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*

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

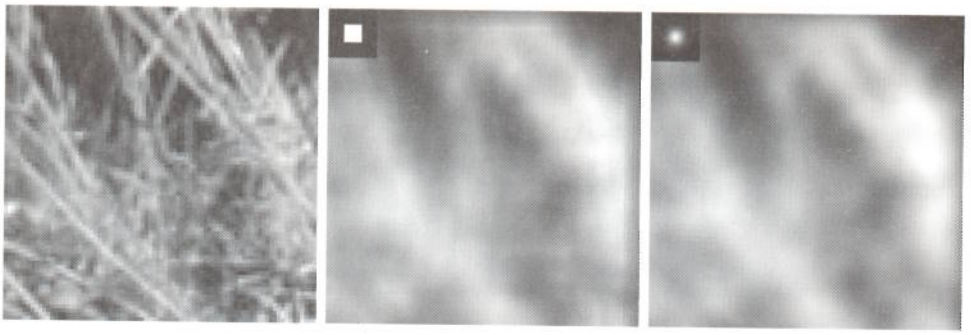

Figure 7.1 Although a uniform local average may seem to give a good blurring model, it generates effects not usually seen in defocusing a lens. The images above compare the effects of a uniform local average with weighted average. The image on the **left** shows a view of grass. In the **center**, the 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

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

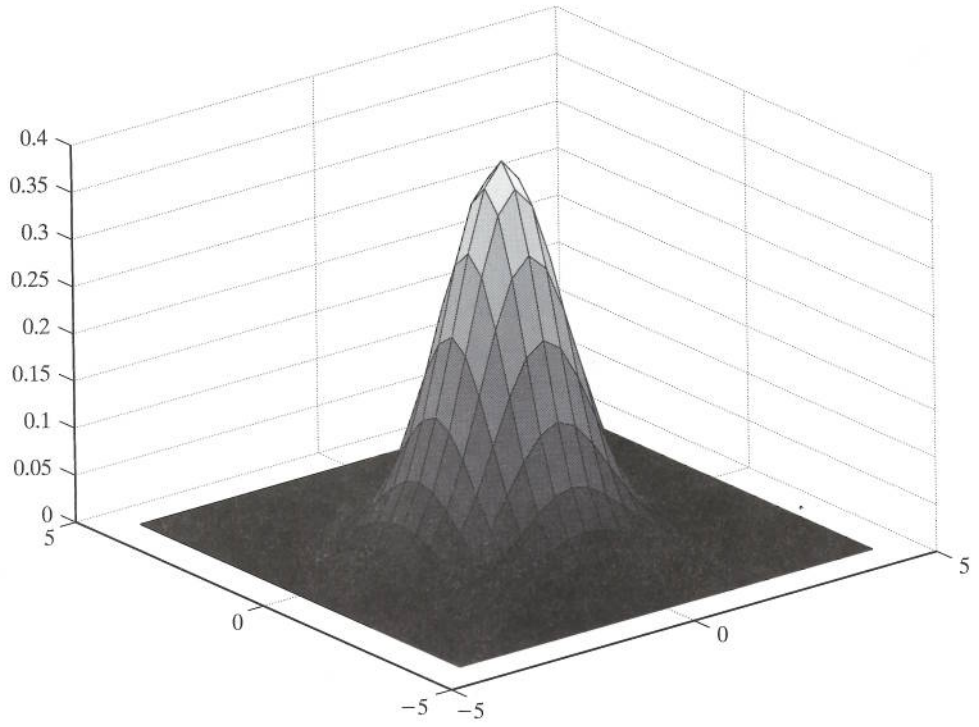

Figure 7.2 The symmetric Gaussian kernel in 2D. This view shows a kernel scaled so that its sum is equal to one; this scaling is quite often omitted. The kernel shown has $\sigma = 1$. Convolution with this kernel forms a weighted average that stresses the point at the center of the convolution window 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

$$\mathcal{H} = \begin{Bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{Bmatrix}.$$

