# Chapter 3

# Image processing




R. Szeliski, *Computer Vision: Algorithms and Applications*, Texts in Computer Science,
DOI 10.1007/978-1-84882-935-0_3, © Springer-Verlag London Limited 2011

(a) (b) (c) (d) (e) (f)

**Figure 3.1** Some common image processing operations: (a) original image; (b) increased contrast; (c) change in hue; (d) "posterized" (quantized colors); (e) blurred; (f) rotated.

Now that we have seen how images are formed through the interaction of 3D scene elements, lighting, and camera optics and sensors, let us look at the first stage in most computer vision applications, namely the use of image processing to preprocess the image and convert it into a form suitable for further analysis. Examples of such operations include exposure correction and color balancing, the reduction of image noise, increasing sharpness, or straightening the image by rotating it (Figure 3.1). While some may consider image processing to be outside the purview of computer vision, most computer vision applications, such as computational photography and even recognition, require care in designing the image processing stages in order to achieve acceptable results.

In this chapter, we review standard image processing operators that map pixel values from one image to another. Image processing is often taught in electrical engineering departments as a follow-on course to an introductory course in signal processing (Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999). There are several popular textbooks for image processing (Crane 1997; Gomes and Velho 1997; Jähne 1997; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzales and Woods 2008).

We begin this chapter with the simplest kind of image transforms, namely those that manipulate each pixel independently of its neighbors (Section 3.1). Such transforms are often called *point operators* or *point processes*. Next, we examine *neighborhood* (area-based) operators, where each new pixel's value depends on a small number of neighboring input values (Sections 3.2 and 3.3). A convenient tool to analyze (and sometimes accelerate) such neighborhood operations is the *Fourier Transform*, which we cover in Section 3.4. Neighborhood operators can be cascaded to form *image pyramids* and *wavelets*, which are useful for analyzing images at a variety of resolutions (scales) and for accelerating certain operations (Section 3.5). Another important class of global operators are *geometric transformations*, such as rotations, shears, and perspective deformations (Section 3.6). Finally, we introduce *global optimization* approaches to image processing, which involve the minimization of an energy functional or, equivalently, optimal estimation using Bayesian *Markov random field* models (Section 3.7).

## 3.1 Point operators

The simplest kinds of image processing transforms are *point operators*, where each output pixel's value depends on only the corresponding input pixel value (plus, potentially, some globally collected information or parameters). Examples of such operators include brightness and contrast adjustments (Figure 3.2) as well as color correction and transformations. In the image processing literature, such operations are also known as *point processes* (Crane 1997).

We begin this section with a quick review of simple point operators such as brightness scaling and image addition. Next, we discuss how colors in images can be manipulated. We then present *image compositing* and *matting* operations, which play an important role in computational photography (Chapter 10) and computer graphics applications. Finally, we describe the more global process of *histogram equalization*. We close with an example application that manipulates *tonal values* (exposure and contrast) to improve image appearance.

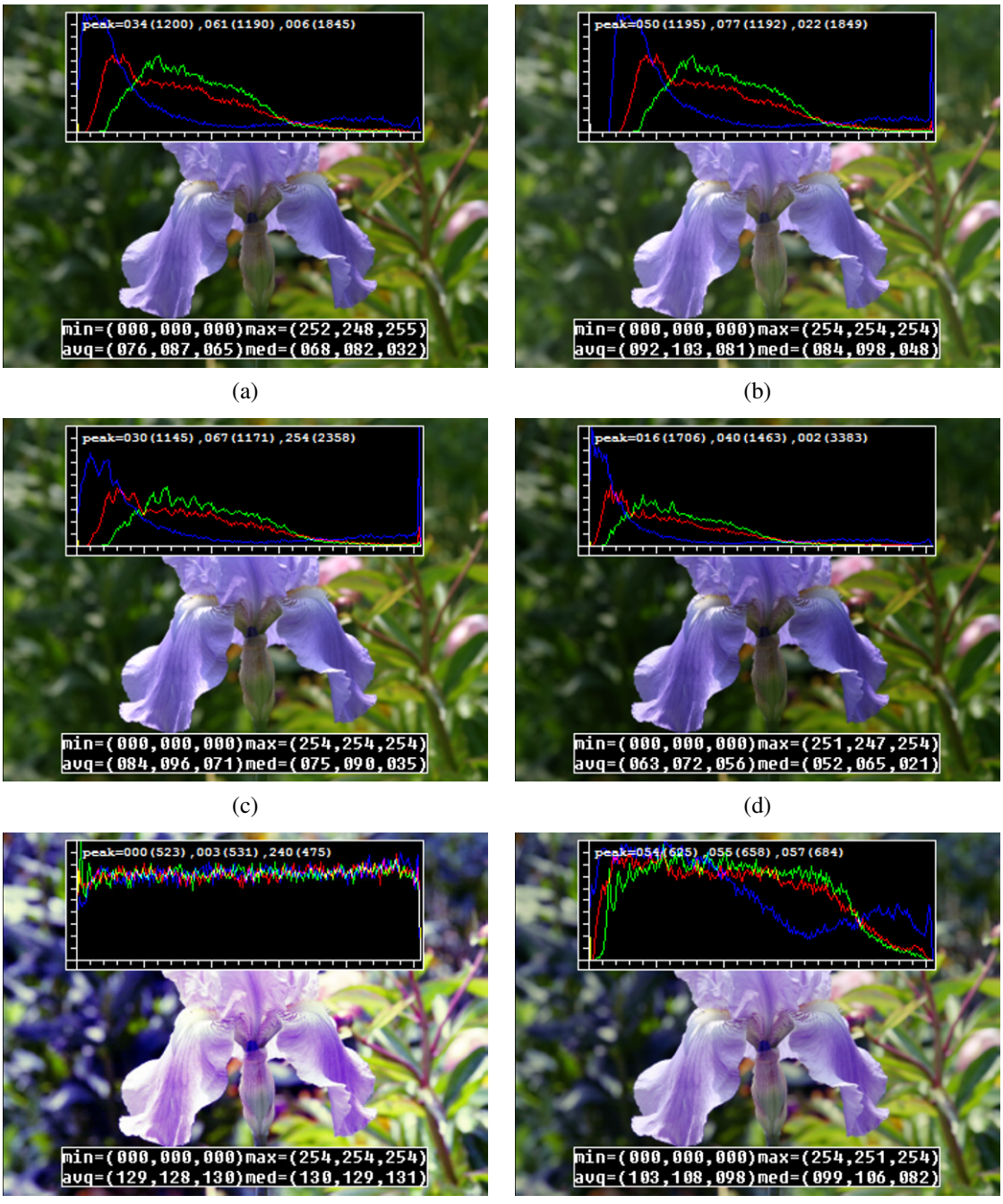
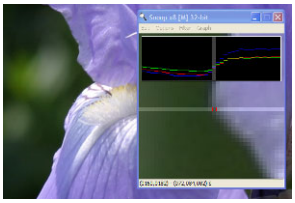



**Figure 3.2** Some local image processing operations: (a) original image along with its three color (per-channel) histograms; (b) brightness increased (additive offset, $b = 16$); (c) contrast increased (multiplicative gain, $a = 1.1$); (d) gamma (partially) linearized ($\gamma = 1.2$); (e) full histogram equalization; (f) partial histogram equalization.

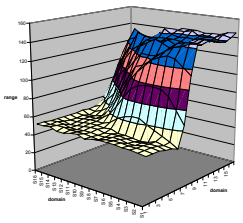
(a)



(b)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 45 | 60 | 98 | 127 | 132 | 133 | 137 | 133 |
| 46 | 65 | 98 | 123 | 126 | 128 | 131 | 133 |
| 47 | 65 | 96 | 115 | 119 | 123 | 135 | 137 |
| 47 | 63 | 91 | 107 | 113 | 122 | 138 | 134 |
| 50 | 59 | 80 | 97 | 110 | 123 | 133 | 134 |
| 49 | 53 | 68 | 83 | 97 | 113 | 128 | 133 |
| 50 | 50 | 58 | 70 | 84 | 102 | 116 | 126 |
| 50 | 50 | 52 | 58 | 69 | 86 | 101 | 120 |

(c)



(d)

**Figure 3.3** Visualizing image data: (a) original image; (b) cropped portion and scanline plot using an image inspection tool; (c) grid of numbers; (d) surface plot. For figures (c)–(d), the image was first converted to grayscale.

### 3.1.1 Pixel transforms

A general image processing *operator* is a function that takes one or more input images and produces an output image. In the continuous domain, this can be denoted as

$$g(\boldsymbol{x}) = h(f(\boldsymbol{x})) \quad \text{or} \quad g(\boldsymbol{x}) = h(f_0(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x})), \tag{3.1}$$

where $\boldsymbol{x}$ is in the D-dimensional *domain* of the functions (usually $D = 2$ for images) and the functions $f$ and $g$ operate over some *range*, which can either be scalar or vector-valued, e.g., for color images or 2D motion. For discrete (sampled) images, the domain consists of a finite number of *pixel locations*, $\boldsymbol{x} = (i, j)$, and we can write

$$g(i, j) = h(f(i, j)). \tag{3.2}$$

Figure 3.3 shows how an image can be represented either by its color (appearance), as a grid of numbers, or as a two-dimensional function (surface plot).

Two commonly used point processes are multiplication and addition with a constant,

$$g(\boldsymbol{x}) = af(\boldsymbol{x}) + b. \tag{3.3}$$

The parameters $a > 0$ and $b$ are often called the *gain* and *bias* parameters; sometimes these parameters are said to control *contrast* and *brightness*, respectively (Figures 3.2b–c).[1] The bias and gain parameters can also be spatially varying,

$$g(\boldsymbol{x}) = a(\boldsymbol{x})f(\boldsymbol{x}) + b(\boldsymbol{x}), \tag{3.4}$$

e.g., when simulating the *graded density filter* used by photographers to selectively darken the sky or when modeling vignetting in an optical system.

Multiplicative gain (both global and spatially varying) is a *linear* operation, since it obeys the *superposition principle*,

$$h(f_0 + f_1) = h(f_0) + h(f_1). \tag{3.5}$$

(We will have more to say about linear shift invariant operators in Section 3.2.) Operators such as image squaring (which is often used to get a local estimate of the *energy* in a band-pass filtered signal, see Section 3.5) are not linear.

[1] An image's luminance characteristics can also be summarized by its *key* (average luminanance) and *range* (Kopf, Uyttendaele, Deussen *et al.* 2007).

Another commonly used *dyadic* (two-input) operator is the *linear blend* operator,

$$g(\boldsymbol{x}) = (1 - \alpha)f_0(\boldsymbol{x}) + \alpha f_1(\boldsymbol{x}). \tag{3.6}$$

By varying $\alpha$ from $0 \rightarrow 1$, this operator can be used to perform a temporal *cross-dissolve* between two images or videos, as seen in slide shows and film production, or as a component of image *morphing* algorithms (Section 3.6.3).

One highly used non-linear transform that is often applied to images before further processing is *gamma correction*, which is used to remove the non-linear mapping between input radiance and quantized pixel values (Section 2.3.2). To invert the gamma mapping applied by the sensor, we can use

$$g(\boldsymbol{x}) = [f(\boldsymbol{x})]^{1/\gamma}, \tag{3.7}$$

where a gamma value of $\gamma \approx 2.2$ is a reasonable fit for most digital cameras.

### 3.1.2 Color transforms

While color images can be treated as arbitrary vector-valued functions or collections of independent bands, it usually makes sense to think about them as highly correlated signals with strong connections to the image formation process (Section 2.2), sensor design (Section 2.3), and human perception (Section 2.3.2). Consider, for example, brightening a picture by adding a constant value to all three channels, as shown in Figure 3.2b. Can you tell if this achieves the desired effect of making the image look brighter? Can you see any undesirable side-effects or artifacts?

In fact, adding the same value to each color channel not only increases the apparent *intensity* of each pixel, it can also affect the pixel's *hue* and *saturation*. How can we define and manipulate such quantities in order to achieve the desired perceptual effects?

As discussed in Section 2.3.2, chromaticity coordinates (2.104) or even simpler color ratios (2.116) can first be computed and then used after manipulating (e.g., brightening) the luminance $Y$ to re-compute a valid RGB image with the same hue and saturation. Figure 2.32g–i shows some color ratio images multiplied by the middle gray value for better visualization.

Similarly, color balancing (e.g., to compensate for incandescent lighting) can be performed either by multiplying each channel with a different scale factor or by the more complex process of mapping to XYZ color space, changing the nominal white point, and mapping back to RGB, which can be written down using a linear $3 \times 3$ *color twist* transform matrix. Exercises 2.9 and 3.1 have you explore some of these issues.

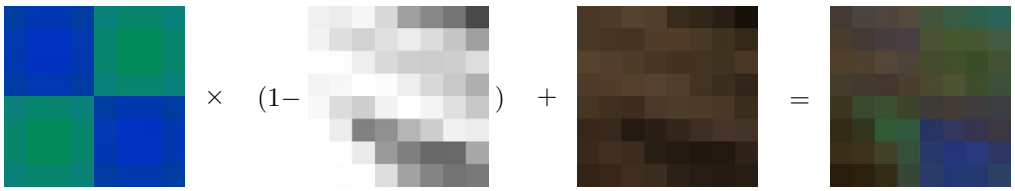
Another fun project, best attempted after you have mastered the rest of the material in this chapter, is to take a picture with a rainbow in it and enhance the strength of the rainbow (Exercise 3.29).

### 3.1.3 Compositing and matting

In many photo editing and visual effects applications, it is often desirable to cut a *foreground* object out of one scene and put it on top of a different *background* (Figure 3.4). The process of extracting the object from the original image is often called *matting* (Smith and Blinn

(a) (b) (c) (d)

**Figure 3.4** Image matting and compositing (Chuang, Curless, Salesin *et al.* 2001) © 2001 IEEE: (a) source image; (b) extracted foreground object $F$; (c) alpha matte $\alpha$ shown in grayscale; (d) new composite $C$.




$B$ $\alpha$ $\alpha F$ $C$

(a) (b) (c) (d)

**Figure 3.5** Compositing equation $C = (1 - \alpha)B + \alpha F$. The images are taken from a close-up of the region of the hair in the upper right part of the lion in Figure 3.4.

1996), while the process of inserting it into another image (without visible artifacts) is called *compositing* (Porter and Duff 1984; Blinn 1994a).

The intermediate representation used for the foreground object between these two stages is called an *alpha-matted color image* (Figure 3.4b–c). In addition to the three color RGB channels, an alpha-matted image contains a fourth *alpha* channel $\alpha$ (or A) that describes the relative amount of *opacity* or *fractional coverage* at each pixel (Figures 3.4c and 3.5b). The opacity is the opposite of the *transparency*. Pixels within the object are fully opaque ($\alpha = 1$), while pixels fully outside the object are transparent ($\alpha = 0$). Pixels on the boundary of the object vary smoothly between these two extremes, which hides the perceptual visible *jaggies* that occur if only binary opacities are used.

To composite a new (or foreground) image on top of an old (background) image, the *over operator*, first proposed by Porter and Duff (1984) and then studied extensively by Blinn (1994a; 1994b), is used,

$$C = (1 - \alpha)B + \alpha F. \tag{3.8}$$

This operator *attenuates* the influence of the background image $B$ by a factor $(1 - \alpha)$ and then adds in the color (and opacity) values corresponding to the foreground layer $F$, as shown in Figure 3.5.

In many situations, it is convenient to represent the foreground colors in *pre-multiplied* form, i.e., to store (and manipulate) the $\alpha F$ values directly. As Blinn (1994b) shows, the pre-multiplied RGBA representation is preferred for several reasons, including the ability to blur or resample (e.g., rotate) alpha-matted images without any additional complications (just treating each RGBA band independently). However, when matting using local color consistency (Ruzon and Tomasi 2000; Chuang, Curless, Salesin *et al.* 2001), the pure un-

**Figure 3.6** An example of light reflecting off the transparent glass of a picture frame (Black and Anandan 1996) © 1996 Elsevier. You can clearly see the woman's portrait inside the picture frame superimposed with the reflection of a man's face off the glass.

multiplied foreground colors $F$ are used, since these remain constant (or vary slowly) in the vicinity of the object edge.

The over operation is not the only kind of compositing operation that can be used. Porter and Duff (1984) describe a number of additional operations that can be useful in photo editing and visual effects applications. In this book, we concern ourselves with only one additional, commonly occurring case (but see Exercise 3.2).

When light reflects off clean transparent glass, the light passing through the glass and the light reflecting off the glass are simply added together (Figure 3.6). This model is useful in the analysis of *transparent motion* (Black and Anandan 1996; Szeliski, Avidan, and Anandan 2000), which occurs when such scenes are observed from a moving camera (see Section 8.5.2).

The actual process of *matting*, i.e., recovering the foreground, background, and alpha matte values from one or more images, has a rich history, which we study in Section 10.4. Smith and Blinn (1996) have a nice survey of traditional *blue-screen matting* techniques, while Toyama, Krumm, Brumitt *et al.* (1999) review *difference matting*. More recently, there has been a lot of activity in computational photography relating to *natural image matting* (Ruzon and Tomasi 2000; Chuang, Curless, Salesin *et al.* 2001; Wang and Cohen 2007a), which attempts to extract the mattes from a single natural image (Figure 3.4a) or from extended video sequences (Chuang, Agarwala, Curless *et al.* 2002). All of these techniques are described in more detail in Section 10.4.

### 3.1.4 Histogram equalization

While the brightness and gain controls described in Section 3.1.1 can improve the appearance of an image, how can we automatically determine their best values? One approach might be to look at the darkest and brightest pixel values in an image and map them to pure black and pure white. Another approach might be to find the *average* value in the image, push it towards middle gray, and expand the *range* so that it more closely fills the displayable values (Kopf, Uyttendaele, Deussen *et al.* 2007).

How can we visualize the set of lightness values in an image in order to test some of

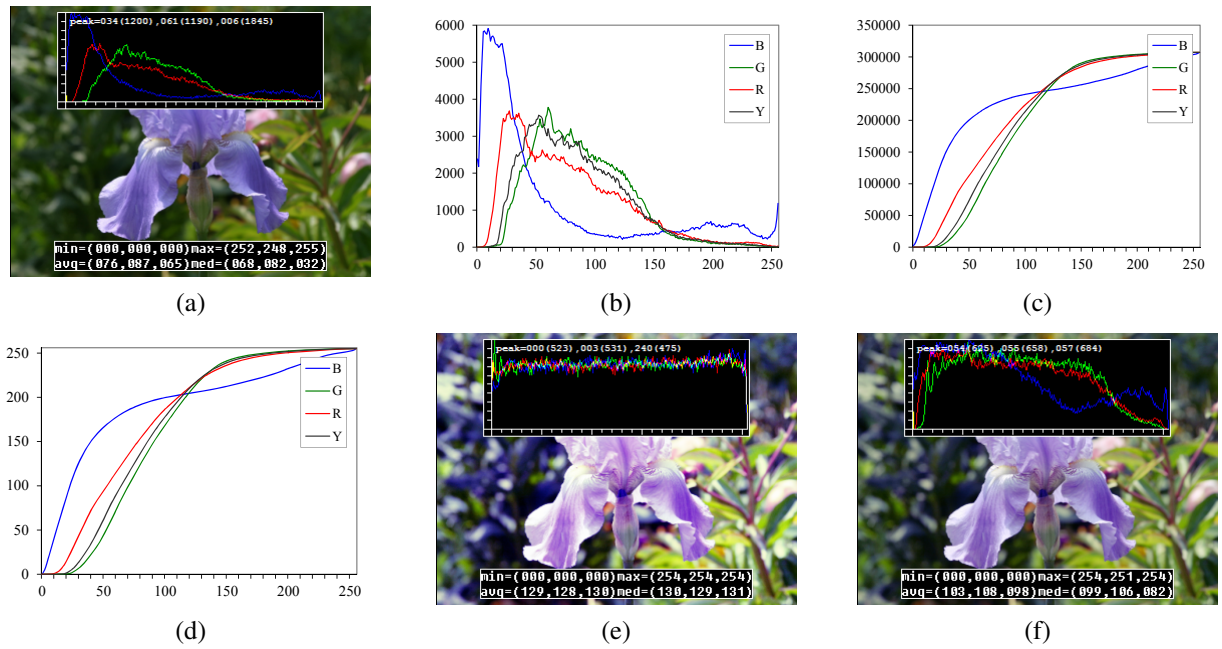



**Figure 3.7** Histogram analysis and equalization: (a) original image (b) color channel and intensity (luminance) histograms; (c) cumulative distribution functions; (d) equalization (transfer) functions; (e) full histogram equalization; (f) partial histogram equalization.

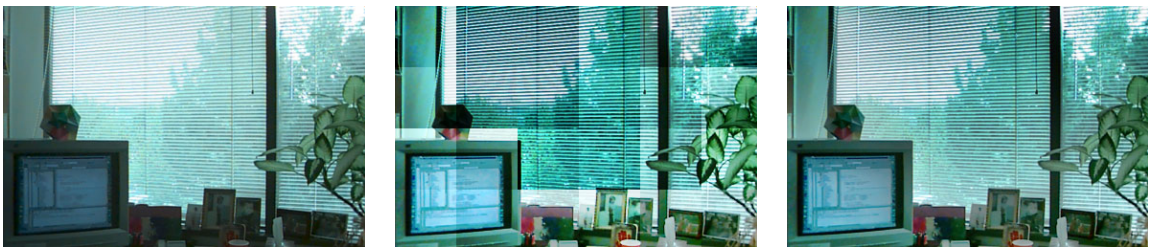
these heuristics? The answer is to plot the *histogram* of the individual color channels and luminance values, as shown in Figure 3.7b.[2] From this distribution, we can compute relevant statistics such as the minimum, maximum, and average intensity values. Notice that the image in Figure 3.7a has both an excess of dark values and light values, but that the mid-range values are largely under-populated. Would it not be better if we could simultaneously brighten some dark values and darken some light values, while still using the full extent of the available dynamic range? Can you think of a mapping that might do this?

One popular answer to this question is to perform *histogram equalization*, i.e., to find an intensity mapping function $f(I)$ such that the resulting histogram is flat. The trick to finding such a mapping is the same one that people use to generate random samples from a *probability density function*, which is to first compute the *cumulative distribution function* shown in Figure 3.7c.

Think of the original histogram $h(I)$ as the distribution of grades in a class after some exam. How can we map a particular grade to its corresponding *percentile*, so that students at the 75% percentile range scored better than $^3/_4$ of their classmates? The answer is to integrate the distribution $h(I)$ to obtain the cumulative distribution $c(I)$,

$$c(I) = \frac{1}{N}\sum_{i=0}^{I} h(i) = c(I-1) + \frac{1}{N}h(I), \tag{3.9}$$

[2] The histogram is simply the *count* of the number of pixels at each gray level value. For an eight-bit image, an accumulation table with 256 entries is needed. For higher bit depths, a table with the appropriate number of entries (probably fewer than the full number of gray levels) should be used.



(a) (b) (c)

**Figure 3.8** Locally adaptive histogram equalization: (a) original image; (b) block histogram equalization; (c) full locally adaptive equalization.

where $N$ is the number of pixels in the image or students in the class. For any given grade or intensity, we can look up its corresponding percentile $c(I)$ and determine the final value that pixel should take. When working with eight-bit pixel values, the $I$ and $c$ axes are rescaled from $[0, 255]$.

Figure 3.7d shows the result of applying $f(I) = c(I)$ to the original image. As we can see, the resulting histogram is flat; so is the resulting image (it is "flat" in the sense of a lack of contrast and being muddy looking). One way to compensate for this is to only *partially* compensate for the histogram unevenness, e.g., by using a mapping function $f(I) = \alpha c(I) + (1 - \alpha)I$, which is a linear blend between the cumulative distribution function and the identity transform (a straight line). As you can see in Figure 3.7e, the resulting image maintains more of its original grayscale distribution while having a more appealing balance.

Another potential problem with histogram equalization (or, in general, image brightening) is that noise in dark regions can be amplified and become more visible. Exercise 3.6 suggests some possible ways to mitigate this, as well as alternative techniques to maintain contrast and "punch" in the original images (Larson, Rushmeier, and Piatko 1997; Stark 2000).

### Locally adaptive histogram equalization

While global histogram equalization can be useful, for some images it might be preferable to apply different kinds of equalization in different regions. Consider for example the image in Figure 3.8a, which has a wide range of luminance values. Instead of computing a single curve, what if we were to subdivide the image into $M \times M$ pixel blocks and perform separate histogram equalization in each sub-block? As you can see in Figure 3.8b, the resulting image exhibits a lot of blocking artifacts, i.e., intensity discontinuities at block boundaries.

One way to eliminate blocking artifacts is to use a *moving window*, i.e., to recompute the histogram for every $M \times M$ block centered at each pixel. This process can be quite slow ($M^2$ operations per pixel), although with clever programming only the histogram entries corresponding to the pixels entering and leaving the block (in a raster scan across the image) need to be updated ($M$ operations per pixel). Note that this operation is an example of the *non-linear neighborhood operations* we study in more detail in Section 3.3.1.

