response** of the system—as g and have omitted the limits of the integral. These integrals could be from $-\infty$ to ∞, but more stringent limits could apply if g and h have compact support. This operation is called **convolution** (again), and we write the foregoing expression as

$$R(f) = (g * f).$$

Convolution is *symmetric*, meaning

$$(g * h)(x) = (h * g)(x).$$

Convolution is *associative*, meaning that

$$(f * (g * h)) = ((f * g) * h).$$

This latter property means that we can find a single shift invariant linear system that behaves like the composition of two different systems. This comes in useful when we discuss sampling.

Convolution in Two Dimensions The derivation of convolution in two dimensions requires more notation. A box function is now given by $box_{\epsilon^2}(x, y) = box_\epsilon(x)box_\epsilon(y)$; we now have

$$d_\epsilon(x, y) = \frac{box_{\epsilon^2}(x, y)}{\epsilon^2}.$$

The δ-function is the limit of $d_\epsilon(x, y)$ function as $\epsilon \to 0$. Finally, there are more terms in the sum. All this activity results in the expression

$$\begin{aligned} R(h)(x,y) &= \int\int g(x-x', y-y')h(x',y')\,dx\,dy \\ &= (g**h)(x,y), \end{aligned}$$

where we have used two $*$s to indicate a two-dimensional convolution. Convolution *in 2D* is *symmetric*, meaning that

$$(g**h) = (h**g)$$

and *associative*, meaning that

$$((f**g)**h) = (f**(g**h)).$$

A natural model for the impulse response of a two-dimensional system is to think of the pattern seen in a camera viewing a very small, distant light source (which subtends a very small viewing angle). In practical lenses, this view results in some form of fuzzy blob, justifying the name **point spread function**, which is often used for the impulse response of a 2D system. The point spread function of a linear system is often known as its *kernel*.

7.2.3 Edge Effects in Discrete Convolutions

In practical systems, we cannot have infinite arrays of data. This means that when we compute the convolution, we need to contend with the edges of the image; at the edges, there are pixel locations where computing the value of the convolved image requires image values that don't exist. There are a variety of strategies we can adopt:

- **Ignore these locations**—this means that we report only values for which every required image location exists. This has the advantage of probity, but the disadvantage that the output is smaller than the input. Repeated convolutions can cause the image to shrink quite drastically.
- **Pad the image with constant values**—this means that, as we look at output values closer to the edge of the image, the extent to which the output of the convolution depends on the image goes down. This is a convenient trick because we can ensure that the image doesn't shrink, but it has the disadvantage that it can create the appearance of substantial gradients near the boundary.
- **Pad the image in some other way**—for example, we might think of the image as a doubly periodic function so that if we have an $n \times m$ image, then column $m+1$—required for the purposes of convolution—would be the same as column $m-1$. This can create the appearance of substantial second derivative values near the boundary.

7.3 SPATIAL FREQUENCY AND FOURIER TRANSFORMS

We have used the trick of thinking of a signal $g(x,y)$ as a weighted sum of a large (or infinite) number of small (or infinitely small) box functions. This model emphasizes that a signal is an element of a vector space—the box functions form a convenient basis, and the weights are coefficients on this basis. We need a new technique to deal with two related problems so far left open:

- Although it is clear that a discrete image version cannot represent the full information in a signal, we have not yet indicated what is lost;

- It is clear that we cannot shrink an image simply by taking every kth pixel—this could turn a checkerboard image all white or all black—and we should like to know how to shrink an image safely.

All of these problems are related to the presence of fast changes in an image. For example, shrinking an image is most likely to miss fast effects because they could slip between samples; similarly, the derivative is large at fast changes.

These effects can be studied by *a change of basis*. We change the basis to be a set of sinusoids and represent the signal as an infinite weighted sum of an infinite number of sinusoids. This means that fast changes in the signal are obvious, because they correspond to large amounts of high-frequency sinusoids in the new basis.

7.3.1 Fourier Transforms

The change of basis is effected by a *Fourier transform*. We define the Fourier transform of a signal $g(x, y)$ to be

$$\mathcal{F}(g(x, y))(u, v) = \iint_{-\infty}^{\infty} g(x, y) e^{-i2\pi(ux+vy)}\, dx\, dy.$$

Assume that appropriate technical conditions are true to make this integral exist. It is sufficient for all moments of g to be finite; a variety of other possible conditions are available (Bracewell, 1995). The process takes a complex valued function of x, y and returns a complex valued function of u, v (images are complex valued functions with zero imaginary component).

For the moment, fix u and v, and let us consider the meaning of the value of the transform at that point. The exponential can be rewritten

$$e^{-i2\pi(ux+vy)} = \cos(2\pi(ux + vy)) + i \sin(2\pi(ux + vy)).$$

These terms are sinusoids on the x, y plane, whose orientation and frequency are given by u, v. For example, consider the real term, which is constant when $ux + vy$ is constant (i.e., along a straight line in the x, y plane whose orientation is given by $\tan\theta = v/u$). The gradient of this term is perpendicular to lines where $ux + vy$ is constant, and the frequency of the sinusoid is $\sqrt{u^2 + v^2}$. These sinusoids are often referred to as *spatial frequency components*; a variety are illustrated in Figure 7.5.

The integral should be seen as a dot product. If we fix u and v, the value of the integral is the dot product between a sinusoid in x and y and the original function. This is a useful analogy because dot products measure the amount of one vector in the direction of another.

In the same way, the value of the transform at a particular u and v can be seen as measuring the amount of the sinusoid with given frequency and orientation in the signal. The transform takes a function of x and y to the function of u and v whose value at any particular (u, v) is the amount of that particular sinusoid in the original function. This view justifies the model of a Fourier transform as a change of basis.

Linearity The Fourier transform is linear:

$$\mathcal{F}(g(x, y) + h(x, y)) = \mathcal{F}(g(x, y)) + \mathcal{F}(h(x, y))$$

and

$$\mathcal{F}(kg(x, y)) = k\mathcal{F}(g(x, y)).$$

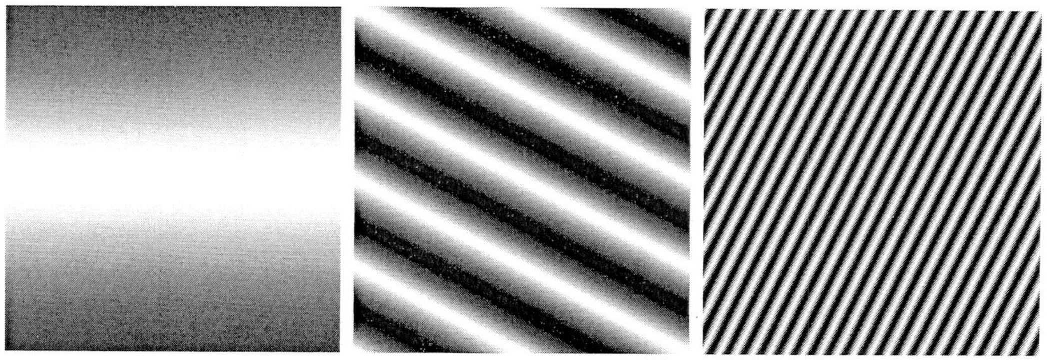

Figure 7.5 The real component of Fourier basis elements shown as intensity images. The brightest point has value one, and the darkest point has value zero. The domain is $[-1, 1] \times [-1, 1]$, with the origin at the center of the image. On the left, $(u, v) = (0, 0.4)$; in the center, $(u, v) = (1, 2)$ and on the right $(u, v) = (10, -5)$. These are sinusoids of various frequencies and orientations described in the text.

The Inverse Fourier Transform It is useful to recover a signal from its Fourier transform. This is another change of basis with the form

$$g(x, y) = \iint\limits_{-\infty}^{\infty} \mathcal{F}(g(x, y))(u, v) e^{i2\pi(ux+vy)}\, du\, dv.$$

Fourier Transform Pairs Fourier transforms are known in closed form for a variety of useful cases; a large set of examples appears in Bracewell (1995). We list a few in Table 7.1 for reference. The last line of Table 7.1 contains the *convolution theorem*; convolution in the signal domain is the same as multiplication in the Fourier domain. We use this important fact several times in what follows (Section 9.2.2).