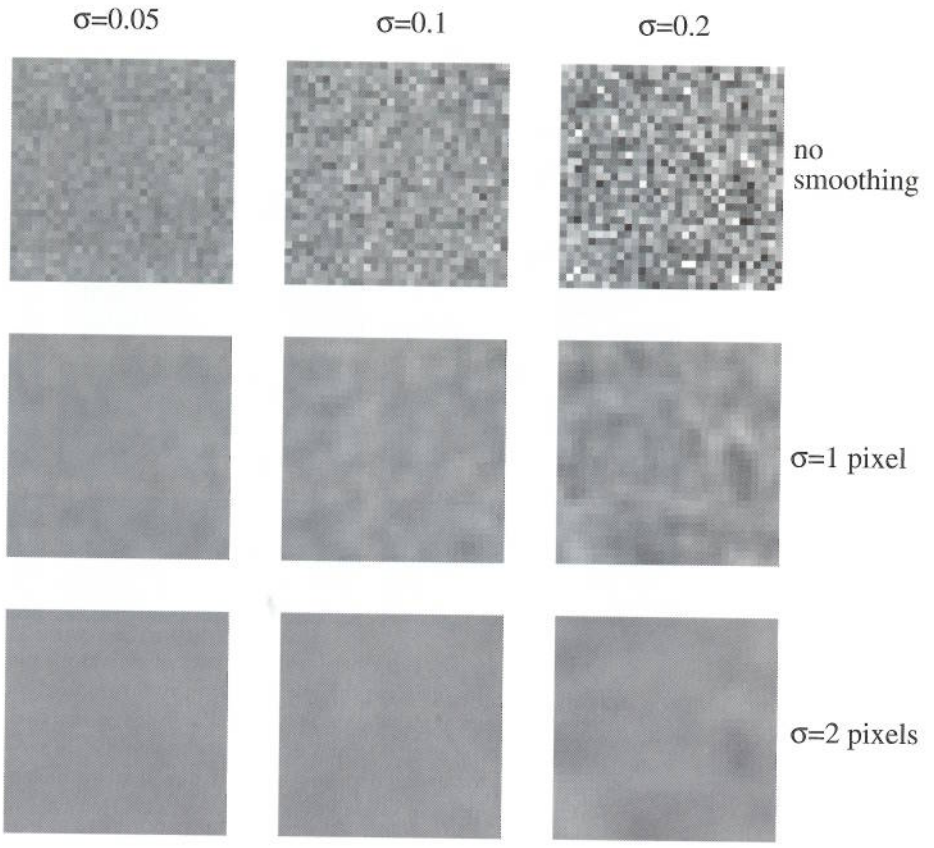

Figure 7.3 The **top row** shows images of a constant mid-gray level corrupted by additive Gaussian noise. In this noise model, each pixel has a zero-mean normal random variable added to it. The range of pixel values is from zero to one, so that the standard deviation of the noise in the first column 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

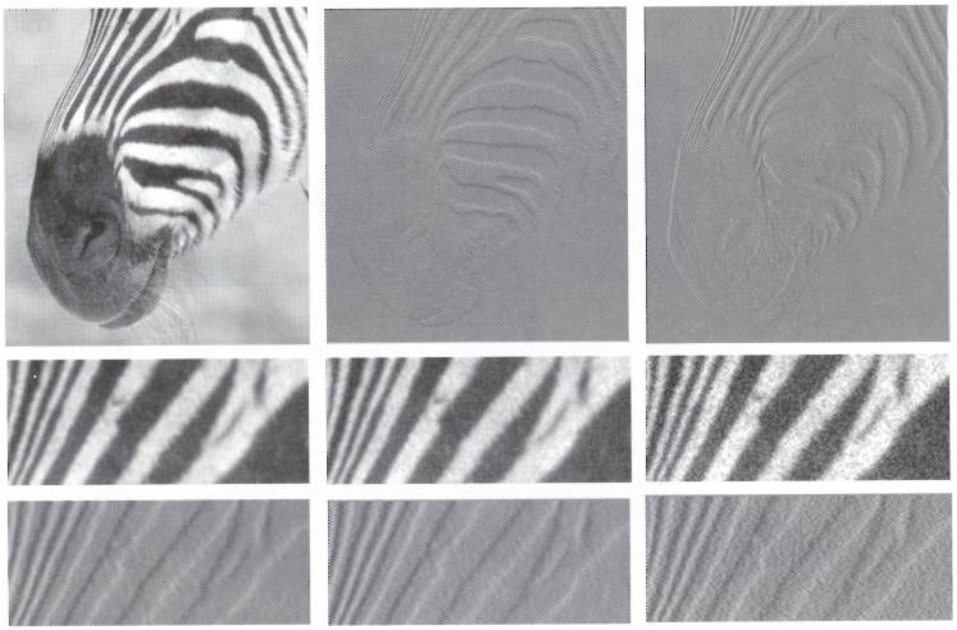

Figure 7.4 The **top row** shows estimates of derivatives obtained by finite differences. The image at the **left** shows a detail from a picture of a zebra. The **center** image shows the partial derivative in the y-direction— which responds strongly to horizontal stripes and weakly to vertical stripes—and 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 $\texttt{Shift}(\boldsymbol{f}, i)$ has, as its jth component, the $j - i$th component of $\boldsymbol{f}$. For example, $\texttt{Shift}(\boldsymbol{e}_0, 1)$ has a zero in the first component. Now we can write

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

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

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

Furthermore, because it is linear, we have

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

This means that

$$\begin{aligned} R(\boldsymbol{f}) &= R\left(\sum_i f_i \, \texttt{Shift}(\boldsymbol{e}_0, i)\right) \\ &= \sum_i R(f_i \, \texttt{Shift}(\boldsymbol{e}_0, i)) \\ &= \sum_i f_i R(\texttt{Shift}(\boldsymbol{e}_0, i)) \\ &= \sum_i f_i \, \texttt{Shift}(R(\boldsymbol{e}_0), i)). \end{aligned}$$

This means that, to obtain the system's response to any data vector, we need to know only its response to $\boldsymbol{e}_0$. This is usually called the system's *impulse response*. Assume that the impulse response can be written as $\boldsymbol{g}$. We have

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

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 $R(\boldsymbol{f})$, 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

$$\mathcal{E}_{00} = \begin{matrix} \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 1 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{matrix}$$

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 $\boldsymbol{f}$ 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. 7.3

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

$$R(h)(x, y) = \int\int g(x - x', y - y')h(x', y')\,dx\,dy$$
$$= (g * *h)(x, y),$$

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;