A more efficient approach is to compute non-overlapped block-based equalization functions as before, but to then smoothly interpolate the transfer functions as we move between blocks. This technique is known as *adaptive histogram equalization* (AHE) and its contrast-

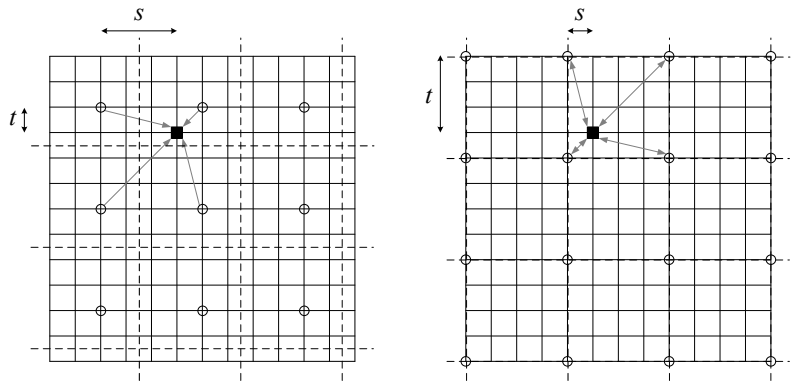



**Figure 3.9** Local histogram interpolation using relative $(s, t)$ coordinates: (a) block-based histograms, with block centers shown as circles; (b) corner-based "spline" histograms. Pixels are located on grid intersections. The black square pixel's transfer function is interpolated from the four adjacent lookup tables (gray arrows) using the computed $(s, t)$ values. Block boundaries are shown as dashed lines.

limited (gain-limited) version is known as CLAHE (Pizer, Amburn, Austin *et al.* 1987).[3] The weighting function for a given pixel $(i, j)$ can be computed as a function of its horizontal and vertical position $(s, t)$ within a block, as shown in Figure 3.9a. To blend the four lookup functions $\{f_{00}, \ldots, f_{11}\}$, a *bilinear* blending function,

$$f_{s,t}(I) = (1-s)(1-t)f_{00}(I) + s(1-t)f_{10}(I) + (1-s)tf_{01}(I) + stf_{11}(I) \quad (3.10)$$

can be used. (See Section 3.5.2 for higher-order generalizations of such *spline* functions.) Note that instead of blending the four lookup tables for each output pixel (which would be quite slow), we can instead blend the results of mapping a given pixel through the four neighboring lookups.

A variant on this algorithm is to place the lookup tables at the *corners* of each $M \times M$ block (see Figure 3.9b and Exercise 3.7). In addition to blending four lookups to compute the final value, we can also *distribute* each input pixel into four adjacent lookup tables during the histogram accumulation phase (notice that the gray arrows in Figure 3.9b point both ways), i.e.,

$$h_{k,l}(I(i, j)) \mathrel{+}= w(i, j, k, l), \quad (3.11)$$

where $w(i, j, k, l)$ is the bilinear weighting function between pixel $(i, j)$ and lookup table $(k, l)$. This is an example of *soft histogramming*, which is used in a variety of other applications, including the construction of SIFT feature descriptors (Section 4.1.3) and vocabulary trees (Section 14.3.2).

### 3.1.5 *Application*: Tonal adjustment

One of the most widely used applications of point-wise image processing operators is the manipulation of contrast or *tone* in photographs, to make them look either more attractive or

[3] This algorithm is implemented in the MATLAB `adapthist` function.

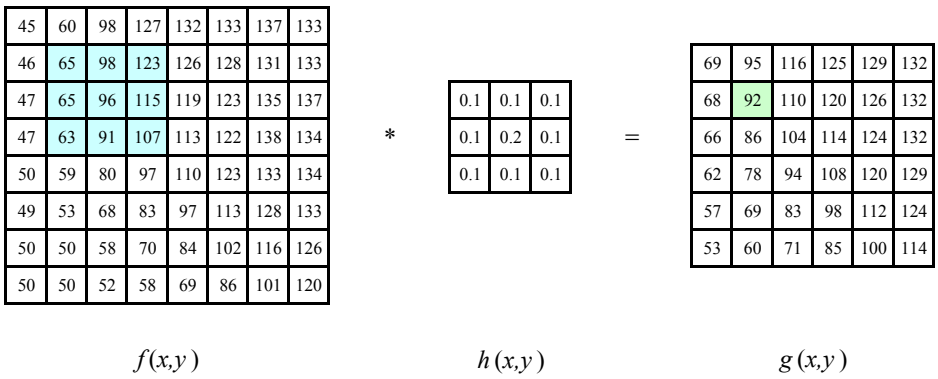



**Figure 3.10** Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.

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),

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

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

$$g = f \otimes h. \tag{3.13}$$

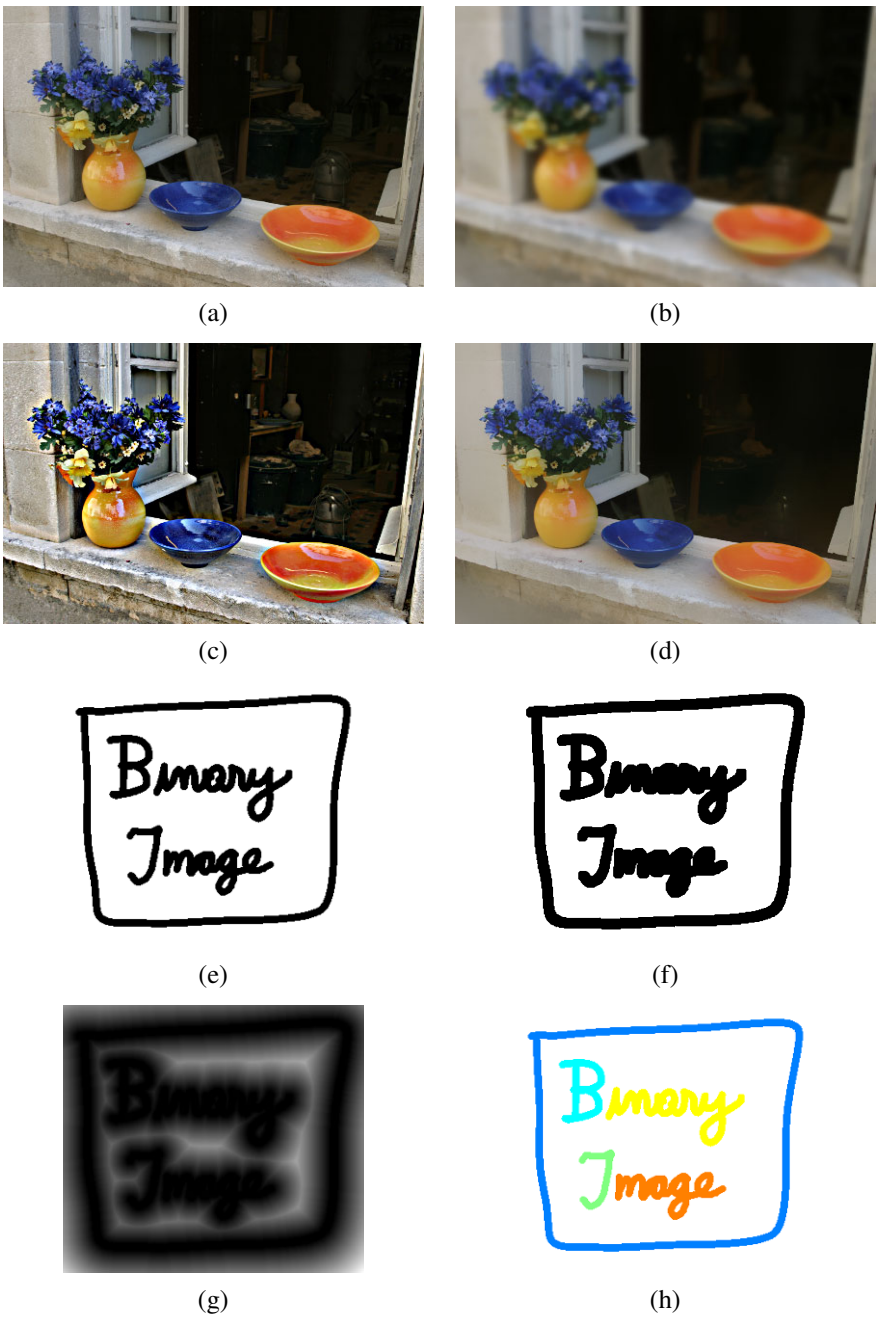



**Figure 3.11** Some neighborhood operations: (a) original image; (b) blurred; (c) sharpened; (d) smoothed with edge-preserving filter; (e) binary image; (f) dilated; (g) distance transform; (h) connected components. For the dilation and connected components, black (ink) pixels are assumed to be active,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

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

**Figure 3.12** One-dimensional signal convolution as a sparse matrix-vector multiply, $\boldsymbol{g} = \boldsymbol{H}\boldsymbol{f}$.

A common variant on this formula is

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

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

$$g = f * h, \tag{3.15}$$

and $h$ is then called the *impulse response function*.[4] 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),

$$h \circ (f_0 + f_1) = h \circ f_0 + h \circ f_1, \tag{3.16}$$

and the *shift invariance* principle,

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

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.,

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

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 $\boldsymbol{f}$ and $\boldsymbol{g}$,

$$\boldsymbol{g} = \boldsymbol{H}\boldsymbol{f}, \tag{3.19}$$

[4] The continuous version of convolution can be written as $g(\boldsymbol{x}) = \int f(\boldsymbol{x} - \boldsymbol{u})h(\boldsymbol{u})d\boldsymbol{u}$.

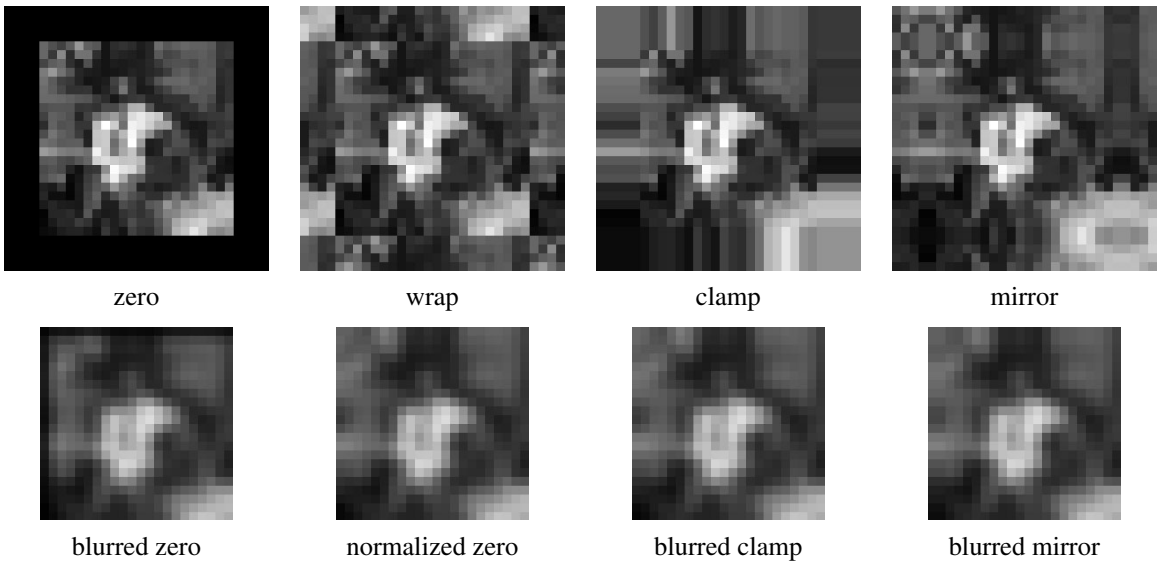



**Figure 3.13** Border padding (top row) and the results of blurring the padded image (bottom row). The normalized zero image is the result of dividing (normalizing) the blurred zero-padded RGBA image by its corresponding soft alpha value.

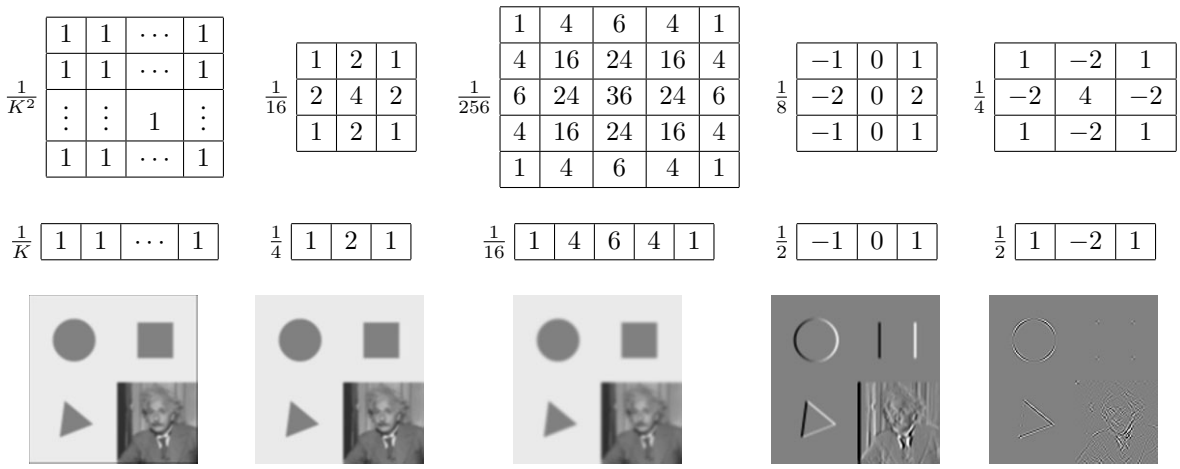
where the (sparse) $\boldsymbol{H}$ matrix contains the convolution kernels. Figure 3.12 shows how a one-dimensional convolution can be represented in matrix-vector form.

### 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.



(a) box, $K = 5$ (b) bilinear (c) "Gaussian" (d) Sobel (e) corner

**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 $\boldsymbol{K}$ corresponding to successive convolution with a horizontal kernel $\boldsymbol{h}$ and a vertical kernel $\boldsymbol{v}$ is the *outer product* of the two kernels,

$$\boldsymbol{K} = \boldsymbol{v}\boldsymbol{h}^T \tag{3.20}$$

(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 $\boldsymbol{K}$ 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 $\boldsymbol{K}$ and to take its singular value decomposition (SVD),

$$\boldsymbol{K} = \sum_i \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T \tag{3.21}$$

(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 $\sqrt{\sigma_0}\boldsymbol{u}_0$ and $\sqrt{\sigma_0}\boldsymbol{v}_0^T$ 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,

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

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 misfocusing, then overlay the two negatives before printing the final image, which corresponds to

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

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,

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

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,

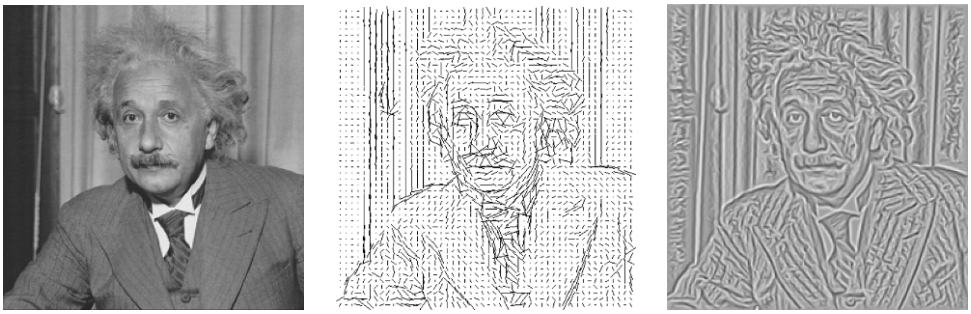
$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 y}{\partial y^2}, \tag{3.25}$$

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,

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

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 obtained by smoothing with a Gaussian (or some other filter) and then taking a



(a) (b) (c)

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

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

The smoothed directional derivative filter,

$$G_{\hat{\boldsymbol{u}}} = uG_x + vG_y = u\frac{\partial G}{\partial x} + v\frac{\partial G}{\partial y}, \tag{3.28}$$

where $\hat{\boldsymbol{u}} = (u, v)$, is an example of a *steerable* filter, since the value of an image convolved with $G_{\hat{\boldsymbol{u}}}$ 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 $\hat{\boldsymbol{u}}$ (Freeman and Adelson 1991). The advantage of this approach is that a whole *family* of filters can be evaluated with very little cost.

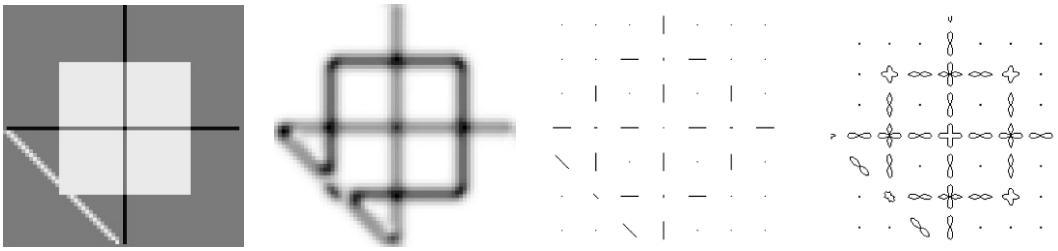
How about steering a directional second derivative filter $\nabla_{\hat{\boldsymbol{u}}} \cdot \nabla_{\hat{\boldsymbol{u}}} G_{\hat{\boldsymbol{u}}}$, which is the result of taking a (smoothed) directional derivative and then taking the directional derivative again? For example, $G_{xx}$ is the second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

At first glance, it would appear that the steering trick will not work, since for every direction $\hat{\boldsymbol{u}}$, 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,

$$G_{\hat{\boldsymbol{u}}\hat{\boldsymbol{u}}} = u^2 G_{xx} + 2uvG_{xy} + v^2 G_{yy}. \tag{3.29}$$

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



(a) (b) (c) (d)

**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,

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

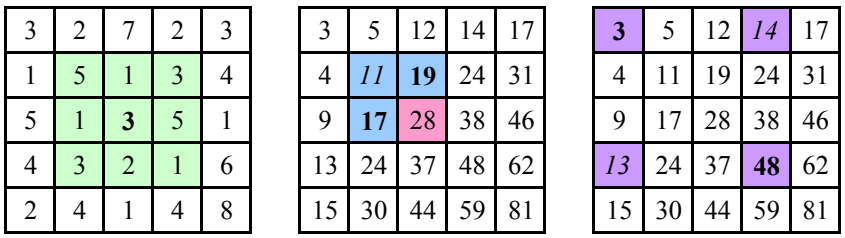
This can be efficiently computed using a recursive (raster-scan) algorithm,

$$s(i,j) = s(i-1,j) + s(i,j-1) - s(i-1,j-1) + f(i,j). \tag{3.31}$$

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,

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

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




(a) S = 24 (b) s = 28 (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).

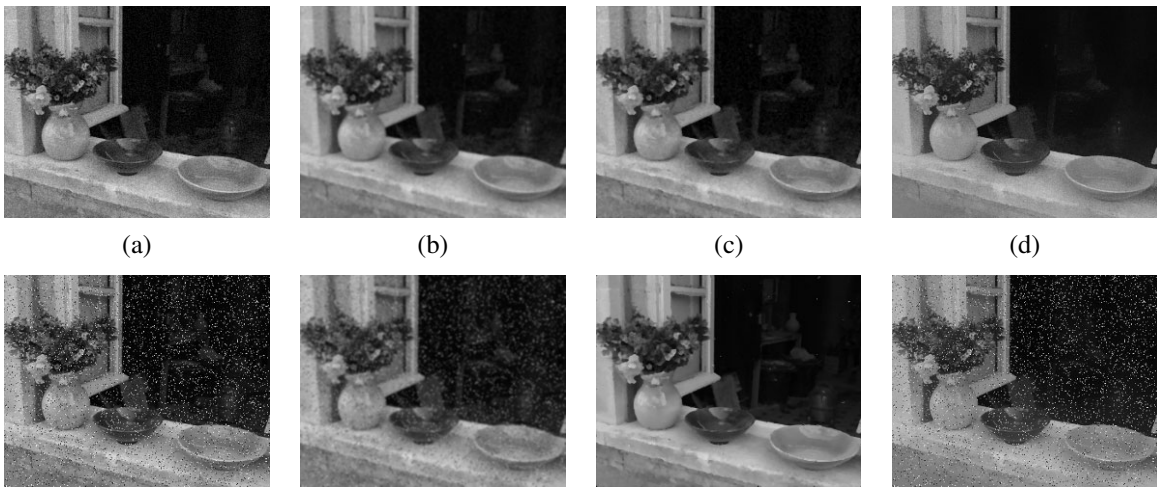
In computer vision, summed area tables have been used in face detection (Viola and Jones 2004) to compute simple multi-scale low-level features. Such features, which consist of adjacent rectangles of positive and negative values, are also known as *boxlets* (Simard, Bottou, Haffner *et al.* 1998). In 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.



(a) (b) (c) (d)

(e) (f) (g) (h)

**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).

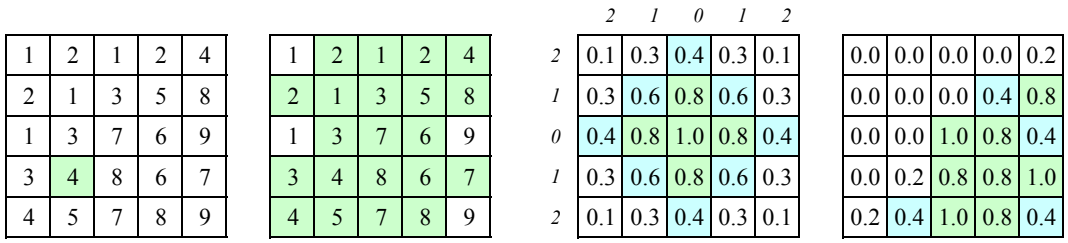
### 3.3.1 Non-linear filtering

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




(a) median = 4 (b) α-mean= 4.6 (c) domain filter (d) range filter

**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

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

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

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

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

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

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

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

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

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

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.[5]

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.

[5] 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).

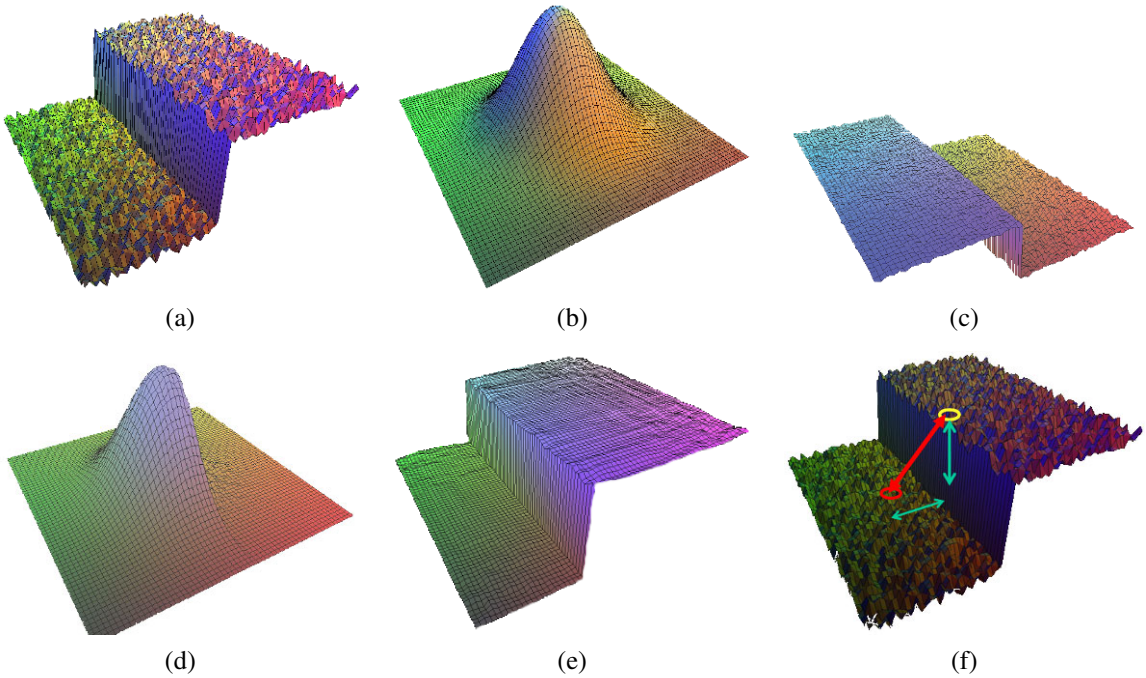



**Figure 3.20** Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.

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

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

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

We can thus re-write (3.34) as

$$f^{(t+1)}(i,j) = \frac{f^{(t)}(i,j)+\eta\sum_{k,l} f^{(t)}(k,l)r(i,j,k,l)}{1+\eta\sum_{k,l} r(i,j,k,l)} \tag{3.40}$$

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

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).[6] 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

[6] The $1/(1+\eta R)$ factor is not present in anisotropic diffusion but becomes negligible as $\eta \to 0$.

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 $r'(x) \to 0$ as $x \to \infty$. 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,

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

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

$$c = f \otimes s \tag{3.42}$$

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

(a) (b) (c) (d) (e) (f)

**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

$$d_1(k, l) = |k| + |l| \tag{3.43}$$

and the *Euclidean* distance

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

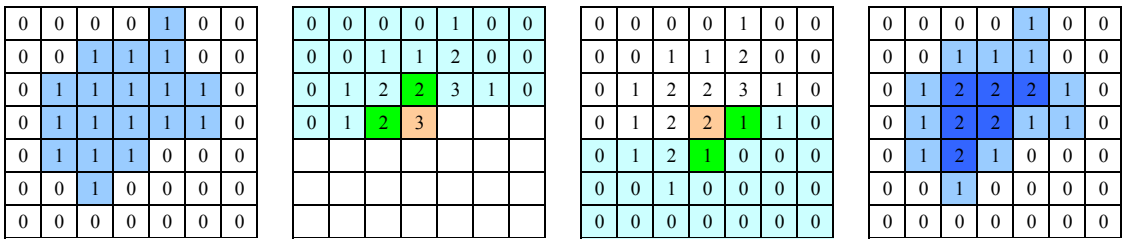



**Figure 3.22** City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange value; (d) final distance transform.

The distance transform is then defined as

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

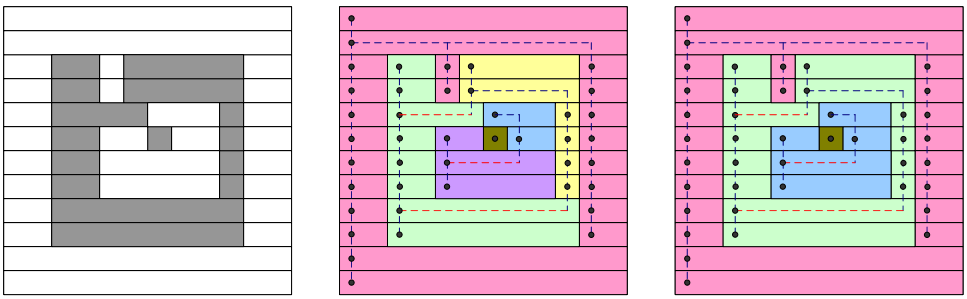
i.e., it is the distance to the *nearest* background pixel whose value is 0.

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



(a) (b) (c)

**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,

$$\boldsymbol{M} = \sum_{(x,y)\in\mathcal{R}} \begin{bmatrix} x-\overline{x} \\ y-\overline{y} \end{bmatrix} \begin{bmatrix} x-\overline{x} & y-\overline{y} \end{bmatrix}, \tag{3.46}$$

  from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis.[7]

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 Schafer (1996); Oppenheim, Schafer, 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

$$s(x) = \sin(2\pi f x + \phi_i) = \sin(\omega x + \phi_i) \tag{3.47}$$

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 Schafer 1996; Oppenheim, Schafer, and Buck 1999).

[7] Moments can also be computed using Green's theorem applied to the boundary pixels (Yang and Albregtsen 1996).

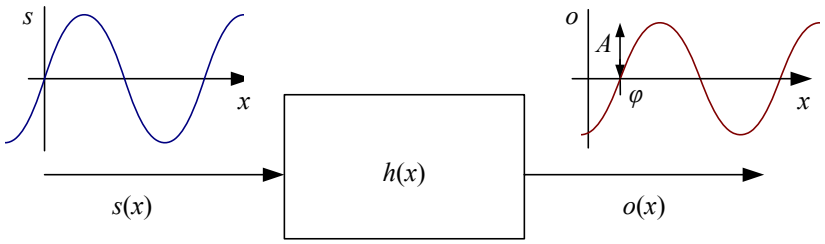



**Figure 3.24** The Fourier Transform as the response of a filter $h(x)$ to an input sinusoid $s(x) = e^{j\omega x}$ yielding an output sinusoid $o(x) = h(x) * s(x) = Ae^{j\omega x+\phi}$.

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 $\phi_o$,

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

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.[8] 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

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

In that case, we can simply write,

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

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

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

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

$$h(x) \overset{\mathcal{F}}{\leftrightarrow} H(\omega). \tag{3.52}$$

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,

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

[8] 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 | $f(ax)$ | | $1/aF(\omega/a)$ |
| 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,

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

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, Schafer, 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

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

has a sinc Fourier transform,

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

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.

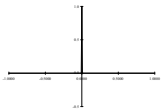
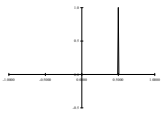
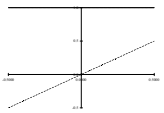
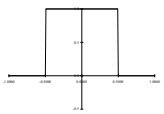
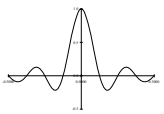
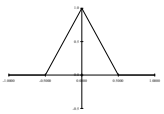
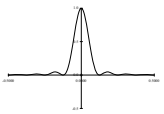
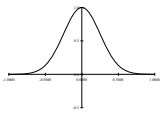
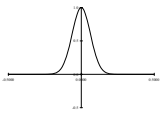
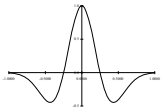
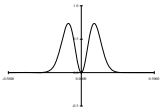
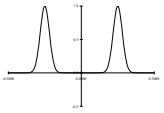
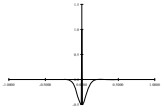
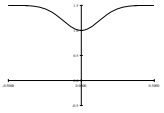
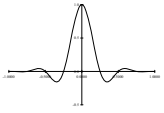
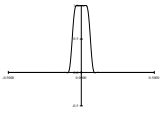
| Name | Signal | | Transform | |
|---|---|---|---|---|
| impulse |  | $\delta(x)$ | $\Leftrightarrow$ | $1$  |
| shifted impulse |  | $\delta(x-u)$ | $\Leftrightarrow$ | $e^{-j\omega u}$  |
| box filter |  | $\text{box}(x/a)$ | $\Leftrightarrow$ | $a\text{sinc}(a\omega)$  |
| tent |  | $\text{tent}(x/a)$ | $\Leftrightarrow$ | $a\text{sinc}^2(a\omega)$  |
| 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,

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

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

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

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

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

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

$$LoG(x;\sigma) = (\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2})G(x;\sigma) \tag{3.59}$$

  has a band-pass response of

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

  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,

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

  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).

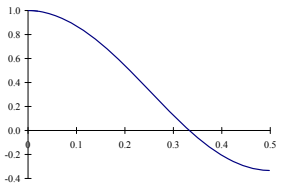
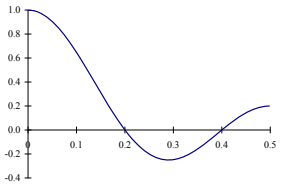
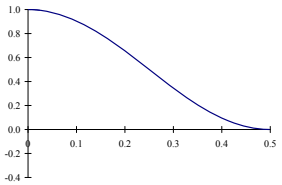
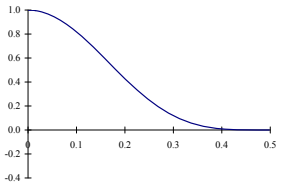
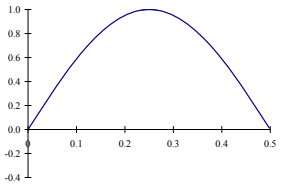
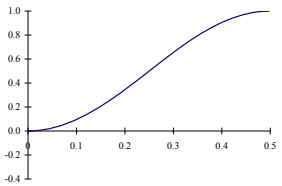
| Name | Kernel | Transform | Plot |
|---|---|---|---|
| box-3 | $\frac{1}{3}\begin{bmatrix}1&1&1\end{bmatrix}$ | $\frac{1}{3}(1+2\cos\omega)$ |  |
| box-5 | $\frac{1}{5}\begin{bmatrix}1&1&1&1&1\end{bmatrix}$ | $\frac{1}{5}(1+2\cos\omega+2\cos 2\omega)$ |  |
| linear | $\frac{1}{4}\begin{bmatrix}1&2&1\end{bmatrix}$ | $\frac{1}{2}(1+\cos\omega)$ |  |
| 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 $(\omega_x, \omega_y)$,

$$s(x, y) = \sin(\omega_x x + \omega_y y). \tag{3.62}$$

The corresponding two-dimensional Fourier transforms are then

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

and in the discrete domain,

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

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 $\boldsymbol{x} = (x, y)$, $\boldsymbol{\omega} = (\omega_x, \omega_y)$, $\boldsymbol{x}_0 = (x_0, y_0)$, and $\boldsymbol{a} = (a_x, a_y)$, 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.,

$$\left\langle [S(\omega_x, \omega_y)]^2 \right\rangle = P_s(\omega_x, \omega_y), \tag{3.65}$$

where the angle brackets $\langle \cdot \rangle$ denote the expected (mean) value of a random variable.[9] 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[10] 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*.[11]

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

$$o(x, y) = s(x, y) + n(x, y), \tag{3.66}$$

[9] The notation $E[\cdot]$ is also commonly used.

[10] 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.

[11] 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

$$O(\omega_x, \omega_y) = S(\omega_x, \omega_y) + N(\omega_x, \omega_y), \tag{3.67}$$

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

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

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

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

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

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

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

$$\begin{aligned} -\log p(S|O) &= -\log p(O|S) - \log p(S) + C & (3.71)\\ &= {}^1\!/_2 P_n^{-1}(S-O)^2 + {}^1\!/_2 P_s^{-1} S^2 + C, & (3.72) \end{aligned}$$

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

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

The quantity

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

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.

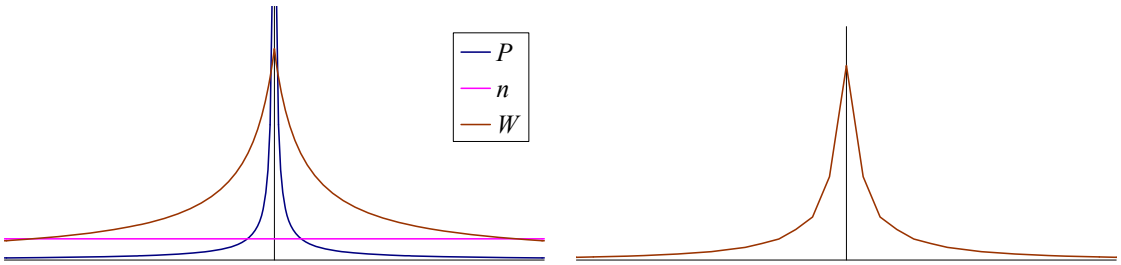



**Figure 3.25** One-dimensional Wiener filter: (a) power spectrum of signal $P_s(f)$, noise level $\sigma^2$, and Wiener filter transform $W(f)$; (b) Wiener filter spatial kernel.

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,

$$o(x, y) = b(x, y) * s(x, y) + n(x, y), \tag{3.75}$$

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,

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

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.

In 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,

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

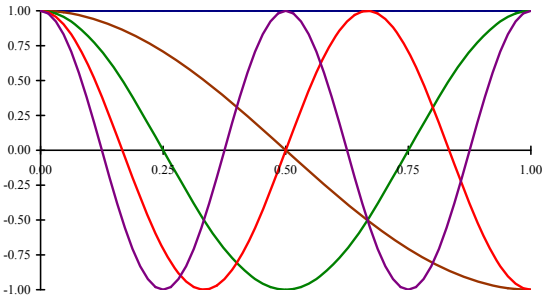



**Figure 3.26** Discrete cosine transform (DCT) basis functions: The first DC (i.e., constant) basis is the horizontal blue line, the second is the brown half-cycle waveform, etc. These bases are widely used in image and video compression standards such as JPEG.

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.[12] 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 $I(\boldsymbol{x})$ is the original (noise-free) image and $\hat{I}(\boldsymbol{x})$ 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).

[12] 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,

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

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.

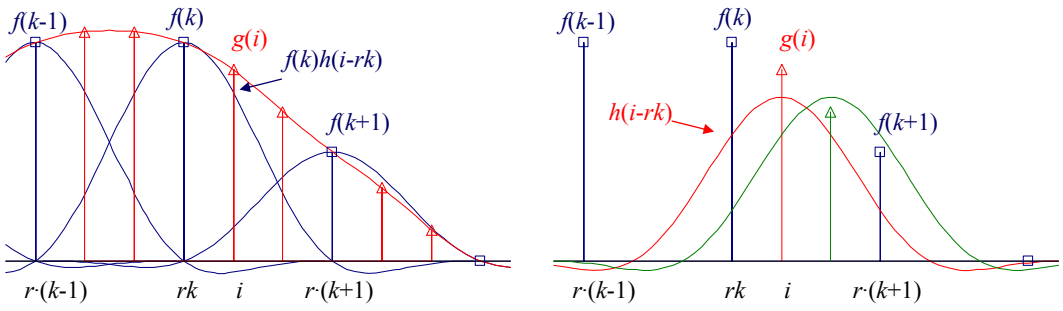



**Figure 3.27** Signal interpolation, $g(i) = \sum_k f(k)h(i - rk)$: (a) weighted summation of input values; (b) polyphase filter interpretation.

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.[13] 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")[14] whose equation is

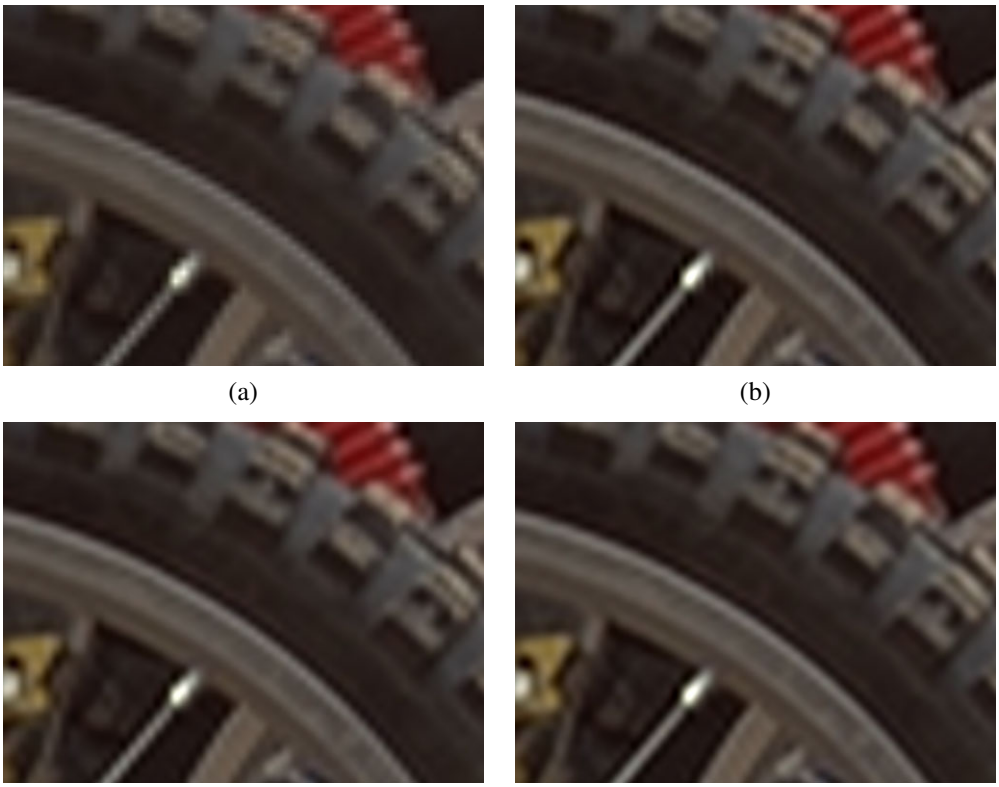
$$h(x) = \begin{cases} 1 - (a+3)x^2 + (a+2)|x|^3 & \text{if } |x| < 1 \\ a(|x|-1)(|x|-2)^2 & \text{if } 1 \le |x| < 2 \\ 0 & \text{otherwise,} \end{cases} \tag{3.79}$$

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-

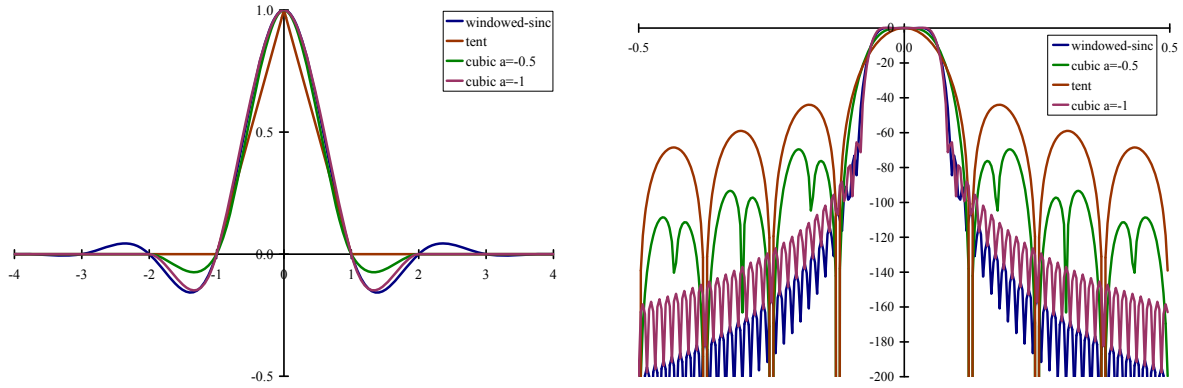
[13] The smoothing kernels in Table 3.3 have a unit area. To turn them into interpolating kernels, we simply scale them up by the interpolation rate $r$.

[14] 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.



(a) (b) (c) (d)

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




(a) (b)

**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.[15] 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,

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

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

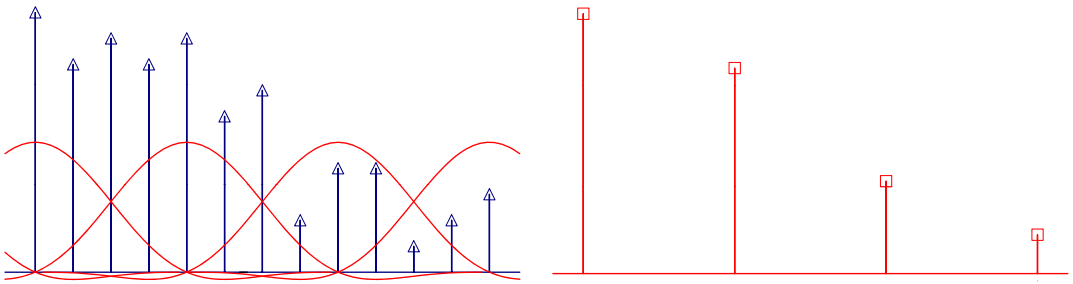
$$g(i,j) = \frac{1}{r}\sum_{k,l} f(k,l)h(i-k/r, j-l/r) \tag{3.81}$$

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

[15] 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.



(a) (b)

**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.

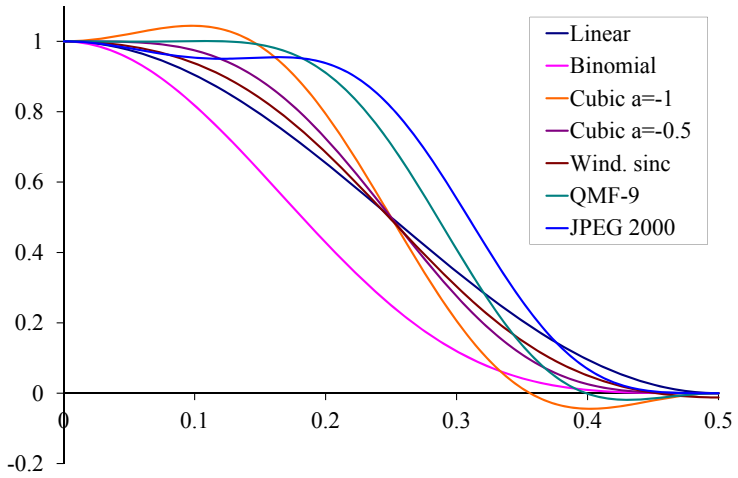



**Figure 3.31** Frequency response for some $2\times$ decimation filters. The cubic $a = -1$ filter has the sharpest fall-off but also a bit of ringing; the wavelet analysis filters (QMF-9 and JPEG 2000), while useful for compression, have more aliasing.

### 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 subsample 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

$$\begin{array}{|c|c|c|c|c|}\hline c & b & a & b & c \\ \hline\end{array}, \tag{3.82}$$

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

$$\frac{1}{16}\begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 6 & 4 & 1 \\ \hline\end{array}, \tag{3.83}$$

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.[16]

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"

[16] Then again, this is true for any smoothing kernel (Wells 1986).

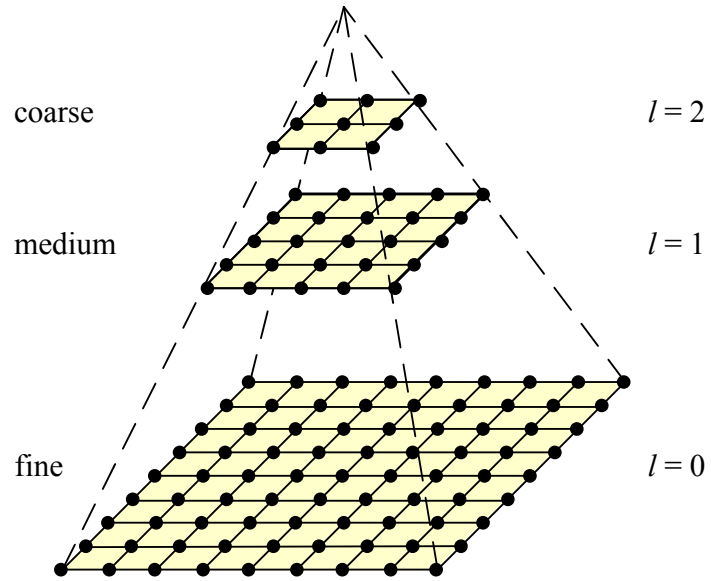



**Figure 3.32** A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level.

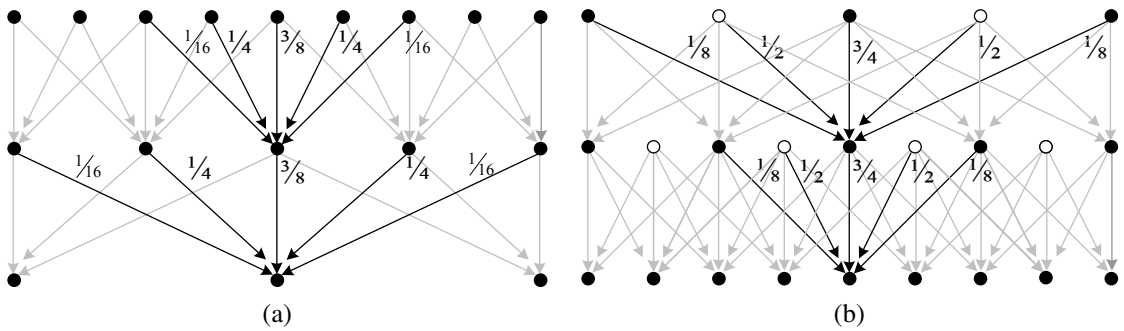



**Figure 3.33** The Gaussian pyramid shown as a signal processing diagram: The (a) analysis and (b) re-synthesis stages are shown as using similar computations. The white circles indicate zero values inserted by the $\uparrow 2$ upsampling operation. Notice how the reconstruction filter coefficients 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*.

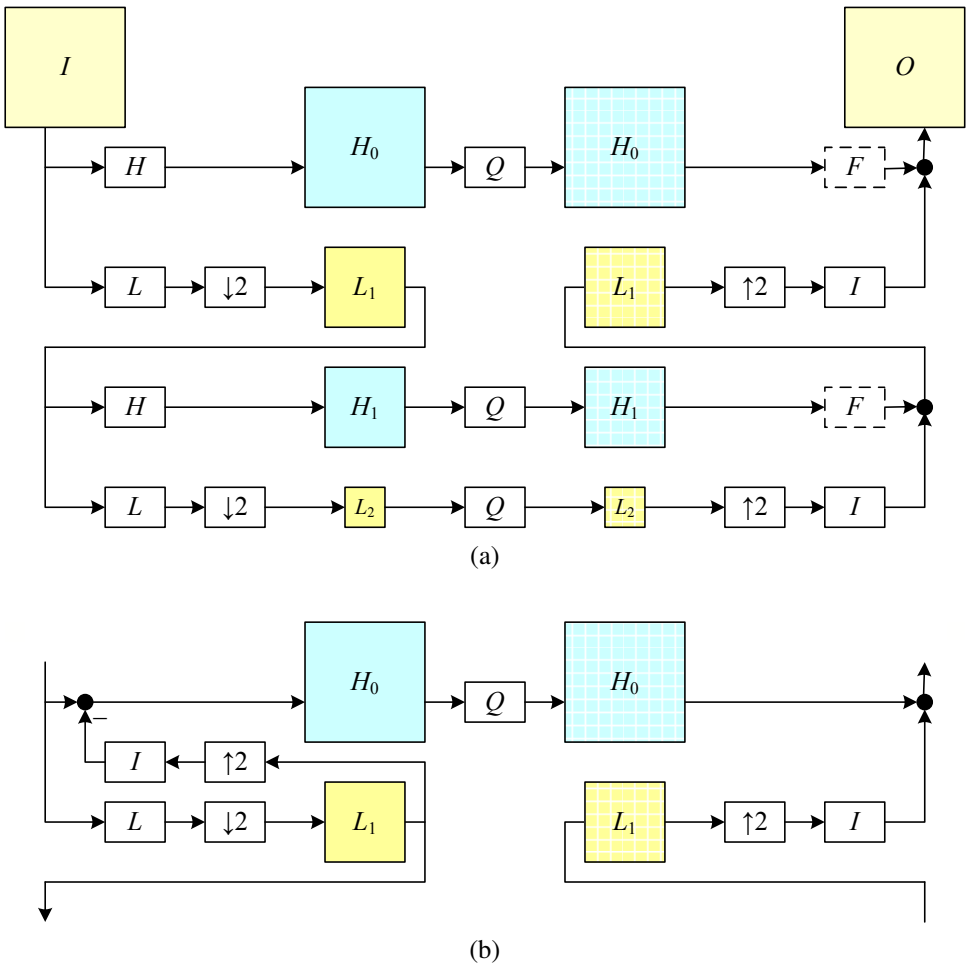



**Figure 3.34** The Laplacian pyramid: (a) The conceptual flow of images through processing stages: images are high-pass and low-pass filtered, and the low-pass filtered images are processed in the next stage of the pyramid. During reconstruction, the interpolated image and the (optionally filtered) 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.

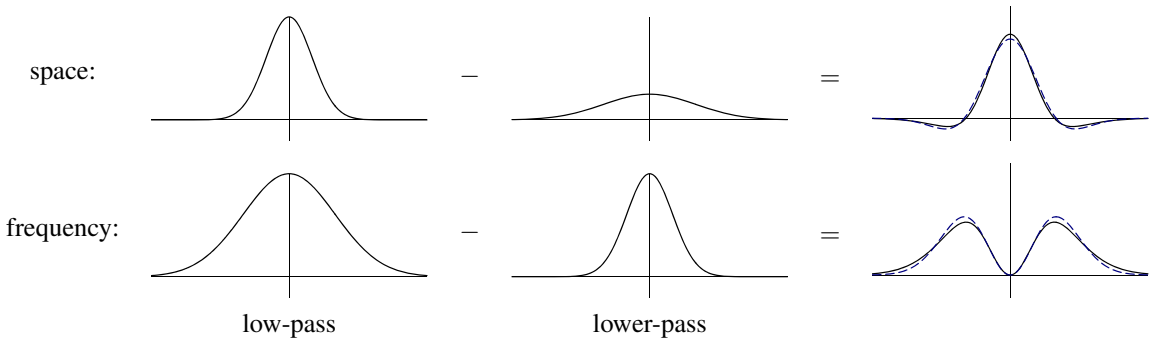



**Figure 3.35** The difference of two low-pass filters results in a band-pass filter. The dashed blue lines show the close fit to a half-octave Laplacian of Gaussian.

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,

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

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

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

where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3.86}$$

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-

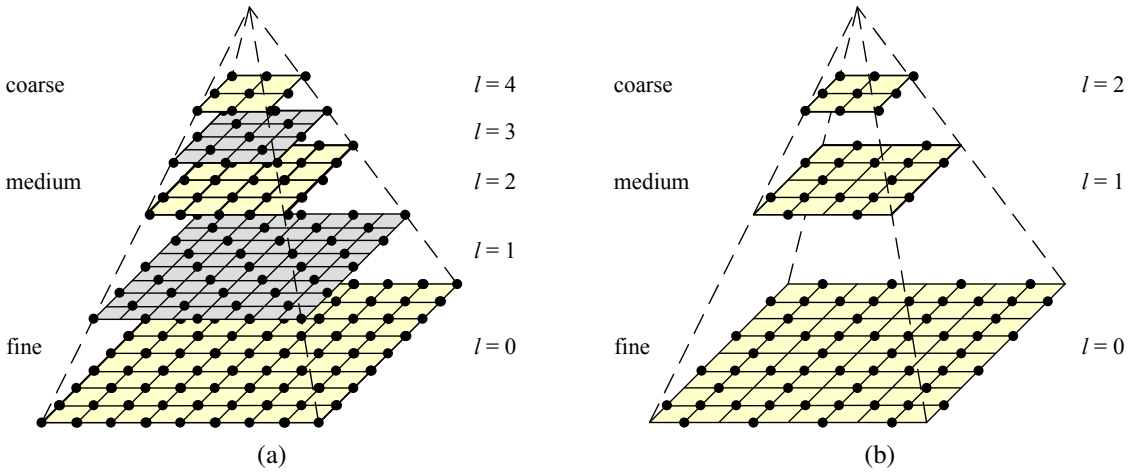



**Figure 3.36** Multiresolution pyramids: (a) pyramid with half-octave (*quincunx*) sampling (odd levels are colored gray for clarity). (b) wavelet pyramid—each wavelet level stores $3/4$ of the original pixels (usually the horizontal, vertical, and mixed gradients), so that the total number of wavelet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

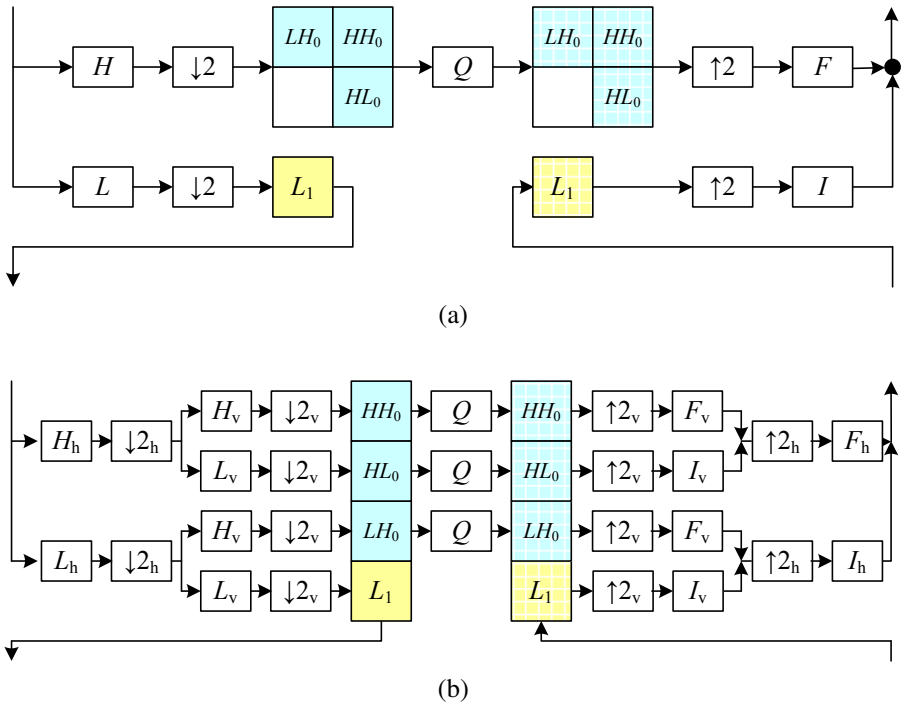



**Figure 3.37** Two-dimensional wavelet decomposition: (a) high-level diagram showing the low-pass and high-pass transforms as single boxes; (b) separable implementation, which involves first performing the wavelet transform horizontally and then vertically. The $I$ and $F$ boxes are the interpolation and filtering boxes required to re-synthesize the image from its wavelet components.

Since both image pyramids and wavelets decompose an image into multi-resolution descriptions that are localized in both space and frequency, how do they differ? The usual answer is that traditional pyramids are *overcomplete*, i.e., they use more pixels than the original image to represent the decomposition, whereas wavelets provide a *tight frame*, i.e., they keep the size of the decomposition the same as the image (Figure 3.36b). However, some wavelet families *are*, in fact, overcomplete in order to provide better shiftability or steering in orientation (Simoncelli, Freeman, Adelson *et al.* 1992). A better distinction, therefore, might be that wavelets are more orientation selective than regular band-pass pyramids.

How are two-dimensional wavelets constructed? Figure 3.37a shows a high-level diagram of one stage of the (recursive) coarse-to-fine construction (analysis) pipeline alongside the complementary re-construction (synthesis) stage. In this diagram, the high-pass filter followed by decimation keeps $3/4$ of the original pixels, while $1/4$ of the low-frequency coefficients are passed on to the next stage for further analysis. In practice, the filtering is usually broken down into two separable sub-stages, as shown in Figure 3.37b. The resulting three wavelet images are sometimes called the high–high ($HH$), high–low ($HL$), and low–high ($LH$) images. The high–low and low–high images accentuate the horizontal and vertical edges and gradients, while the high–high image contains the less frequently occurring mixed derivatives.

How are the high-pass $H$ and low-pass $L$ filters shown in Figure 3.37b chosen and how

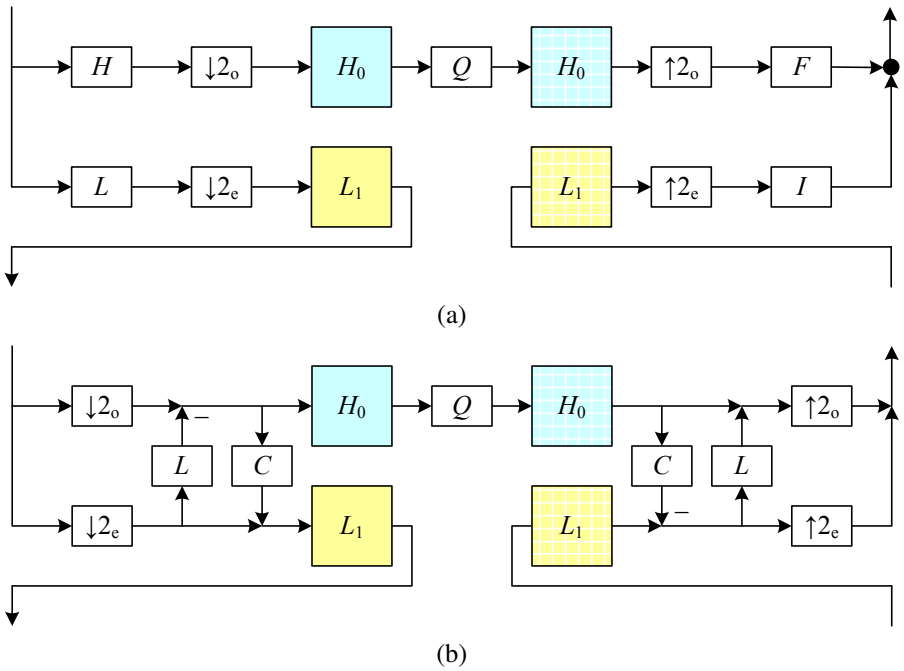



**Figure 3.38** One-dimensional wavelet transform: (a) usual high-pass + low-pass filters followed by odd ($\downarrow 2_o$) and even ($\downarrow 2_e$) downsampling; (b) lifted version, which first selects the odd and even subsequences and then applies a low-pass prediction stage $L$ and a high-pass correction stage $C$ in an easily reversible manner.

can the corresponding reconstruction filters $I$ and $F$ be computed? Can filters be designed that all have finite impulse responses? This topic has been the main subject of study in the wavelet community for over two decades. The answer depends largely on the intended application, e.g., whether the wavelets are being used for compression, image analysis (feature finding), or denoising. Simoncelli and Adelson (1990b) show (in Table 4.1) some good odd-length quadrature mirror filter (QMF) coefficients that seem to work well in practice.

Since the design of wavelet filters is such a tricky art, is there perhaps a better way? Indeed, a simpler procedure is to split the signal into its even and odd components and then perform trivially reversible filtering operations on each sequence to produce what are called *lifted wavelets* (Figures 3.38 and 3.39). Sweldens (1996) gives a wonderfully understandable introduction to the *lifting scheme* for *second-generation wavelets*, followed by a comprehensive review (Sweldens 1997).

As Figure 3.38 demonstrates, rather than first filtering the whole input sequence (image) with high-pass and low-pass filters and then keeping the odd and even sub-sequences, the lifting scheme first splits the sequence into its even and odd sub-components. Filtering the even sequence with a low-pass filter $L$ and subtracting the result from the even sequence is trivially reversible: simply perform the same filtering and then add the result back in. Furthermore, this operation can be performed in place, resulting in significant space savings. The same applies to filtering the even sequence with the correction filter $C$, which is used to ensure that the even sequence is low-pass. A series of such *lifting* steps can be used to create more complex filter responses with low computational cost and guaranteed reversibility.

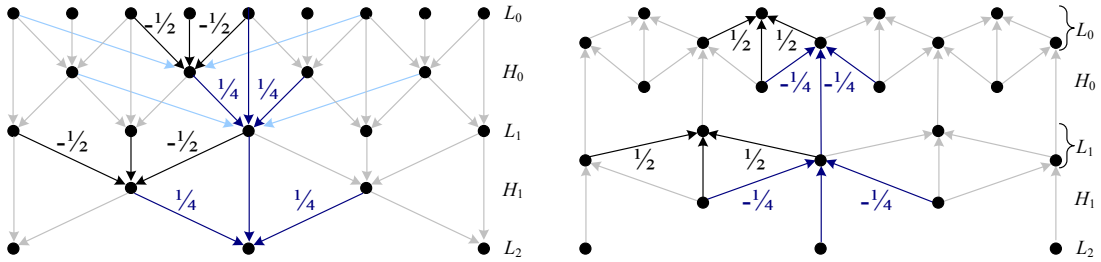



**Figure 3.39** Lifted transform shown as a signal processing diagram: (a) The analysis stage first predicts the odd value from its even neighbors, stores the difference wavelet, and then compensates the coarser even value by adding in a fraction of the wavelet. (b) The synthesis stage simply reverses the flow of computation and the signs of some of the filters and operations. The light blue lines show what happens if we use four taps for the prediction and correction instead of just two.

This process can perhaps be more easily understood by considering the signal processing diagram in Figure 3.39. During analysis, the average of the even values is subtracted from the odd value to obtain a high-pass wavelet coefficient. However, the even samples still contain an aliased sample of the low-frequency signal. To compensate for this, a small amount of the high-pass wavelet is added back to the even sequence so that it is properly low-pass filtered. (It is easy to show that the effective low-pass filter is $[-1/8, 1/4, 3/4, 1/4, -1/8]$, which is indeed a low-pass filter.) During synthesis, the same operations are reversed with a judicious change in sign.

Of course, we need not restrict ourselves to two-tap filters. Figure 3.39 shows as light blue arrows additional filter coefficients that could optionally be added to the lifting scheme without affecting its reversibility. In fact, the low-pass and high-pass filtering operations can be interchanged, e.g., we could use a five-tap cubic low-pass filter on the odd sequence (plus center value) first, followed by a four-tap cubic low-pass predictor to estimate the wavelet, although I have not seen this scheme written down.

Lifted wavelets are called *second-generation wavelets* because they can easily adapt to non-regular sampling topologies, e.g., those that arise in computer graphics applications such as multi-resolution surface manipulation (Schröder and Sweldens 1995). It also turns out that lifted *weighted wavelets*, i.e., wavelets whose coefficients adapt to the underlying problem being solved (Fattal 2009), can be extremely effective for low-level image manipulation tasks and also for preconditioning the kinds of sparse linear systems that arise in the optimization-based approaches to vision algorithms that we discuss in Section 3.7 (Szeliski 2006b).

An alternative to the widely used "separable" approach to wavelet construction, which decomposes each level into horizontal, vertical, and "cross" sub-bands, is to use a representation that is more rotationally symmetric and orientationally selective and also avoids the aliasing inherent in sampling signals below their Nyquist frequency.[17] Simoncelli, Freeman, Adelson *et al.* (1992) introduce such a representation, which they call a *pyramidal radial frequency implementation* of *shiftable multi-scale transforms* or, more succinctly, *steerable pyramids*.

[17] Such aliasing can often be seen as the signal content moving between bands as the original signal is slowly shifted.

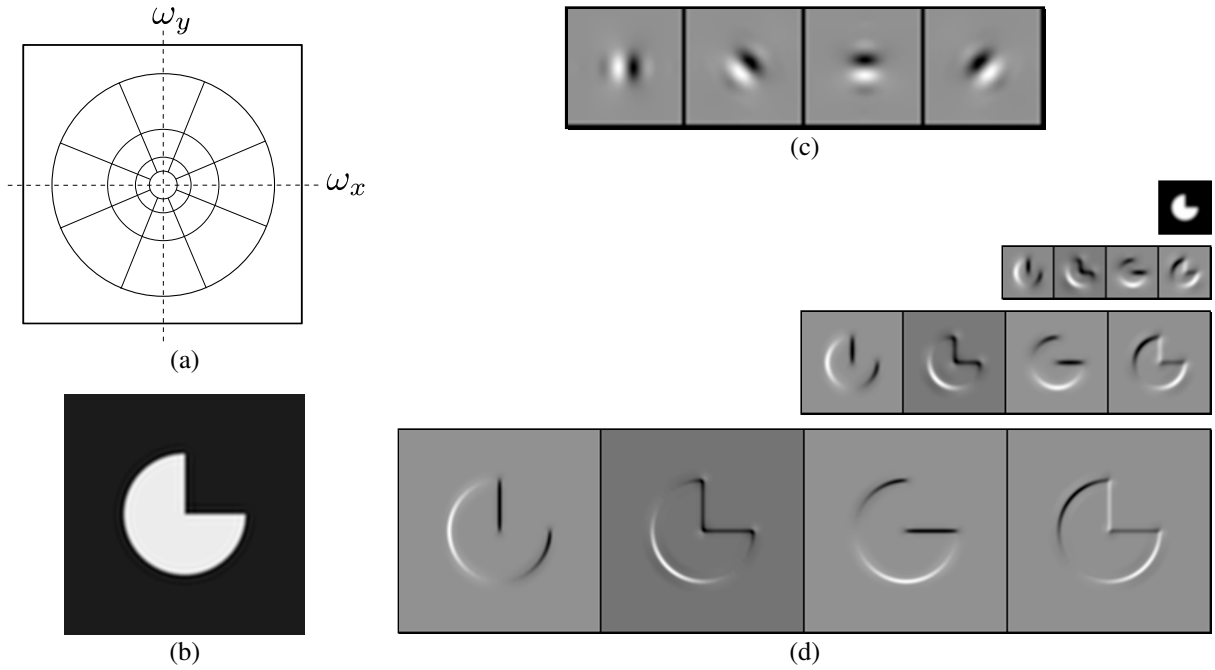


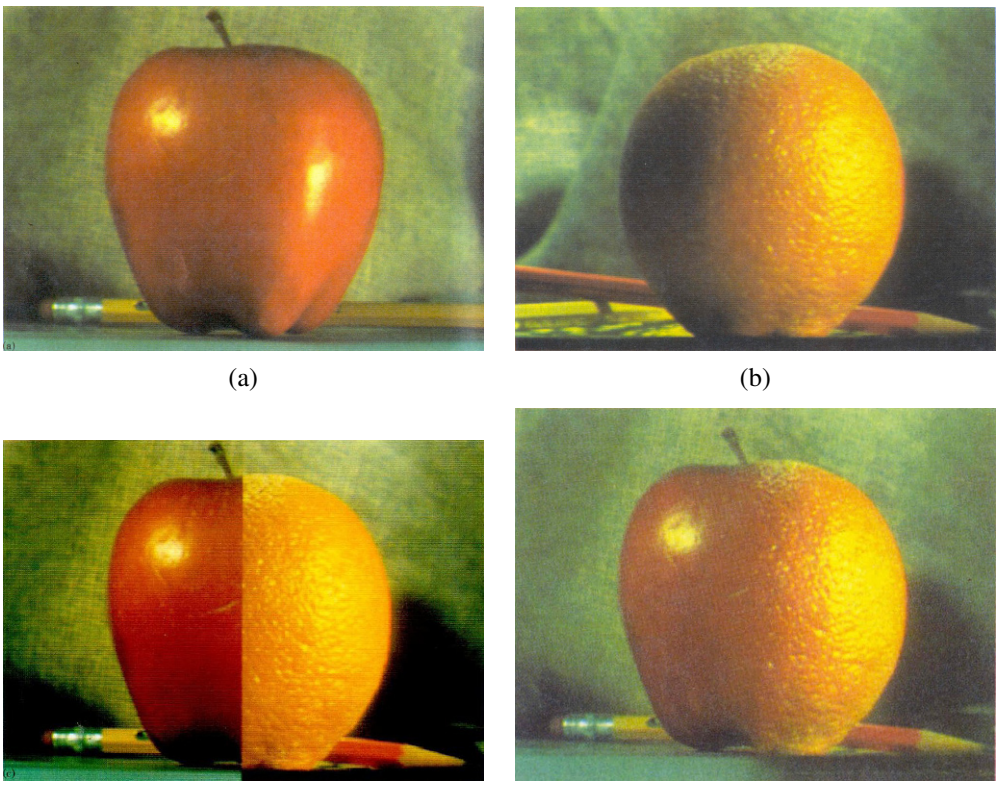

**Figure 3.40** Steerable shiftable multiscale transforms (Simoncelli, Freeman, Adelson *et al.* 1992) © 1992 IEEE: (a) radial multi-scale frequency domain decomposition; (b) original image; (c) a set of four steerable filters; (d) the radial multi-scale wavelet decomposition.

Their representation is not only overcomplete (which eliminates the aliasing problem) but is also orientationally selective and has identical analysis and synthesis basis functions, i.e., it is *self-inverting*, just like "regular" wavelets. As a result, this makes steerable pyramids a much more useful basis for the structural analysis and matching tasks commonly used in computer vision.

Figure 3.40a shows how such a decomposition looks in frequency space. Instead of recursively dividing the frequency domain into $2 \times 2$ squares, which results in checkerboard high frequencies, radial arcs are used instead. Figure 3.40b illustrates the resulting pyramid sub-bands. Even through the representation is *overcomplete*, i.e., there are more wavelet coefficients than input pixels, the additional frequency and orientation selectivity makes this representation preferable for tasks such as texture analysis and synthesis (Portilla and Simoncelli 2000) and image denoising (Portilla, Strela, Wainwright *et al.* 2003; Lyu and Simoncelli 2009).

### 3.5.5 *Application*: Image blending

One of the most engaging and fun applications of the Laplacian pyramid presented in Section 3.5.3 is the creation of blended composite images, as shown in Figure 3.41 (Burt and Adelson 1983b). While splicing the apple and orange images together along the midline produces a noticeable cut, *splining* them together (as Burt and Adelson (1983b) called their procedure) creates a beautiful illusion of a truly hybrid fruit. The key to their approach is that the low-frequency color variations between the red apple and the orange are smoothly



(a) (b) (c) (d)

**Figure 3.41** Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM: (a) original image of apple, (b) original image of orange, (c) regular splice, (d) pyramid blend.

blended, while the higher-frequency textures on each fruit are blended more quickly to avoid "ghosting" effects when two textures are overlaid.

To create the blended image, each source image is first decomposed into its own Laplacian pyramid (Figure 3.42, left and middle columns). Each band is then multiplied by a smooth weighting function whose extent is proportional to the pyramid level. The simplest and most general way to create these weights is to take a binary mask image (Figure 3.43c) and to construct a *Gaussian* pyramid from this mask. Each Laplacian pyramid image is then multiplied by its corresponding Gaussian mask and the sum of these two weighted pyramids is then used to construct the final image (Figure 3.42, right column).

Figure 3.43 shows that this process can be applied to arbitrary mask images with surprising results. It is also straightforward to extend the pyramid blend to an arbitrary number of images whose pixel provenance is indicated by an integer-valued label image (see Exercise 3.20). This is particularly useful in image stitching and compositing applications, where the exposures may vary between different images, as described in Section 9.3.4.

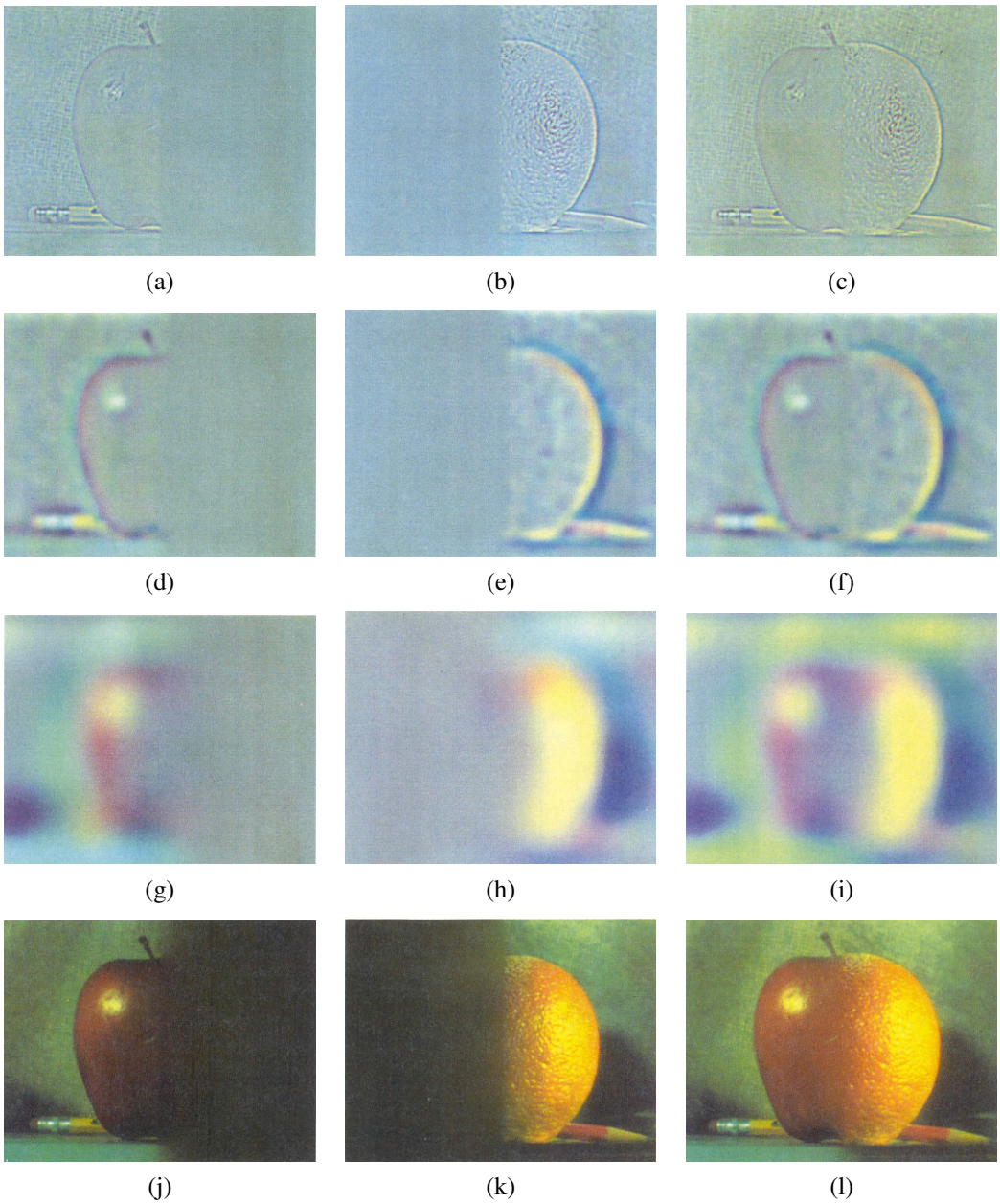
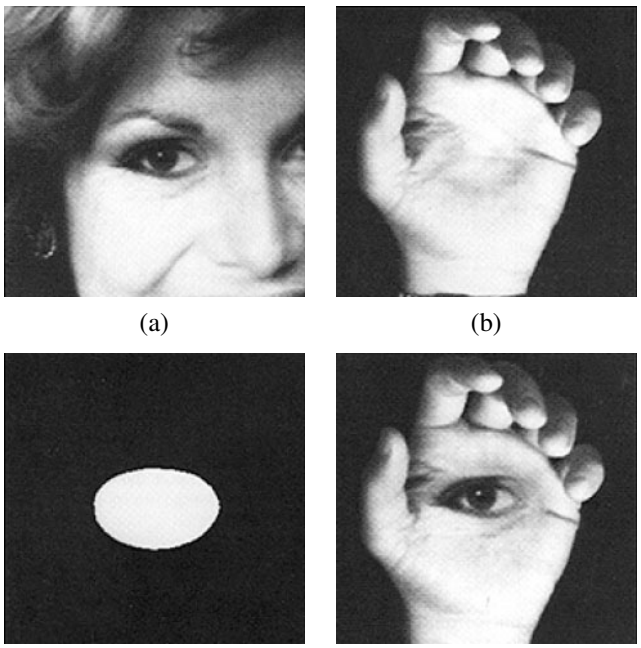



**Figure 3.42** Laplacian pyramid blending details (Burt and Adelson 1983b) © 1983 ACM. The first three rows show the high, medium, and low frequency parts of the Laplacian pyramid (taken from levels 0, 2, and 4). The left and middle columns show the original apple and orange images weighted by the smooth interpolation functions, while the right column shows the averaged contributions.



(a) (b) (c) (d)

**Figure 3.43** Laplacian pyramid blend of two images of arbitrary shape (Burt and Adelson 1983b) © 1983 ACM: (a) first input image; (b) second input image; (c) region mask; (d) blended image.

## 3.6 Geometric transformations

In the previous sections, we saw how interpolation and decimation could be used to change the *resolution* of an image. In this section, we look at how to perform more general transformations, such as image rotations or general warps. In contrast to the point processes we saw in Section 3.1, where the function applied to an image transforms the *range* of the image,

$$g(\boldsymbol{x}) = h(f(\boldsymbol{x})), \tag{3.87}$$

here we look at functions that transform the *domain*,

$$g(\boldsymbol{x}) = f(\boldsymbol{h}(\boldsymbol{x})) \tag{3.88}$$

(see Figure 3.44).

We begin by studying the global *parametric* 2D transformation first introduced in Section 2.1.2. (Such a transformation is called parametric because it is controlled by a small number of parameters.) We then turn our attention to more local general deformations such as those defined on meshes (Section 3.6.2). Finally, we show how image warps can be combined with cross-dissolves to create interesting *morphs* (in-between animations) in Section 3.6.3. For readers interested in more details on these topics, there is an excellent survey by Heckbert (1986) as well as very accessible textbooks by Wolberg (1990), Gomes, Darsa, Costa *et al.* (1999) and Akenine-Möller and Haines (2002). Note that Heckbert's survey is on *texture mapping*, which is how the computer graphics community refers to the topic of warping images onto surfaces.

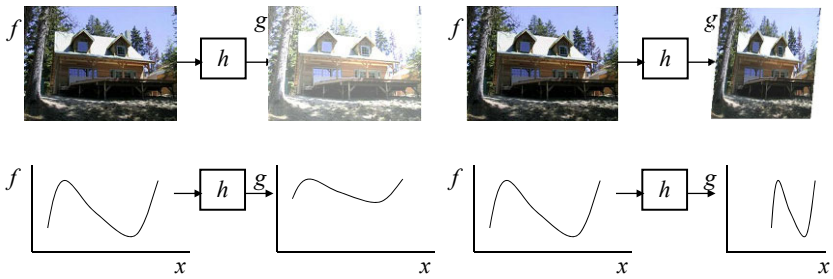



**Figure 3.44** Image warping involves modifying the *domain* of an image function rather than its *range*.

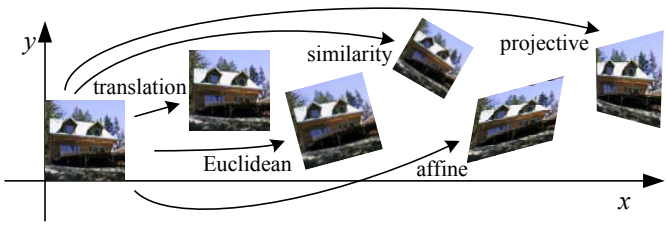



**Figure 3.45** Basic set of 2D geometric image transformations.

| Transformation | Matrix | # DoF | Preserves | Icon |
|---|---|---|---|---|
| translation | $\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$ | 2 | orientation | |
| rigid (Euclidean) | $\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$ | 3 | lengths | |
| similarity | $\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$ | 4 | angles | |
| affine | $\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times 3}$ | 6 | parallelism | |
| projective | $\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times 3}$ | 8 | straight lines | |

**Table 3.5** Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The $2 \times 3$ matrices are extended with a third $[\mathbf{0}^T\ 1]$ row to form a full $3 \times 3$ matrix for homogeneous coordinate transformations.

**procedure** *forwardWarp*($f, \boldsymbol{h}$, **out** $g$):

For every pixel $\boldsymbol{x}$ in $f(\boldsymbol{x})$

1. Compute the destination location $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$.
2. Copy the pixel $f(\boldsymbol{x})$ to $g(\boldsymbol{x}')$.

**Algorithm 3.1** Forward warping algorithm for transforming an image $f(\boldsymbol{x})$ into an image $g(\boldsymbol{x}')$ through the parametric transform $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$.


**Figure 3.46** Forward warping algorithm: (a) a pixel $f(\boldsymbol{x})$ is copied to its corresponding location $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$ in image $g(\boldsymbol{x}')$; (b) detail of the source and destination pixel locations.

### 3.6.1 Parametric transformations

Parametric transformations apply a global deformation to an image, where the behavior of the transformation is controlled by a small number of parameters. Figure 3.45 shows a few examples of such transformations, which are based on the 2D geometric transformations shown in Figure 2.4. The formulas for these transformations were originally given in Table 2.1 and are reproduced here in Table 3.5 for ease of reference.

In general, given a transformation specified by a formula $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$ and a source image $f(\boldsymbol{x})$, how do we compute the values of the pixels in the new image $g(\boldsymbol{x})$, as given in (3.88)? Think about this for a minute before proceeding and see if you can figure it out.

If you are like most people, you will come up with an algorithm that looks something like Algorithm 3.1. This process is called *forward warping* or *forward mapping* and is shown in Figure 3.46a. Can you think of any problems with this approach?

In fact, this approach suffers from several limitations. The process of copying a pixel $f(\boldsymbol{x})$ to a location $\boldsymbol{x}'$ in $g$ is not well defined when $\boldsymbol{x}'$ has a non-integer value. What do we do in such a case? What would you do?

You can round the value of $\boldsymbol{x}'$ to the nearest integer coordinate and copy the pixel there, but the resulting image has severe aliasing and pixels that jump around a lot when animating the transformation. You can also "distribute" the value among its four nearest neighbors in a weighted (bilinear) fashion, keeping track of the per-pixel weights and normalizing at the end. This technique is called *splatting* and is sometimes used for volume rendering in the graphics community (Levoy and Whitted 1985; Levoy 1988; Westover 1989; Rusinkiewicz and Levoy 2000). Unfortunately, it suffers from both moderate amounts of aliasing and a fair amount of blur (loss of high-resolution detail).

**procedure** *inverseWarp*($f, \boldsymbol{h}$, **out** $g$):

For every pixel $\boldsymbol{x}'$ in $g(\boldsymbol{x}')$

1. Compute the source location $\boldsymbol{x} = \hat{\boldsymbol{h}}(\boldsymbol{x}')$
2. Resample $f(\boldsymbol{x})$ at location $\boldsymbol{x}$ and copy to $g(\boldsymbol{x}')$

**Algorithm 3.2** Inverse warping algorithm for creating an image $g(\boldsymbol{x}')$ from an image $f(\boldsymbol{x})$ using the parametric transform $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$.


**Figure 3.47** Inverse warping algorithm: (a) a pixel $g(\boldsymbol{x}')$ is sampled from its corresponding location $\boldsymbol{x} = \hat{\boldsymbol{h}}(\boldsymbol{x}')$ in image $f(\boldsymbol{x})$; (b) detail of the source and destination pixel locations.

The second major problem with forward warping is the appearance of cracks and holes, especially when magnifying an image. Filling such holes with their nearby neighbors can lead to further aliasing and blurring.

What can we do instead? A preferable solution is to use *inverse warping* (Algorithm 3.2), where each pixel in the destination image $g(\boldsymbol{x}')$ is sampled from the original image $f(\boldsymbol{x})$ (Figure 3.47).

How does this differ from the forward warping algorithm? For one thing, since $\hat{\boldsymbol{h}}(\boldsymbol{x}')$ is (presumably) defined for all pixels in $g(\boldsymbol{x}')$, we no longer have holes. More importantly, resampling an image at non-integer locations is a well-studied problem (general image interpolation, see Section 3.5.2) and high-quality filters that control aliasing can be used.

Where does the function $\hat{\boldsymbol{h}}(\boldsymbol{x}')$ come from? Quite often, it can simply be computed as the inverse of $\boldsymbol{h}(\boldsymbol{x})$. In fact, all of the parametric transforms listed in Table 3.5 have closed form solutions for the inverse transform: simply take the inverse of the $3 \times 3$ matrix specifying the transform.

In other cases, it is preferable to formulate the problem of image warping as that of resampling a source image $f(\boldsymbol{x})$ given a mapping $\boldsymbol{x} = \hat{\boldsymbol{h}}(\boldsymbol{x}')$ from destination pixels $\boldsymbol{x}'$ to source pixels $\boldsymbol{x}$. For example, in optical flow (Section 8.4), we estimate the flow field as the location of the *source* pixel which produced the current pixel whose flow is being estimated, as opposed to computing the *destination* pixel to which it is going. Similarly, when correcting for radial distortion (Section 2.1.6), we calibrate the lens by computing for each pixel in the final (undistorted) image the corresponding pixel location in the original (distorted) image.

What kinds of interpolation filter are suitable for the resampling process? Any of the filters we studied in Section 3.5.2 can be used, including nearest neighbor, bilinear, bicubic, and

windowed sinc functions. While bilinear is often used for speed (e.g., inside the inner loop of a patch-tracking algorithm, see Section 8.1.3), bicubic, and windowed sinc are preferable where visual quality is important.

To compute the value of $f(\boldsymbol{x})$ at a non-integer location $\boldsymbol{x}$, we simply apply our usual FIR resampling filter,

$$g(x,y) = \sum_{k,l} f(k,l)h(x-k, y-l), \tag{3.89}$$

where $(x, y)$ are the sub-pixel coordinate values and $h(x, y)$ is some interpolating or smoothing kernel. Recall from Section 3.5.2 that when decimation is being performed, the smoothing kernel is stretched and re-scaled according to the downsampling rate $r$.

Unfortunately, for a general (non-zoom) image transformation, the resampling rate $r$ is not well defined. Consider a transformation that stretches the $x$ dimensions while squashing the $y$ dimensions. The resampling kernel should be performing regular interpolation along the $x$ dimension and smoothing (to anti-alias the blurred image) in the $y$ direction. This gets even more complicated for the case of general affine or perspective transforms.

What can we do? Fortunately, Fourier analysis can help. The two-dimensional generalization of the one-dimensional *domain scaling* law given in Table 3.1 is

$$g(\boldsymbol{A}\boldsymbol{x}) \Leftrightarrow |\boldsymbol{A}|^{-1} G(\boldsymbol{A}^{-T}\boldsymbol{f}). \tag{3.90}$$

For all of the transforms in Table 3.5 except perspective, the matrix $\boldsymbol{A}$ is already defined. For perspective transformations, the matrix $\boldsymbol{A}$ is the linearized *derivative* of the perspective transformation (Figure 3.48a), i.e., the local affine approximation to the stretching induced by the projection (Heckbert 1986; Wolberg 1990; Gomes, Darsa, Costa *et al.* 1999; Akenine-Möller and Haines 2002).

To prevent aliasing, we need to pre-filter the image $f(\boldsymbol{x})$ with a filter whose frequency response is the projection of the final desired spectrum through the $\boldsymbol{A}^{-T}$ transform (Szeliski, Winder, and Uyttendaele 2010). In general (for non-zoom transforms), this filter is non-separable and hence is very slow to compute. Therefore, a number of approximations to this filter are used in practice, include MIP-mapping, elliptically weighted Gaussian averaging, and anisotropic filtering (Akenine-Möller and Haines 2002).

### MIP-mapping

MIP-mapping was first proposed by Williams (1983) as a means to rapidly pre-filter images being used for *texture mapping* in computer graphics. A MIP-map[18] is a standard image pyramid (Figure 3.32), where each level is pre-filtered with a high-quality filter rather than a poorer quality approximation, such as Burt and Adelson's (1983b) five-tap binomial. To resample an image from a MIP-map, a scalar estimate of the resampling rate $r$ is first computed. For example, $r$ can be the maximum of the absolute values in $\boldsymbol{A}$ (which suppresses aliasing) or it can be the minimum (which reduces blurring). Akenine-Möller and Haines (2002) discuss these issues in more detail.

Once a resampling rate has been specified, a *fractional* pyramid level is computed using the base 2 logarithm,

$$l = \log_2 r. \tag{3.91}$$

[18] The term 'MIP' stands for *multi in parvo*, meaning 'many in one'.

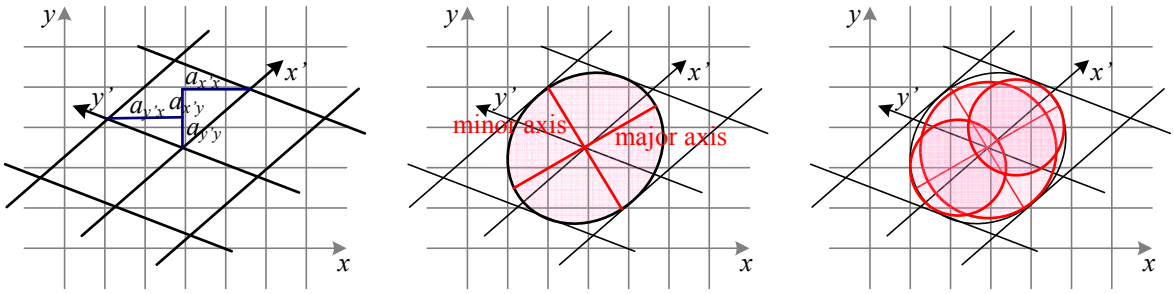



**Figure 3.48** Anisotropic texture filtering: (a) Jacobian of transform $\boldsymbol{A}$ and the induced horizontal and vertical resampling rates $\{a_{x'x}, a_{x'y}, a_{y'x}, a_{y'y}\}$; (b) elliptical footprint of an EWA smoothing kernel; (c) anisotropic filtering using multiple samples along the major axis. Image pixels lie at line intersections.

One simple solution is to resample the texture from the next higher or lower pyramid level, depending on whether it is preferable to reduce aliasing or blur. A better solution is to resample *both* images and blend them linearly using the fractional component of $l$. Since most MIP-map implementations use bilinear resampling within each level, this approach is usually called *trilinear MIP-mapping*. Computer graphics rendering APIs, such as OpenGL and Direct3D, have parameters that can be used to select which variant of MIP-mapping (and of the sampling rate $r$ computation) should be used, depending on the desired tradeoff between speed and quality. Exercise 3.22 has you examine some of these tradeoffs in more detail.

### Elliptical Weighted Average

The Elliptical Weighted Average (EWA) filter invented by Greene and Heckbert (1986) is based on the observation that the affine mapping $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x}'$ defines a skewed two-dimensional coordinate system in the vicinity of each source pixel $\boldsymbol{x}$ (Figure 3.48a). For every destination pixel $\boldsymbol{x}'$, the ellipsoidal projection of a small pixel grid in $\boldsymbol{x}'$ onto $\boldsymbol{x}$ is computed (Figure 3.48b). This is then used to filter the source image $g(\boldsymbol{x})$ with a Gaussian whose inverse covariance matrix is this ellipsoid.

Despite its reputation as a high-quality filter (Akenine-Möller and Haines 2002), we have found in our work (Szeliski, Winder, and Uyttendaele 2010) that because a Gaussian kernel is used, the technique suffers simultaneously from both blurring and aliasing, compared to higher-quality filters. The EWA is also quite slow, although faster variants based on MIP-mapping have been proposed (Szeliski, Winder, and Uyttendaele (2010) provide some additional references).

### Anisotropic filtering

An alternative approach to filtering oriented textures, which is sometimes implemented in graphics hardware (GPUs), is to use anisotropic filtering (Barkans 1997; Akenine-Möller and Haines 2002). In this approach, several samples at different resolutions (fractional levels in the MIP-map) are combined along the major axis of the EWA Gaussian (Figure 3.48c).

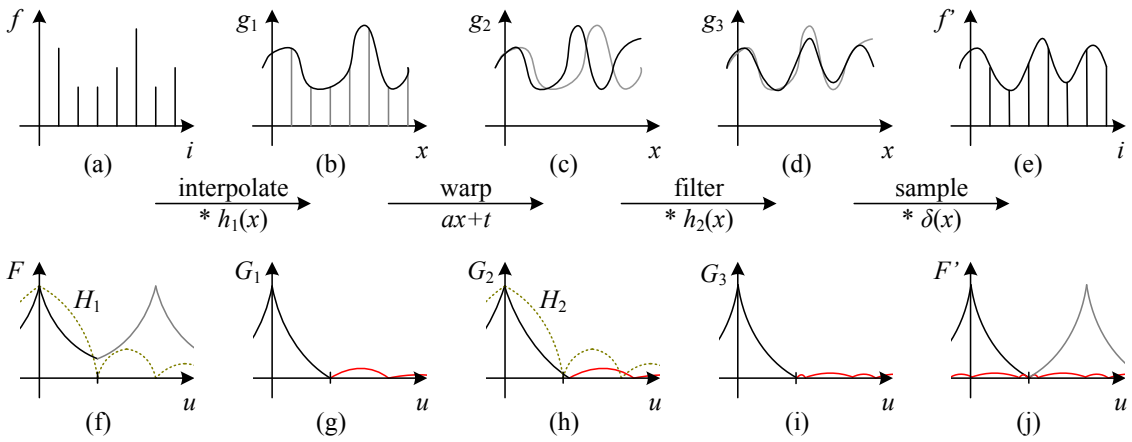



**Figure 3.49** One-dimensional signal resampling (Szeliski, Winder, and Uyttendaele 2010): (a) original sampled signal $f(i)$; (b) interpolated signal $g_1(x)$; (c) warped signal $g_2(x)$; (d) filtered signal $g_3(x)$; (e) sampled signal $f'(i)$. The corresponding spectra are shown below the signals, with the aliased portions shown in red.

### Multi-pass transforms

The optimal approach to warping images without excessive blurring or aliasing is to adaptively pre-filter the source image at each pixel using an ideal low-pass filter, i.e., an oriented skewed sinc or low-order (e.g., cubic) approximation (Figure 3.48a). Figure 3.49 shows how this works in one dimension. The signal is first (theoretically) interpolated to a continuous waveform, (ideally) low-pass filtered to below the new Nyquist rate, and then re-sampled to the final desired resolution. In practice, the interpolation and decimation steps are concatenated into a single *polyphase* digital filtering operation (Szeliski, Winder, and Uyttendaele 2010).

For parametric transforms, the oriented two-dimensional filtering and resampling operations can be approximated using a series of one-dimensional resampling and shearing transforms (Catmull and Smith 1980; Heckbert 1989; Wolberg 1990; Gomes, Darsa, Costa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010). The advantage of using a series of one-dimensional transforms is that they are much more efficient (in terms of basic arithmetic operations) than large, non-separable, two-dimensional filter kernels.

In order to prevent aliasing, however, it may be necessary to upsample in the opposite direction before applying a shearing transformation (Szeliski, Winder, and Uyttendaele 2010). Figure 3.50 shows this process for a rotation, where a vertical upsampling stage is added before the horizontal shearing (and upsampling) stage. The upper image shows the appearance of the letter being rotated, while the lower image shows its corresponding Fourier transform.

## 3.6.2 Mesh-based warping

While parametric transforms specified by a small number of global parameters have many uses, *local* deformations with more degrees of freedom are often required.

Consider, for example, changing the appearance of a face from a frown to a smile (Figure 3.51a). What is needed in this case is to curve the corners of the mouth upwards while

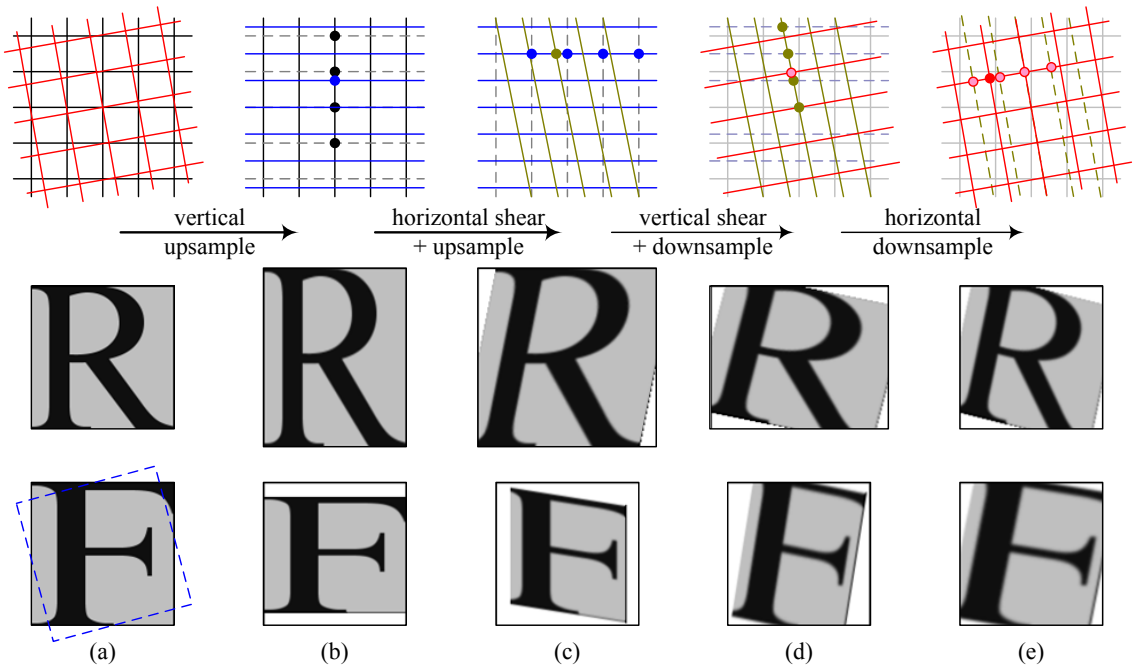



**Figure 3.50** Four-pass rotation (Szeliski, Winder, and Uyttendaele 2010): (a) original pixel grid, image, and its Fourier transform; (b) vertical upsampling; (c) horizontal shear and upsampling; (d) vertical shear and downsampling; (e) horizontal downsampling. The general affine case looks similar except that the first two stages perform general resampling.

leaving the rest of the face intact.[19] To perform such a transformation, different amounts of motion are required in different parts of the image. Figure 3.51 shows some of the commonly used approaches.

The first approach, shown in Figure 3.51a–b, is to specify a *sparse* set of corresponding points. The displacement of these points can then be interpolated to a dense *displacement field* (Chapter 8) using a variety of techniques (Nielson 1993). One possibility is to *triangulate* the set of points in one image (de Berg, Cheong, van Kreveld *et al.* 2006; Litwinowicz and Williams 1994; Buck, Finkelstein, Jacobs *et al.* 2000) and to use an *affine* motion model (Table 3.5), specified by the three triangle vertices, inside each triangle. If the destination image is triangulated according to the new vertex locations, an inverse warping algorithm (Figure 3.47) can be used. If the source image is triangulated and used as a *texture map*, computer graphics rendering algorithms can be used to draw the new image (but care must be taken along triangle edges to avoid potential aliasing).

Alternative methods for interpolating a sparse set of displacements include moving nearby quadrilateral mesh vertices, as shown in Figure 3.51a, using *variational* (energy minimizing) interpolants such as regularization (Litwinowicz and Williams 1994), see Section 3.7.1, or using locally weighted (*radial basis function*) combinations of displacements (Nielson 1993). (See (Section 12.3.1) for additional *scattered data interpolation* techniques.) If quadrilateral

[19] Rowland and Perrett (1995); Pighin, Hecker, Lischinski *et al.* (1998); Blanz and Vetter (1999); Leyvand, Cohen-Or, Dror *et al.* (2008) show more sophisticated examples of changing facial expression and appearance.

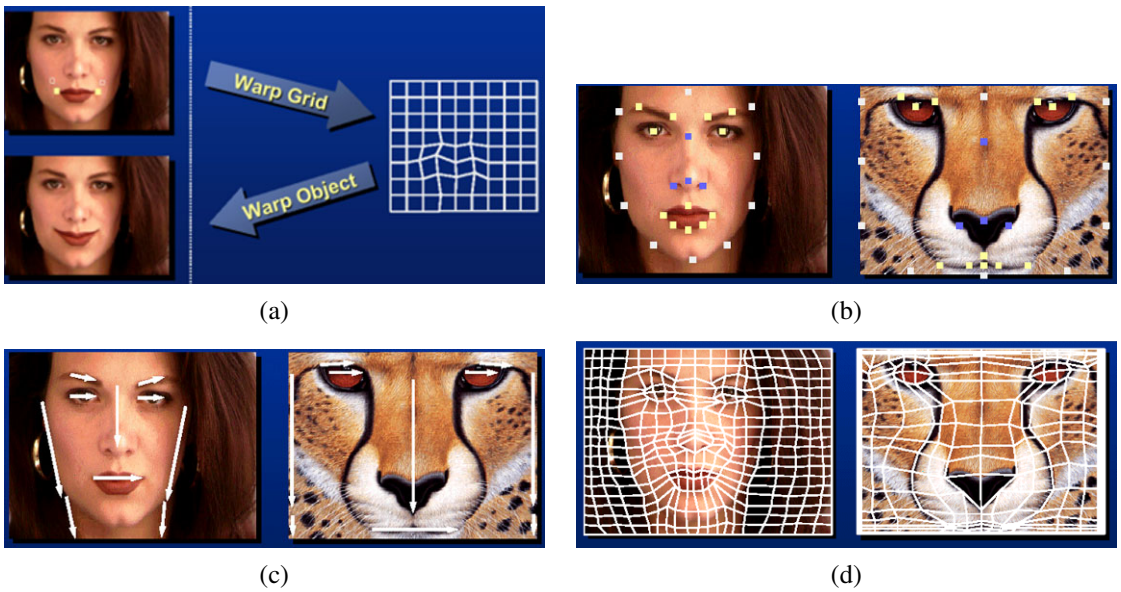


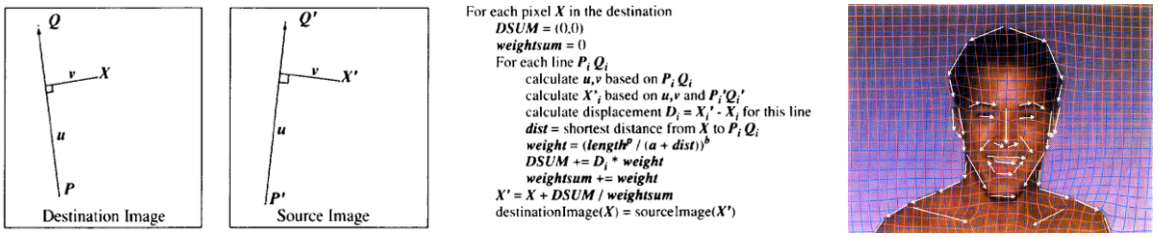

**Figure 3.51** Image warping alternatives (Gomes, Darsa, Costa *et al.* 1999) © 1999 Morgan Kaufmann: (a) sparse control points ⟶ deformation grid; (b) denser set of control point correspondences; (c) oriented line correspondences; (d) uniform quadrilateral grid.

meshes are used, it may be desirable to interpolate displacements down to individual pixel values using a smooth interpolant such as a quadratic B-spline (Farin 1996; Lee, Wolberg, Chwa *et al.* 1996).[20]

In some cases, e.g., if a dense depth map has been estimated for an image (Shade, Gortler, He *et al.* 1998), we only know the forward displacement for each pixel. As mentioned before, drawing source pixels at their destination location, i.e., forward warping (Figure 3.46), suffers from several potential problems, including aliasing and the appearance of small cracks. An alternative technique in this case is to forward warp the *displacement field* (or depth map) to its new location, fill small holes in the resulting map, and then use inverse warping to perform the resampling (Shade, Gortler, He *et al.* 1998). The reason that this generally works better than forward warping is that displacement fields tend to be much smoother than images, so the aliasing introduced during the forward warping of the displacement field is much less noticeable.

A second approach to specifying displacements for local deformations is to use corresponding *oriented line segments* (Beier and Neely 1992), as shown in Figures 3.51c and 3.52. Pixels along each line segment are transferred from source to destination exactly as specified, and other pixels are warped using a smooth interpolation of these displacements. Each line segment correspondence specifies a translation, rotation, and scaling, i.e., a *similarity transform* (Table 3.5), for pixels in its vicinity, as shown in Figure 3.52a. Line segments influence the overall displacement of the image using a weighting function that depends on the minimum distance to the line segment ($v$ in Figure 3.52a if $u \in [0, 1]$, else the shorter of the two

[20] Note that the *block-based* motion models used by many video compression standards (Le Gall 1991) can be thought of as a 0th-order (piecewise-constant) displacement field.




(a) (b) (c)

**Figure 3.52** Line-based image warping (Beier and Neely 1992) © 1992 ACM: (a) distance computation and position transfer; (b) rendering algorithm; (c) two intermediate warps used for morphing.

distances to $P$ and $Q$).

For each pixel $X$, the target location $X'$ for each line correspondence is computed along with a weight that depends on the distance and the line segment length (Figure 3.52b). The weighted average of all target locations $X'_i$ then becomes the final destination location. Note that while Beier and Neely describe this algorithm as a forward warp, an equivalent algorithm can be written by sequencing through the destination pixels. The resulting warps are not identical because line lengths or distances to lines may be different. Exercise 3.23 has you implement the Beier–Neely (line-based) warp and compare it to a number of other local deformation methods.

Yet another way of specifying correspondences in order to create image warps is to use snakes (Section 5.1.1) combined with B-splines (Lee, Wolberg, Chwa *et al.* 1996). This technique is used in Apple's Shake software and is popular in the medical imaging community.

One final possibility for specifying displacement fields is to use a mesh specifically *adapted* to the underlying image content, as shown in Figure 3.51d. Specifying such meshes by hand can involve a fair amount of work; Gomes, Darsa, Costa *et al.* (1999) describe an interactive system for doing this. Once the two meshes have been specified, intermediate warps can be generated using linear interpolation and the displacements at mesh nodes can be interpolated using splines.

### 3.6.3 *Application*: Feature-based morphing

While warps can be used to change the appearance of or to animate a *single* image, even more powerful effects can be obtained by warping and blending two or more images using a process now commonly known as *morphing* (Beier and Neely 1992; Lee, Wolberg, Chwa *et al.* 1996; Gomes, Darsa, Costa *et al.* 1999).

Figure 3.53 shows the essence of image morphing. Instead of simply cross-dissolving between two images, which leads to ghosting as shown in the top row, each image is warped toward the other image before blending, as shown in the bottom row. If the correspondences have been set up well (using any of the techniques shown in Figure 3.51), corresponding features are aligned and no ghosting results.

The above process is repeated for each intermediate frame being generated during a morph, using different blends (and amounts of deformation) at each interval. Let $t \in [0, 1]$ be

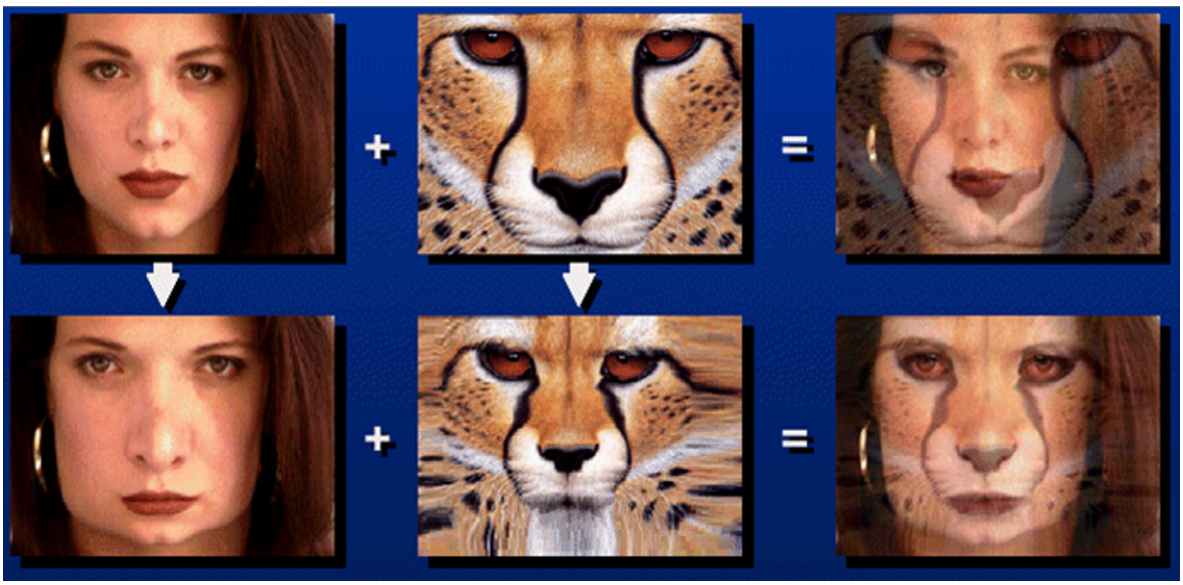


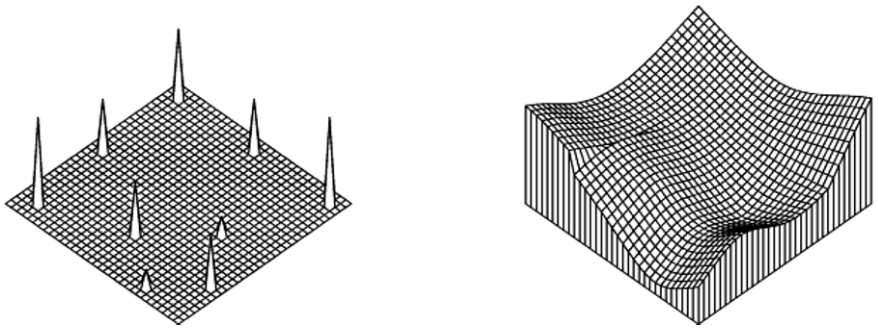
**Figure 3.53** Image morphing (Gomes, Darsa, Costa *et al.* 1999) © 1999 Morgan Kaufmann. Top row: if the two images are just blended, visible ghosting results. Bottom row: both images are first warped to the same intermediate location (e.g., halfway towards the other image) and the resulting warped images are then blended resulting in a seamless morph.

the time parameter that describes the sequence of interpolated frames. The weighting functions for the two warped images in the blend go as $(1-t)$ and $t$. Conversely, the amount of motion that image 0 undergoes at time $t$ is $t$ of the total amount of motion that is specified by the correspondences. However, some care must be taken in defining what it means to partially warp an image towards a destination, especially if the desired motion is far from linear (Sederberg, Gao, Wang *et al.* 1993). Exercise 3.25 has you implement a morphing algorithm and test it out under such challenging conditions.

## 3.7 Global optimization

So far in this chapter, we have covered a large number of image processing operators that take as input one or more images and produce some filtered or transformed version of these images. In many applications, it is more useful to first *formulate* the goals of the desired transformation using some optimization criterion and then find or infer the solution that best meets this criterion.

In this final section, we present two different (but closely related) variants on this idea. The first, which is often called *regularization* or *variational methods* (Section 3.7.1), constructs a continuous global energy function that describes the desired characteristics of the solution and then finds a minimum energy solution using sparse linear systems or related iterative techniques. The second formulates the problem using Bayesian statistics, modeling both the noisy measurement process that produced the input images as well as *prior assumptions* about the solution space, which are often encoded using a *Markov random field*



(a) (b)

**Figure 3.54** A simple surface interpolation problem: (a) nine data points of various height scattered on a grid; (b) second-order, controlled-continuity, thin-plate spline interpolator, with a tear along its left edge and a crease along its right (Szeliski 1989) © 1989 Springer.

(Section 3.7.2).

Examples of such problems include surface interpolation from scattered data (Figure 3.54), image denoising and the restoration of missing regions (Figure 3.57), and the segmentation of images into foreground and background regions (Figure 3.61).

### 3.7.1 Regularization

The theory of regularization was first developed by statisticians trying to fit models to data that severely underconstrained the solution space (Tikhonov and Arsenin 1977; Engl, Hanke, and Neubauer 1996). Consider, for example, finding a smooth surface that passes through (or near) a set of measured data points (Figure 3.54). Such a problem is described as *ill-posed* because many possible surfaces can fit this data. Since small changes in the input can sometimes lead to large changes in the fit (e.g., if we use polynomial interpolation), such problems are also often *ill-conditioned*. Since we are trying to recover the unknown function $f(x, y)$ from which the data point $d(x_i, y_i)$ were sampled, such problems are also often called *inverse problems*. Many computer vision tasks can be viewed as inverse problems, since we are trying to recover a full description of the 3D world from a limited set of images.

In order to quantify what it means to find a *smooth* solution, we can define a norm on the solution space. For one-dimensional functions $f(x)$, we can integrate the squared first derivative of the function,

$$\mathcal{E}_1 = \int f_x^2(x)\, dx \tag{3.92}$$

or perhaps integrate the squared second derivative,

$$\mathcal{E}_2 = \int f_{xx}^2(x)\, dx. \tag{3.93}$$

(Here, we use subscripts to denote differentiation.) Such energy measures are examples of *functionals*, which are operators that map functions to scalar values. They are also often called *variational methods*, because they measure the variation (non-smoothness) in a function.

In two dimensions (e.g., for images, flow fields, or surfaces), the corresponding smoothness functionals are

$$\mathcal{E}_1 = \int f_x^2(x,y) + f_y^2(x,y)\,dx\,dy = \int \|\nabla f(x,y)\|^2\,dx\,dy \tag{3.94}$$

and

$$\mathcal{E}_2 = \int f_{xx}^2(x,y) + 2f_{xy}^2(x,y) + f_{yy}^2(x,y)\,dx\,dy, \tag{3.95}$$

where the mixed $2f_{xy}^2$ term is needed to make the measure rotationally invariant (Grimson 1983).

The first derivative norm is often called the *membrane*, since interpolating a set of data points using this measure results in a tent-like structure. (In fact, this formula is a small-deflection approximation to the surface area, which is what soap bubbles minimize.) The second-order norm is called the *thin-plate spline*, since it approximates the behavior of thin plates (e.g., flexible steel) under small deformations. A blend of the two is called the *thin-plate spline under tension*; versions of these formulas where each derivative term is multiplied by a local weighting function are called *controlled-continuity splines* (Terzopoulos 1988). Figure 3.54 shows a simple example of a controlled-continuity interpolator fit to nine scattered data points. In practice, it is more common to find first-order smoothness terms used with images and flow fields (Section 8.4) and second-order smoothness associated with surfaces (Section 12.3.1).

In addition to the smoothness term, regularization also requires a data term (or *data penalty*). For scattered data interpolation (Nielson 1993), the data term measures the distance between the function $f(x,y)$ and a set of data points $d_i = d(x_i, y_i)$,

$$\mathcal{E}_d = \sum_i [f(x_i, y_i) - d_i]^2. \tag{3.96}$$

For a problem like noise removal, a continuous version of this measure can be used,

$$\mathcal{E}_d = \int [f(x,y) - d(x,y)]^2\,dx\,dy. \tag{3.97}$$

To obtain a global energy that can be minimized, the two energy terms are usually added together,

$$\mathcal{E} = \mathcal{E}_d + \lambda \mathcal{E}_s, \tag{3.98}$$

where $\mathcal{E}_s$ is the *smoothness penalty* ($\mathcal{E}_1$, $\mathcal{E}_2$ or some weighted blend) and $\lambda$ is the *regularization parameter*, which controls how smooth the solution should be.

In order to find the minimum of this continuous problem, the function $f(x,y)$ is usually first discretized on a regular grid.[21] The most principled way to perform this discretization is to use *finite element analysis*, i.e., to approximate the function with a piecewise continuous spline, and then perform the analytic integration (Bathe 2007).

Fortunately, for both the first-order and second-order smoothness functionals, the judicious selection of appropriate finite elements results in particularly simple discrete forms

[21] The alternative of using *kernel basis functions* centered on the data points (Boult and Kender 1986; Nielson 1993) is discussed in more detail in Section 12.3.1.

(Terzopoulos 1983). The corresponding *discrete* smoothness energy functions become

$$E_1 = \sum_{i,j} s_x(i,j)[f(i+1,j) - f(i,j) - g_x(i,j)]^2 + s_y(i,j)[f(i,j+1) - f(i,j) - g_y(i,j)]^2 \tag{3.99}$$

and

$$\begin{aligned} E_2 = {} & h^{-2} \sum_{i,j} c_x(i,j)[f(i+1,j) - 2f(i,j) + f(i-1,j)]^2 \\ & + 2c_m(i,j)[f(i+1,j+1) - f(i+1,j) - f(i,j+1) + f(i,j)]^2 \\ & + c_y(i,j)[f(i,j+1) - 2f(i,j) + f(i,j-1)]^2, \end{aligned} \tag{3.100}$$

where $h$ is the size of the finite element grid. The $h$ factor is only important if the energy is being discretized at a variety of resolutions, as in coarse-to-fine or multigrid techniques.

The optional smoothness weights $s_x(i,j)$ and $s_y(i,j)$ control the location of horizontal and vertical tears (or weaknesses) in the surface. For other problems, such as colorization (Levin, Lischinski, and Weiss 2004) and interactive tone mapping (Lischinski, Farbman, Uyttendaele *et al.* 2006a), they control the smoothness in the interpolated chroma or exposure field and are often set inversely proportional to the local luminance gradient strength. For second-order problems, the crease variables $c_x(i,j)$, $c_m(i,j)$, and $c_y(i,j)$ control the locations of creases in the surface (Terzopoulos 1988; Szeliski 1990a).

The data values $g_x(i,j)$ and $g_y(i,j)$ are gradient data terms (constraints) used by algorithms, such as photometric stereo (Section 12.1.1), HDR tone mapping (Section 10.2.1) (Fattal, Lischinski, and Werman 2002), Poisson blending (Section 9.3.4) (Pérez, Gangnet, and Blake 2003), and gradient-domain blending (Section 9.3.4) (Levin, Zomet, Peleg *et al.* 2004). They are set to zero when just discretizing the conventional first-order smoothness functional (3.94).

The two-dimensional discrete data energy is written as

$$E_d = \sum_{i,j} w(i,j)[f(i,j) - d(i,j)]^2, \tag{3.101}$$

where the local weights $w(i,j)$ control how strongly the data constraint is enforced. These values are set to zero where there is no data and can be set to the inverse variance of the data measurements when there is data (as discussed by Szeliski (1989) and in Section 3.7.2).

The total energy of the discretized problem can now be written as a *quadratic form*

$$E = E_d + \lambda E_s = \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} - 2\boldsymbol{x}^T \boldsymbol{b} + c, \tag{3.102}$$

where $\boldsymbol{x} = [f(0,0) \ldots f(m-1,n-1)]$ is called the *state vector*.[22]

The sparse symmetric positive-definite matrix $\boldsymbol{A}$ is called the *Hessian* since it encodes the second derivative of the energy function.[23] For the one-dimensional, first-order problem, $\boldsymbol{A}$

[22] We use $\boldsymbol{x}$ instead of $\boldsymbol{f}$ because this is the more common form in the numerical analysis literature (Golub and Van Loan 1996).

[23] In numerical analysis, $\boldsymbol{A}$ is called the *coefficient* matrix (Saad 2003); in finite element analysis (Bathe 2007), it is called the *stiffness* matrix.

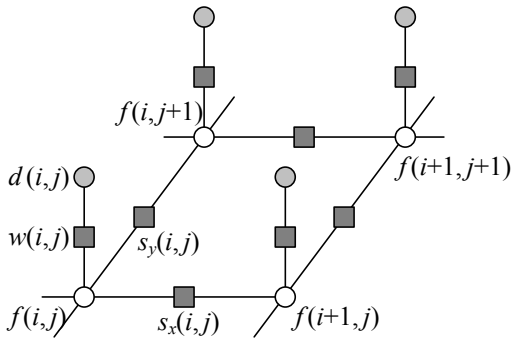



**Figure 3.55** Graphical model interpretation of first-order regularization. The white circles are the unknowns $f(i,j)$ while the dark circles are the input data $d(i,j)$. In the resistive grid interpretation, the $d$ and $f$ values encode input and output voltages and the black squares denote resistors whose *conductance* is set to $s_x(i,j)$, $s_y(i,j)$, and $w(i,j)$. In the spring-mass system analogy, the circles denote elevations and the black squares denote springs. The same graphical model can be used to depict a first-order Markov random field (Figure 3.56).

is tridiagonal; for the two-dimensional, first-order problem, it is multi-banded with five non-zero entries per row. We call $\boldsymbol{b}$ the *weighted data vector*. Minimizing the above quadratic form is equivalent to solving the sparse linear system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.103}$$

which can be done using a variety of sparse matrix techniques, such as multigrid (Briggs, Henson, and McCormick 2000) and hierarchical preconditioners (Szeliski 2006b), as described in Appendix A.5.

While regularization was first introduced to the vision community by Poggio, Torre, and Koch (1985) and Terzopoulos (1986b) for problems such as surface interpolation, it was quickly adopted by other vision researchers for such varied problems as edge detection (Section 4.2), optical flow (Section 8.4), and shape from shading (Section 12.1) (Poggio, Torre, and Koch 1985; Horn and Brooks 1986; Terzopoulos 1986b; Bertero, Poggio, and Torre 1988; Brox, Bruhn, Papenberg *et al.* 2004). Poggio, Torre, and Koch (1985) also showed how the discrete energy defined by Equations (3.100–3.101) could be implemented in a resistive grid, as shown in Figure 3.55. In computational photography (Chapter 10), regularization and its variants are commonly used to solve problems such as high-dynamic range tone mapping (Fattal, Lischinski, and Werman 2002; Lischinski, Farbman, Uyttendaele *et al.* 2006a), Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet, Peleg *et al.* 2004; Agarwala, Dontcheva, Agrawala *et al.* 2004), colorization (Levin, Lischinski, and Weiss 2004), and natural image matting (Levin, Lischinski, and Weiss 2008).

## Robust regularization

While regularization is most commonly formulated using quadratic ($L_2$) norms (compare with the squared derivatives in (3.92–3.95) and squared differences in (3.100–3.101)), it can also be formulated using non-quadratic *robust* penalty functions (Appendix B.3). For exam-

ple, (3.100) can be generalized to

$$E_{1r} = \sum_{i,j} s_x(i,j)\rho(f(i+1,j) - f(i,j)) + s_y(i,j)\rho(f(i,j+1) - f(i,j)), \tag{3.104}$$

where $\rho(x)$ is some monotonically increasing penalty function. For example, the family of norms $\rho(x) = |x|^p$ is called $p$-norms. When $p < 2$, the resulting smoothness terms become more piecewise continuous than totally smooth, which can better model the discontinuous nature of images, flow fields, and 3D surfaces.

An early example of robust regularization is the *graduated non-convexity* (GNC) algorithm introduced by Blake and Zisserman (1987). Here, the norms on the data and derivatives are clamped to a maximum value

$$\rho(x) = \min(x^2, V). \tag{3.105}$$

Because the resulting problem is highly non-convex (it has many local minima), a *continuation* method is proposed, where a quadratic norm (which is convex) is gradually replaced by the non-convex robust norm (Allgower and Georg 2003). (Around the same time, Terzopoulos (1988) was also using continuation to infer the tear and crease variables in his surface interpolation problems.)

Today, it is more common to use the $L_1$ ($p = 1$) norm, which is often called *total variation* (Chan, Osher, and Shen 2001; Tschumperlé and Deriche 2005; Tschumperlé 2006; Kaftory, Schechner, and Zeevi 2007). Other norms, for which the *influence* (derivative) more quickly decays to zero, are presented by Black and Rangarajan (1996); Black, Sapiro, Marimont *et al.* (1998) and discussed in Appendix B.3.

Even more recently, *hyper-Laplacian* norms with $p < 1$ have gained popularity, based on the observation that the log-likelihood distribution of image derivatives follows a $p \approx 0.5 - 0.8$ slope and is therefore a hyper-Laplacian distribution (Simoncelli 1999; Levin and Weiss 2007; Weiss and Freeman 2007; Krishnan and Fergus 2009). Such norms have an even stronger tendency to prefer large discontinuities over small ones. See the related discussion in Section 3.7.2 (3.114).

While least squares regularized problems using $L_2$ norms can be solved using linear systems, other $p$-norms require different iterative techniques, such as iteratively reweighted least squares (IRLS), Levenberg–Marquardt, or alternation between local non-linear subproblems and global quadratic regularization (Krishnan and Fergus 2009). Such techniques are discussed in Section 6.1.3 and Appendices A.3 and B.3.

### 3.7.2 Markov random fields

As we have just seen, regularization, which involves the minimization of energy functionals defined over (piecewise) continuous functions, can be used to formulate and solve a variety of low-level computer vision problems. An alternative technique is to formulate a *Bayesian* model, which separately models the noisy image formation (*measurement*) process, as well as assuming a statistical *prior* model over the solution space. In this section, we look at priors based on Markov random fields, whose log-likelihood can be described using local neighborhood interaction (or penalty) terms (Kindermann and Snell 1980; Geman and Geman 1984; Marroquin, Mitter, and Poggio 1987; Li 1995; Szeliski, Zabih, Scharstein *et al.* 2008).

The use of Bayesian modeling has several potential advantages over regularization (see also Appendix B). The ability to model measurement processes statistically enables us to extract the maximum information possible from each measurement, rather than just guessing what weighting to give the data. Similarly, the parameters of the prior distribution can often be *learned* by observing samples from the class we are modeling (Roth and Black 2007a; Tappen 2007; Li and Huttenlocher 2008). Furthermore, because our model is probabilistic, it is possible to estimate (in principle) complete probability *distributions* over the unknowns being recovered and, in particular, to model the *uncertainty* in the solution, which can be useful in latter processing stages. Finally, Markov random field models can be defined over *discrete* variables, such as image labels (where the variables have no proper ordering), for which regularization does not apply.

Recall from (3.68) in Section 3.4.3 (or see Appendix B.4) that, according to Bayes' Rule, the *posterior* distribution for a given set of measurements $\boldsymbol{y}$, $p(\boldsymbol{y}|\boldsymbol{x})$, combined with a prior $p(\boldsymbol{x})$ over the unknowns $\boldsymbol{x}$, is given by

$$p(\boldsymbol{x}|\boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x})}{p(\boldsymbol{y})}, \tag{3.106}$$

where $p(\boldsymbol{y}) = \int_{\boldsymbol{x}} p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x})$ is a normalizing constant used to make the $p(\boldsymbol{x}|\boldsymbol{y})$ distribution *proper* (integrate to 1). Taking the negative logarithm of both sides of (3.106), we get

$$-\log p(\boldsymbol{x}|\boldsymbol{y}) = -\log p(\boldsymbol{y}|\boldsymbol{x}) - \log p(\boldsymbol{x}) + C, \tag{3.107}$$

which is the *negative posterior log likelihood*.

To find the most likely (*maximum a posteriori* or MAP) solution $\boldsymbol{x}$ given some measurements $\boldsymbol{y}$, we simply minimize this negative log likelihood, which can also be thought of as an *energy*,

$$E(\boldsymbol{x}, \boldsymbol{y}) = E_d(\boldsymbol{x}, \boldsymbol{y}) + E_p(\boldsymbol{x}). \tag{3.108}$$

(We drop the constant $C$ because its value does not matter during energy minimization.) The first term $E_d(\boldsymbol{x}, \boldsymbol{y})$ is the *data energy* or *data penalty*; it measures the negative log likelihood that the data were observed given the unknown state $\boldsymbol{x}$. The second term $E_p(\boldsymbol{x})$ is the *prior energy*; it plays a role analogous to the smoothness energy in regularization. Note that the MAP estimate may not always be desirable, since it selects the "peak" in the posterior distribution rather than some more stable statistic—see the discussion in Appendix B.2 and by Levin, Weiss, Durand *et al.* (2009).

For image processing applications, the unknowns $\boldsymbol{x}$ are the set of output pixels

$$\boldsymbol{x} = [f(0,0) \ldots f(m-1, n-1)],$$

and the data are (in the simplest case) the input pixels

$$\boldsymbol{y} = [d(0,0) \ldots d(m-1, n-1)]$$

as shown in Figure 3.56.

For a Markov random field, the probability $p(\boldsymbol{x})$ is a *Gibbs* or *Boltzmann distribution*, whose negative log likelihood (according to the Hammersley–Clifford theorem) can be written as a sum of pairwise interaction potentials,

$$E_p(\boldsymbol{x}) = \sum_{\{(i,j),(k,l)\} \in \mathcal{N}} V_{i,j,k,l}(f(i,j), f(k,l)), \tag{3.109}$$

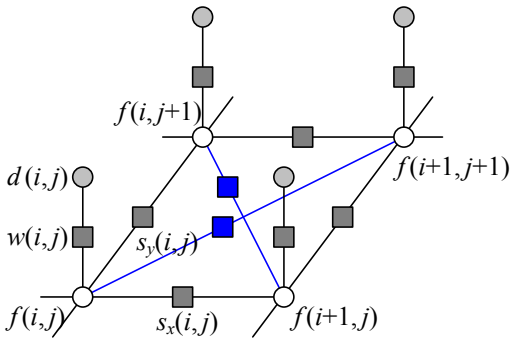



**Figure 3.56** Graphical model for an $\mathcal{N}_4$ neighborhood Markov random field. (The blue edges are added for an $\mathcal{N}_8$ neighborhood.) The white circles are the unknowns $f(i,j)$, while the dark circles are the input data $d(i,j)$. The $s_x(i,j)$ and $s_y(i,j)$ black boxes denote arbitrary *interaction potentials* between adjacent nodes in the random field, and the $w(i,j)$ denote the *data penalty* functions. The same graphical model can be used to depict a discrete version of a first-order regularization problem (Figure 3.55).

where $\mathcal{N}(i,j)$ denotes the *neighbors* of pixel $(i,j)$. In fact, the general version of the theorem says that the energy may have to be evaluated over a larger set of *cliques*, which depend on the *order* of the Markov random field (Kindermann and Snell 1980; Geman and Geman 1984; Bishop 2006; Kohli, Ladický, and Torr 2009; Kohli, Kumar, and Torr 2009).

The most commonly used neighborhood in Markov random field modeling is the $\mathcal{N}_4$ neighborhood, where each pixel in the field $f(i,j)$ interacts only with its immediate neighbors. The model in Figure 3.56, which we previously used in Figure 3.55 to illustrate the discrete version of first-order regularization, shows an $\mathcal{N}_4$ MRF. The $s_x(i,j)$ and $s_y(i,j)$ black boxes denote arbitrary *interaction potentials* between adjacent nodes in the random field and the $w(i,j)$ denote the data penalty functions. These square nodes can also be interpreted as *factors* in a *factor graph* version of the (undirected) graphical model (Bishop 2006), which is another name for interaction potentials. (Strictly speaking, the factors are (improper) probability functions whose product is the (un-normalized) posterior distribution.)

As we will see in (3.112–3.113), there is a close relationship between these interaction potentials and the discretized versions of regularized image restoration problems. Thus, to a first approximation, we can view energy minimization being performed when solving a regularized problem and the maximum a posteriori inference being performed in an MRF as equivalent.

While $\mathcal{N}_4$ neighborhoods are most commonly used, in some applications $\mathcal{N}_8$ (or even higher order) neighborhoods perform better at tasks such as image segmentation because they can better model discontinuities at different orientations (Boykov and Kolmogorov 2003; Rother, Kohli, Feng *et al.* 2009; Kohli, Ladický, and Torr 2009; Kohli, Kumar, and Torr 2009).

### Binary MRFs

The simplest possible example of a Markov random field is a binary field. Examples of such fields include 1-bit (black and white) scanned document images as well as images segmented into foreground and background regions.

To denoise a scanned image, we set the data penalty to reflect the agreement between the

scanned and final images,

$$E_d(i,j) = w\delta(f(i,j), d(i,j)) \tag{3.110}$$

and the smoothness penalty to reflect the agreement between neighboring pixels

$$E_p(i,j) = E_x(i,j) + E_y(i,j) = s\delta(f(i,j), f(i+1,j)) + s\delta(f(i,j), f(i,j+1)). \tag{3.111}$$

Once we have formulated the energy, how do we minimize it? The simplest approach is to perform gradient descent, flipping one state at a time if it produces a lower energy. This approach is known as *contextual classification* (Kittler and Föglein 1984), *iterated conditional modes* (ICM) (Besag 1986), or *highest confidence first* (HCF) (Chou and Brown 1990) if the pixel with the largest energy decrease is selected first.

Unfortunately, these downhill methods tend to get easily stuck in local minima. An alternative approach is to add some randomness to the process, which is known as *stochastic gradient descent* (Metropolis, Rosenbluth, Rosenbluth *et al.* 1953; Geman and Geman 1984). When the amount of noise is decreased over time, this technique is known as *simulated annealing* (Kirkpatrick, Gelatt, and Vecchi 1983; Carnevali, Coletti, and Patarnello 1985; Wolberg and Pavlidis 1985; Swendsen and Wang 1987) and was first popularized in computer vision by Geman and Geman (1984) and later applied to stereo matching by Barnard (1989), among others.

Even this technique, however, does not perform that well (Boykov, Veksler, and Zabih 2001). For binary images, a much better technique, introduced to the computer vision community by Boykov, Veksler, and Zabih (2001) is to re-formulate the energy minimization as a *max-flow/min-cut* graph optimization problem (Greig, Porteous, and Seheult 1989). This technique has informally come to be known as *graph cuts* in the computer vision community (Boykov and Kolmogorov 2010). For simple energy functions, e.g., those where the penalty for non-identical neighboring pixels is a constant, this algorithm is guaranteed to produce the *global minimum*. Kolmogorov and Zabih (2004) formally characterize the class of binary energy potentials (*regularity conditions*) for which these results hold, while newer work by Komodakis, Tziritas, and Paragios (2008) and Rother, Kolmogorov, Lempitsky *et al.* (2007) provide good algorithms for the cases when they do not.

In addition to the above mentioned techniques, a number of other optimization approaches have been developed for MRF energy minimization, such as (loopy) belief propagation and dynamic programming (for one-dimensional problems). These are discussed in more detail in Appendix B.5 as well as the comparative survey paper by Szeliski, Zabih, Scharstein *et al.* (2008).

### Ordinal-valued MRFs

In addition to binary images, Markov random fields can be applied to ordinal-valued labels such as grayscale images or depth maps. The term "ordinal" indicates that the labels have an implied ordering, e.g., that higher values are lighter pixels. In the next section, we look at unordered labels, such as source image labels for image compositing.

In many cases, it is common to extend the binary data and smoothness prior terms as

$$E_d(i,j) = w(i,j)\rho_d(f(i,j) - d(i,j)) \tag{3.112}$$

(a) (b) (c) (d)

**Figure 3.57** Grayscale image denoising and inpainting: (a) original image; (b) image corrupted by noise and with missing data (black bar); (c) image restored using loopy belief propagation; (d) image restored using expansion move graph cuts. Images are from http://vision.middlebury.edu/MRF/results/ (Szeliski, Zabih, Scharstein *et al.* 2008).

and

$$E_p(i,j) = s_x(i,j)\rho_p(f(i,j) - f(i+1,j)) + s_y(i,j)\rho_p(f(i,j) - f(i,j+1)), \quad (3.113)$$

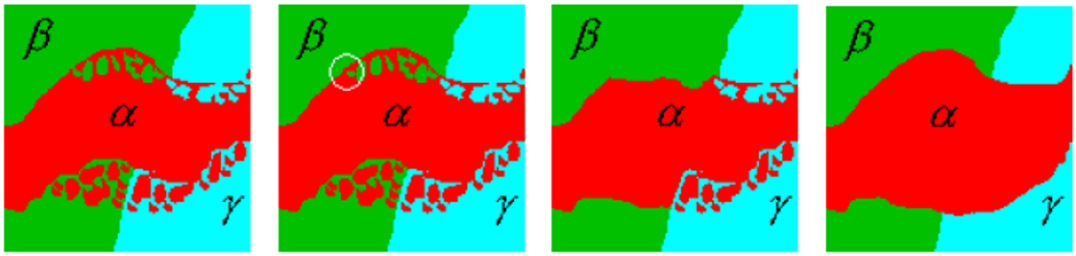
which are robust generalizations of the quadratic penalty terms (3.101) and (3.100), first introduced in (3.105). As before, the $w(i,j)$, $s_x(i,j)$ and $s_y(i,j)$ weights can be used to locally control the data weighting and the horizontal and vertical smoothness. Instead of using a quadratic penalty, however, a general monotonically increasing penalty function $\rho()$ is used. (Different functions can be used for the data and smoothness terms.) For example, $\rho_p$ can be a hyper-Laplacian penalty

$$\rho_p(d) = |d|^p, \;\; p < 1, \quad (3.114)$$

which better encodes the distribution of gradients (mainly edges) in an image than either a quadratic or linear (total variation) penalty.[24] Levin and Weiss (2007) use such a penalty to separate a transmitted and reflected image (Figure 8.17) by encouraging gradients to lie in one or the other image, but not both. More recently, Levin, Fergus, Durand *et al.* (2007) use the hyper-Laplacian as a prior for image deconvolution (deblurring) and Krishnan and Fergus (2009) develop a faster algorithm for solving such problems. For the data penalty, $\rho_d$ can be quadratic (to model Gaussian noise) or the log of a *contaminated Gaussian* (Appendix B.3).

When $\rho_p$ is a quadratic function, the resulting Markov random field is called a Gaussian Markov random field (GMRF) and its minimum can be found by sparse linear system solving (3.103). When the weighting functions are uniform, the GMRF becomes a special case of Wiener filtering (Section 3.4.3). Allowing the weighting functions to depend on the input image (a special kind of conditional random field, which we describe below) enables quite sophisticated image processing algorithms to be performed, including colorization (Levin, Lischinski, and Weiss 2004), interactive tone mapping (Lischinski, Farbman, Uyttendaele *et*

[24] Note that, unlike a quadratic penalty, the sum of the horizontal and vertical derivative $p$-norms is not rotationally invariant. A better approach may be to locally estimate the gradient direction and to impose different norms on the perpendicular and parallel components, which Roth and Black (2007b) call a *steerable random field*.




(a) initial labeling (b) standard move (c) $\alpha$-$\beta$-swap (d) $\alpha$-expansion

**Figure 3.58** Multi-level graph optimization from (Boykov, Veksler, and Zabih 2001) © 2001 IEEE: (a) initial problem configuration; (b) the standard move only changes one pixel; (c) the $\alpha$-$\beta$-swap optimally exchanges all $\alpha$ and $\beta$-labeled pixels; (d) the $\alpha$-expansion move optimally selects among current pixel values and the $\alpha$ label.

*al.* 2006a), natural image matting (Levin, Lischinski, and Weiss 2008), and image restoration (Tappen, Liu, Freeman *et al.* 2007).

When $\rho_d$ or $\rho_p$ are non-quadratic functions, gradient descent techniques such as non-linear least squares or iteratively re-weighted least squares can sometimes be used (Appendix A.3). However, if the search space has lots of local minima, as is the case for stereo matching (Barnard 1989; Boykov, Veksler, and Zabih 2001), more sophisticated techniques are required.

The extension of graph cut techniques to multi-valued problems was first proposed by Boykov, Veksler, and Zabih (2001). In their paper, they develop two different algorithms, called the *swap move* and the *expansion move*, which iterate among a series of binary labeling sub-problems to find a good solution (Figure 3.58). Note that a global solution is generally not achievable, as the problem is provably NP-hard for general energy functions. Because both these algorithms use a binary MRF optimization inside their inner loop, they are subject to the kind of constraints on the energy functions that occur in the binary labeling case (Kolmogorov and Zabih 2004). Appendix B.5.4 discusses these algorithms in more detail, along with some more recently developed approaches to this problem.

Another MRF inference technique is *belief propagation* (BP). While belief propagation was originally developed for inference over trees, where it is exact (Pearl 1988), it has more recently been applied to graphs with loops such as Markov random fields (Freeman, Pasztor, and Carmichael 2000; Yedidia, Freeman, and Weiss 2001). In fact, some of the better performing stereo-matching algorithms use loopy belief propagation (LBP) to perform their inference (Sun, Zheng, and Shum 2003). LBP is discussed in more detail in Appendix B.5.3 as well as the comparative survey paper on MRF optimization (Szeliski, Zabih, Scharstein *et al.* 2008).

Figure 3.57 shows an example of image denoising and inpainting (hole filling) using a non-quadratic energy function (non-Gaussian MRF). The original image has been corrupted by noise and a portion of the data has been removed (the black bar). In this case, the loopy belief propagation algorithm computes a slightly lower energy and also a smoother image than the alpha-expansion graph cut algorithm.

Of course, the above formula (3.113) for the smoothness term $E_p(i, j)$ just shows the simplest case. In more recent work, Roth and Black (2009) propose a *Field of Experts* (FoE) model, which sums up a large number of exponentiated local filter outputs to arrive at the

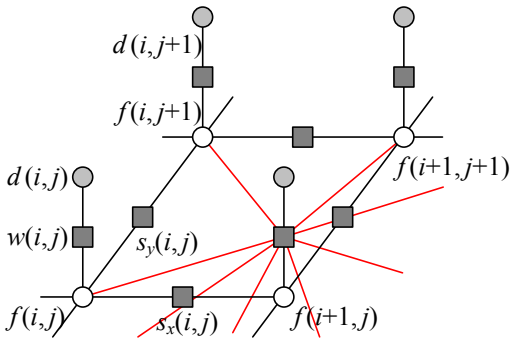



**Figure 3.59** Graphical model for a Markov random field with a more complex measurement model. The additional colored edges show how combinations of unknown values (say, in a sharp image) produce the measured values (a noisy blurred image). The resulting graphical model is still a classic MRF and is just as easy to sample from, but some inference algorithms (e.g., those based on graph cuts) may not be applicable because of the increased network complexity, since state changes during the inference become more entangled and the posterior MRF has much larger cliques.

smoothness penalty. Weiss and Freeman (2007) analyze this approach and compare it to the simpler hyper-Laplacian model of natural image statistics. Lyu and Simoncelli (2009) use *Gaussian Scale Mixtures* (GSMs) to construct an inhomogeneous multi-scale MRF, with one (positive exponential) GMRF modulating the variance (amplitude) of another Gaussian MRF.

It is also possible to extend the *measurement* model to make the sampled (noise-corrupted) input pixels correspond to blends of unknown (latent) image pixels, as in Figure 3.59. This is the commonly occurring case when trying to de-blur an image. While this kind of a model is still a traditional generative Markov random field, finding an optimal solution can be difficult because the clique sizes get larger. In such situations, gradient descent techniques, such as iteratively reweighted least squares, can be used (Joshi, Zitnick, Szeliski *et al.* 2009). Exercise 3.31 has you explore some of these issues.

### Unordered labels

Another case with multi-valued labels where Markov random fields are often applied are *unordered labels*, i.e., labels where there is no semantic meaning to the numerical difference between the values of two labels. For example, if we are classifying terrain from aerial imagery, it makes no sense to take the numeric difference between the labels assigned to forest, field, water, and pavement. In fact, the adjacencies of these various kinds of terrain each have different likelihoods, so it makes more sense to use a prior of the form

$$E_p(i,j) = s_x(i,j)V(l(i,j), l(i+1,j)) + s_y(i,j)V(l(i,j), l(i,j+1)), \tag{3.115}$$

where $V(l_0, l_1)$ is a general *compatibility* or *potential* function. (Note that we have also replaced $f(i,j)$ with $l(i,j)$ to make it clearer that these are labels rather than discrete function samples.) An alternative way to write this prior energy (Boykov, Veksler, and Zabih 2001;

**Figure 3.60** An unordered label MRF (Agarwala, Dontcheva, Agrawala *et al.* 2004) © 2004 ACM: Strokes in each of the source images on the left are used as constraints on an MRF optimization, which is solved using graph cuts. The resulting multi-valued label field is shown as a color overlay in the middle image, and the final composite is shown on the right.

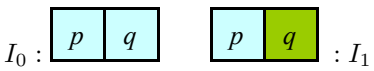
Szeliski, Zabih, Scharstein *et al.* 2008) is

$$E_p = \sum_{(p,q)\in\mathcal{N}} V_{p,q}(l_p, l_q), \tag{3.116}$$

where the $(p, q)$ are neighboring pixels and a spatially varying potential function $V_{p,q}$ is evaluated for each neighboring pair.

An important application of unordered MRF labeling is seam finding in image compositing (Davis 1998; Agarwala, Dontcheva, Agrawala *et al.* 2004) (see Figure 3.60, which is explained in more detail in Section 9.3.2). Here, the compatibility $V_{p,q}(l_p, l_q)$ measures the quality of the visual appearance that would result from placing a pixel $p$ from image $l_p$ next to a pixel $q$ from image $l_q$. As with most MRFs, we assume that $V_{p,q}(l, l) = 0$, i.e., it is perfectly fine to choose contiguous pixels from the same image. For different labels, however, the compatibility $V_{p,q}(l_p, l_q)$ may depend on the values of the underlying pixels $I_{l_p}(p)$ and $I_{l_q}(q)$.

Consider, for example, where one image $I_0$ is all sky blue, i.e., $I_0(p) = I_0(q) = B$, while the other image $I_1$ has a transition from sky blue, $I_1(p) = B$, to forest green, $I_1(q) = G$.




In this case, $V_{p,q}(1, 0) = 0$ (the colors agree), while $V_{p,q}(0, 1) > 0$ (the colors disagree).

### Conditional random fields

In a classic Bayesian model (3.106–3.108),

$$p(\boldsymbol{x}|\boldsymbol{y}) \propto p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x}), \tag{3.117}$$

the prior distribution $p(\boldsymbol{x})$ is independent of the observations $\boldsymbol{y}$. Sometimes, however, it is useful to modify our prior assumptions, say about the smoothness of the field we are trying to estimate, in response to the sensed data. Whether this makes sense from a probability viewpoint is something we discuss once we have explained the new model.

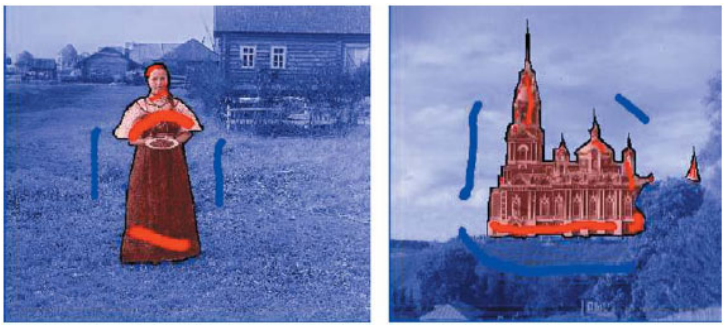


**Figure 3.61** Image segmentation (Boykov and Funka-Lea 2006) © 2006 Springer: The user draws a few red strokes in the foreground object and a few blue ones in the background. The system computes color distributions for the foreground and background and solves a binary MRF. The smoothness weights are modulated by the intensity gradients (edges), which makes this a conditional random field (CRF).

Consider the interactive image segmentation problem shown in Figure 3.61 (Boykov and Funka-Lea 2006). In this application, the user draws foreground (red) and background (blue) strokes, and the system then solves a binary MRF labeling problem to estimate the extent of the foreground object. In addition to minimizing a data term, which measures the pointwise similarity between pixel colors and the inferred region distributions (Section 5.5), the MRF is modified so that the smoothness terms $s_x(x, y)$ and $s_y(x, y)$ in Figure 3.56 and (3.113) depend on the magnitude of the gradient between adjacent pixels.[25]

Since the smoothness term now depends on the data, Bayes' Rule (3.117) no longer applies. Instead, we use a direct model for the posterior distribution $p(\boldsymbol{x}|\boldsymbol{y})$, whose negative log likelihood can be written as

$$\begin{aligned} E(\boldsymbol{x}|\boldsymbol{y}) &= E_d(\boldsymbol{x}, \boldsymbol{y}) + E_s(\boldsymbol{x}, \boldsymbol{y}) \\ &= \sum_p V_p(x_p, \boldsymbol{y}) + \sum_{(p,q)\in\mathcal{N}} V_{p,q}(x_p, x_q, \boldsymbol{y}), \end{aligned} \tag{3.118}$$

using the notation introduced in (3.116). The resulting probability distribution is called a *conditional random field* (CRF) and was first introduced to the computer vision field by Kumar and Hebert (2003), based on earlier work in text modeling by Lafferty, McCallum, and Pereira (2001).

Figure 3.62 shows a graphical model where the smoothness terms depend on the data values. In this particular model, each smoothness term depends only on its adjacent pair of data values, i.e., terms are of the form $V_{p,q}(x_p, x_q, y_p, y_q)$ in (3.118).

The idea of modifying smoothness terms in response to input data is not new. For example, Boykov and Jolly (2001) used this idea for interactive segmentation, as shown in Figure 3.61, and it is now widely used in image segmentation (Section 5.5) (Blake, Rother, Brown *et al.* 2004; Rother, Kolmogorov, and Blake 2004), denoising (Tappen, Liu, Freeman *et al.* 2007), and object recognition (Section 14.4.3) (Winn and Shotton 2006; Shotton, Winn, Rother *et al.* 2009).

[25] An alternative formulation that also uses detected edges to modulate the smoothness of a depth or motion field and hence to integrate multiple lower level vision modules is presented by Poggio, Gamble, and Little (1988).

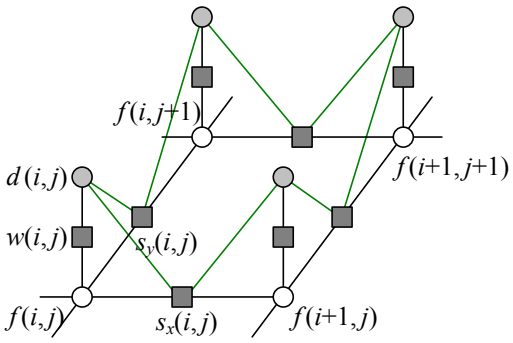



**Figure 3.62** Graphical model for a conditional random field (CRF). The additional green edges show how combinations of sensed data influence the smoothness in the underlying MRF prior model, i.e., $s_x(i,j)$ and $s_y(i,j)$ in (3.113) depend on adjacent $d(i,j)$ values. These additional links (factors) enable the smoothness to depend on the input data. However, they make sampling from this MRF more complex.

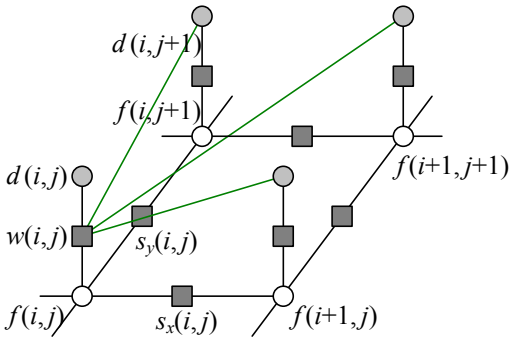



**Figure 3.63** Graphical model for a discriminative random field (DRF). The additional green edges show how combinations of sensed data, e.g., $d(i,j+1)$, influence the data term for $f(i,j)$. The generative model is therefore more complex, i.e., we cannot just apply a simple function to the unknown variables and add noise.

In stereo matching, the idea of encouraging disparity discontinuities to coincide with intensity edges goes back even further to the early days of optimization and MRF-based algorithms (Poggio, Gamble, and Little 1988; Fua 1993; Bobick and Intille 1999; Boykov, Veksler, and Zabih 2001) and is discussed in more detail in (Section 11.5).

In addition to using smoothness terms that adapt to the input data, Kumar and Hebert (2003) also compute a neighborhood function over the input data for each $V_p(x_p, \boldsymbol{y})$ term, as illustrated in Figure 3.63, instead of using the classic unary MRF data term $V_p(x_p, y_p)$ shown in Figure 3.56.[26] Because such neighborhood functions can be thought of as *discriminant* functions (a term widely used in machine learning (Bishop 2006)), they call the resulting graphical model a *discriminative random field* (DRF). In their paper, Kumar and Hebert (2006) show that DRFs outperform similar CRFs on a number of applications, such as structure detection (Figure 3.64) and binary image denoising.

Here again, one could argue that previous stereo correspondence algorithms also look at

[26] Kumar and Hebert (2006) call the unary potentials $V_p(x_p, \boldsymbol{y})$ *association potentials* and the pairwise potentials $V_{p,q}(x_p, y_q, \boldsymbol{y})$ *interaction potentials*.

**Figure 3.64** Structure detection results using an MRF (left) and a DRF (right) (Kumar and Hebert 2006) © 2006 Springer.

a neighborhood of input data, either explicitly, because they compute correlation measures (Criminisi, Cross, Blake *et al.* 2006) as data terms, or implicitly, because even pixel-wise disparity costs look at several pixels in either the left or right image (Barnard 1989; Boykov, Veksler, and Zabih 2001).

What, then are the advantages and disadvantages of using conditional or discriminative random fields instead of MRFs?

Classic Bayesian inference (MRF) assumes that the prior distribution of the data is independent of the measurements. This makes a lot of sense: if you see a pair of sixes when you first throw a pair of dice, it would be unwise to assume that they will always show up thereafter. However, if after playing for a long time you detect a statistically significant bias, you may want to adjust your prior. What CRFs do, in essence, is to select or modify the prior model based on observed data. This can be viewed as making a partial inference over additional hidden variables or correlations between the unknowns (say, a label, depth, or clean image) and the knowns (observed images).

In some cases, the CRF approach makes a lot of sense and is, in fact, the only plausible way to proceed. For example, in grayscale image colorization (Section 10.3.2) (Levin, Lischinski, and Weiss 2004), the best way to transfer the continuity information from the input grayscale image to the unknown color image is to modify local smoothness constraints. Similarly, for simultaneous segmentation and recognition (Winn and Shotton 2006; Shotton, Winn, Rother *et al.* 2009), it makes a lot of sense to permit strong color edges to influence the semantic image label continuities.

In other cases, such as image denoising, the situation is more subtle. Using a non-quadratic (robust) smoothness term as in (3.113) plays a qualitatively similar role to setting the smoothness based on local gradient information in a Gaussian MRF (GMRF) (Tappen, Liu, Freeman *et al.* 2007). (In more recent work, Tanaka and Okutomi (2008) use a larger neighborhood and full covariance matrix on a related Gaussian MRF.) The advantage of Gaussian MRFs, when the smoothness can be correctly inferred, is that the resulting quadratic energy can be minimized in a single step. However, for situations where the discontinuities are not self-evident in the input data, such as for piecewise-smooth sparse data interpolation (Blake and Zisserman 1987; Terzopoulos 1988), classic robust smoothness energy minimization may be preferable. Thus, as with most computer vision algorithms, a careful analysis of

the problem at hand and desired robustness and computation constraints may be required to choose the best technique.

Perhaps the biggest advantage of CRFs and DRFs, as argued by Kumar and Hebert (2006), Tappen, Liu, Freeman *et al.* (2007) and Blake, Rother, Brown *et al.* (2004), is that learning the model parameters is sometimes easier. While learning parameters in MRFs and their variants is not a topic that we cover in this book, interested readers can find more details in recently published articles (Kumar and Hebert 2006; Roth and Black 2007a; Tappen, Liu, Freeman *et al.* 2007; Tappen 2007; Li and Huttenlocher 2008).

### 3.7.3 *Application*: Image restoration

In Section 3.4.4, we saw how two-dimensional linear and non-linear filters can be used to remove noise or enhance sharpness in images. Sometimes, however, images are degraded by larger problems, such as scratches and blotches (Kokaram 2004). In this case, Bayesian methods such as MRFs, which can model spatially varying per-pixel measurement noise, can be used instead. An alternative is to use hole filling or inpainting techniques (Bertalmio, Sapiro, Caselles *et al.* 2000; Bertalmio, Vese, Sapiro *et al.* 2003; Criminisi, Pérez, and Toyama 2004), as discussed in Sections 5.1.4 and 10.5.1.

Figure 3.57 shows an example of image denoising and inpainting (hole filling) using a Markov random field. The original image has been corrupted by noise and a portion of the data has been removed. In this case, the loopy belief propagation algorithm computes a slightly lower energy and also a smoother image than the alpha-expansion graph cut algorithm.

## 3.8 Additional reading

If you are interested in exploring the topic of image processing in more depth, some popular textbooks have been written by Lim (1990); Crane (1997); Gomes and Velho (1997); Jähne (1997); Pratt (2007); Russ (2007); Burger and Burge (2008); Gonzales and Woods (2008). The pre-eminent conference and journal in this field are the IEEE Conference on Image Processsing and the IEEE Transactions on Image Processing.

For image compositing operators, the seminal reference is by Porter and Duff (1984) while Blinn (1994a,b) provides a more detailed tutorial. For image compositing, Smith and Blinn (1996) were the first to bring this topic to the attention of the graphics community, while Wang and Cohen (2007a) provide a recent in-depth survey.

In the realm of linear filtering, Freeman and Adelson (1991) provide a great introduction to separable and steerable oriented band-pass filters, while Perona (1995) shows how to approximate any filter as a sum of separable components.

The literature on non-linear filtering is quite wide and varied; it includes such topics as bilateral filtering (Tomasi and Manduchi 1998; Durand and Dorsey 2002; Paris and Durand 2006; Chen, Paris, and Durand 2007; Paris, Kornprobst, Tumblin *et al.* 2008), related iterative algorithms (Saint-Marc, Chen, and Medioni 1991; Nielsen, Florack, and Deriche 1997; Black, Sapiro, Marimont *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 1998; Weickert 1998; Barash 2002; Scharr, Black, and Haussecker 2003; Barash and Comaniciu 2004),

and variational approaches (Chan, Osher, and Shen 2001; Tschumperlé and Deriche 2005; Tschumperlé 2006; Kaftory, Schechner, and Zeevi 2007).

Good references to image morphology include (Haralick and Shapiro 1992, Section 5.2; Bovik 2000, Section 2.2; Ritter and Wilson 2000, Section 7; Serra 1982; Serra and Vincent 1992; Yuille, Vincent, and Geiger 1992; Soille 2006).

The classic papers for image pyramids and pyramid blending are by Burt and Adelson (1983a,b). Wavelets were first introduced to the computer vision community by Mallat (1989) and good tutorial and review papers and books are available (Strang 1989; Simoncelli and Adelson 1990b; Rioul and Vetterli 1991; Chui 1992; Meyer 1993; Sweldens 1997). Wavelets are widely used in the computer graphics community to perform multi-resolution geometric processing (Stollnitz, DeRose, and Salesin 1996) and have 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).

While image pyramids (Section 3.5.3) are usually constructed using linear filtering operators, some recent work has started investigating non-linear filters, since these can better preserve details and other salient features. Some representative papers in the computer vision literature are by Gluckman (2006a,b); Lyu and Simoncelli (2008) and in computational photography by Bae, Paris, and Durand (2006); Farbman, Fattal, Lischinski *et al.* (2008); Fattal (2009).

High-quality algorithms for image warping and resampling are covered both in the image processing literature (Wolberg 1990; Dodgson 1992; Gomes, Darsa, Costa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010) and in computer graphics (Williams 1983; Heckbert 1986; Barkans 1997; Akenine-Möller and Haines 2002), where they go under the name of *texture mapping*. Combination of image warping and image blending techniques are used to enable *morphing* between images, which is covered in a series of seminal papers and books (Beier and Neely 1992; Gomes, Darsa, Costa *et al.* 1999).

The regularization approach to computer vision problems was first introduced to the vision community by Poggio, Torre, and Koch (1985) and Terzopoulos (1986a,b, 1988) and continues to be a popular framework for formulating and solving low-level vision problems (Ju, Black, and Jepson 1996; Nielsen, Florack, and Deriche 1997; Nordström 1990; Brox, Bruhn, Papenberg *et al.* 2004; Levin, Lischinski, and Weiss 2008). More detailed mathematical treatment and additional applications can be found in the applied mathematics and statistics literature (Tikhonov and Arsenin 1977; Engl, Hanke, and Neubauer 1996).

The literature on Markov random fields is truly immense, with publications in related fields such as optimization and control theory of which few vision practitioners are even aware. A good guide to the latest techniques is the book edited by Blake, Kohli, and Rother (2010). Other recent articles that contain nice literature reviews or experimental comparisons include (Boykov and Funka-Lea 2006; Szeliski, Zabih, Scharstein *et al.* 2008; Kumar, Veksler, and Torr 2010).

The seminal paper on Markov random fields is the work of Geman and Geman (1984), who introduced this formalism to computer vision researchers and also introduced the notion of *line processes*, additional binary variables that control whether smoothness penalties are enforced or not. Black and Rangarajan (1996) showed how independent line processes

could be replaced with robust pairwise potentials; Boykov, Veksler, and Zabih (2001) developed iterative binary, graph cut algorithms for optimizing multi-label MRFs; Kolmogorov and Zabih (2004) characterized the class of binary energy potentials required for these techniques to work; and Freeman, Pasztor, and Carmichael (2000) popularized the use of loopy belief propagation for MRF inference. Many more additional references can be found in Sections 3.7.2 and 5.5, and Appendix B.5.

## 3.9 Exercises

**Ex 3.1: Color balance** Write a simple application to change the color balance of an image by multiplying each color value by a different user-specified constant. If you want to get fancy, you can make this application interactive, with sliders.

1. Do you get different results if you take out the gamma transformation before or after doing the multiplication? Why or why not?
2. Take the same picture with your digital camera using different color balance settings (most cameras control the color balance from one of the menus). Can you recover what the color balance ratios are between the different settings? You may need to put your camera on a tripod and align the images manually or automatically to make this work. Alternatively, use a color checker chart (Figure 10.3b), as discussed in Sections 2.3 and 10.1.1.
3. If you have access to the RAW image for the camera, perform the demosaicing yourself (Section 10.3.1) or downsample the image resolution to get a "true" RGB image. Does your camera perform a simple linear mapping between RAW values and the color-balanced values in a JPEG? Some high-end cameras have a RAW+JPEG mode, which makes this comparison much easier.
4. Can you think of any reason why you might want to perform a color twist (Section 3.1.2) on the images? See also Exercise 2.9 for some related ideas.

**Ex 3.2: Compositing and reflections** Section 3.1.3 describes the process of compositing an alpha-matted image on top of another. Answer the following questions and optionally validate them experimentally:

1. Most captured images have gamma correction applied to them. Does this invalidate the basic compositing equation (3.8); if so, how should it be fixed?
2. The additive (pure reflection) model may have limitations. What happens if the glass is tinted, especially to a non-gray hue? How about if the glass is dirty or smudged? How could you model wavy glass or other kinds of refractive objects?

**Ex 3.3: Blue screen matting** Set up a blue or green background, e.g., by buying a large piece of colored posterboard. Take a picture of the empty background, and then of the background with a new object in front of it. *Pull the matte* using the difference between each colored pixel and its assumed corresponding background pixel, using one of the techniques described in Section 3.1.3) or by Smith and Blinn (1996).

**Ex 3.4: Difference keying** Implement a difference keying algorithm (see Section 3.1.3) (Toyama, Krumm, Brumitt *et al.* 1999), consisting of the following steps:

1. Compute the mean and variance (or median and robust variance) at each pixel in an "empty" video sequence.
2. For each new frame, classify each pixel as foreground or background (set the background pixels to RGBA=0).
3. (Optional) Compute the alpha channel and composite over a new background.
4. (Optional) Clean up the image using morphology (Section 3.3.1), label the connected components (Section 3.3.4), compute their centroids, and track them from frame to frame. Use this to build a "people counter".

**Ex 3.5: Photo effects** Write a variety of photo enhancement or effects filters: contrast, solarization (quantization), etc. Which ones are useful (perform sensible corrections) and which ones are more creative (create unusual images)?

**Ex 3.6: Histogram equalization** Compute the gray level (luminance) histogram for an image and equalize it so that the tones look better (and the image is less sensitive to exposure settings). You may want to use the following steps:

1. Convert the color image to luminance (Section 3.1.2).
2. Compute the histogram, the cumulative distribution, and the compensation transfer function (Section 3.1.4).
3. (Optional) Try to increase the "punch" in the image by ensuring that a certain fraction of pixels (say, 5%) are mapped to pure black and white.
4. (Optional) Limit the local *gain* $f'(I)$ in the transfer function. One way to do this is to limit $f(I) < \gamma I$ or $f'(I) < \gamma$ while performing the accumulation (3.9), keeping any unaccumulated values "in reserve". (I'll let you figure out the exact details.)
5. Compensate the luminance channel through the lookup table and re-generate the color image using color ratios (2.116).
6. (Optional) Color values that are *clipped* in the original image, i.e., have one or more saturated color channels, may appear unnatural when remapped to a non-clipped value. Extend your algorithm to handle this case in some useful way.

**Ex 3.7: Local histogram equalization** Compute the gray level (luminance) histograms for each patch, but add to vertices based on distance (a spline).

1. Build on Exercise 3.6 (luminance computation).
2. Distribute values (counts) to adjacent vertices (bilinear).
3. Convert to CDF (look-up functions).
4. (Optional) Use low-pass filtering of CDFs.

5. Interpolate adjacent CDFs for final lookup.

**Ex 3.8: Padding for neighborhood operations** Write down the formulas for computing the padded pixel values $\tilde{f}(i,j)$ as a function of the original pixel values $f(k,l)$ and the image width and height $(M,N)$ for *each* of the padding modes shown in Figure 3.13. For example, for replication (clamping),

$$\tilde{f}(i,j) = f(k,l), \quad \begin{matrix} k = \max(0, \min(M-1, i)), \\ l = \max(0, \min(N-1, j)), \end{matrix}$$

(Hint: you may want to use the min, max, mod, and absolute value operators in addition to the regular arithmetic operators.)

- Describe in more detail the advantages and disadvantages of these various modes.
- (Optional) Check what your graphics card does by drawing a texture-mapped rectangle where the texture coordinates lie beyond the $[0.0, 1.0]$ range and using different texture clamping modes.

**Ex 3.9: Separable filters** Implement convolution with a separable kernel. The input should be a grayscale or color image along with the horizontal and vertical kernels. Make sure you support the padding mechanisms developed in the previous exercise. You will need this functionality for some of the later exercises. If you already have access to separable filtering in an image processing package you are using (such as IPL), skip this exercise.

- (Optional) Use Pietro Perona's (1995) technique to approximate convolution as a sum of a number of separable kernels. Let the user specify the number of kernels and report back some sensible metric of the approximation fidelity.

**Ex 3.10: Discrete Gaussian filters** Discuss the following issues with implementing a discrete Gaussian filter:

- If you just sample the equation of a continuous Gaussian filter at discrete locations, will you get the desired properties, e.g., will the coefficients sum up to 0? Similarly, if you sample a derivative of a Gaussian, do the samples sum up to 0 or have vanishing higher-order moments?
- Would it be preferable to take the original signal, interpolate it with a sinc, blur with a continuous Gaussian, then pre-filter with a sinc before re-sampling? Is there a simpler way to do this in the frequency domain?
- Would it make more sense to produce a Gaussian frequency response in the Fourier domain and to then take an inverse FFT to obtain a discrete filter?
- How does truncation of the filter change its frequency response? Does it introduce any additional artifacts?
- Are the resulting two-dimensional filters as rotationally invariant as their continuous analogs? Is there some way to improve this? In fact, can any 2D discrete (separable or non-separable) filter be truly rotationally invariant?

**Ex 3.11: Sharpening, blur, and noise removal** Implement some softening, sharpening, and non-linear diffusion (selective sharpening or noise removal) filters, such as Gaussian, median, and bilateral (Section 3.3.1), as discussed in Section 3.4.4.

Take blurry or noisy images (shooting in low light is a good way to get both) and try to improve their appearance and legibility.

**Ex 3.12: Steerable filters** Implement Freeman and Adelson's (1991) steerable filter algorithm. The input should be a grayscale or color image and the output should be a multi-banded image consisting of $G_1^{0^\circ}$ and $G_1^{90^\circ}$. The coefficients for the filters can be found in the paper by Freeman and Adelson (1991).

Test the various order filters on a number of images of your choice and see if you can reliably find corner and intersection features. These filters will be quite useful later to detect elongated structures, such as lines (Section 4.3).

**Ex 3.13: Distance transform** Implement some (raster-scan) algorithms for city block and Euclidean distance transforms. Can you do it without peeking at the literature (Danielsson 1980; Borgefors 1986)? If so, what problems did you come across and resolve?

Later on, you can use the distance functions you compute to perform *feathering* during image stitching (Section 9.3.2).

**Ex 3.14: Connected components** Implement one of the connected component algorithms from Section 3.3.4 or Section 2.3 from Haralick and Shapiro's book (1992) and discuss its computational complexity.

- Threshold or quantize an image to obtain a variety of input labels and then compute the area statistics for the regions that you find.
- Use the connected components that you have found to track or match regions in different images or video frames.

**Ex 3.15: Fourier transform** Prove the properties of the Fourier transform listed in Table 3.1 and derive the formulas for the Fourier transforms listed in Tables 3.2 and 3.3. These exercises are very useful if you want to become comfortable working with Fourier transforms, which is a very useful skill when analyzing and designing the behavior and efficiency of many computer vision algorithms.

**Ex 3.16: Wiener filtering** Estimate the frequency spectrum of your personal photo collection and use it to perform Wiener filtering on a few images with varying degrees of noise.

1. Collect a few hundred of your images by re-scaling them to fit within a $512 \times 512$ window and cropping them.
2. Take their Fourier transforms, throw away the phase information, and average together all of the spectra.
3. Pick two of your favorite images and add varying amounts of Gaussian noise, $\sigma_n \in \{1, 2, 5, 10, 20\}$ gray levels.
4. For each combination of image and noise, determine by eye which width of a Gaussian blurring filter $\sigma_s$ gives the best denoised result. You will have to make a subjective decision between sharpness and noise.

5. Compute the Wiener filtered version of all the noised images and compare them against your hand-tuned Gaussian-smoothed images.

6. (Optional) Do your image spectra have a lot of energy concentrated along the horizontal and vertical axes ($f_x = 0$ and $f_y = 0$)? Can you think of an explanation for this? Does rotating your image samples by $45°$ move this energy to the diagonals? If not, could it be due to edge effects in the Fourier transform? Can you suggest some techniques for reducing such effects?

**Ex 3.17: Deblurring using Wiener filtering** