

**Figure 3.10** Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.

more interpretable. You can get a good sense of the range of operations possible by opening up any photo manipulation tool and trying out a variety of contrast, brightness, and color manipulation options, as shown in Figures 3.2 and 3.7.

Exercises 3.1, 3.5, and 3.6 have you implement some of these operations, in order to become familiar with basic image processing operators. More sophisticated techniques for tonal adjustment (Reinhard, Ward, Pattanaik *et al.* 2005; Bae, Paris, and Durand 2006) are described in the section on high dynamic range tone mapping (Section 10.2.1).

## 3.2 Linear filtering

Locally adaptive histogram equalization is an example of a *neighborhood operator* or *local operator*, which uses a collection of pixel values in the vicinity of a given pixel to determine its final output value (Figure 3.10). In addition to performing local tone adjustment, neighborhood operators can be used to *filter* images in order to add soft blur, sharpen details, accentuate edges, or remove noise (Figure 3.11b–d). In this section, we look at *linear* filtering operators, which involve weighted combinations of pixels in small neighborhoods. In Section 3.3, we look at non-linear operators such as morphological filters and distance transforms.

The most commonly used type of neighborhood operator is a *linear filter*, in which an output pixel’s value is determined as a weighted sum of input pixel values (Figure 3.10),

$$g(i, j) = \sum_{k, l} f(i + k, j + l) h(k, l). \quad (3.12)$$

The entries in the weight *kernel* or *mask*  $h(k, l)$  are often called the *filter coefficients*. The above *correlation* operator can be more compactly notated as

$$g = f \otimes h. \quad (3.13)$$



**Figure 3.11** Some neighborhood operations: (a) original image; (b) blurred; (c) sharpened; (d) smoothed with edge-preserving filter; (e) binary image; (f) dilated; (g) distance transform; (h) connected components. For the dilation and connected components, black (ink) pixels are assumed to be active, i.e., to have a value of 1 in Equations (3.41–3.45).

$$\begin{bmatrix} 72 & 88 & 62 & 52 & 37 \end{bmatrix} * \begin{bmatrix} 1/4 & 1/2 & 1/4 \end{bmatrix} \Leftrightarrow \frac{1}{4} \begin{bmatrix} 2 & 1 & . & . & . \\ 1 & 2 & 1 & . & . \\ . & 1 & 2 & 1 & . \\ . & . & 1 & 2 & 1 \\ . & . & . & 1 & 2 \end{bmatrix} \begin{bmatrix} 72 \\ 88 \\ 62 \\ 52 \\ 37 \end{bmatrix}$$

**Figure 3.12** One-dimensional signal convolution as a sparse matrix-vector multiply,  $g = Hf$ .

A common variant on this formula is

$$g(i, j) = \sum_{k, l} f(i - k, j - l) h(k, l) = \sum_{k, l} f(k, l) h(i - k, j - l), \quad (3.14)$$

where the sign of the offsets in  $f$  has been reversed. This is called the *convolution* operator,

$$g = f * h, \quad (3.15)$$

and  $h$  is then called the *impulse response function*.<sup>4</sup> The reason for this name is that the kernel function,  $h$ , convolved with an impulse signal,  $\delta(i, j)$  (an image that is 0 everywhere except at the origin) reproduces itself,  $h * \delta = h$ , whereas correlation produces the reflected signal. (Try this yourself to verify that it is so.)

In fact, Equation (3.14) can be interpreted as the superposition (addition) of shifted impulse response functions  $h(i - k, j - l)$  multiplied by the input pixel values  $f(k, l)$ . Convolution has additional nice properties, e.g., it is both commutative and associative. As well, the Fourier transform of two convolved images is the product of their individual Fourier transforms (Section 3.4).

Both correlation and convolution are *linear shift-invariant* (LSI) operators, which obey both the superposition principle (3.5),

$$h \circ (f_0 + f_1) = h \circ f_0 + h \circ f_1, \quad (3.16)$$

and the *shift invariance* principle,

$$g(i, j) = f(i + k, j + l) \Leftrightarrow (h \circ g)(i, j) = (h \circ f)(i + k, j + l), \quad (3.17)$$

which means that shifting a signal commutes with applying the operator ( $\circ$  stands for the LSI operator). Another way to think of shift invariance is that the operator “behaves the same everywhere”.

Occasionally, a shift-variant version of correlation or convolution may be used, e.g.,

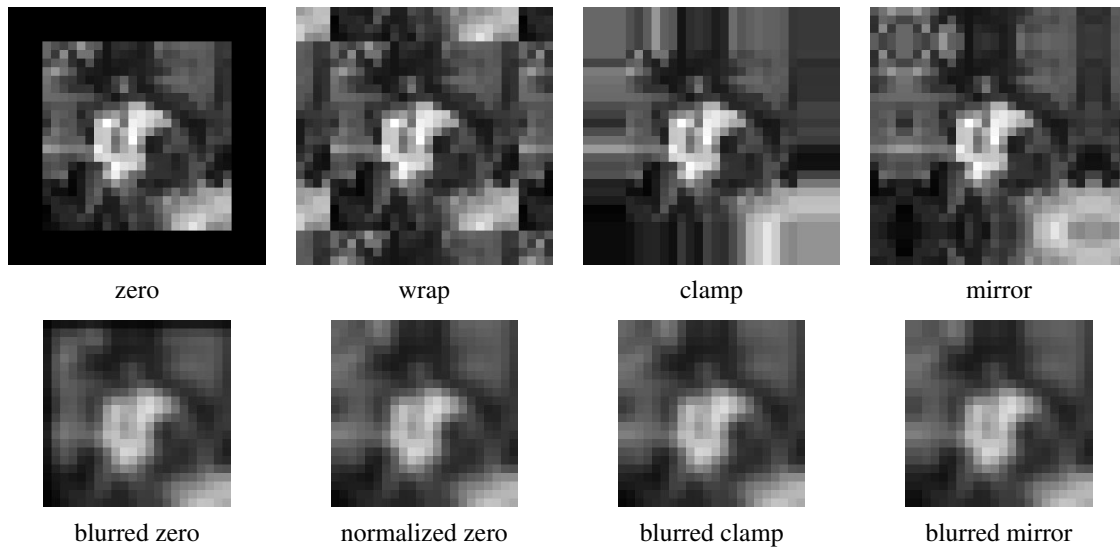
$$g(i, j) = \sum_{k, l} f(i - k, j - l) h(k, l; i, j), \quad (3.18)$$

where  $h(k, l; i, j)$  is the convolution kernel at pixel  $(i, j)$ . For example, such a spatially varying kernel can be used to model blur in an image due to variable depth-dependent defocus.

Correlation and convolution can both be written as a matrix-vector multiply, if we first convert the two-dimensional images  $f(i, j)$  and  $g(i, j)$  into raster-ordered vectors  $\mathbf{f}$  and  $\mathbf{g}$ ,

$$\mathbf{g} = H\mathbf{f}, \quad (3.19)$$

<sup>4</sup> The continuous version of convolution can be written as  $g(\mathbf{x}) = \int f(\mathbf{x} - \mathbf{u})h(\mathbf{u})d\mathbf{u}$ .



**Figure 3.13** Border padding (top row) and the results of blurring the padded image (bottom row). The normalized zero image is the result of dividing (normalizing) the blurred zero-padded RGBA image by its corresponding soft alpha value.

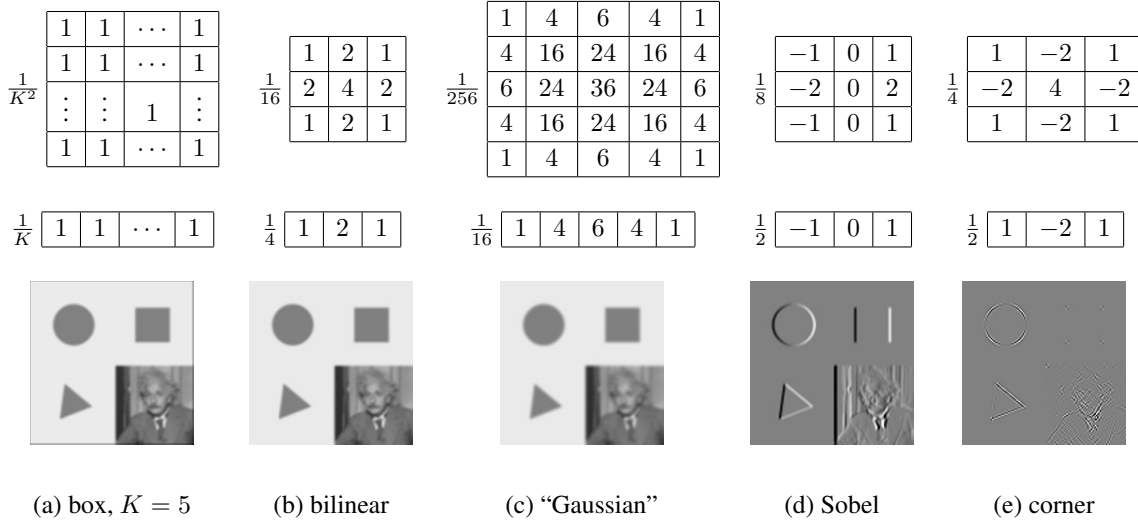
where the (sparse)  $H$  matrix contains the convolution kernels. Figure 3.12 shows how a one-dimensional convolution can be represented in matrix-vector form.

### Padding (border effects)

The astute reader will notice that the matrix multiply shown in Figure 3.12 suffers from *boundary effects*, i.e., the results of filtering the image in this form will lead to a *darkening* of the corner pixels. This is because the original image is effectively being padded with 0 values wherever the convolution kernel extends beyond the original image boundaries.

To compensate for this, a number of alternative *padding* or extension modes have been developed (Figure 3.13):

- *zero*: set all pixels outside the source image to 0 (a good choice for alpha-matted cutout images);
- *constant (border color)*: set all pixels outside the source image to a specified *border* value;
- *clamp (replicate or clamp to edge)*: repeat edge pixels indefinitely;
- *(cyclic) wrap (repeat or tile)*: loop “around” the image in a “toroidal” configuration;
- *mirror*: reflect pixels across the image edge;
- *extend*: extend the signal by subtracting the mirrored version of the signal from the edge pixel value.



**Figure 3.14** Separable linear filters: For each image (a)–(e), we show the 2D filter kernel (top), the corresponding horizontal 1D kernel (middle), and the filtered image (bottom). The filtered Sobel and corner images are signed, scaled up by  $2\times$  and  $4\times$ , respectively, and added to a gray offset before display.

In the computer graphics literature (Akenine-Möller and Haines 2002, p. 124), these mechanisms are known as the *wrapping mode* (OpenGL) or *texture addressing mode* (Direct3D). The formulas for each of these modes are left to the reader (Exercise 3.8).

Figure 3.13 shows the effects of padding an image with each of the above mechanisms and then blurring the resulting padded image. As you can see, zero padding darkens the edges, clamp (replication) padding propagates border values inward, mirror (reflection) padding preserves colors near the borders. Extension padding (not shown) keeps the border pixels fixed (during blur).

An alternative to padding is to blur the zero-padded RGBA image and to then divide the resulting image by its alpha value to remove the darkening effect. The results can be quite good, as seen in the normalized zero image in Figure 3.13.

### 3.2.1 Separable filtering

The process of performing a convolution requires  $K^2$  (multiply-add) operations per pixel, where  $K$  is the size (width or height) of the convolution kernel, e.g., the box filter in Figure 3.14a. In many cases, this operation can be significantly sped up by first performing a one-dimensional horizontal convolution followed by a one-dimensional vertical convolution (which requires a total of  $2K$  operations per pixel). A convolution kernel for which this is possible is said to be *separable*.

It is easy to show that the two-dimensional kernel  $K$  corresponding to successive convolution with a horizontal kernel  $h$  and a vertical kernel  $v$  is the *outer product* of the two kernels,

$$K = v h^T \quad (3.20)$$

(see Figure 3.14 for some examples). Because of the increased efficiency, the design of

convolution kernels for computer vision applications is often influenced by their separability.

How can we tell if a given kernel  $K$  is indeed separable? This can often be done by inspection or by looking at the analytic form of the kernel (Freeman and Adelson 1991). A more direct method is to treat the 2D kernel as a 2D matrix  $K$  and to take its singular value decomposition (SVD),

$$K = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (3.21)$$

(see Appendix A.1.1 for the definition of the SVD). If only the first singular value  $\sigma_0$  is non-zero, the kernel is separable and  $\sqrt{\sigma_0} \mathbf{u}_0$  and  $\sqrt{\sigma_0} \mathbf{v}_0^T$  provide the vertical and horizontal kernels (Perona 1995). For example, the Laplacian of Gaussian kernel (3.26 and 4.23) can be implemented as the sum of two separable filters (4.24) (Wiejak, Buxton, and Buxton 1985).

What if your kernel is not separable and yet you still want a faster way to implement it? Perona (1995), who first made the link between kernel separability and SVD, suggests using more terms in the (3.21) series, i.e., summing up a number of separable convolutions. Whether this is worth doing or not depends on the relative sizes of  $K$  and the number of significant singular values, as well as other considerations, such as cache coherency and memory locality.

### 3.2.2 Examples of linear filtering

Now that we have described the process for performing linear filtering, let us examine a number of frequently used filters.

The simplest filter to implement is the *moving average* or *box* filter, which simply averages the pixel values in a  $K \times K$  window. This is equivalent to convolving the image with a kernel of all ones and then scaling (Figure 3.14a). For large kernels, a more efficient implementation is to slide a moving window across each scanline (in a separable filter) while adding the newest pixel and subtracting the oldest pixel from the running sum. This is related to the concept of *summed area tables*, which we describe shortly.

A smoother image can be obtained by separably convolving the image with a piecewise linear “tent” function (also known as a *Bartlett* filter). Figure 3.14b shows a  $3 \times 3$  version of this filter, which is called the *bilinear* kernel, since it is the outer product of two linear (first-order) splines (see Section 3.5.2).

Convolving the linear tent function with itself yields the cubic approximating spline, which is called the “Gaussian” kernel (Figure 3.14c) in Burt and Adelson’s (1983a) *Laplacian pyramid* representation (Section 3.5). Note that approximate Gaussian kernels can also be obtained by iterated convolution with box filters (Wells 1986). In applications where the filters really need to be rotationally symmetric, carefully tuned versions of sampled Gaussians should be used (Freeman and Adelson 1991) (Exercise 3.10).

The kernels we just discussed are all examples of blurring (smoothing) or *low-pass* kernels (since they pass through the lower frequencies while attenuating higher frequencies). How good are they at doing this? In Section 3.4, we use frequency-space Fourier analysis to examine the exact frequency response of these filters. We also introduce the *sinc*  $((\sin x)/x)$  filter, which performs *ideal* low-pass filtering.

In practice, smoothing kernels are often used to reduce high-frequency noise. We have much more to say about using variants on smoothing to remove noise later (see Sections 3.3.1, 3.4, and 3.7).

Surprisingly, smoothing kernels can also be used to *sharpen* images using a process called *unsharp masking*. Since blurring the image reduces high frequencies, adding some of the difference between the original and the blurred image makes it sharper,

$$g_{\text{sharp}} = f + \gamma(f - h_{\text{blur}} * f). \quad (3.22)$$

In fact, before the advent of digital photography, this was the standard way to sharpen images in the darkroom: create a blurred (“positive”) negative from the original negative by mis-focusing, then overlay the two negatives before printing the final image, which corresponds to

$$g_{\text{unsharp}} = f(1 - \gamma h_{\text{blur}} * f). \quad (3.23)$$

This is no longer a linear filter but it still works well.

Linear filtering can also be used as a pre-processing stage to edge extraction (Section 4.2) and interest point detection (Section 4.1) algorithms. Figure 3.14d shows a simple  $3 \times 3$  edge extractor called the Sobel operator, which is a separable combination of a horizontal *central difference* (so called because the horizontal derivative is centered on the pixel) and a vertical tent filter (to smooth the results). As you can see in the image below the kernel, this filter effectively emphasizes horizontal edges.

The simple corner detector (Figure 3.14e) looks for simultaneous horizontal and vertical second derivatives. As you can see however, it responds not only to the corners of the square, but also along diagonal edges. Better corner detectors, or at least interest point detectors that are more rotationally invariant, are described in Section 4.1.

### 3.2.3 Band-pass and steerable filters

The Sobel and corner operators are simple examples of band-pass and oriented filters. More sophisticated kernels can be created by first smoothing the image with a (unit area) Gaussian filter,

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \quad (3.24)$$

and then taking the first or second derivatives (Marr 1982; Witkin 1983; Freeman and Adelson 1991). Such filters are known collectively as *band-pass filters*, since they filter out both low and high frequencies.

The (undirected) second derivative of a two-dimensional image,

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}, \quad (3.25)$$

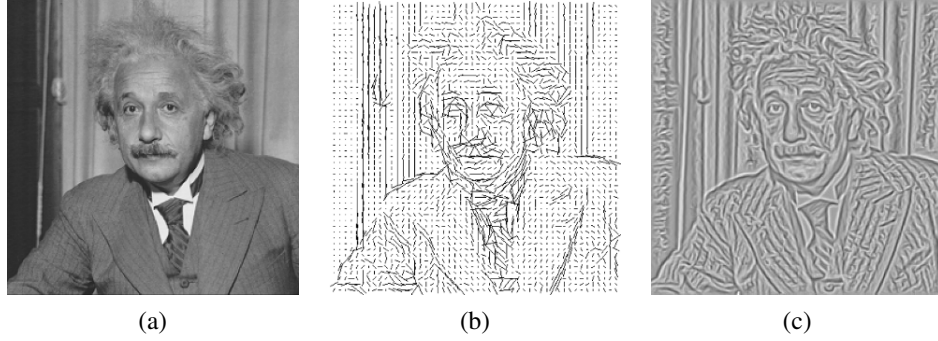
is known as the *Laplacian* operator. Blurring an image with a Gaussian and then taking its Laplacian is equivalent to convolving directly with the *Laplacian of Gaussian* (LoG) filter,

$$\nabla^2 G(x, y; \sigma) = \left( \frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2} \right) G(x, y; \sigma), \quad (3.26)$$

which has certain nice *scale-space properties* (Witkin 1983; Witkin, Terzopoulos, and Kass 1986). The five-point Laplacian is just a compact approximation to this more sophisticated filter.

Likewise, the Sobel operator is a simple approximation to a *directional* or *oriented* filter, which can be obtained by smoothing with a Gaussian (or some other filter) and then taking a





**Figure 3.15** Second-order steerable filter (Freeman 1992) © 1992 IEEE: (a) original image of Einstein; (b) orientation map computed from the second-order oriented energy; (c) original image with oriented structures enhanced.

directional derivative  $\nabla_{\hat{\mathbf{u}}} = \frac{\partial}{\partial \hat{\mathbf{u}}}$ , which is obtained by taking the dot product between the gradient field  $\nabla$  and a unit direction  $\hat{\mathbf{u}} = (\cos \theta, \sin \theta)$ ,

$$\hat{\mathbf{u}} \cdot \nabla (G * f) = \nabla_{\hat{\mathbf{u}}} (G * f) = (\nabla_{\hat{\mathbf{u}}} G) * f. \quad (3.27)$$

The smoothed directional derivative filter,

$$G_{\hat{\mathbf{u}}} = uG_x + vG_y = u \frac{\partial G}{\partial x} + v \frac{\partial G}{\partial y}, \quad (3.28)$$

where  $\hat{\mathbf{u}} = (u, v)$ , is an example of a *steerable* filter, since the value of an image convolved with  $G_{\hat{\mathbf{u}}}$  can be computed by first convolving with the pair of filters  $(G_x, G_y)$  and then *steering* the filter (potentially locally) by multiplying this gradient field with a unit vector  $\hat{\mathbf{u}}$  (Freeman and Adelson 1991). The advantage of this approach is that a whole *family* of filters can be evaluated with very little cost.

How about steering a directional second derivative filter  $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G_{\hat{\mathbf{u}}}$ , which is the result of taking a (smoothed) directional derivative and then taking the directional derivative again? For example,  $G_{xx}$  is the second directional derivative in the  $x$  direction.

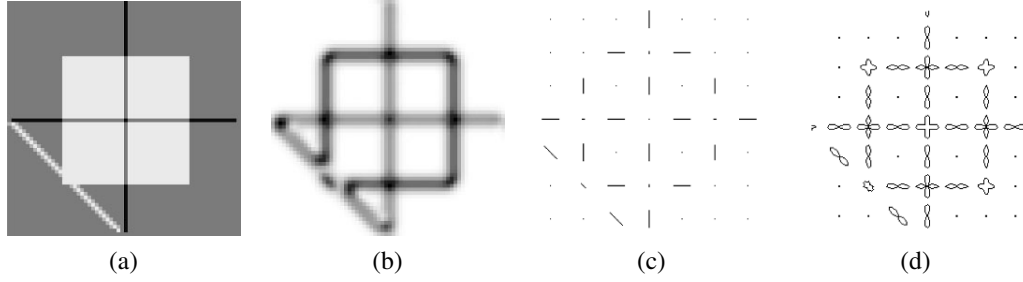
At first glance, it would appear that the steering trick will not work, since for every direction  $\hat{\mathbf{u}}$ , we need to compute a different first directional derivative. Somewhat surprisingly, Freeman and Adelson (1991) showed that, for directional Gaussian derivatives, it is possible to steer *any* order of derivative with a relatively small number of basis functions. For example, only three basis functions are required for the second-order directional derivative,

$$G_{\hat{\mathbf{u}}\hat{\mathbf{u}}} = u^2 G_{xx} + 2uv G_{xy} + v^2 G_{yy}. \quad (3.29)$$

Furthermore, each of the basis filters, while not itself necessarily separable, can be computed using a linear combination of a small number of separable filters (Freeman and Adelson 1991).

This remarkable result makes it possible to construct directional derivative filters of increasingly greater *directional selectivity*, i.e., filters that only respond to edges that have strong local consistency in orientation (Figure 3.15). Furthermore, higher order steerable





**Figure 3.16** Fourth-order steerable filter (Freeman and Adelson 1991) © 1991 IEEE: (a) test image containing bars (lines) and step edges at different orientations; (b) average oriented energy; (c) dominant orientation; (d) oriented energy as a function of angle (polar plot).

filters can respond to potentially more than a single edge orientation at a given location, and they can respond to both *bar* edges (thin lines) and the classic step edges (Figure 3.16). In order to do this, however, full *Hilbert transform pairs* need to be used for second-order and higher filters, as described in (Freeman and Adelson 1991).

Steerable filters are often used to construct both feature descriptors (Section 4.1.3) and edge detectors (Section 4.2). While the filters developed by Freeman and Adelson (1991) are best suited for detecting linear (edge-like) structures, more recent work by Koethe (2003) shows how a combined  $2 \times 2$  boundary tensor can be used to encode both edge and junction (“corner”) features. Exercise 3.12 has you implement such steerable filters and apply them to finding both edge and corner features.

### Summed area table (integral image)

If an image is going to be repeatedly convolved with different box filters (and especially filters of different sizes at different locations), you can precompute the *summed area table* (Crow 1984), which is just the running sum of all the pixel values from the origin,

$$s(i, j) = \sum_{k=0}^i \sum_{l=0}^j f(k, l). \quad (3.30)$$

This can be efficiently computed using a recursive (raster-scan) algorithm,

$$s(i, j) = s(i-1, j) + s(i, j-1) - s(i-1, j-1) + f(i, j). \quad (3.31)$$

The image  $s(i, j)$  is also often called an *integral image* (see Figure 3.17) and can actually be computed using only two additions per pixel if separate row sums are used (Viola and Jones 2004). To find the summed area (integral) inside a rectangle  $[i_0, i_1] \times [j_0, j_1]$ , we simply combine four samples from the summed area table,

$$S(i_0 \dots i_1, j_0 \dots j_1) = \sum_{i=i_0}^{i_1} \sum_{j=j_0}^{j_1} s(i_1, j_1) - s(i_1, j_0 - 1) - s(i_0 - 1, j_1) + s(i_0 - 1, j_0 - 1). \quad (3.32)$$

A potential disadvantage of summed area tables is that they require  $\log M + \log N$  extra bits in the accumulation image compared to the original image, where  $M$  and  $N$  are the image

3	2	7	2	3
1	5	1	3	4
5	1	<b>3</b>	5	1
4	3	2	1	6
2	4	1	4	8

(a)  $S = 24$

3	5	12	14	17
4	<i>11</i>	<b>19</b>	24	31
9	<b>17</b>	<b>28</b>	38	46
13	24	37	48	62
15	30	44	59	81

(b)  $s = 28$

<b>3</b>	5	12	<b>14</b>	17
4	11	19	24	31
9	17	28	38	46
<b>13</b>	24	37	<b>48</b>	62
15	30	44	59	81

(c)  $S = 24$

**Figure 3.17** Summed area tables: (a) original image; (b) summed area table; (c) computation of area sum. Each value in the summed area table  $s(i, j)$  (red) is computed recursively from its three adjacent (blue) neighbors (3.31). Area sums  $S$  (green) are computed by combining the four values at the rectangle corners (purple) (3.32). Positive values are shown in **bold** and negative values in *italics*.

width and height. Extensions of summed area tables can also be used to approximate other convolution kernels (Wolberg (1990, Section 6.5.2) contains a review).

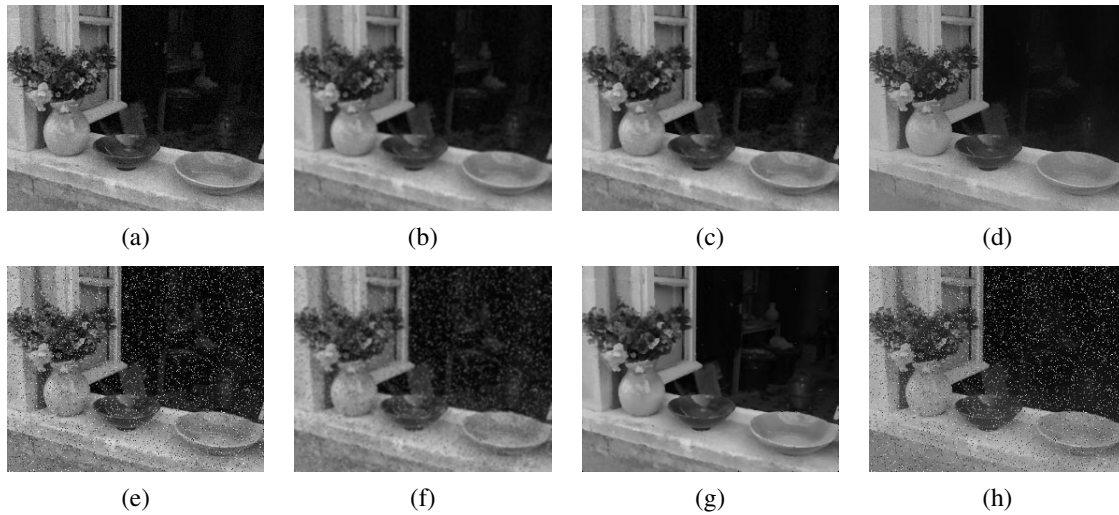
In computer vision, summed area tables have been used in face detection (Viola and Jones 2004) to compute simple multi-scale low-level features. Such features, which consist of adjacent rectangles of positive and negative values, are also known as *boxlets* (Simard, Bottou, Haffner *et al.* 1998). In principle, summed area tables could also be used to compute the sums in the sum of squared differences (SSD) stereo and motion algorithms (Section 11.4). In practice, separable moving average filters are usually preferred (Kanade, Yoshida, Oda *et al.* 1996), unless many different window shapes and sizes are being considered (Veksler 2003).

### Recursive filtering

The incremental formula (3.31) for the summed area is an example of a *recursive filter*, i.e., one whose values depends on previous filter outputs. In the signal processing literature, such filters are known as *infinite impulse response* (IIR), since the output of the filter to an impulse (single non-zero value) goes on forever. For example, for a summed area table, an impulse generates an infinite rectangle of 1s below and to the right of the impulse. The filters we have previously studied in this chapter, which involve the image with a finite extent kernel, are known as *finite impulse response* (FIR).

Two-dimensional IIR filters and recursive formulas are sometimes used to compute quantities that involve large area interactions, such as two-dimensional distance functions (Section 3.3.3) and connected components (Section 3.3.4).

More commonly, however, IIR filters are used inside one-dimensional separable filtering stages to compute large-extent smoothing kernels, such as efficient approximations to Gaussians and edge filters (Deriche 1990; Nielsen, Florack, and Deriche 1997). Pyramid-based algorithms (Section 3.5) can also be used to perform such large-area smoothing computations.



**Figure 3.18** Median and bilateral filtering: (a) original image with Gaussian noise; (b) Gaussian filtered; (c) median filtered; (d) bilaterally filtered; (e) original image with shot noise; (f) Gaussian filtered; (g) median filtered; (h) bilaterally filtered. Note that the bilateral filter fails to remove the shot noise because the noisy pixels are too different from their neighbors.

### 3.3 More neighborhood operators

As we have just seen, linear filters can perform a wide variety of image transformations. However non-linear filters, such as edge-preserving median or bilateral filters, can sometimes perform even better. Other examples of neighborhood operators include *morphological* operators that operate on binary images, as well as *semi-global* operators that compute *distance transforms* and find *connected components* in binary images (Figure 3.11f–h).

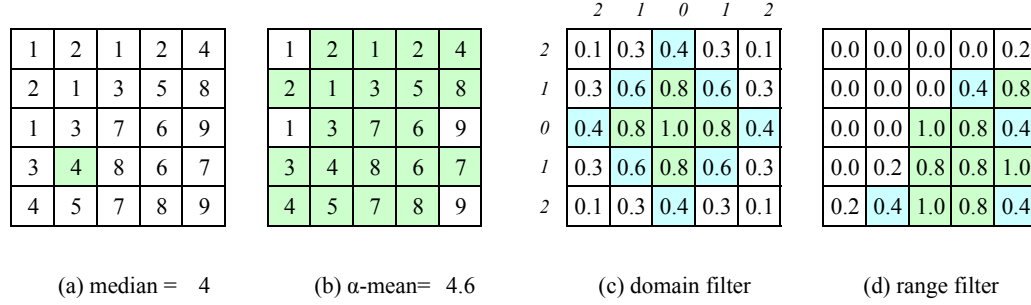
#### 3.3.1 Non-linear filtering

The filters we have looked at so far have all been *linear*, i.e., their response to a sum of two signals is the same as the sum of the individual responses. This is equivalent to saying that each output pixel is a weighted summation of some number of input pixels (3.19). Linear filters are easier to compose and are amenable to frequency response analysis (Section 3.4).

In many cases, however, better performance can be obtained by using a *non-linear* combination of neighboring pixels. Consider for example the image in Figure 3.18e, where the noise, rather than being Gaussian, is *shot noise*, i.e., it occasionally has very large values. In this case, regular blurring with a Gaussian filter fails to remove the noisy pixels and instead turns them into softer (but still visible) spots (Figure 3.18f).

##### Median filtering

A better filter to use in this case is the *median* filter, which selects the median value from each pixel's neighborhood (Figure 3.19a). Median values can be computed in expected linear time using a randomized select algorithm (Cormen 2001) and incremental variants have also been



**Figure 3.19** Median and bilateral filtering: (a) median pixel (green); (b) selected  $\alpha$ -trimmed mean pixels; (c) domain filter (numbers along edge are pixel distances); (d) range filter.

developed by Tomasi and Manduchi (1998) and Bovik (2000, Section 3.2). Since the shot noise value usually lies well outside the true values in the neighborhood, the median filter is able to filter away such bad pixels (Figure 3.18c).

One downside of the median filter, in addition to its moderate computational cost, is that since it selects only one input pixel value to replace each output pixel, it is not as *efficient* at averaging away regular Gaussian noise (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986; Stewart 1999). A better choice may be the  $\alpha$ -trimmed mean (Lee and Redner 1990) (Crane 1997, p. 109), which averages together all of the pixels except for the  $\alpha$  fraction that are the smallest and the largest (Figure 3.19b).

Another possibility is to compute a *weighted median*, in which each pixel is used a number of times depending on its distance from the center. This turns out to be equivalent to minimizing the weighted objective function

$$\sum_{k,l} w(k,l) |f(i+k, j+l) - g(i,j)|^p, \quad (3.33)$$

where  $g(i,j)$  is the desired output value and  $p = 1$  for the weighted median. The value  $p = 2$  is the usual *weighted mean*, which is equivalent to correlation (3.12) after normalizing by the sum of the weights (Bovik 2000, Section 3.2) (Haralick and Shapiro 1992, Section 7.2.6). The weighted mean also has deep connections to other methods in robust statistics (see Appendix B.3), such as influence functions (Huber 1981; Hampel, Ronchetti, Rousseeuw *et al.* 1986).

Non-linear smoothing has another, perhaps even more important property, especially since shot noise is rare in today's cameras. Such filtering is more *edge preserving*, i.e., it has less tendency to soften edges while filtering away high-frequency noise.

Consider the noisy image in Figure 3.18a. In order to remove most of the noise, the Gaussian filter is forced to smooth away high-frequency detail, which is most noticeable near strong edges. Median filtering does better but, as mentioned before, does not do as good a job at smoothing away from discontinuities. See (Tomasi and Manduchi 1998) for some additional references to edge-preserving smoothing techniques.

While we could try to use the  $\alpha$ -trimmed mean or weighted median, these techniques still have a tendency to round sharp corners, since the majority of pixels in the smoothing area come from the background distribution.

### Bilateral filtering

What if we were to combine the idea of a weighted filter kernel with a better version of outlier rejection? What if instead of rejecting a fixed percentage  $\alpha$ , we simply reject (in a soft way) pixels whose *values* differ too much from the central pixel value? This is the essential idea in *bilateral filtering*, which was first popularized in the computer vision community by Tomasi and Manduchi (1998). Chen, Paris, and Durand (2007) and Paris, Kornprobst, Tumblin *et al.* (2008) cite similar earlier work (Aurich and Weule 1995; Smith and Brady 1997) as well as the wealth of subsequent applications in computer vision and computational photography.

In the bilateral filter, the output pixel value depends on a weighted combination of neighboring pixel values

$$g(i, j) = \frac{\sum_{k,l} f(k, l) w(i, j, k, l)}{\sum_{k,l} w(i, j, k, l)}. \quad (3.34)$$

The weighting coefficient  $w(i, j, k, l)$  depends on the product of a *domain kernel* (Figure 3.19c),

$$d(i, j, k, l) = \exp \left( -\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} \right), \quad (3.35)$$

and a data-dependent *range kernel* (Figure 3.19d),

$$r(i, j, k, l) = \exp \left( -\frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2} \right). \quad (3.36)$$

When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i, j, k, l) = \exp \left( -\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2} \right). \quad (3.37)$$

Figure 3.20 shows an example of the bilateral filtering of a noisy step edge. Note how the domain kernel is the usual Gaussian, the range kernel measures appearance (intensity) similarity to the center pixel, and the bilateral filter kernel is a product of these two.

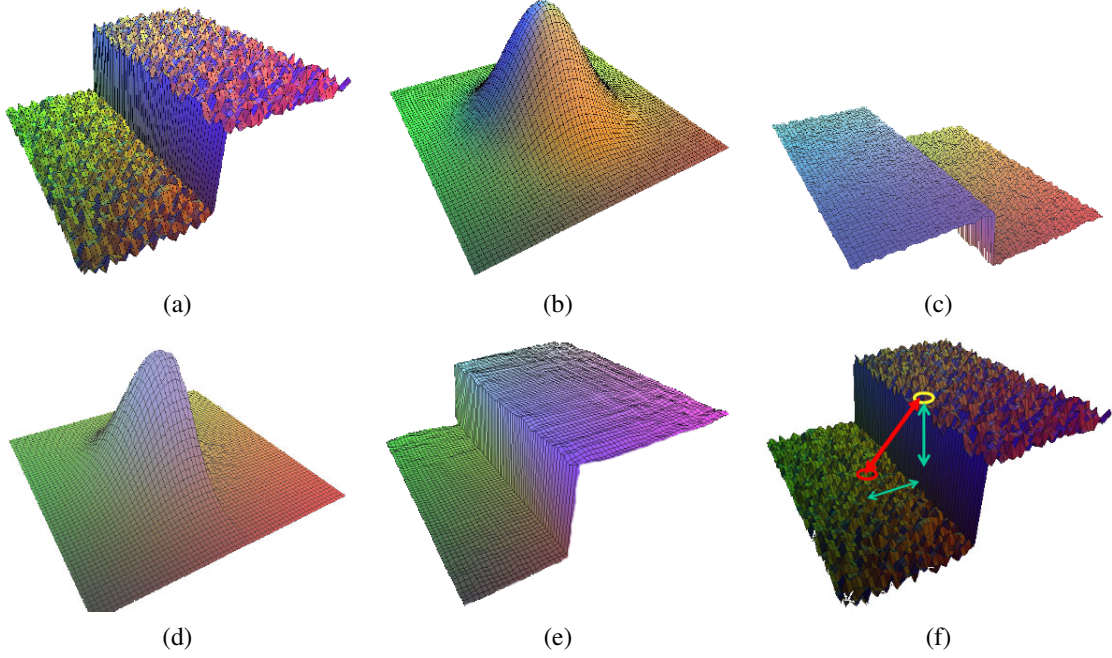
Notice that the range filter (3.36) uses the *vector distance* between the center and the neighboring pixel. This is important in color images, since an edge in any *one* of the color bands signals a change in material and hence the need to downweight a pixel's influence.<sup>5</sup>

Since bilateral filtering is quite slow compared to regular separable filtering, a number of acceleration techniques have been developed (Durand and Dorsey 2002; Paris and Durand 2006; Chen, Paris, and Durand 2007; Paris, Kornprobst, Tumblin *et al.* 2008). Unfortunately, these techniques tend to use more memory than regular filtering and are hence not directly applicable to filtering full-color images.

### Iterated adaptive smoothing and anisotropic diffusion

Bilateral (and other) filters can also be applied in an iterative fashion, especially if an appearance more like a “cartoon” is desired (Tomasi and Manduchi 1998). When iterated filtering is applied, a much smaller neighborhood can often be used.

<sup>5</sup> Tomasi and Manduchi (1998) show that using the vector distance (as opposed to filtering each color band separately) reduces color fringing effects. They also recommend taking the color difference in the more perceptually uniform CIELAB color space (see Section 2.3.2).



**Figure 3.20** Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.

Consider, for example, using only the four nearest neighbors, i.e., restricting  $|k - i| + |l - j| \leq 1$  in (3.34). Observe that

$$d(i, j, k, l) = \exp \left( -\frac{(i - k)^2 + (j - l)^2}{2\sigma_d^2} \right) \quad (3.38)$$

$$= \begin{cases} 1, & |k - i| + |l - j| = 0, \\ \lambda = e^{-1/2\sigma_d^2}, & |k - i| + |l - j| = 1. \end{cases} \quad (3.39)$$

We can thus re-write (3.34) as

$$\begin{aligned} f^{(t+1)}(i, j) &= \frac{f^{(t)}(i, j) + \eta \sum_{k,l} f^{(t)}(k, l) r(i, j, k, l)}{1 + \eta \sum_{k,l} r(i, j, k, l)} \\ &= f^{(t)}(i, j) + \frac{\eta}{1 + \eta R} \sum_{k,l} r(i, j, k, l) [f^{(t)}(k, l) - f^{(t)}(i, j)], \end{aligned} \quad (3.40)$$

where  $R = \sum_{(k,l)} r(i, j, k, l)$ ,  $(k, l)$  are the  $\mathcal{N}_4$  neighbors of  $(i, j)$ , and we have made the iterative nature of the filtering explicit.

As Barash (2002) notes, (3.40) is the same as the discrete *anisotropic diffusion* equation first proposed by Perona and Malik (1990b).<sup>6</sup> Since its original introduction, anisotropic diffusion has been extended and applied to a wide range of problems (Nielsen, Florack, and Deriche 1997; Black, Sapiro, Marimont *et al.* 1998; Weickert, ter Haar Romeny, and Viergever

<sup>6</sup> The  $1/(1 + \eta R)$  factor is not present in anisotropic diffusion but becomes negligible as  $\eta \rightarrow 0$ .



1998; Weickert 1998). It has also been shown to be closely related to other *adaptive smoothing* techniques (Saint-Marc, Chen, and Medioni 1991; Barash 2002; Barash and Comaniciu 2004) as well as Bayesian regularization with a non-linear smoothness term that can be derived from image statistics (Scharr, Black, and Haussecker 2003).

In its general form, the range kernel  $r(i, j, k, l) = r(\|f(i, j) - f(k, l)\|)$ , which is usually called the *gain* or *edge-stopping* function, or diffusion coefficient, can be any monotonically increasing function with  $r'(x) \rightarrow 0$  as  $x \rightarrow \infty$ . Black, Sapiro, Marimont *et al.* (1998) show how anisotropic diffusion is equivalent to minimizing a robust penalty function on the image gradients, which we discuss in Sections 3.7.1 and 3.7.2). Scharr, Black, and Haussecker (2003) show how the edge-stopping function can be derived in a principled manner from local image statistics. They also extend the diffusion neighborhood from  $\mathcal{N}_4$  to  $\mathcal{N}_8$ , which allows them to create a diffusion operator that is both rotationally invariant and incorporates information about the eigenvalues of the local structure tensor.

Note that, without a bias term towards the original image, anisotropic diffusion and iterative adaptive smoothing converge to a constant image. Unless a small number of iterations is used (e.g., for speed), it is usually preferable to formulate the smoothing problem as a joint minimization of a smoothness term and a data fidelity term, as discussed in Sections 3.7.1 and 3.7.2 and by Scharr, Black, and Haussecker (2003), which introduce such a bias in a principled manner.

### 3.3.2 Morphology

While non-linear filters are often used to enhance grayscale and color images, they are also used extensively to process binary images. Such images often occur after a *thresholding* operation,

$$\theta(f, t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else,} \end{cases} \quad (3.41)$$

e.g., converting a scanned grayscale document into a binary image for further processing such as *optical character recognition*.

The most common binary image operations are called *morphological operations*, since they change the *shape* of the underlying binary objects (Ritter and Wilson 2000, Chapter 7). To perform such an operation, we first convolve the binary image with a binary *structuring element* and then select a binary output value depending on the thresholded result of the convolution. (This is not the usual way in which these operations are described, but I find it a nice simple way to unify the processes.) The structuring element can be any shape, from a simple  $3 \times 3$  box filter, to more complicated disc structures. It can even correspond to a particular shape that is being sought for in the image.

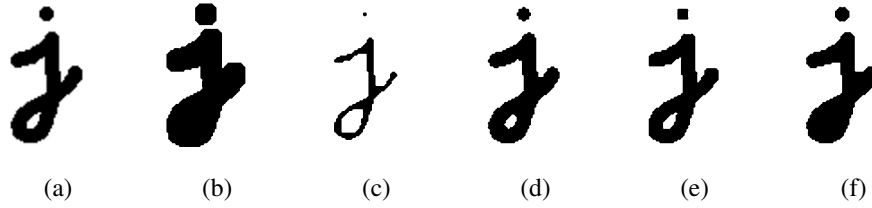
Figure 3.21 shows a close-up of the convolution of a binary image  $f$  with a  $3 \times 3$  structuring element  $s$  and the resulting images for the operations described below. Let

$$c = f \otimes s \quad (3.42)$$

be the integer-valued *count* of the number of 1s inside each structuring element as it is scanned over the image and  $S$  be the size of the structuring element (number of pixels). The standard operations used in binary morphology include:

- **dilation:**  $\text{dilate}(f, s) = \theta(c, 1)$ ;





**Figure 3.21** Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing. The structuring element for all examples is a  $5 \times 5$  square. The effects of majority are a subtle rounding of sharp corners. Opening fails to eliminate the dot, since it is not wide enough.

- **erosion:**  $\text{erode}(f, s) = \theta(c, S)$ ;
- **majority:**  $\text{maj}(f, s) = \theta(c, S/2)$ ;
- **opening:**  $\text{open}(f, s) = \text{dilate}(\text{erode}(f, s), s)$ ;
- **closing:**  $\text{close}(f, s) = \text{erode}(\text{dilate}(f, s), s)$ .

As we can see from Figure 3.21, dilation grows (thickens) objects consisting of 1s, while erosion shrinks (thins) them. The opening and closing operations tend to leave large regions and smooth boundaries unaffected, while removing small objects or holes and smoothing boundaries.

While we will not use mathematical morphology much in the rest of this book, it is a handy tool to have around whenever you need to clean up some thresholded images. You can find additional details on morphology in other textbooks on computer vision and image processing (Haralick and Shapiro 1992, Section 5.2) (Bovik 2000, Section 2.2) (Ritter and Wilson 2000, Section 7) as well as articles and books specifically on this topic (Serra 1982; Serra and Vincent 1992; Yuille, Vincent, and Geiger 1992; Soille 2006).

### 3.3.3 Distance transforms

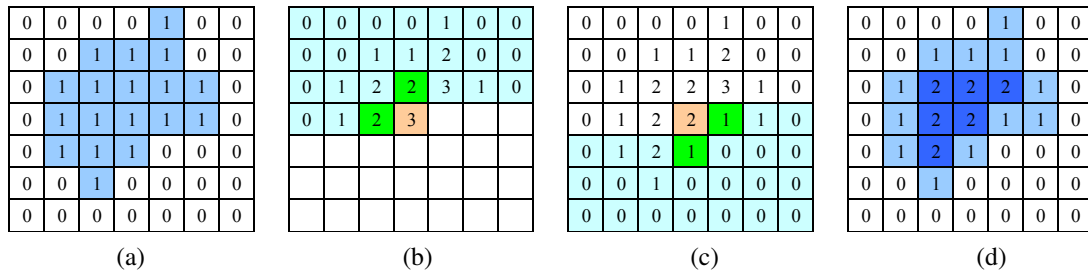
The distance transform is useful in quickly precomputing the distance to a curve or set of points using a two-pass raster algorithm (Rosenfeld and Pfaltz 1966; Danielsson 1980; Borgefors 1986; Paglieroni 1992; Breu, Gil, Kirkpatrick *et al.* 1995; Felzenszwalb and Huttenlocher 2004a; Fabbri, Costa, Torelli *et al.* 2008). It has many applications, including level sets (Section 5.1.4), fast *chamfer matching* (binary image alignment) (Huttenlocher, Klanderman, and Rucklidge 1993), feathering in image stitching and blending (Section 9.3.2), and nearest point alignment (Section 12.2.1).

The distance transform  $D(i, j)$  of a binary image  $b(i, j)$  is defined as follows. Let  $d(k, l)$  be some *distance metric* between pixel offsets. Two commonly used metrics include the *city block* or *Manhattan* distance

$$d_1(k, l) = |k| + |l| \quad (3.43)$$

and the *Euclidean* distance

$$d_2(k, l) = \sqrt{k^2 + l^2}. \quad (3.44)$$



**Figure 3.22** City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange value; (d) final distance transform.

The distance transform is then defined as

$$D(i, j) = \min_{k, l: b(k, l) = 0} d(i - k, j - l), \quad (3.45)$$

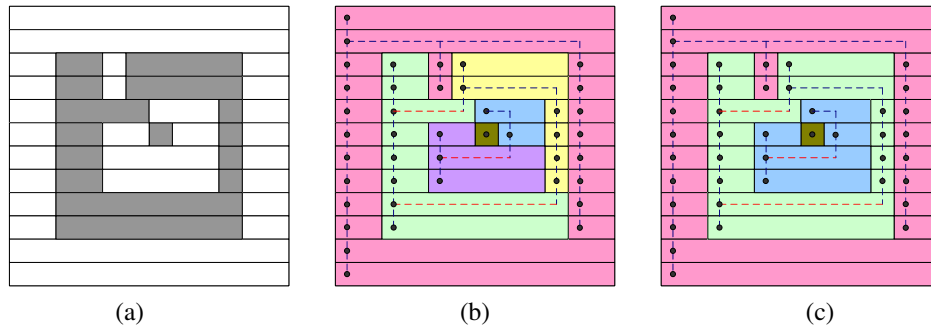
i.e., it is the distance to the *nearest* background pixel whose value is 0.

The  $D_1$  city block distance transform can be efficiently computed using a forward and backward pass of a simple raster-scan algorithm, as shown in Figure 3.22. During the forward pass, each non-zero pixel in  $b$  is replaced by the minimum of 1 + the distance of its north or west neighbor. During the backward pass, the same occurs, except that the minimum is both over the current value  $D$  and 1 + the distance of the south and east neighbors (Figure 3.22).

Efficiently computing the Euclidean distance transform is more complicated. Here, just keeping the minimum scalar distance to the boundary during the two passes is not sufficient. Instead, a *vector-valued* distance consisting of both the  $x$  and  $y$  coordinates of the distance to the boundary must be kept and compared using the squared distance (hypotenuse) rule. As well, larger search regions need to be used to obtain reasonable results. Rather than explaining the algorithm (Danielsson 1980; Borgefors 1986) in more detail, we leave it as an exercise for the motivated reader (Exercise 3.13).

Figure 3.11g shows a distance transform computed from a binary image. Notice how the values grow away from the black (ink) regions and form ridges in the white area of the original image. Because of this linear growth from the starting boundary pixels, the distance transform is also sometimes known as the *grassfire transform*, since it describes the time at which a fire starting inside the black region would consume any given pixel, or a *chamfer*, because it resembles similar shapes used in woodworking and industrial design. The ridges in the distance transform become the *skeleton* (or *medial axis transform* (MAT)) of the region where the transform is computed, and consist of pixels that are of equal distance to two (or more) boundaries (Tek and Kimia 2003; Sebastian and Kimia 2005).

A useful extension of the basic distance transform is the *signed distance transform*, which computes distances to boundary pixels for *all* the pixels (Lavallée and Szeliski 1995). The simplest way to create this is to compute the distance transforms for both the original binary image and its complement and to negate one of them before combining. Because such distance fields tend to be smooth, it is possible to store them more compactly (with minimal loss in *relative* accuracy) using a spline defined over a quadtree or octree data structure



**Figure 3.23** Connected component computation: (a) original grayscale image; (b) horizontal runs (nodes) connected by vertical (graph) edges (dashed blue)—runs are pseudocolored with unique colors inherited from parent nodes; (c) re-coloring after merging adjacent segments.

(Lavallée and Szeliski 1995; Szeliski and Lavallée 1996; Frisken, Perry, Rockwood *et al.* 2000). Such precomputed signed distance transforms can be extremely useful in efficiently aligning and merging 2D curves and 3D surfaces (Huttenlocher, Klanderman, and Rucklidge 1993; Szeliski and Lavallée 1996; Curless and Levoy 1996), especially if the *vectorial* version of the distance transform, i.e., a pointer from each pixel or voxel to the nearest boundary or surface element, is stored and interpolated. Signed distance fields are also an essential component of level set evolution (Section 5.1.4), where they are called *characteristic functions*.

### 3.3.4 Connected components

Another useful semi-global image operation is finding *connected components*, which are defined as regions of adjacent pixels that have the same input value (or label). (In the remainder of this section, consider pixels to be *adjacent* if they are immediate  $\mathcal{N}_4$  neighbors and they have the same input value.) Connected components can be used in a variety of applications, such as finding individual letters in a scanned document or finding objects (say, cells) in a thresholded image and computing their area statistics.

Consider the grayscale image in Figure 3.23a. There are four connected components in this figure: the outermost set of white pixels, the large ring of gray pixels, the white enclosed region, and the single gray pixel. These are shown pseudocolored in Figure 3.23c as pink, green, blue, and brown.

To compute the connected components of an image, we first (conceptually) split the image into horizontal *runs* of adjacent pixels, and then color the runs with unique labels, re-using the labels of vertically adjacent runs whenever possible. In a second phase, adjacent runs of different colors are then merged.

While this description is a little sketchy, it should be enough to enable a motivated student to implement this algorithm (Exercise 3.14). Haralick and Shapiro (1992, Section 2.3) give a much longer description of various connected component algorithms, including ones that avoid the creation of a potentially large re-coloring (equivalence) table. Well-debugged connected component algorithms are also available in most image processing libraries.

Once a binary or multi-valued image has been segmented into its connected components,

it is often useful to compute the area statistics for each individual region  $\mathcal{R}$ . Such statistics include:

- the area (number of pixels);
- the perimeter (number of boundary pixels);
- the centroid (average  $x$  and  $y$  values);
- the second moments,

$$\mathbf{M} = \sum_{(x,y) \in \mathcal{R}} \begin{bmatrix} x - \bar{x} \\ y - \bar{y} \end{bmatrix} \begin{bmatrix} x - \bar{x} & y - \bar{y} \end{bmatrix}, \quad (3.46)$$

from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis.<sup>7</sup>

These statistics can then be used for further processing, e.g., for sorting the regions by the area size (to consider the largest regions first) or for preliminary matching of regions in different images.

### 3.4 Fourier transforms

In Section 3.2, we mentioned that Fourier analysis could be used to analyze the frequency characteristics of various filters. In this section, we explain both how Fourier analysis lets us determine these characteristics (or equivalently, the frequency *content* of an image) and how using the Fast Fourier Transform (FFT) lets us perform large-kernel convolutions in time that is independent of the kernel's size. More comprehensive introductions to Fourier transforms are provided by Bracewell (1986); Glassner (1995); Oppenheim and Schaffer (1996); Oppenheim, Schaffer, and Buck (1999).

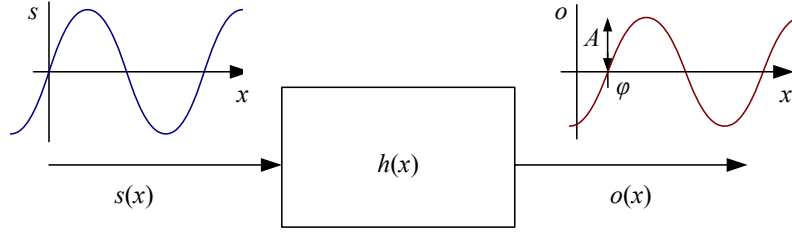
How can we analyze what a given filter does to high, medium, and low frequencies? The answer is to simply pass a sinusoid of known frequency through the filter and to observe by how much it is attenuated. Let

$$s(x) = \sin(2\pi f x + \phi_i) = \sin(\omega x + \phi_i) \quad (3.47)$$

be the input sinusoid whose *frequency* is  $f$ , *angular frequency* is  $\omega = 2\pi f$ , and *phase* is  $\phi_i$ . Note that in this section, we use the variables  $x$  and  $y$  to denote the spatial coordinates of an image, rather than  $i$  and  $j$  as in the previous sections. This is both because the letters  $i$  and  $j$  are used for the *imaginary* number (the usage depends on whether you are reading complex variables or electrical engineering literature) and because it is clearer how to distinguish the horizontal ( $x$ ) and vertical ( $y$ ) components in frequency space. In this section, we use the letter  $j$  for the imaginary number, since that is the form more commonly found in the signal processing literature (Bracewell 1986; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999).

---

<sup>7</sup> Moments can also be computed using Green's theorem applied to the boundary pixels (Yang and Albregtsen 1996).



**Figure 3.24** The Fourier Transform as the response of a filter  $h(x)$  to an input sinusoid  $s(x) = e^{j\omega x}$  yielding an output sinusoid  $o(x) = h(x) * s(x) = Ae^{j\omega x + \phi}$ .

If we convolve the sinusoidal signal  $s(x)$  with a filter whose impulse response is  $h(x)$ , we get another sinusoid of the same frequency but different magnitude  $A$  and phase  $\phi_o$ ,

$$o(x) = h(x) * s(x) = A \sin(\omega x + \phi_o), \quad (3.48)$$

as shown in Figure 3.24. To see that this is the case, remember that a convolution can be expressed as a weighted summation of shifted input signals (3.14) and that the summation of a bunch of shifted sinusoids of the same frequency is just a single sinusoid at that frequency.<sup>8</sup> The new magnitude  $A$  is called the *gain* or *magnitude* of the filter, while the phase difference  $\Delta\phi = \phi_o - \phi_i$  is called the *shift* or *phase*.

In fact, a more compact notation is to use the complex-valued sinusoid

$$s(x) = e^{j\omega x} = \cos \omega x + j \sin \omega x. \quad (3.49)$$

In that case, we can simply write,

$$o(x) = h(x) * s(x) = Ae^{j\omega x + \phi}. \quad (3.50)$$

The *Fourier transform* is simply a tabulation of the magnitude and phase response at each frequency,

$$H(\omega) = \mathcal{F}\{h(x)\} = Ae^{j\phi}, \quad (3.51)$$

i.e., it is the response to a complex sinusoid of frequency  $\omega$  passed through the filter  $h(x)$ . The Fourier transform pair is also often written as

$$h(x) \xleftrightarrow{\mathcal{F}} H(\omega). \quad (3.52)$$

Unfortunately, (3.51) does not give an actual *formula* for computing the Fourier transform. Instead, it gives a *recipe*, i.e., convolve the filter with a sinusoid, observe the magnitude and phase shift, repeat. Fortunately, closed form equations for the Fourier transform exist both in the continuous domain,

$$H(\omega) = \int_{-\infty}^{\infty} h(x) e^{-j\omega x} dx, \quad (3.53)$$

<sup>8</sup> If  $h$  is a general (non-linear) transform, additional *harmonic* frequencies are introduced. This was traditionally the bane of audiophiles, who insisted on equipment with no *harmonic distortion*. Now that digital audio has introduced pure distortion-free sound, some audiophiles are buying retro tube amplifiers or digital signal processors that simulate such distortions because of their “warmer sound”.

Property	Signal	Transform
superposition	$f_1(x) + f_2(x)$	$F_1(\omega) + F_2(\omega)$
shift	$f(x - x_0)$	$F(\omega)e^{-j\omega x_0}$
reversal	$f(-x)$	$F^*(\omega)$
convolution	$f(x) * h(x)$	$F(\omega)H(\omega)$
correlation	$f(x) \otimes h(x)$	$F(\omega)H^*(\omega)$
multiplication	$f(x)h(x)$	$F(\omega) * H(\omega)$
differentiation	$f'(x)$	$j\omega F(\omega)$
domain scaling	$f(ax)$	$1/a F(\omega/a)$
real images	$f(x) = f^*(x)$	$\Leftrightarrow F(\omega) = F^*(-\omega)$
Parseval's Theorem	$\sum_x [f(x)]^2$	$= \sum_\omega [F(\omega)]^2$

**Table 3.1** Some useful properties of Fourier transforms. The original transform pair is  $F(\omega) = \mathcal{F}\{f(x)\}$ .

and in the discrete domain,

$$H(k) = \frac{1}{N} \sum_{x=0}^{N-1} h(x) e^{-j \frac{2\pi kx}{N}}, \quad (3.54)$$

where  $N$  is the length of the signal or region of analysis. These formulas apply both to filters, such as  $h(x)$ , and to signals or images, such as  $s(x)$  or  $g(x)$ .

The discrete form of the Fourier transform (3.54) is known as the *Discrete Fourier Transform* (DFT). Note that while (3.54) can be evaluated for any value of  $k$ , it only makes sense for values in the range  $k \in [-\frac{N}{2}, \frac{N}{2}]$ . This is because larger values of  $k$  *alias* with lower frequencies and hence provide no additional information, as explained in the discussion on aliasing in Section 2.3.1.

At face value, the DFT takes  $O(N^2)$  operations (multiply-adds) to evaluate. Fortunately, there exists a faster algorithm called the *Fast Fourier Transform* (FFT), which requires only  $O(N \log_2 N)$  operations (Bracewell 1986; Oppenheim, Schaffer, and Buck 1999). We do not explain the details of the algorithm here, except to say that it involves a series of  $\log_2 N$  stages, where each stage performs small  $2 \times 2$  transforms (matrix multiplications with known coefficients) followed by some semi-global permutations. (You will often see the term *butterfly* applied to these stages because of the pictorial shape of the signal processing graphs involved.) Implementations for the FFT can be found in most numerical and signal processing libraries.

Now that we have defined the Fourier transform, what are some of its properties and how can they be used? Table 3.1 lists a number of useful properties, which we describe in a little more detail below:

- **Superposition:** The Fourier transform of a sum of signals is the sum of their Fourier transforms. Thus, the Fourier transform is a linear operator.
- **Shift:** The Fourier transform of a shifted signal is the transform of the original signal multiplied by a *linear phase shift* (complex sinusoid).

- **Reversal:** The Fourier transform of a reversed signal is the complex conjugate of the signal's transform.
- **Convolution:** The Fourier transform of a pair of convolved signals is the product of their transforms.
- **Correlation:** The Fourier transform of a correlation is the product of the first transform times the complex conjugate of the second one.
- **Multiplication:** The Fourier transform of the product of two signals is the convolution of their transforms.
- **Differentiation:** The Fourier transform of the derivative of a signal is that signal's transform multiplied by the frequency. In other words, differentiation linearly emphasizes (magnifies) higher frequencies.
- **Domain scaling:** The Fourier transform of a stretched signal is the equivalently compressed (and scaled) version of the original transform and *vice versa*.
- **Real images:** The Fourier transform of a real-valued signal is symmetric around the origin. This fact can be used to save space and to double the speed of image FFTs by packing alternating scanlines into the real and imaginary parts of the signal being transformed.
- **Parseval's Theorem:** The energy (sum of squared values) of a signal is the same as the energy of its Fourier transform.

All of these properties are relatively straightforward to prove (see Exercise 3.15) and they will come in handy later in the book, e.g., when designing optimum Wiener filters (Section 3.4.3) or performing fast image correlations (Section 8.1.2).

### 3.4.1 Fourier transform pairs

Now that we have these properties in place, let us look at the Fourier transform pairs of some commonly occurring filters and signals, as listed in Table 3.2. In more detail, these pairs are as follows:

- **Impulse:** The impulse response has a constant (all frequency) transform.
- **Shifted impulse:** The shifted impulse has unit magnitude and linear phase.
- **Box filter:** The box (moving average) filter

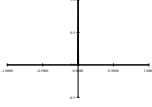

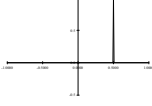

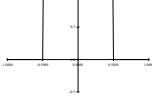


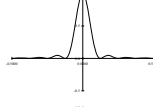
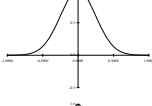
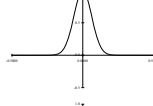
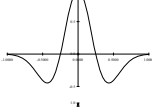
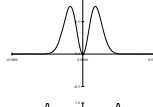
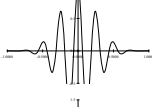
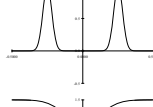
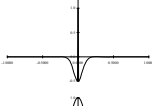
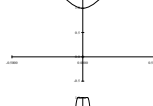
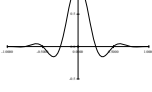
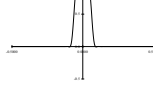
$$\text{box}(x) = \begin{cases} 1 & \text{if } |x| \leq 1 \\ 0 & \text{else} \end{cases} \quad (3.55)$$

has a sinc Fourier transform,

$$\text{sinc}(\omega) = \frac{\sin \omega}{\omega}, \quad (3.56)$$

which has an infinite number of side lobes. Conversely, the sinc filter is an ideal low-pass filter. For a non-unit box, the width of the box  $a$  and the spacing of the zero crossings in the sinc  $1/a$  are inversely proportional.



Name	Signal	Transform
impulse		$\delta(x) \Leftrightarrow 1$ 
shifted impulse		$\delta(x-u) \Leftrightarrow e^{-j\omega u}$ 
box filter		$\text{box}(x/a) \Leftrightarrow a\text{sinc}(a\omega)$ 
tent		$\text{tent}(x/a) \Leftrightarrow a\text{sinc}^2(a\omega)$ 
Gaussian		$G(x; \sigma) \Leftrightarrow \frac{\sqrt{2\pi}}{\sigma} G(\omega; \sigma^{-1})$ 
Laplacian of Gaussian		$(\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2})G(x; \sigma) \Leftrightarrow -\frac{\sqrt{2\pi}}{\sigma} \omega^2 G(\omega; \sigma^{-1})$ 
Gabor		$\cos(\omega_0 x)G(x; \sigma) \Leftrightarrow \frac{\sqrt{2\pi}}{\sigma} G(\omega \pm \omega_0; \sigma^{-1})$ 
unsharp mask		$(1 + \gamma)\delta(x) - \gamma G(x; \sigma) \Leftrightarrow (1 + \gamma) - \frac{\sqrt{2\pi}\gamma}{\sigma} G(\omega; \sigma^{-1})$ 
windowed sinc		$\text{rcos}(x/(aW)) \text{sinc}(x/a) \Leftrightarrow$ (see Figure 3.29) 

**Table 3.2** Some useful (continuous) Fourier transform pairs: The dashed line in the Fourier transform of the shifted impulse indicates its (linear) phase. All other transforms have zero phase (they are real-valued). Note that the figures are not necessarily drawn to scale but are drawn to illustrate the general shape and characteristics of the filter or its response. In particular, the Laplacian of Gaussian is drawn inverted because it resembles more a “Mexican hat”, as it is sometimes called.

- **Tent:** The piecewise linear tent function,

$$\text{tent}(x) = \max(0, 1 - |x|), \quad (3.57)$$

has a  $\text{sinc}^2$  Fourier transform.

- **Gaussian:** The (unit area) Gaussian of width  $\sigma$ ,

$$G(x; \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}, \quad (3.58)$$

has a (unit height) Gaussian of width  $\sigma^{-1}$  as its Fourier transform.

- **Laplacian of Gaussian:** The second derivative of a Gaussian of width  $\sigma$ ,

$$\text{LoG}(x; \sigma) = \left(\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2}\right)G(x; \sigma) \quad (3.59)$$

has a band-pass response of

$$-\frac{\sqrt{2\pi}}{\sigma} \omega^2 G(\omega; \sigma^{-1}) \quad (3.60)$$

as its Fourier transform.

- **Gabor:** The even Gabor function, which is the product of a cosine of frequency  $\omega_0$  and a Gaussian of width  $\sigma$ , has as its transform the sum of the two Gaussians of width  $\sigma^{-1}$  centered at  $\omega = \pm\omega_0$ . The odd Gabor function, which uses a sine, is the difference of two such Gaussians. Gabor functions are often used for oriented and band-pass filtering, since they can be more frequency selective than Gaussian derivatives.
- **Unsharp mask:** The unsharp mask introduced in (3.22) has as its transform a unit response with a slight boost at higher frequencies.
- **Windowed sinc:** The windowed (masked) sinc function shown in Table 3.2 has a response function that approximates an ideal low-pass filter better and better as additional side lobes are added ( $W$  is increased). Figure 3.29 shows the shapes of these such filters along with their Fourier transforms. For these examples, we use a one-lobe raised cosine,

$$\text{rcos}(x) = \frac{1}{2}(1 + \cos \pi x)\text{box}(x), \quad (3.61)$$

also known as the *Hann window*, as the windowing function. Wolberg (1990) and Oppenheim, Schafer, and Buck (1999) discuss additional windowing functions, which include the *Lanczos* window, the positive first lobe of a sinc function.

We can also compute the Fourier transforms for the small discrete kernels shown in Figure 3.14 (see Table 3.3). Notice how the moving average filters do not uniformly dampen higher frequencies and hence can lead to ringing artifacts. The binomial filter (Gomes and Velho 1997) used as the “Gaussian” in Burt and Adelson’s (1983a) Laplacian pyramid (see Section 3.5), does a decent job of separating the high and low frequencies, but still leaves a fair amount of high-frequency detail, which can lead to aliasing after downsampling. The Sobel edge detector at first linearly accentuates frequencies, but then decays at higher frequencies, and hence has trouble detecting fine-scale edges, e.g., adjacent black and white columns. We look at additional examples of small kernel Fourier transforms in Section 3.5.2, where we study better kernels for pre-filtering before decimation (size reduction).

Name	Kernel	Transform	Plot
box-3	$\frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$	$\frac{1}{3}(1 + 2 \cos \omega)$	
box-5	$\frac{1}{5} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$	$\frac{1}{5}(1 + 2 \cos \omega + 2 \cos 2\omega)$	
linear	$\frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$	$\frac{1}{2}(1 + \cos \omega)$	
binomial	$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	$\frac{1}{4}(1 + \cos \omega)^2$	
Sobel	$\frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$	$\sin \omega$	
corner	$\frac{1}{2} \begin{bmatrix} -1 & 2 & -1 \end{bmatrix}$	$\frac{1}{2}(1 - \cos \omega)$	

**Table 3.3** Fourier transforms of the separable kernels shown in Figure 3.14.

### 3.4.2 Two-dimensional Fourier transforms

The formulas and insights we have developed for one-dimensional signals and their transforms translate directly to two-dimensional images. Here, instead of just specifying a horizontal or vertical frequency  $\omega_x$  or  $\omega_y$ , we can create an oriented sinusoid of frequency  $(\omega_x, \omega_y)$ ,

$$s(x, y) = \sin(\omega_x x + \omega_y y). \quad (3.62)$$

The corresponding two-dimensional Fourier transforms are then

$$H(\omega_x, \omega_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x, y) e^{-j(\omega_x x + \omega_y y)} dx dy, \quad (3.63)$$

and in the discrete domain,

$$H(k_x, k_y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-j2\pi \frac{k_x x + k_y y}{MN}}, \quad (3.64)$$

where  $M$  and  $N$  are the width and height of the image.

All of the Fourier transform properties from Table 3.1 carry over to two dimensions if we replace the scalar variables  $x$ ,  $\omega$ ,  $x_0$  and  $a$  with their 2D vector counterparts  $\mathbf{x} = (x, y)$ ,  $\boldsymbol{\omega} = (\omega_x, \omega_y)$ ,  $\mathbf{x}_0 = (x_0, y_0)$ , and  $\mathbf{a} = (a_x, a_y)$ , and use vector inner products instead of multiplications.

### 3.4.3 Wiener filtering

While the Fourier transform is a useful tool for analyzing the frequency characteristics of a filter kernel or image, it can also be used to analyze the frequency spectrum of a whole *class* of images.

A simple model for images is to assume that they are random noise fields whose expected magnitude at each frequency is given by this *power spectrum*  $P_s(\omega_x, \omega_y)$ , i.e.,

$$\langle [S(\omega_x, \omega_y)]^2 \rangle = P_s(\omega_x, \omega_y), \quad (3.65)$$

where the angle brackets  $\langle \cdot \rangle$  denote the expected (mean) value of a random variable.<sup>9</sup> To generate such an image, we simply create a random Gaussian noise image  $S(\omega_x, \omega_y)$  where each “pixel” is a zero-mean Gaussian<sup>10</sup> of variance  $P_s(\omega_x, \omega_y)$  and then take its inverse FFT.

The observation that signal spectra capture a first-order description of spatial statistics is widely used in signal and image processing. In particular, assuming that an image is a sample from a correlated Gaussian random noise field combined with a statistical model of the measurement process yields an optimum restoration filter known as the *Wiener filter*.<sup>11</sup>

To derive the Wiener filter, we analyze each frequency component of a signal’s Fourier transform independently. The noisy image formation process can be written as

$$o(x, y) = s(x, y) + n(x, y), \quad (3.66)$$

<sup>9</sup> The notation  $E[\cdot]$  is also commonly used.

<sup>10</sup> We set the DC (i.e., constant) component at  $S(0, 0)$  to the mean grey level. See Algorithm C.1 in Appendix C.2 for code to generate Gaussian noise.

<sup>11</sup> Wiener is pronounced “veener” since, in German, the “w” is pronounced “v”. Remember that next time you order “Wiener schnitzel”.

where  $s(x, y)$  is the (unknown) image we are trying to recover,  $n(x, y)$  is the additive noise signal, and  $o(x, y)$  is the *observed* noisy image. Because of the linearity of the Fourier transform, we can write

$$O(\omega_x, \omega_y) = S(\omega_x, \omega_y) + N(\omega_x, \omega_y), \quad (3.67)$$

where each quantity in the above equation is the Fourier transform of the corresponding image.

At each frequency  $(\omega_x, \omega_y)$ , we know from our image spectrum that the unknown transform component  $S(\omega_x, \omega_y)$  has a *prior* distribution which is a zero-mean Gaussian with variance  $P_s(\omega_x, \omega_y)$ . We also have noisy measurement  $O(\omega_x, \omega_y)$  whose variance is  $P_n(\omega_x, \omega_y)$ , i.e., the power spectrum of the noise, which is usually assumed to be constant (white),  $P_n(\omega_x, \omega_y) = \sigma_n^2$ .

According to Bayes' Rule (Appendix B.4), the *posterior estimate* of  $S$  can be written as

$$p(S|O) = \frac{p(O|S)p(S)}{p(O)}, \quad (3.68)$$

where  $p(O) = \int_S p(O|S)p(S)$  is a normalizing constant used to make the  $p(S|O)$  distribution *proper* (integrate to 1). The prior distribution  $p(S)$  is given by

$$p(S) = e^{-\frac{(S-\mu)^2}{2P_s}}, \quad (3.69)$$

where  $\mu$  is the expected mean at that frequency (0 everywhere except at the origin) and the measurement distribution  $P(O|S)$  is given by

$$p(S) = e^{-\frac{(S-O)^2}{2P_n}}. \quad (3.70)$$

Taking the negative logarithm of both sides of (3.68) and setting  $\mu = 0$  for simplicity, we get

$$-\log p(S|O) = -\log p(O|S) - \log p(S) + C \quad (3.71)$$

$$= 1/2 P_n^{-1} (S - O)^2 + 1/2 P_s^{-1} S^2 + C, \quad (3.72)$$

which is the *negative posterior log likelihood*. The minimum of this quantity is easy to compute,

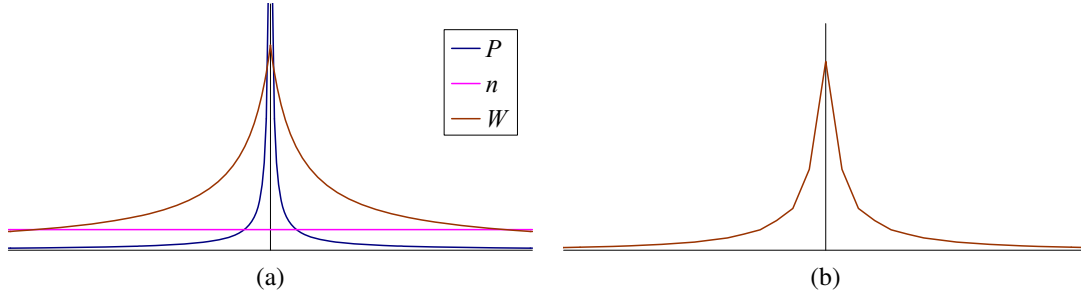
$$S_{\text{opt}} = \frac{P_n^{-1}}{P_n^{-1} + P_s^{-1}} O = \frac{P_s}{P_s + P_n} O = \frac{1}{1 + P_n/P_s} O. \quad (3.73)$$

The quantity

$$W(\omega_x, \omega_y) = \frac{1}{1 + \sigma_n^2/P_s(\omega_x, \omega_y)} \quad (3.74)$$

is the Fourier transform of the optimum *Wiener filter* needed to remove the noise from an image whose power spectrum is  $P_s(\omega_x, \omega_y)$ .

Notice that this filter has the right qualitative properties, i.e., for low frequencies where  $P_s \gg \sigma_n^2$ , it has unit gain, whereas for high frequencies, it attenuates the noise by a factor  $P_s/\sigma_n^2$ . Figure 3.25 shows the one-dimensional transform  $W(f)$  and the corresponding filter kernel  $w(x)$  for the commonly assumed case of  $P(f) = f^{-2}$  (Field 1987). Exercise 3.16 has you compare the Wiener filter as a denoising algorithm to hand-tuned Gaussian smoothing.



**Figure 3.25** One-dimensional Wiener filter: (a) power spectrum of signal  $P_s(f)$ , noise level  $\sigma^2$ , and Wiener filter transform  $W(f)$ ; (b) Wiener filter spatial kernel.

The methodology given above for deriving the Wiener filter can easily be extended to the case where the observed image is a noisy blurred version of the original image,

$$o(x, y) = b(x, y) * s(x, y) + n(x, y), \quad (3.75)$$

where  $b(x, y)$  is the known blur kernel. Rather than deriving the corresponding Wiener filter, we leave it as an exercise (Exercise 3.17), which also encourages you to compare your de-blurring results with unsharp masking and naïve inverse filtering. More sophisticated algorithms for blur removal are discussed in Sections 3.7 and 10.3.

### Discrete cosine transform

The *discrete cosine transform* (DCT) is a variant of the Fourier transform particularly well-suited to compressing images in a block-wise fashion. The one-dimensional DCT is computed by taking the dot product of each  $N$ -wide block of pixels with a set of cosines of different frequencies,

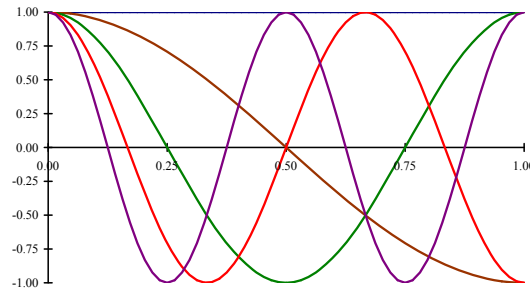
$$F(k) = \sum_{i=0}^{N-1} \cos\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)k\right) f(i), \quad (3.76)$$

where  $k$  is the coefficient (frequency) index, and the  $1/2$ -pixel offset is used to make the basis coefficients symmetric (Wallace 1991). Some of the discrete cosine basis functions are shown in Figure 3.26. As you can see, the first basis function (the straight blue line) encodes the average DC value in the block of pixels, while the second encodes a slightly curvy version of the slope.

It turns out that the DCT is a good approximation to the optimal Karhunen–Loève decomposition of natural image statistics over small patches, which can be obtained by performing a principal component analysis (PCA) of images, as described in Section 14.2.1. The KL-transform de-correlates the signal optimally (assuming the signal is described by its spectrum) and thus, theoretically, leads to optimal compression.

The two-dimensional version of the DCT is defined similarly,

$$F(k, l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)k\right) \cos\left(\frac{\pi}{N}\left(j + \frac{1}{2}\right)l\right) f(i, j). \quad (3.77)$$



**Figure 3.26** Discrete cosine transform (DCT) basis functions: The first DC (i.e., constant) basis is the horizontal blue line, the second is the brown half-cycle waveform, etc. These bases are widely used in image and video compression standards such as JPEG.

Like the 2D Fast Fourier Transform, the 2D DCT can be implemented separably, i.e., first computing the DCT of each line in the block and then computing the DCT of each resulting column. Like the FFT, each of the DCTs can also be computed in  $O(N \log N)$  time.

As we mentioned in Section 2.3.3, the DCT is widely used in today’s image and video compression algorithms, although it is slowly being supplanted by wavelet algorithms (Simoncelli and Adelson 1990b), as discussed in Section 3.5.4, and overlapped variants of the DCT (Malvar 1990, 1998, 2000), which are used in the new JPEG XR standard.<sup>12</sup> These newer algorithms suffer less from the *blocking artifacts* (visible edge-aligned discontinuities) that result from the pixels in each block (typically  $8 \times 8$ ) being transformed and quantized independently. See Exercise 3.30 for ideas on how to remove blocking artifacts from compressed JPEG images.

### 3.4.4 Application: Sharpening, blur, and noise removal

Another common application of image processing is the enhancement of images through the use of sharpening and noise removal operations, which require some kind of neighborhood processing. Traditionally, these kinds of operation were performed using linear filtering (see Sections 3.2 and Section 3.4.3). Today, it is more common to use non-linear filters (Section 3.3.1), such as the weighted median or bilateral filter (3.34–3.37), anisotropic diffusion (3.39–3.40), or non-local means (Buades, Coll, and Morel 2008). Variational methods (Section 3.7.1), especially those using non-quadratic (robust) norms such as the  $L_1$  norm (which is called *total variation*), are also often used. Figure 3.19 shows some examples of linear and non-linear filters being used to remove noise.

When measuring the effectiveness of image denoising algorithms, it is common to report the results as a *peak signal-to-noise ratio (PSNR)* measurement (2.119), where  $I(x)$  is the original (noise-free) image and  $\hat{I}(x)$  is the image after denoising; this is for the case where the noisy image has been synthetically generated, so that the clean image is known. A better way to measure the quality is to use a perceptually based similarity metric, such as the structural similarity (SSIM) index (Wang, Bovik, Sheikh *et al.* 2004; Wang, Bovik, and Simoncelli 2005).

<sup>12</sup> <http://www.itu.int/rec/T-REC-T.832-200903-I/en>.



Exercises 3.11, 3.16, 3.17, 3.21, and 3.28 have you implement some of these operations and compare their effectiveness. More sophisticated techniques for blur removal and the related task of super-resolution are discussed in Section 10.3.

## 3.5 Pyramids and wavelets

So far in this chapter, all of the image transformations we have studied produce output images of the same size as the inputs. Often, however, we may wish to change the resolution of an image before proceeding further. For example, we may need to interpolate a small image to make its resolution match that of the output printer or computer screen. Alternatively, we may want to reduce the size of an image to speed up the execution of an algorithm or to save on storage space or transmission time.

Sometimes, we do not even know what the appropriate resolution for the image should be. Consider, for example, the task of finding a face in an image (Section 14.1.1). Since we do not know the scale at which the face will appear, we need to generate a whole *pyramid* of differently sized images and scan each one for possible faces. (Biological visual systems also operate on a hierarchy of scales (Marr 1982).) Such a pyramid can also be very helpful in accelerating the search for an object by first finding a smaller instance of that object at a coarser level of the pyramid and then looking for the full resolution object only in the vicinity of coarse-level detections (Section 8.1.1). Finally, image pyramids are extremely useful for performing multi-scale editing operations such as blending images while maintaining details.

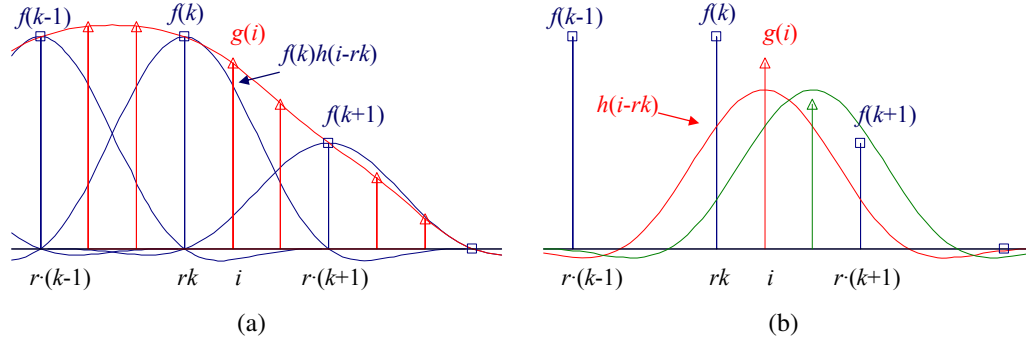
In this section, we first discuss good filters for changing image resolution, i.e., upsampling (*interpolation*, Section 3.5.1) and downsampling (*decimation*, Section 3.5.2). We then present the concept of multi-resolution pyramids, which can be used to create a complete hierarchy of differently sized images and to enable a variety of applications (Section 3.5.3). A closely related concept is that of *wavelets*, which are a special kind of pyramid with higher frequency selectivity and other useful properties (Section 3.5.4). Finally, we present a useful application of pyramids, namely the blending of different images in a way that hides the seams between the image boundaries (Section 3.5.5).

### 3.5.1 Interpolation

In order to *interpolate* (or *upsample*) an image to a higher resolution, we need to select some interpolation kernel with which to convolve the image,

$$g(i, j) = \sum_{k, l} f(k, l) h(i - rk, j - rl). \quad (3.78)$$

This formula is related to the discrete convolution formula (3.14), except that we replace  $k$  and  $l$  in  $h()$  with  $rk$  and  $rl$ , where  $r$  is the upsampling rate. Figure 3.27a shows how to think of this process as the superposition of sample weighted interpolation kernels, one centered at each input sample  $k$ . An alternative mental model is shown in Figure 3.27b, where the kernel is centered at the output pixel value  $i$  (the two forms are equivalent). The latter form is sometimes called the *polyphase filter* form, since the kernel values  $h(i)$  can be stored as  $r$  separate kernels, each of which is selected for convolution with the input samples depending on the *phase* of  $i$  relative to the upsampled grid.



**Figure 3.27** Signal interpolation,  $g(i) = \sum_k f(k)h(i - rk)$ : (a) weighted summation of input values; (b) polyphase filter interpretation.

What kinds of kernel make good interpolators? The answer depends on the application and the computation time involved. Any of the smoothing kernels shown in Tables 3.2 and 3.3 can be used after appropriate re-scaling.<sup>13</sup> The *linear* interpolator (corresponding to the tent kernel) produces interpolating piecewise linear curves, which result in unappealing *creases* when applied to images (Figure 3.28a). The cubic B-spline, whose discrete  $1/2$ -pixel sampling appears as the *binomial kernel* in Table 3.3, is an *approximating* kernel (the interpolated image does not pass through the input data points) that produces soft images with reduced high-frequency detail. The equation for the cubic B-spline is easiest to derive by convolving the tent function (linear B-spline) with itself.

While most graphics cards use the bilinear kernel (optionally combined with a MIP-map—see Section 3.5.3), most photo editing packages use *bicubic* interpolation. The cubic interpolant is a  $C^1$  (derivative-continuous) piecewise-cubic *spline* (the term “spline” is synonymous with “piecewise-polynomial”)<sup>14</sup> whose equation is

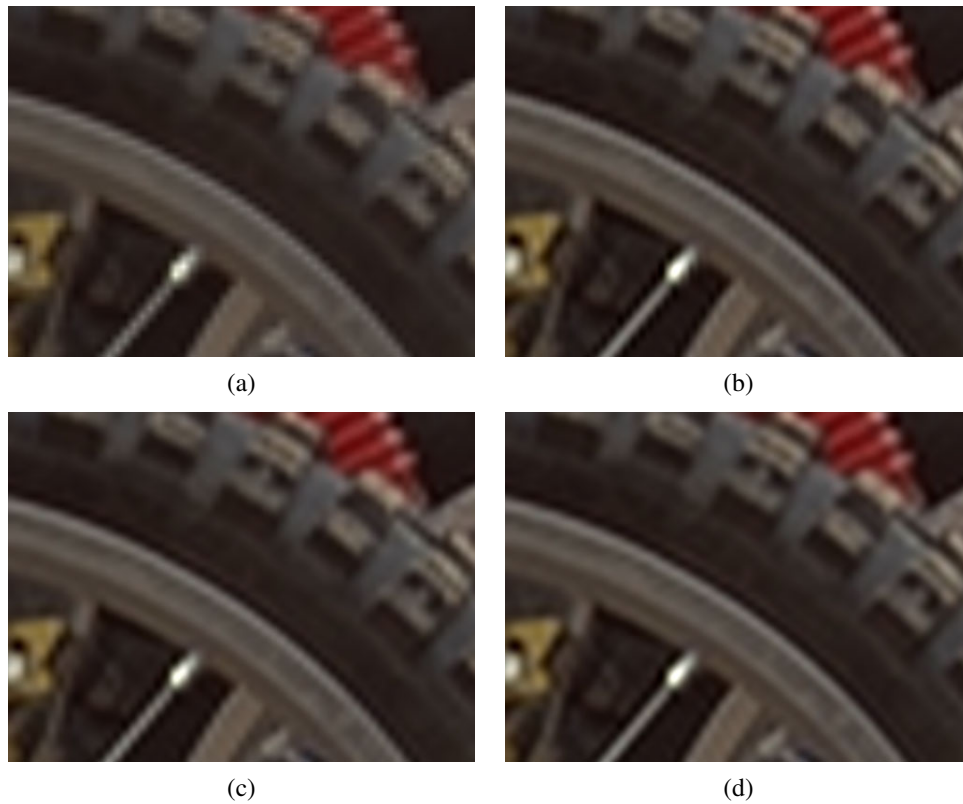
$$h(x) = \begin{cases} 1 - (a + 3)x^2 + (a + 2)|x|^3 & \text{if } |x| < 1 \\ a(|x| - 1)(|x| - 2)^2 & \text{if } 1 \leq |x| < 2 \\ 0 & \text{otherwise,} \end{cases} \quad (3.79)$$

where  $a$  specifies the derivative at  $x = 1$  (Parker, Kenyon, and Troxel 1983). The value of  $a$  is often set to  $-1$ , since this best matches the frequency characteristics of a sinc function (Figure 3.29). It also introduces a small amount of sharpening, which can be visually appealing. Unfortunately, this choice does not linearly interpolate straight lines (intensity ramps), so some visible ringing may occur. A better choice for large amounts of interpolation is probably  $a = -0.5$ , which produces a *quadratic reproducing* spline; it interpolates linear and quadratic functions exactly (Wolberg 1990, Section 5.4.3). Figure 3.29 shows the  $a = -1$  and  $a = -0.5$  cubic interpolating kernel along with their Fourier transforms; Figure 3.28b and c shows them being applied to two-dimensional interpolation.

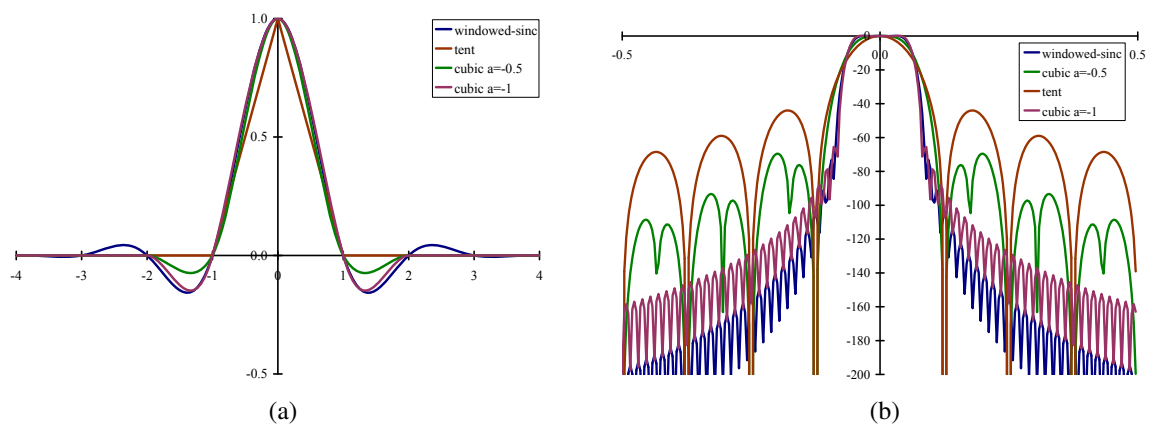
Splines have long been used for function and data value interpolation because of the abil-

<sup>13</sup> The smoothing kernels in Table 3.3 have a unit area. To turn them into interpolating kernels, we simply scale them up by the interpolation rate  $r$ .

<sup>14</sup> The term “spline” comes from the draughtsman’s workshop, where it was the name of a flexible piece of wood or metal used to draw smooth curves.



**Figure 3.28** Two-dimensional image interpolation: (a) bilinear; (b) bicubic ( $a = -1$ ); (c) bicubic ( $a = -0.5$ ); (d) windowed sinc (nine taps).



**Figure 3.29** (a) Some windowed sinc functions and (b) their log Fourier transforms: raised-cosine windowed sinc in blue, cubic interpolators ( $a = -1$  and  $a = -0.5$ ) in green and purple, and tent function in brown. They are often used to perform high-accuracy low-pass filtering operations.

ity to precisely specify derivatives at control points and efficient *incremental* algorithms for their evaluation (Bartels, Beatty, and Barsky 1987; Farin 1992, 1996). Splines are widely used in geometric modeling and computer-aided design (CAD) applications, although they have started being displaced by subdivision surfaces (Zorin, Schröder, and Sweldens 1996; Peters and Reif 2008). In computer vision, splines are often used for elastic image deformations (Section 3.6.2), motion estimation (Section 8.3), and surface interpolation (Section 12.3). In fact, it is possible to carry out most image processing operations by representing images as splines and manipulating them in a multi-resolution framework (Unser 1999).

The highest quality interpolator is generally believed to be the windowed sinc function because it both preserves details in the lower resolution image and avoids aliasing. (It is also possible to construct a  $C^1$  piecewise-cubic approximation to the windowed sinc by matching its derivatives at zero crossing (Szeliski and Ito 1986).) However, some people object to the excessive *ringing* that can be introduced by the windowed sinc and to the repetitive nature of the ringing frequencies (see Figure 3.28d). For this reason, some photographers prefer to repeatedly interpolate images by a small fractional amount (this tends to de-correlate the original pixel grid with the final image). Additional possibilities include using the bilateral filter as an interpolator (Kopf, Cohen, Lischinski *et al.* 2007), using global optimization (Section 3.6) or hallucinating details (Section 10.3).

### 3.5.2 Decimation

While interpolation can be used to increase the resolution of an image, decimation (downsampling) is required to reduce the resolution.<sup>15</sup> To perform decimation, we first (conceptually) convolve the image with a low-pass filter (to avoid aliasing) and then keep every  $r$ th sample. In practice, we usually only evaluate the convolution at every  $r$ th sample,

$$g(i, j) = \sum_{k, l} f(k, l) h(ri - k, rj - l), \quad (3.80)$$

as shown in Figure 3.30. Note that the smoothing kernel  $h(k, l)$ , in this case, is often a stretched and re-scaled version of an interpolation kernel. Alternatively, we can write

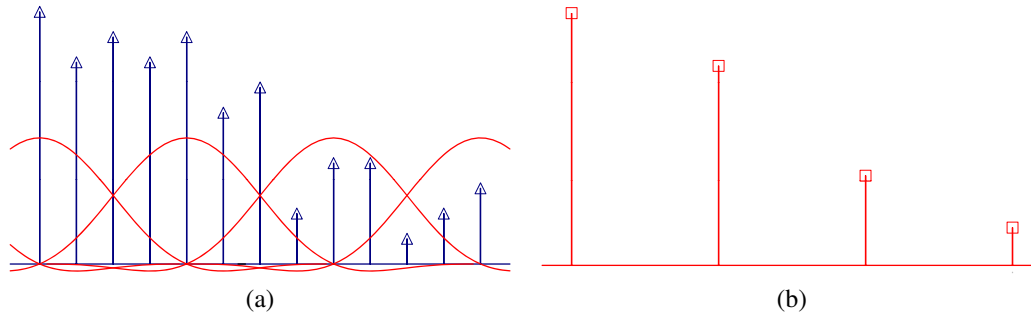
$$g(i, j) = \frac{1}{r} \sum_{k, l} f(k, l) h(i - k/r, j - l/r) \quad (3.81)$$

and keep the same kernel  $h(k, l)$  for both interpolation and decimation.

One commonly used ( $r = 2$ ) decimation filter is the *binomial* filter introduced by Burt and Adelson (1983a). As shown in Table 3.3, this kernel does a decent job of separating the high and low frequencies, but still leaves a fair amount of high-frequency detail, which can lead to aliasing after downsampling. However, for applications such as image blending (discussed later in this section), this aliasing is of little concern.

If, however, the downsampled images will be displayed directly to the user or, perhaps, blended with other resolutions (as in MIP-mapping, Section 3.5.3), a higher-quality filter is

<sup>15</sup> The term “decimation” has a gruesome etymology relating to the practice of killing every tenth soldier in a Roman unit guilty of cowardice. It is generally used in signal processing to mean any downsampling or rate reduction operation.



**Figure 3.30** Signal decimation: (a) the original samples are (b) convolved with a low-pass filter before being downsampled.

desired. For high downsampling rates, the windowed sinc pre-filter is a good choice (Figure 3.29). However, for small downsampling rates, e.g.,  $r = 2$ , more careful filter design is required.

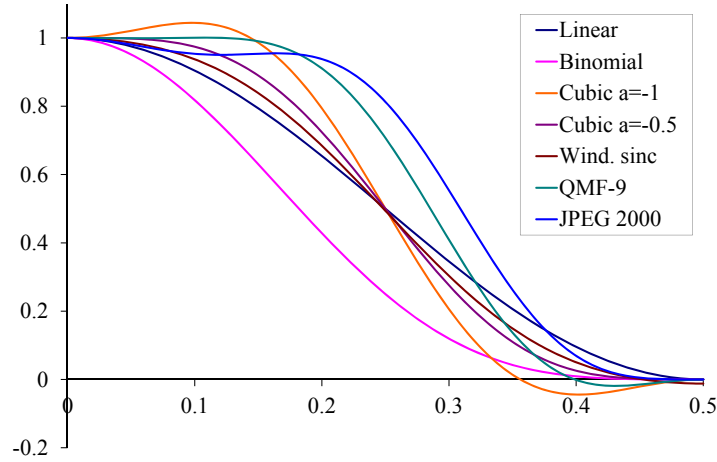
Table 3.4 shows a number of commonly used  $r = 2$  downsampling filters, while Figure 3.31 shows their corresponding frequency responses. These filters include:

- the linear  $[1, 2, 1]$  filter gives a relatively poor response;
- the binomial  $[1, 4, 6, 4, 1]$  filter cuts off a lot of frequencies but is useful for computer vision analysis pyramids;
- the cubic filters from (3.79); the  $a = -1$  filter has a sharper fall-off than the  $a = -0.5$  filter (Figure 3.31);
- a cosine-windowed sinc function (Table 3.2);
- the QMF-9 filter of Simoncelli and Adelson (1990b) is used for wavelet denoising and aliases a fair amount (note that the original filter coefficients are normalized to  $\sqrt{2}$  gain so they can be “self-inverting”);
- the 9/7 analysis filter from JPEG 2000 (Taubman and Marcellin 2002).

Please see the original papers for the full-precision values of some of these coefficients.

$ n $	Linear	Binomial	Cubic $a = -1$	Cubic $a = -0.5$	Windowed sinc	QMF-9	JPEG 2000
0	0.50	0.3750	0.5000	0.50000	0.4939	0.5638	0.6029
1	0.25	0.2500	0.3125	0.28125	0.2684	0.2932	0.2669
2		0.0625	0.0000	0.00000	0.0000	-0.0519	-0.0782
3			-0.0625	-0.03125	-0.0153	-0.0431	-0.0169
4					0.0000	0.0198	0.0267

**Table 3.4** Filter coefficients for  $2 \times$  decimation. These filters are of odd length, are symmetric, and are normalized to have unit DC gain (sum up to 1). See Figure 3.31 for their associated frequency responses.



**Figure 3.31** Frequency response for some  $2\times$  decimation filters. The cubic  $a = -1$  filter has the sharpest fall-off but also a bit of ringing; the wavelet analysis filters (QMF-9 and JPEG 2000), while useful for compression, have more aliasing.

### 3.5.3 Multi-resolution representations

Now that we have described interpolation and decimation algorithms, we can build a complete image pyramid (Figure 3.32). As we mentioned before, pyramids can be used to accelerate coarse-to-fine search algorithms, to look for objects or patterns at different scales, and to perform multi-resolution blending operations. They are also widely used in computer graphics hardware and software to perform fractional-level decimation using the MIP-map, which we cover in Section 3.6.

The best known (and probably most widely used) pyramid in computer vision is Burt and Adelson’s (1983a) Laplacian pyramid. To construct the pyramid, we first blur and sub-sample the original image by a factor of two and store this in the next level of the pyramid (Figure 3.33). Because adjacent levels in the pyramid are related by a sampling rate  $r = 2$ , this kind of pyramid is known as an *octave pyramid*. Burt and Adelson originally proposed a five-tap kernel of the form

$$\begin{bmatrix} c & b & a & b & c \end{bmatrix}, \quad (3.82)$$

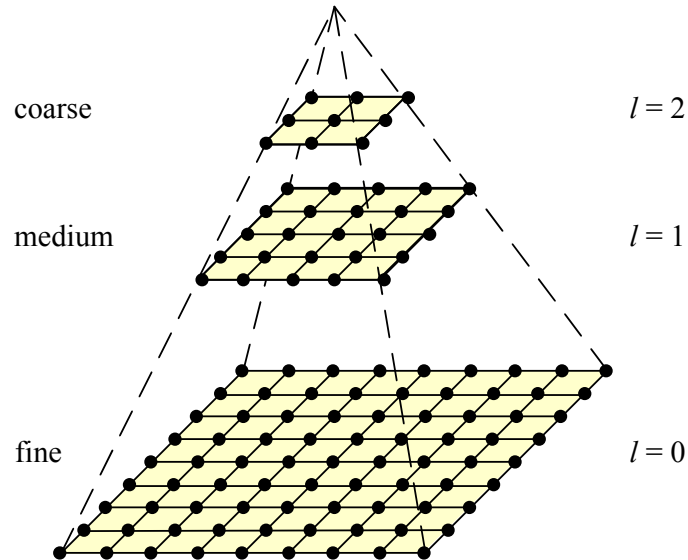
with  $b = 1/4$  and  $c = 1/4 - a/2$ . In practice,  $a = 3/8$ , which results in the familiar binomial kernel,

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}, \quad (3.83)$$

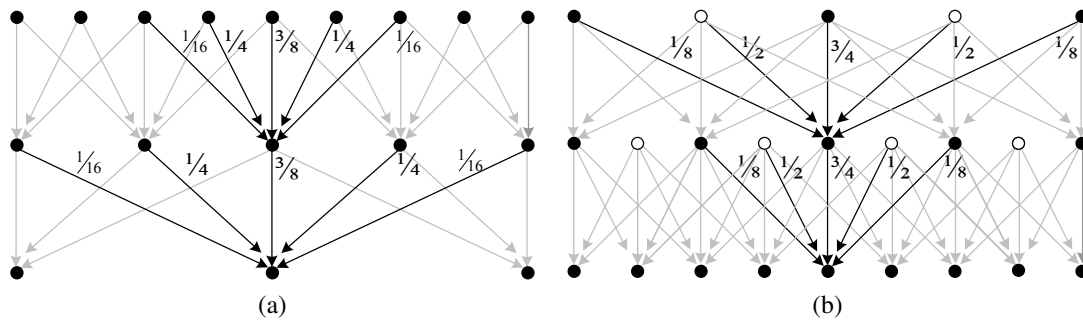
which is particularly easy to implement using shifts and adds. (This was important in the days when multipliers were expensive.) The reason they call their resulting pyramid a *Gaussian* pyramid is that repeated convolutions of the binomial kernel converge to a Gaussian.<sup>16</sup>

To compute the *Laplacian* pyramid, Burt and Adelson first interpolate a lower resolution image to obtain a *reconstructed* low-pass version of the original image (Figure 3.34b). They then subtract this low-pass version from the original to yield the band-pass “Laplacian”

<sup>16</sup> Then again, this is true for any smoothing kernel (Wells 1986).

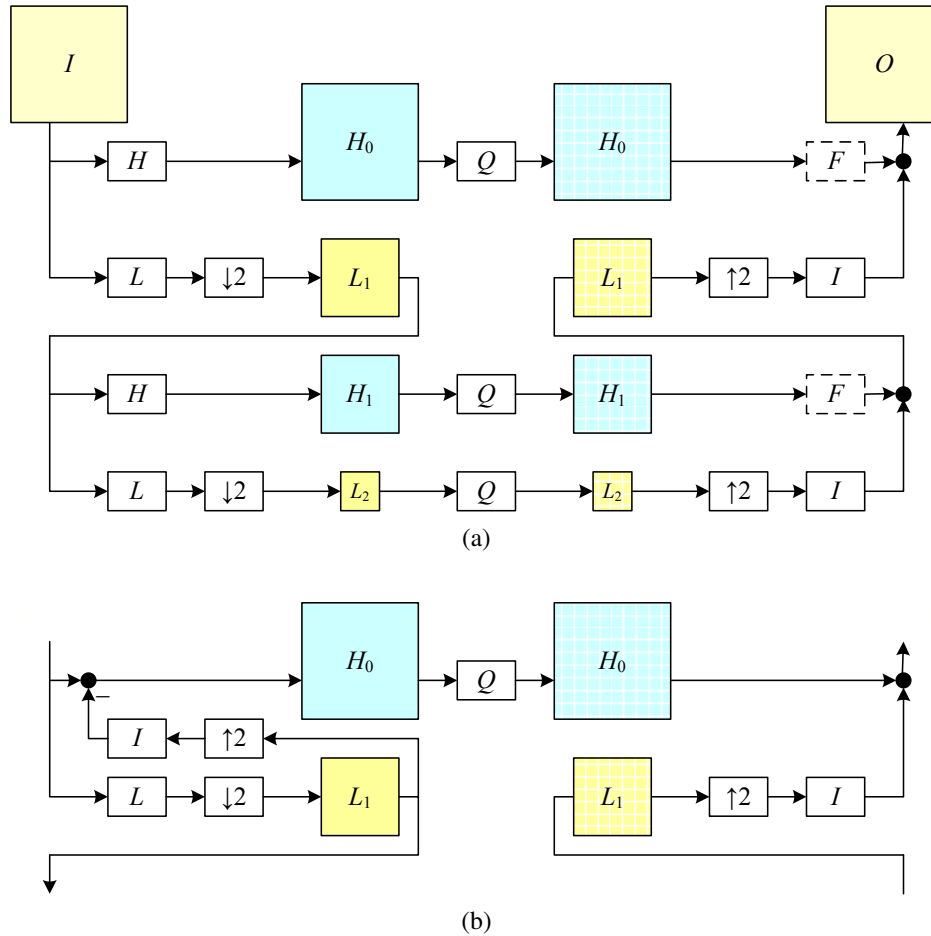


**Figure 3.32** A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level.

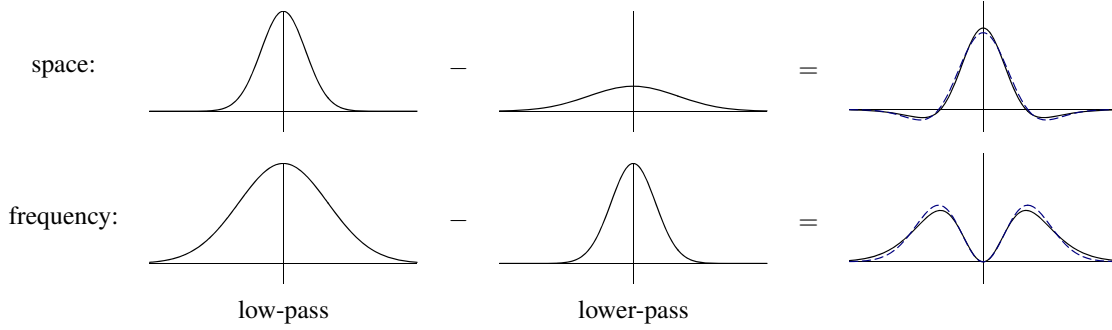


**Figure 3.33** The Gaussian pyramid shown as a signal processing diagram: The (a) analysis and (b) re-synthesis stages are shown as using similar computations. The white circles indicate zero values inserted by the  $\uparrow 2$  upsampling operation. Notice how the reconstruction filter coefficients are twice the analysis coefficients. The computation is shown as flowing down the page, regardless of whether we are going from coarse to fine or *vice versa*.





**Figure 3.34** The Laplacian pyramid: (a) The conceptual flow of images through processing stages: images are high-pass and low-pass filtered, and the low-pass filtered images are processed in the next stage of the pyramid. During reconstruction, the interpolated image and the (optionally filtered) high-pass image are added back together. The  $Q$  box indicates quantization or some other pyramid processing, e.g., noise removal by *coring* (setting small wavelet values to 0). (b) The actual computation of the high-pass filter involves first interpolating the downsampled low-pass image and then subtracting it. This results in perfect reconstruction when  $Q$  is the identity. The high-pass (or band-pass) images are typically called *Laplacian* images, while the low-pass images are called *Gaussian* images.



**Figure 3.35** The difference of two low-pass filters results in a band-pass filter. The dashed blue lines show the close fit to a half-octave Laplacian of Gaussian.

image, which can be stored away for further processing. The resulting pyramid has *perfect reconstruction*, i.e., the Laplacian images plus the base-level Gaussian ( $L_2$  in Figure 3.34b) are sufficient to exactly reconstruct the original image. Figure 3.33 shows the same computation in one dimension as a signal processing diagram, which completely captures the computations being performed during the analysis and re-synthesis stages.

Burt and Adelson also describe a variant on the Laplacian pyramid, where the low-pass image is taken from the original blurred image rather than the reconstructed pyramid (piping the output of the  $L$  box directly to the subtraction in Figure 3.34b). This variant has less aliasing, since it avoids one downsampling and upsampling round-trip, but it is not self-inverting, since the Laplacian images are no longer adequate to reproduce the original image.

As with the Gaussian pyramid, the term Laplacian is a bit of a misnomer, since their band-pass images are really differences of (approximate) Gaussians, or DoGs,

$$\text{DoG}\{I; \sigma_1, \sigma_2\} = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I. \quad (3.84)$$

A Laplacian of Gaussian (which we saw in (3.26)) is actually its second derivative,

$$\text{LoG}\{I; \sigma\} = \nabla^2(G_{\sigma} * I) = (\nabla^2 G_{\sigma}) * I, \quad (3.85)$$

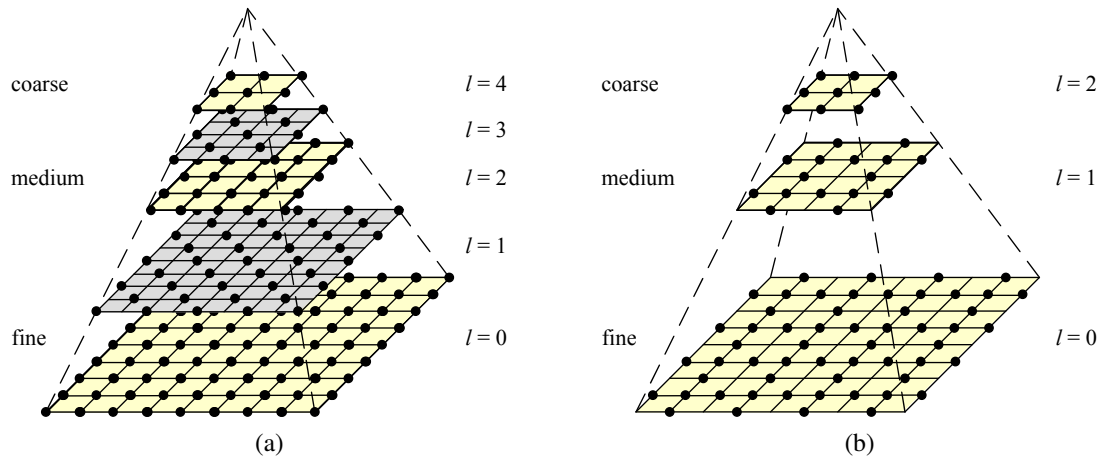
where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \quad (3.86)$$

is the Laplacian (operator) of a function. Figure 3.35 shows how the Differences of Gaussian and Laplacians of Gaussian look in both space and frequency.

Laplacians of Gaussian have elegant mathematical properties, which have been widely studied in the *scale-space* community (Witkin 1983; Witkin, Terzopoulos, and Kass 1986; Lindeberg 1990; Nielsen, Florack, and Deriche 1997) and can be used for a variety of applications including edge detection (Marr and Hildreth 1980; Perona and Malik 1990b), stereo matching (Witkin, Terzopoulos, and Kass 1987), and image enhancement (Nielsen, Florack, and Deriche 1997).

A less widely used variant is *half-octave pyramids*, shown in Figure 3.36a. These were first introduced to the vision community by Crowley and Stern (1984), who call them *Difference of Low-Pass* (DOLP) transforms. Because of the small scale change between adja-



**Figure 3.36** Multiresolution pyramids: (a) pyramid with half-octave (*quincunx*) sampling (odd levels are colored gray for clarity). (b) wavelet pyramid—each wavelet level stores 3/4 of the original pixels (usually the horizontal, vertical, and mixed gradients), so that the total number of wavelet coefficients and original pixels is the same.

cent levels, the authors claim that coarse-to-fine algorithms perform better. In the image-processing community, half-octave pyramids combined with checkerboard sampling grids are known as *quincunx* sampling (Feilner, Van De Ville, and Unser 2005). In detecting multi-scale features (Section 4.1.1), it is often common to use half-octave or even quarter-octave pyramids (Lowe 2004; Triggs 2004). However, in this case, the subsampling only occurs at every octave level, i.e., the image is repeatedly blurred with wider Gaussians until a full octave of resolution change has been achieved (Figure 4.11).

### 3.5.4 Wavelets

While pyramids are used extensively in computer vision applications, some people use *wavelet* decompositions as an alternative. Wavelets are filters that localize a signal in both space and frequency (like the Gabor filter in Table 3.2) and are defined over a hierarchy of scales. Wavelets provide a smooth way to decompose a signal into frequency components without blocking and are closely related to pyramids.

Wavelets were originally developed in the applied math and signal processing communities and were introduced to the computer vision community by Mallat (1989). Strang (1989); Simoncelli and Adelson (1990b); Rioul and Vetterli (1991); Chui (1992); Meyer (1993) all provide nice introductions to the subject along with historical reviews, while Chui (1992) provides a more comprehensive review and survey of applications. Sweldens (1997) describes the more recent *lifting* approach to wavelets that we discuss shortly.

Wavelets are widely used in the computer graphics community to perform multi-resolution geometric processing (Stollnitz, DeRose, and Salesin 1996) and have also been used in computer vision for similar applications (Szeliski 1990b; Pentland 1994; Gortler and Cohen 1995; Yaou and Chang 1994; Lai and Vemuri 1997; Szeliski 2006b), as well as for multi-scale oriented filtering (Simoncelli, Freeman, Adelson *et al.* 1992) and denoising (Portilla, Strela, Wainwright *et al.* 2003).