Phase and Magnitude The Fourier transform consists of a real and a complex component:

$$\begin{aligned}
\mathcal{F}(g(x, y))(u, v) &= \int\int_{-\infty}^{\infty} g(x, y) \cos(2\pi(ux + vy))\, dx\, dy \\
&\quad + i \int\int_{-\infty}^{\infty} g(x, y) \sin(2\pi(ux + vy))\, dx\, dy \\
&= \Re(\mathcal{F}(g)) + i * \Im(\mathcal{F}(g)) \\
&= \mathcal{F}_R(g) + i * \mathcal{F}_I(g).
\end{aligned}$$

It is usually inconvenient to draw complex functions of the plane. One solution is to plot $\mathcal{F}_R(g)$ and $\mathcal{F}_I(g)$ separately; another is to consider the *magnitude* and *phase* of the complex functions, and to plot these instead. These are then called the *magnitude spectrum* and *phase spectrum*, respectively.

The value of the Fourier transform of a function at a particular u, v point depends on the whole function. This is obvious from the definition because the domain of the integral is the whole domain of the function. It leads to some subtle properties, however. First, a local change in the function (e.g., zeroing out a block of points) is going to lead to a change *at every point* in the Fourier transform. This means that the Fourier transform is quite difficult to use as a

TABLE 7.1 A variety of functions of two dimensions and their Fourier transforms. This table can be used in two directions (with appropriate substitutions for u, v and x, y) because the Fourier transform of the Fourier transform of a function is the function. Observant readers may suspect that the results on infinite sums of δ functions contradict the linearity of Fourier transforms. By careful inspection of limits, it is possible to show that they do not (see, e.g., Bracewell, 1995). Observant readers may also have noted that an expression for $\mathcal{F}(\frac{\partial f}{\partial y})$ can be obtained by combining two lines of this table.

Function	Fourier transform
$g(x, y)$	$\iint_{-\infty}^{\infty} g(x, y)e^{-i2\pi(ux+vy)}\, dx\, dy$
$\iint_{-\infty}^{\infty} \mathcal{F}(g(x, y))(u, v)e^{i2\pi(ux+vy)}\, du\, dv$	$\mathcal{F}(g(x, y))(u, v)$
$\delta(x, y)$	1
$\frac{\partial f}{\partial x}(x, y)$	$u\mathcal{F}(f)(u, v)$
$0.5\delta(x + a, y) + 0.5\delta(x - a, y)$	$\cos 2\pi au$
$e^{-\pi(x^2+y^2)}$	$e^{-\pi(u^2+v^2)}$
$box_1(x, y)$	$\frac{\sin u}{u} \frac{\sin v}{v}$
$f(ax, by)$	$\frac{\mathcal{F}(f)(u/a, v/b)}{ab}$
$\sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \delta(x - i, y - j)$	$\sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \delta(u - i, v - j)$
$(f * *g)(x, y)$	$\mathcal{F}(f)\mathcal{F}(g)(u, v)$
$f(x - a, y - b)$	$e^{-i2\pi(au+bv)}\mathcal{F}(f)$
$f(x \cos\theta - y \sin\theta, x \sin\theta + y \cos\theta)$	$\mathcal{F}(f)(u \cos\theta - v \sin\theta, u \sin\theta + v \cos\theta)$

representation (e.g., it might be very difficult to tell if a pattern was present in an image just by looking at the Fourier transform). Second, the magnitude spectra of images tends to be similar. This appears to be a fact of nature, rather than something that can be proven axiomatically. As a result, the magnitude spectrum of an image is surprisingly uninformative (see Figure 7.6 for an example).

7.4 SAMPLING AND ALIASING

The crucial reason to discuss Fourier transforms is to get some insight into the difference between discrete and continuous images. In particular, it is clear that some information has been lost when we work on a discrete pixel grid, but what? A good, simple example to think about comes from an image of a checkerboard, and is given in Figure 7.7. The problem has to do with the number of samples relative to the function; we can formalize this rather precisely given a sufficiently powerful model.

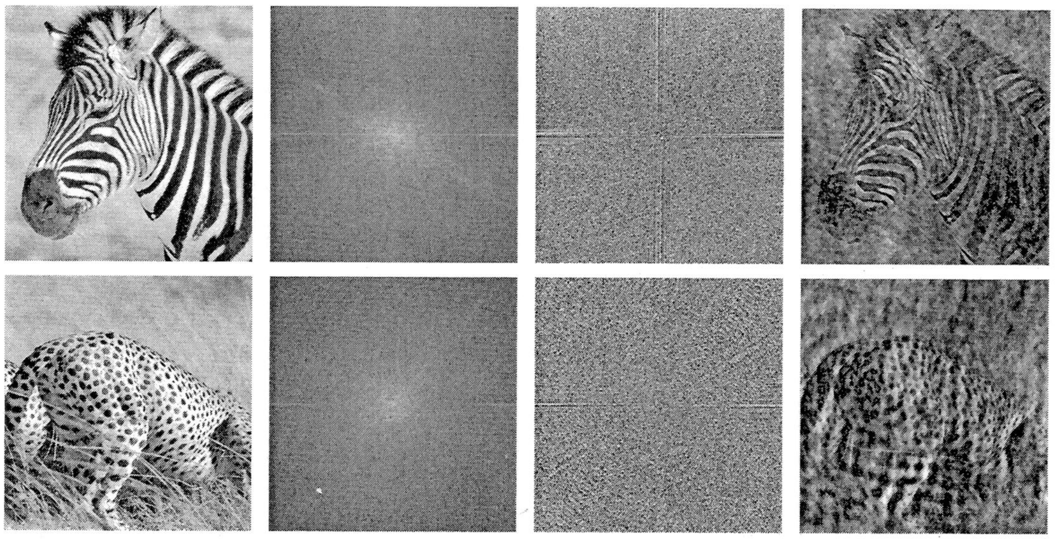

Figure 7.6 The second image in each row shows the log of the magnitude spectrum for the first image in the row; the third image shows the phase spectrum scaled so that $-\pi$ is dark and π is light. The final images are obtained by swapping the magnitude spectra. Although this swap leads to substantial image noise, it doesn't substantially affect the interpretation of the image, suggesting that the phase spectrum is more important for perception than the magnitude spectrum.

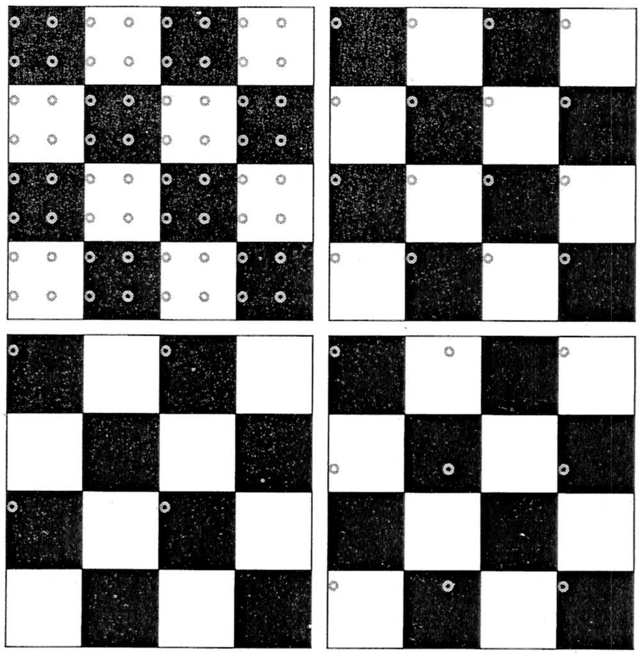

Figure 7.7 The two checkerboards on the **top** illustrate a sampling procedure that appears to be successful (whether it is or not depends on some details that we will deal with later). The grey circles represent the samples; if there are sufficient samples, then the samples represent the detail in the underlying function. The sampling procedures shown on the **bottom** is unequivocally unsuccessful; the samples suggest that there are fewer checks than there are. This illustrates two important phenomena: First, successful sampling schemes sample data often enough; second, unsuccessful sampling schemes cause high-frequency information to appear as lower frequency information.

7.4.1 Sampling

Passing from a continuous function—like the irradiance at the back of a camera system—to a collection of values on a discrete grid—like the pixel values reported by a camera—is referred to as *sampling*. We construct a model that allows us to obtain a precise notion of what is lost in sampling.

Sampling in One Dimension Sampling in one dimension takes a function and returns a discrete set of values. The most important case involves sampling on a uniform discrete grid, and we assume that the samples are defined at integer points. This means we have a process that takes some function and returns a vector of values:

$$\texttt{sample}_{1D}(f(x)) = \boldsymbol{f}.$$

We model this sampling process by assuming that the elements of this vector are the values of the function $f(x)$ at the sample points and allowing negative indices to the vector (Figure 7.8). This means that the ith component of $\boldsymbol{f}$ is $f(x_i)$.

Sampling in Two Dimensions Sampling in 2D is very similar to sampling in 1D. Although sampling can occur on nonregular grids (the best example being the human retina), we proceed on the assumption that samples are drawn at points with integer coordinates. This yields a uniform rectangular grid, which is a good model of most cameras. Our sampled images are then rectangular arrays of finite size (all values outside the grid being zero).

In the formal model, we sample a function of two dimensions, instead of one, yielding an array (Figure 7.9). This array we allow to have negative indices in both dimensions, and can then write

$$\texttt{sample}_{2D}(F(x, y)) = \mathcal{F},$$

where the i, jth element of the array $\mathcal{F}$ is $F(x_i, y_j) = F(i, j)$.

Samples are not always evenly spaced in practical systems. This is quite often due to the pervasive effect of television; television screens have an aspect ratio of 4:3 (width:height). Cameras quite often accommodate this effect by spacing sample points slightly farther apart horizontally than vertically (in jargon, they have *non-square pixels*).

A Continuous Model of a Sampled Signal We need a continuous model of a sampled signal. Generally, this model is used to evaluate integrals—in particular, taking a Fourier

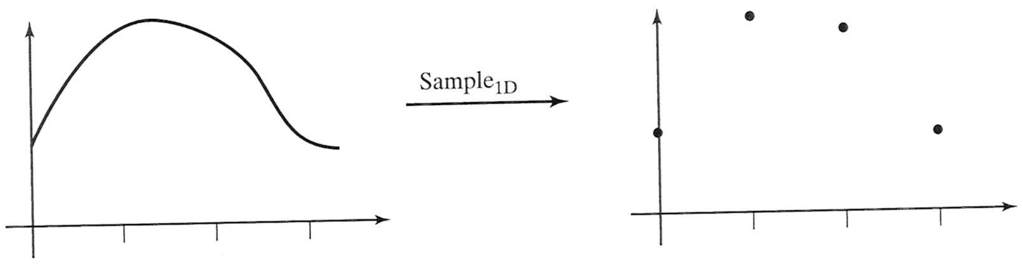

Figure 7.8 Sampling in 1D takes a function and returns a vector whose elements are values of that function at the sample points, as the top figures show. For our purposes, it is enough that the sample points be integer values of the argument. We allow the vector to be infinite dimensional and have negative as well as positive indices.

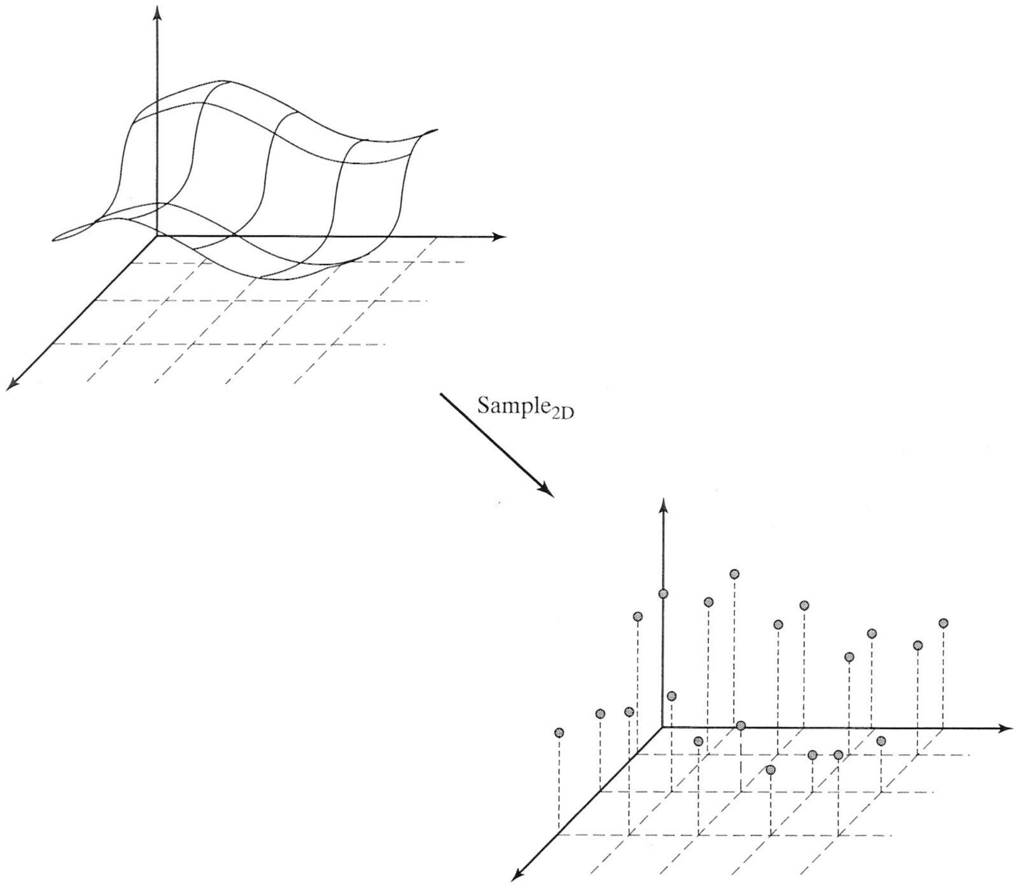

Figure 7.9 Sampling in 2D takes a function and returns an array; again, we allow the array to be infinite dimensional and to have negative as well as positive indices.

transform involves integrating the product of our model with a complex exponential. It is clear how this integral should behave—the value of the integral should be obtained by adding up values at each integer point. This means we cannot model a sampled signal as a function that is zero everywhere except at integer points (where it takes the value of the signal) because this model has a zero integral.

An appropriate continuous model of a sampled signal relies on an important property of the δ function:

$$\begin{aligned}\int_{-\infty}^{\infty} a\delta(x)f(x)\,dx &= a\lim_{\epsilon\to 0}\int_{-\infty}^{\infty} d(x;\epsilon)f(x)\,dx\\ &= a\lim_{\epsilon\to 0}\int_{-\infty}^{\infty}\frac{bar(x;\epsilon)}{\epsilon}(f(x))\,dx\\ &= a\lim_{\epsilon\to 0}\sum_{i=-\infty}^{\infty}\frac{bar(x;\epsilon)}{\epsilon}(f(i\epsilon)bar(x-i\epsilon;\epsilon))\epsilon\\ &= af(0).\end{aligned}$$

Here we have used the idea of an integral as the limit of a sum of small strips.

An appropriate continuous model of a sampled signal consists of a δ-function at each sample point weighted by the value of the sample at that point. We can obtain this model by multiplying the sampled signal by a set of δ-functions, one at each sample point. In one dimension, a function of this form is called a *comb function* (because that's what the graph looks like). In two dimensions, a function of this form is called a *bed-of-nails function* (for the same reason).

Working in 2D and assuming that the samples are at integer points, this procedure gets

$$\begin{aligned}\texttt{sample}_{2D}(f) &= \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f(i,j)\delta(x-i,y-j)\\ &= f(x,y)\left\{\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right\}.\end{aligned}$$

This function is zero except at integer points (because the δ-function is zero except at integer points), and its integral is the sum of the function values at the integer points.

7.4.2 Aliasing

Sampling involves a loss of information. As this section shows, a signal sampled too slowly is misrepresented by the samples; high spatial frequency components of the original signal appear as low spatial frequency components in the sampled signal—an effect known as *aliasing*.

The Fourier Transform of a Sampled Signal A sampled signal is given by a product of the original signal with a bed-of-nails function. By the convolution theorem, the Fourier transform of this product is the convolution of the Fourier transforms of the two functions. This means that the Fourier transform of a sampled signal is obtained by convolving the Fourier transform of the signal with another bed-of-nails function.

Now convolving a function with a shifted δ-function merely shifts the function (see exercises). This means that the Fourier transform of the sampled signal is the sum of a collection of shifted versions of the Fourier transforms of the signal, that is,

$$\begin{aligned}\mathcal{F}(\texttt{sample}_{2D}(f(x,y))) &= \mathcal{F}\left(f(x,y)\left\{\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right\}\right)\\ &= \mathcal{F}(f(x,y)) **\mathcal{F}\left(\left\{\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right\}\right)\\ &= \sum_{i=-\infty}^{\infty} F(u-i,v-j),\end{aligned}$$

where we have written the Fourier transform of $f(x,y)$ as $F(u,v)$.

If the support of these shifted versions of the Fourier transform of the signal does not intersect, we can easily reconstruct the signal from the sampled version. We take the sampled signal, Fourier transform it, and cut out one copy of the Fourier transform of the signal and Fourier transform this back (Figure 7.10).

However, if the support regions *do* overlap, we are not able to reconstruct the signal because we can't determine the Fourier transform of the signal in the regions of overlap, where different copies of the Fourier transform will add. This results in a characteristic effect, usually called **aliasing**, where high spatial frequencies appear to be low spatial frequencies (see Figure 7.12

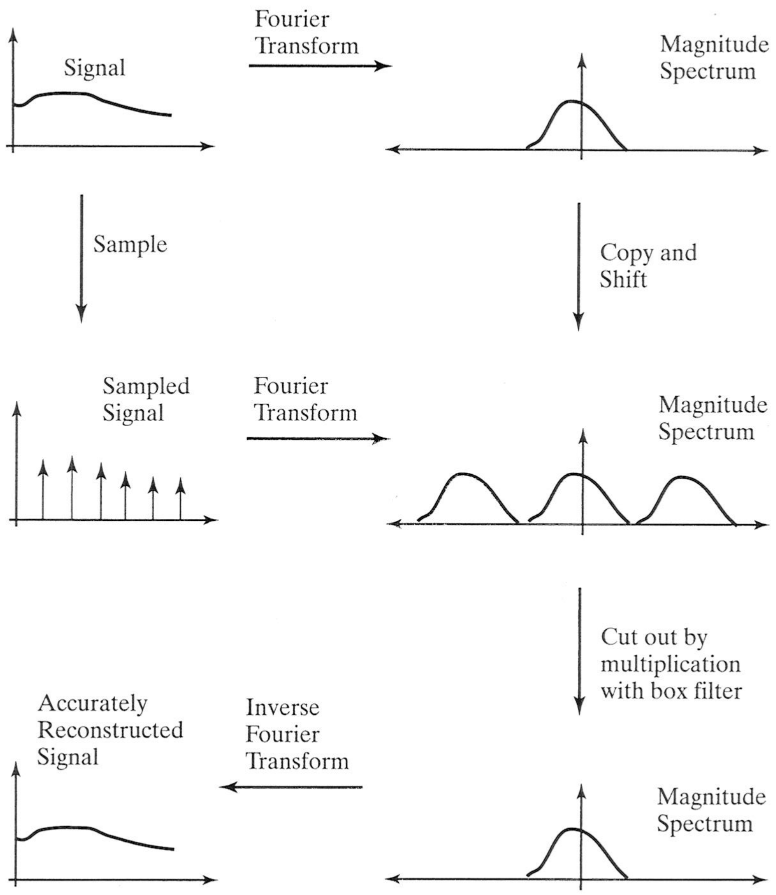

Figure 7.10 The Fourier transform of the sampled signal consists of a sum of copies of the Fourier transform of the original signal, shifted with respect to each other by the sampling frequency. Two possibilities occur. If the shifted copies do not intersect with each other (as in this case), the original signal can be reconstructed from the sampled signal (we just cut out one copy of the Fourier transform and inverse transform it). If they do intersect (as in Figure 7.11), the intersection region is added, and so we cannot obtain a separate copy of the Fourier transform, and the signal has aliased.

and exercises). Our argument also yields *Nyquist's theorem*—the sampling frequency must be at least twice the highest frequency present for a signal to be reconstructed from a sampled version.

7.4.3 Smoothing and Resampling

Nyquist's theorem means it is dangerous to shrink an image by simply taking every kth pixel (as Figure 7.12 confirms). Instead, we need to filter the image so that spatial frequencies above the new sampling frequency are removed. We could do this exactly by multiplying the image Fourier transform by a scaled 2D bar function, which would act as a low-pass filter. Equivalently, we would convolve the image with a kernel of the form $(\sin x \sin y)/(xy)$. This is a difficult and expensive (a polite way of saying *impossible*) convolution because this function has infinite support.

The most interesting case occurs when we want to halve the width and height of the image. We assume that the sampled image has no aliasing (because if it did, there would be nothing we

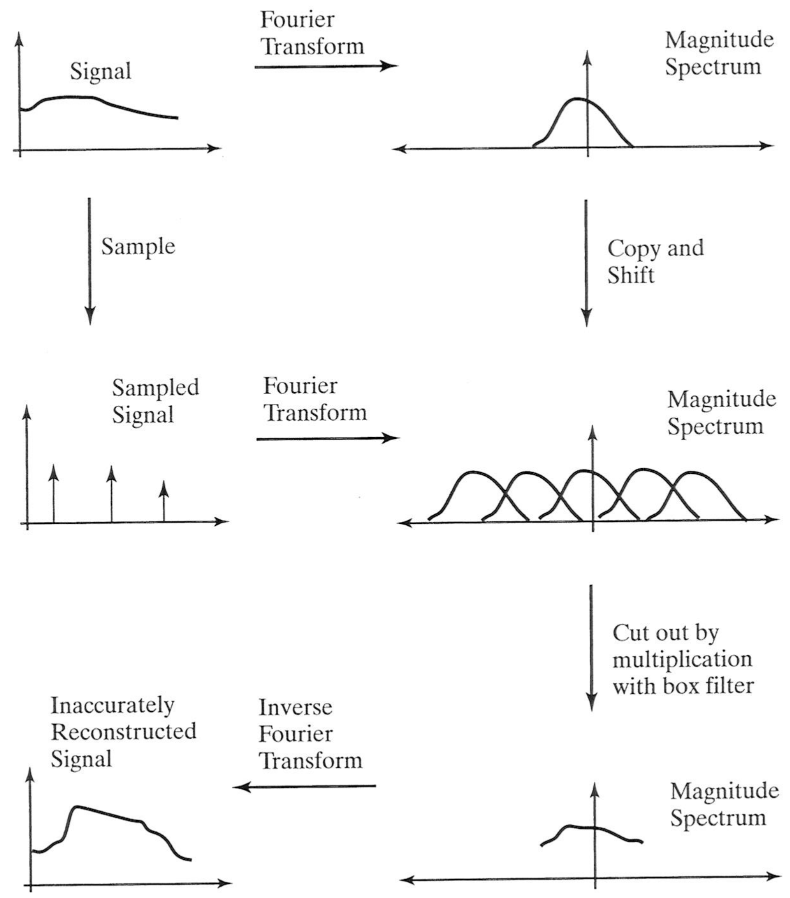

Figure 7.11 The Fourier transform of the sampled signal consists of a sum of copies of the Fourier transform of the original signal, shifted with respect to each other by the sampling frequency. Two possibilities occur. If the shifted copies do not intersect with each other (as in Figure 7.10), the original signal can be reconstructed from the sampled signal (we just cut out one copy of the Fourier transform and inverse transform it). If they do intersect (as in this figure), the intersection region is added, and so we cannot obtain a separate copy of the Fourier transform, and the signal has aliased. This also explains the tendency of high spatial frequencies to alias to lower spatial frequencies.

could do about it anyway; once an image has been sampled, any aliasing that is going to occur has happened, and there's not much we can do about it without an image model). This means that the Fourier transform of the sampled image is going to consist of a set of copies of some Fourier transform, with centers shifted to integer points in u, v space.

If we resample this signal, the copies now have centers on the half-integer points in u, v space. This means that, to avoid aliasing, we need to apply a filter that strongly reduces the content of the original Fourier transform outside the range $|u| < 1/2$, $|v| < 1/2$. Of course, if we reduce the content of the signal *inside* this range, we may lose information, too. Now the Fourier transform of a Gaussian is a Gaussian, and Gaussians die away fairly quickly. Thus, if we were to convolve the image with a Gaussian—or multiply its Fourier transform by a Gaussian, which is the same thing—we could achieve what we want.

The choice of Gaussian depends on the application; if σ is large, there is less aliasing (because the value of the kernel outside our range is very small), but information is lost because the kernel is not flat within our range; similarly, if σ is small, less information is lost within the

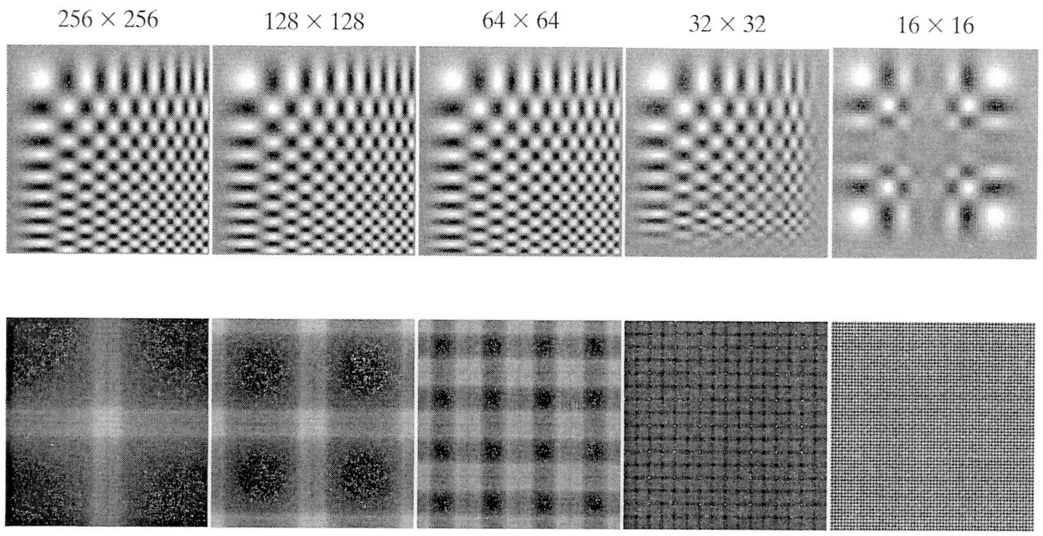

Figure 7.12 The **top row** shows sampled versions of an image of a grid obtained by multiplying two sinusoids with linearly increasing frequency—one in x and one in y. The other images in the series are obtained by resampling by factors of two without smoothing (i.e., the next is a 128x128, then a 64x64, etc., all scaled to the same size). Note the substantial aliasing; high spatial frequencies alias down to low spatial frequencies, and the smallest image is an extremely poor representation of the large image. The **bottom row** shows the magnitude of the Fourier transform of each image displayed as a log to compress the intensity scale. The constant component is at the center. Notice that the Fourier transform of a resampled image is obtained by scaling the Fourier transform of the original image and then tiling the plane. Interference between copies of the original Fourier transform means that we cannot recover its value at some points; this is the mechanism underlying aliasing.

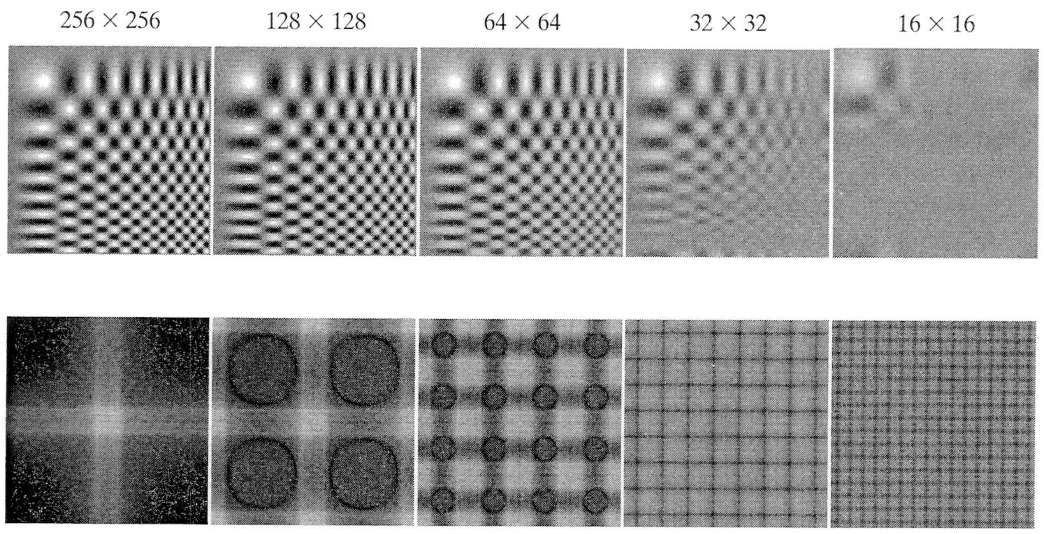

Figure 7.13 **Top:** Resampled versions of the image of Figure 7.12, again by factors of two, but this time each image is smoothed with a Gaussian of σ one pixel before resampling. This filter is a low-pass filter, and so suppresses high spatial frequency components, reducing aliasing. **Bottom:** The effect of the low-pass filter is easily seen in these log-magnitude images; the low-pass filter suppresses the high spatial frequency components so that components interfere less to reduce aliasing.

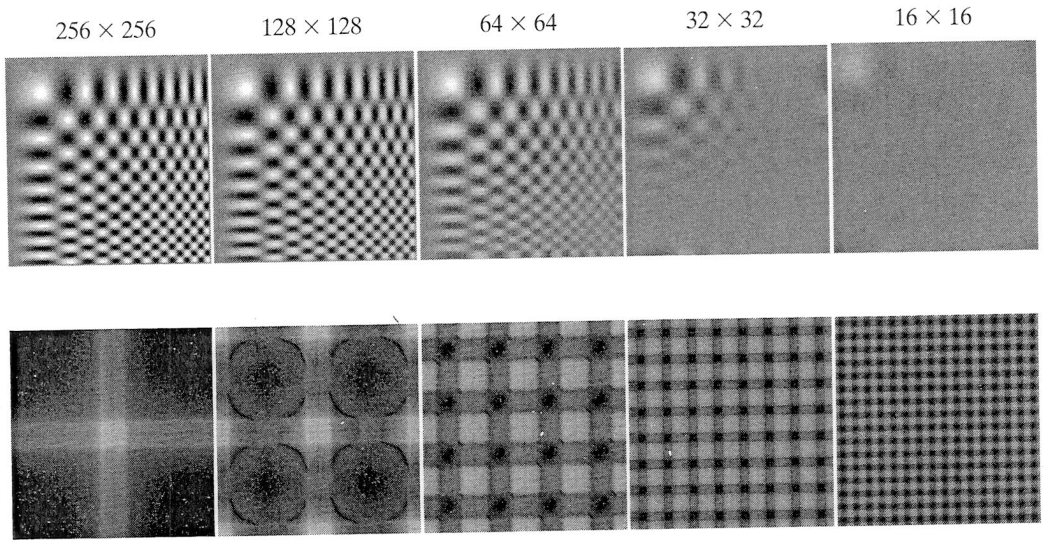

Figure 7.14 **Top:** Resampled versions of the image of Figure 7.12, again by factors of two, but this time each image is smoothed with a Gaussian of σ two pixels before resampling. This filter suppresses high spatial frequency components more aggressively than that of Figure 7.13. **Bottom:** The effect of the low-pass filter is easily seen in these log-magnitude images; the low-pass filter suppresses the high spatial frequency components so that components interfere less, to reduce aliasing.

range, but aliasing can be more substantial. Figures 7.13 and 7.14 illustrate the effects of different choices of σ.

We have been using a Gaussian as a low-pass filter because its response at high spatial frequencies is low and at low spatial frequencies is high. In fact, the Gaussian is not a particularly good low-pass filter. What one wants is a filter whose response is pretty close to constant for some range of low spatial frequencies— the pass band—and whose response is also pretty close to constant—and zero—for higher spatial frequencies—the stop band. It is possible to design low-pass filters that are significantly better than Gaussians. The design process involves detailed compromise between criteria of ripple—how flat is the respnse in the pass band and the stop band?—and roll-off—how quickly does the response fall to zero and stay there? The basic steps for resampling an image are given in Algorithm 7.1.

Algorithm 7.1: Subsampling an Image by a Factor of Two

```
Apply a low-pass filter to the original image
  (a Gaussian with a $\sigma$ of between one
  and two pixels is usually an acceptable choice).
Create a new image whose dimensions on edge are half
  those of the old image
Set the value of the $i, j$th pixel of the new image to the value
  of the $2i, 2j$th pixel of the filtered image
```

7.5 FILTERS AS TEMPLATES

It turns out that filters offer a natural mechanism for finding simple patterns because filters respond most strongly to pattern elements that look like the filter. For example, smoothed derivative filters are intended to give a strong response at a point where the derivative is large. At these points, the kernel of the filter looks like the effect it is intended to detect. The x-derivative filters look like a vertical light blob next to a vertical dark blob (an arrangement where there is a large x-derivative), and so on.

It is generally the case that filters intended to give a strong response to a pattern look like that pattern (Figure 7.15). This is a simple geometric result.

7.5.1 Convolution as a Dot Product

Recall from Section 7.1.1 that, for $\mathcal{G}$ the kernel of some linear filter, the response of this filter to an image $\mathcal{H}$ is given by:

$$R_{ij} = \sum_{u,v} G_{i-u,j-v} H_{uv}.$$

Now consider the response of a filter at the point where i and j are zero. This is

$$R = \sum_{u,v} G_{-u,-v} H_{u,v}.$$

This response is obtained by associating image elements with filter kernel elements, multiplying the associated elements, and summing. We could scan the image into a vector, and the filter kernel into another vector, in such a way that associated elements are in the same component. By inserting zeros as needed, we can ensure that these two vectors have the same dimension. Once this is done, the process of multiplying associated elements and summing is precisely the same as taking a dot product.

This is a powerful analogy because this dot product, like any other, achieves its largest value when the vector representing the image is parallel to the vector representing the filter

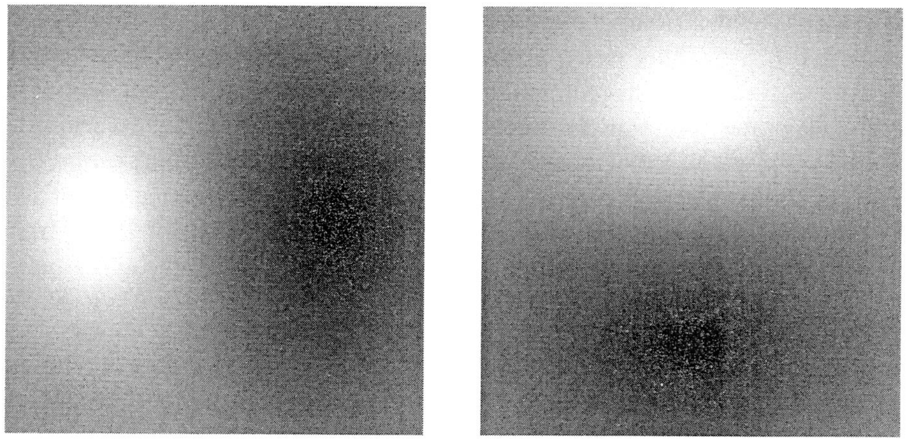

Figure 7.15 Filter kernels look like the effects they are intended to detect. On the **left**, a smoothed derivative of Gaussian filter that looks for large changes in the x-direction (such as a dark blob next to a light blob); on the **right**, a smoothed derivative of Gaussian filter that looks for large changes in the y-direction.

kernel. This means that a filter responds most strongly when it encounters an image pattern that looks like the filter. The response of a filter gets stronger as a region gets brighter, too.

Now consider the response of the image to a filter at some other point. Nothing significant about our model has changed. Again, we can scan the image into one vector and the filter kernel into another vector, such that associated elements lie in the same components. Again, the result of applying this filter is a dot product. There are two useful ways to think about this dot product.

7.5.2 Changing Basis

We can think of convolution as a dot product between the image and *a different vector at each point* (because we have moved the filter kernel to lie over some other point in the image). The new vector is obtained by rearranging the old one so that the elements lie in the right components to make the sum work out. This means that, by convolving an image with a filter, we are representing the image on a new *basis* of the vector space of images—the basis given by the different shifted versions of the filter. The original basis elements were vectors with a zero in all slots except one. The new basis elements are shifted versions of a single pattern.

For many of the kernels discussed, we expect that this process will *lose* information—for the same reason that smoothing suppresses noise—so that the coefficients on this basis are redundant. While the coefficients may be redundant, they expose image structure in a useful way. This basis transformation is valuable in texture analysis. Typically, we choose a basis that consists of small, useful pattern components. Large values of the basis coefficients suggest that a pattern component is present, and texture can be represented by representing the relationships between these pattern components, usually with some form of probability model.

7.6 TECHNIQUE: NORMALIZED CORRELATION AND FINDING PATTERNS

We can think of convolution as comparing a filter with a patch of image centered at the point whose response we are looking at. In this view, the image neighborhood corresponding to the filter kernel is scanned into a vector that is compared with the filter kernel. By itself, this dot product is a poor way to find features because the value may be large simply because the image region is bright. By analogy with vectors, we are interested in the cosine of the angle between the filter vector and the image neighborhood vector; this suggests computing the root sum of squares of the relevant image region (the image elements that would lie under the filter kernel) and dividing the response by that value.

This yields a value that is large and positive when the image region looks like the filter kernel, and small and negative when the image region looks like a contrast-reversed version of the filter kernel. This value could be squared if contrast reversal doesn't matter. This is a cheap and effective method for finding patterns, often called *normalized correlation.*

7.6.1 Controlling the Television by Finding Hands by Normalized Correlation

It would be nice to have systems that could respond to human gestures. For example, you might wave at the light to get the room illuminated, point at the air conditioning to get the room temperature changed, or make an appropriate gesture at an annoying politician on television and get a change in channel. In typical consumer applications, there are quite strict limits to the amount of computation available, meaning that it is essential that the gesture recognition system be simple. However, such systems are usually quite limited in what they need to do, too.

Controlling the Television Typically, a user interface is in some state—perhaps a menu is displayed—and an event occurs—perhaps a button is pressed on a remote control. This event causes the interface to change state—a new menu item is highlighted, say—and the whole process continues. In some states, some events cause the system to perform some action—the channel might change. All this means that a state machine is a natural model for a user interface.

One way for vision to fit into this model is to provide events. This is good because there are generally few different kinds of event, and we know what kinds of event the system should care about in any particular state. As a result, the vision system needs only to determine whether either nothing or one of a small number of known kinds of event has occurred. It is quite often possible to build systems that meet these constraints.

A relatively small set of events is required to simulate a remote control; one needs events that look like button presses (e.g., to turn the television on or off), and events that look like pointer motion (e.g., to increase the volume; it is possible to do this with buttons, too). With these events, the television can be turned on, and an on-screen menu system navigated.

Finding Hands Freeman, Anderson and et al. (1998) produced an interface where an open hand turns the television on. This can be robust because all the system needs to do is determine whether there is a hand in view. Furthermore, the user will cooperate by holding their hand up and open. Because the user is expected to be a fairly constant distance from the camera—so the size of the hand is roughly known, and there is no need to search over scales—and in front of the television, the image region that needs to be searched to determine whether there is a hand is quite small.

The hand is held up in a fairly standard configuration and orientation to turn the television set on, and it usually appears at about the same distance from the television (so we know what it looks like). This means that a normalized correlation score is sufficient to find the hand. Any points in the correlation image where the score is high enough correspond to hands. This approach can be used to control volume and so on, as well as turn the television on and off. To do so, we need some notion of where the hand is going—to one side turns the volume up, to the other turns it down—and this can be obtained by comparing the position in the previous frame with that in the current frame. The system displays an iconic representation of its interpretation of hand position so the user has some feedback as to what the system is doing (Figure 7.16). Notice that an attractive feature of this approach is that it could be self-calibrating. In this approach, when you install your television set, you sit in front of it and show it your hand a few times to allow it to get an estimate of the scale at which the hand appears.

7.7 TECHNIQUE: SCALE AND IMAGE PYRAMIDS

Images look quite different at different scales. For example, the zebra's muzzle in Figure 7.17 can be described in terms of individual hairs—which might be coded in terms of the response of oriented filters that operate at a scale of a small number of pixels—or in terms of the stripes on the zebra. In the case of the zebra, we would not want to apply large filters to find the stripes. This is because these filters are inclined to spurious precision—we don't wish to represent the disposition of each hair on the stripe—inconvenient to build, and slow to apply. A more practical approach than applying large filters is to apply smaller filters to smoothed and resampled versions of the image.

7.7.1 The Gaussian Pyramid

An *image pyramid* is a collection of representations of an image. The name comes from a visual analogy. Typically, each layer of the pyramid is half the width and half the height of the previous layer; if we were to stack the layers on top of each other, a pyramid would result. In a **Gaussian**

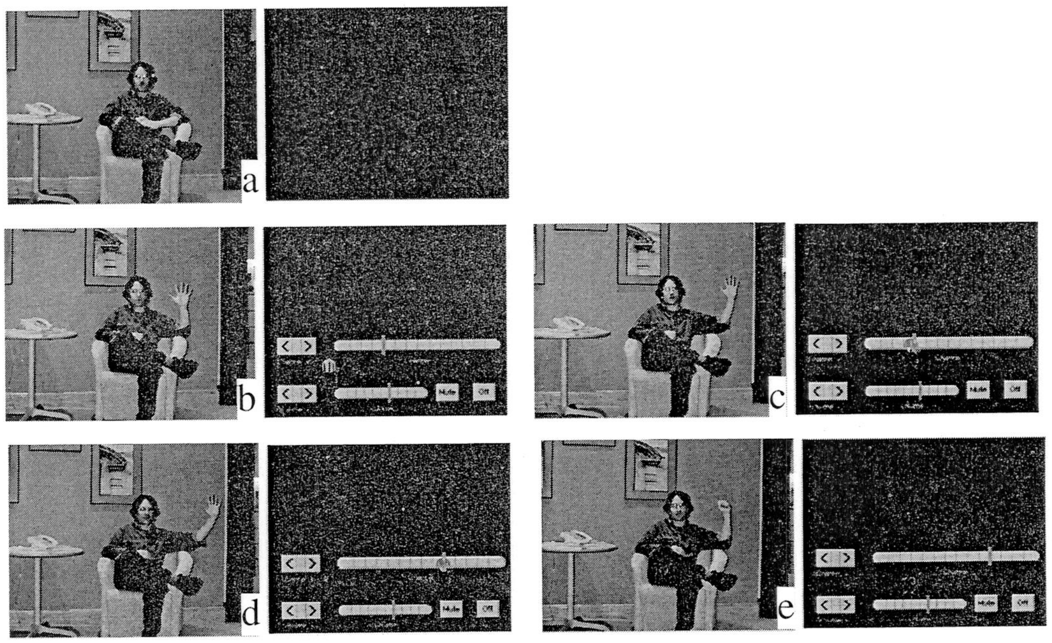

Figure 7.16 Examples of Freeman *et al.*'s system controlling a television set. Each state is illustrated with what the television sees on the **left** and what the user sees on the **right**. In **(a)**, the television is asleep, but a process is watching the user. An open hand causes the television to come on and show its user interface panel **(b)**. Focus on the panel tracks the movement of the user's open hand in **(c)**, and the user can change channel by using this tracking to move an icon on the screen in **(d)**. Finally, the user displays a closed hand in **(e)** to turn off the set. *Reprinted from "Computer Vision for Interactive Computer Graphics," by W.Freeman et al., IEEE Computer Graphics and Applications, 1998 © 1998 IEEE*

Algorithm 7.2: Forming a Gaussian Pyramid

```
Set the finest scale layer to the image
For each layer, going from next to finest to coarsest
   Obtain this layer by smoothing the next finest
   layer with a Gaussian, and then subsampling it
end
```

pyramid, each layer is smoothed by a symmetric Gaussian kernel and resampled to get the next layer (Figure 7.17). These pyramids are most convenient if the image dimensions are a power of two or a multiple of a power of two. The smallest image is the most heavily smoothed; the layers are often referred to as *coarse scale* versions of the image.

With a little notation, we can write simple expressions for the layers of a Gaussian pyramid. The operator $S^{\downarrow}$ downsamples an image; in particular, the j, kth element of $S^{\downarrow}(\mathcal{I})$ is the $2j, 2k$th element of $\mathcal{I}$. The nth level of a pyramid $P(\mathcal{I})$ is denoted $P(\mathcal{I})_n$. With this notation, we have

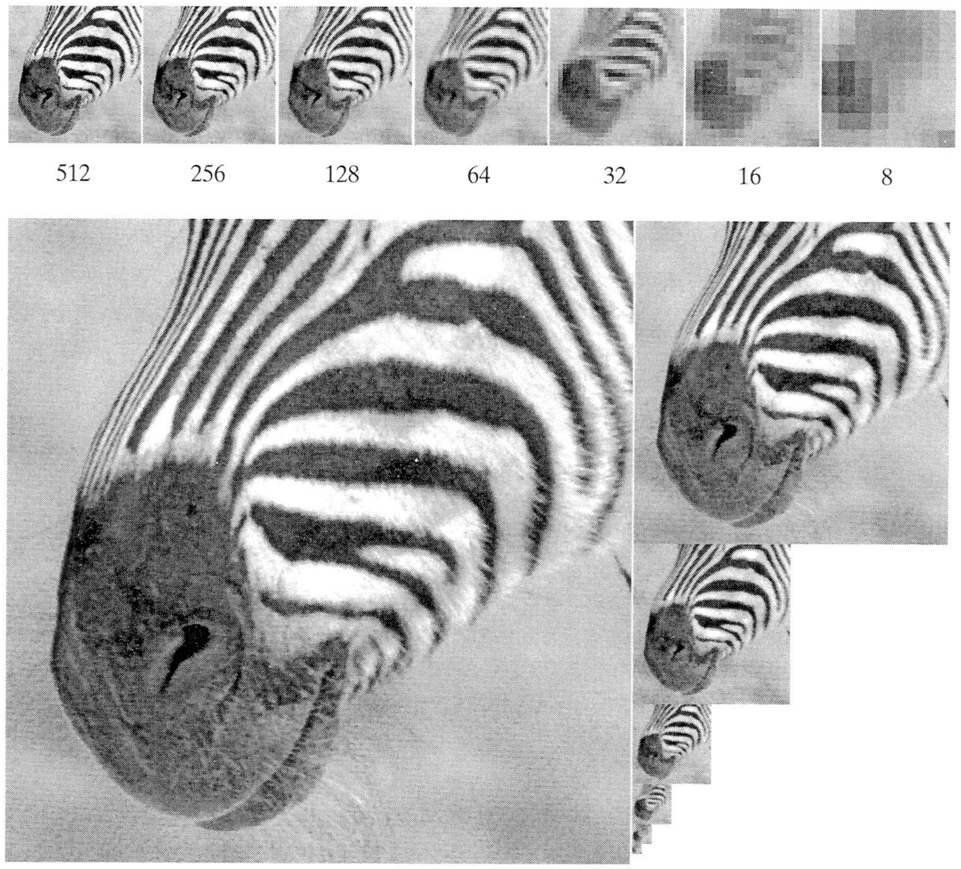

Figure 7.17 A Gaussian pyramid of images running from 512x512 to 8x8. On the top row, we have shown each image at the same size (so that some have bigger pixels than others), and the lower part of the figure shows the images to scale. Notice that if we convolve each image with a fixed size filter, it responds to quite different phenomena. An 8x8 pixel block at the finest scale might contain a few hairs; at a coarser scale, it might contain an entire stripe; and at the coarsest scale, it contains the animal's muzzle.

$$\begin{aligned} P_{\text{Gaussian}}(\mathcal{I})_{n+1} &= S^{\downarrow}(G_\sigma ** P_{\text{Gaussian}}(\mathcal{I})_n) \\ &= S^{\downarrow} G_\sigma (P_{\text{Gaussian}}(\mathcal{I})_n) \end{aligned}$$

(where we have written G_σ for the linear operator that takes an image to the convolution of that image with a Gaussian). The finest scale layer is the original image:

$$P_{\text{Gaussian}}(\mathcal{I})_1 = \mathcal{I}.$$

7.7.2 Applications of Scaled Representations

Gaussian pyramids are useful because they make it possible to extract representations of different types of structure in an image. There are three standard applications.

Search over Scale Numerous objects can be represented as small image patterns. A standard example is a frontal view of a face. Typically, at low resolution, frontal views of faces have a quite distinctive pattern: The eyes form dark pools, under a dark bar (the eyebrows), separated by a lighter bar (specular reflections from the nose), and above a dark bar (the mouth). There are various methods for finding faces that exploit these properties (see chapter 22). These methods all assume that the face lies in a small range of scales. All other faces are found by searching a pyramid. To find bigger faces, we look at coarser scale layers, and to find smaller faces we look at finer scale layers. This useful trick applies to many different kinds of feature, as we see in the chapters that follow.

Spatial Search One application is spatial search, a common theme in computer vision. Typically, we have a point in one image and are trying to find a point in a second image that corresponds to it. This problem occurs in stereopsis—where the point has moved because the two images are obtained from different viewing positions—and in motion analysis—where the image point has moved either because the camera moved or because it is on a moving object.

Searching for a match in the original pairs of images is inefficient because we may have to wade through a great deal of detail. A better approach, which is now pretty much universal, is to look for a match in a heavily smoothed and resampled image and then refine that match by looking at increasingly detailed versions of the image. For example, we might reduce 1024×1024 images down to 4×4 versions, match those, and then look at 8×8 versions (because we know a rough match, it is easy to refine it); we then look at 16×16 versions, and so on, all the way up to 1024×1024. This gives an extremely efficient search because a step of a single pixel in the 4×4 version is equivalent to a step of 256 pixels in the 1024×1024 version. This strategy is known as *coarse-to-fine matching*.

Feature Tracking Most features found at coarse levels of smoothing are associated with large, high-contrast image events because for a feature to be marked at a coarse scale a large pool of pixels need to agree that it is there. Typically, finding coarse scale phenomena misestimates both the size and location of a feature. For example, a single pixel error in a coarse-scale image represents a multiple pixel error in a fine-scale image.

At fine scales, there are many features, some of which are associated with smaller, low-contrast events. One strategy for improving a set of features obtained at a fine scale is to track features across scales to a coarser scale and accept only the fine scale features that have identifiable parents at a coarser scale. This strategy, known as *feature tracking* in principle, can suppress features resulting from textured regions (often referred to as noise) and features resulting from real noise.

7.8 NOTES

We don't claim to be exhaustive in our treatment of linear systems, but it wouldn't be possible to read the literature on filters in vision without a grasp of the ideas in this chapter. We have given a fairly straightforward account here; more details on these topics can be found in the excellent books by Bracewell (1995), (2000).

Real Imaging Systems versus Shift Invariant Linear Systems

Imaging systems are only approximately linear. Film is not linear—it does not respond to weak stimuli, and it saturates for bright stimuli—but one can usually get away with a linear model within a reasonable range. CCD cameras are linear within a working range. They give a small, but non zero response to a zero input as a result of thermal noise (which is why astronomers cool their cameras) and they saturate for very bright stimuli. CCD cameras often contain electronics that transforms their output to make them behave more like film because consumers are used to film. Shift invariance is approximate as well because lenses tend to distort responses near the image boundary. Some lenses—fish-eye lenses are a good example—are not shift invariant.

Scale

There is a large body of work on scale space and scaled representations. The origins appear to lie with Witkin (1983) and the idea was developed by Koenderink and van Doorn (1986). Since then, a huge literature has sprung up (one might start with ter Haar Romeny, Florack, Koenderink and Viergever, 1997 or Nielsen, Johansen, Olsen and Weickert, 1999). We have given only the briefest picture here because the analysis tends to be quite tricky. The usefulness of the techniques is currently hotly debated, too.

Anisotropic Scaling

One important difficulty with scale space models is that the symmetric Gaussian smoothing process tends to blur out edges rather too aggressively for comfort. For example, if we have two trees near one another on a skyline, the large-scale blobs corresponding to each tree may start merging before all the small-scale blobs have finished. This suggests that we should smooth differently at edge points than at other points. For example, we might make an estimate of the magnitude and orientation of the gradient: For large gradients, we would then use an oriented smoothing operator that smoothed aggressively perpendicular to the gradient and little along the gradient; for small gradients, we might use a symmetric smoothing operator. This idea used to be known as *edge-preserving smoothing*.

In the modern, more formal version, due to Perona and Malik (1990*a,b*), we notice the scale space representation family is a solution to the *diffusion equation*

$$\begin{aligned}\frac{\partial \Phi}{\partial \sigma} &= \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} \\ &= \nabla^2 \Phi,\end{aligned}$$

with the initial condition

$$\Phi(x, y, 0) = \mathcal{I}(x, y)$$

If this equation is modified to have the form

$$\begin{aligned}\frac{\partial \Phi}{\partial \sigma} &= \nabla \cdot (c(x, y, \sigma) \nabla \Phi) \\ &= c(x, y, \sigma) \nabla^2 \Phi + (\nabla c(x, y, \sigma)) \cdot (\nabla \Phi)\end{aligned}$$

with the same initial condition, then if $c(x, y, \sigma) = 1$, we have the diffusion equation we started with, and if $c(x, y, \sigma) = 0$ there is no smoothing. We assume that c does not depend on σ. If we knew where the edges were in the image, we could construct a mask that consisted of regions where $c(x, y) = 1$, isolated by patches along the edges where $c(x, y) = 0$; in this case, a solution would smooth *inside* each separate region, but not over the edge. Although we do not know where the edges are — the exercise would be empty if we did—we can obtain reasonable choices of $c(x, y)$ from the magnitude of the image gradient. If the gradient is large, then c should be small and vice versa. There is a substantial literature dealing with this approach; a good place to start is ter Haar Romeny (1994).

ASSIGNMENTS

PROBLEMS

7.1. Show that forming unweighted local averages, which yields an operation of the form

$$\mathcal{R}_{ij} = \frac{1}{(2k+1)^2} \sum_{u=i-k}^{u=i+k} \sum_{v=j-k}^{v=j+k} \mathcal{F}_{uv}$$

is a convolution. What is the kernel of this convolution?

7.2. Write $\mathcal{E}_0$ for an image that consists of all zeros with a single one at the center. Show that convolving this image with the kernel

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{((i-k-1)^2 + (j-k-1)^2)}{2\sigma^2}\right)$$

(which is a discretised Gaussian) yields a circularly symmetric fuzzy blob.

7.3. Show that convolving an image with a discrete, separable 2D filter kernel is equivalent to convolving with two 1D filter kernels. Estimate the number of operations saved for an NxN image and a $2k+1 \times 2k+1$ kernel.

7.4. Show that convolving a function with a δ function simply reproduces the original function. Now show that convolving a function with a shifted δ function shifts the function.

7.5. We said that convolving the image with a kernel of the form $(\sin x \sin y)/(xy)$ is impossible because this function has infinite support. Why would it be impossible to Fourier transform the image, multiply the Fourier transform by a box function, and then inverse-Fourier transform the result? Hint: Think support.

7.6. Aliasing takes high spatial frequencies to low spatial frequencies. Explain why the following effects occur:

(a) In old cowboy films that show wagons moving, the wheel often seems to be stationary or moving in the wrong direction (i.e., the wagon moves from left to right and the wheel seems to be turning counterclockwise).

(b) White shirts with thin dark pinstripes often generate a shimmering array of colors on television.

(c) In ray-traced pictures, soft shadows generated by area sources look blocky.

Programming Assignments

7.7. One way to obtain a Gaussian kernel is to convolve a constant kernel with itself many times. Compare this strategy with evaluating a Gaussian kernel.

(a) How many repeated convolutions do you need to get a reasonable approximation? (You need to establish what a reasonable approximation is; you might plot the quality of the approximation against the number of repeated convolutions).

(b) Are there any benefits that can be obtained like this? (Hint: Not every computer comes with an FPU.)

7.8. Write a program that produces a Gaussian pyramid from an image.

7.9. A sampled Gaussian kernel must alias because the kernel contains components at arbitrarily high spatial frequencies. Assume that the kernel is sampled on an infinite grid. As the standard deviation gets smaller, the aliased energy must increase. Plot the energy that aliases against the standard deviation of the Gaussian kernel in pixels. Now assume that the Gaussian kernel is given on a 7x7 grid. If the aliased energy must be of the same order of magnitude as the error due to truncating the Gaussian, what is the smallest standard deviation that can be expressed on this grid?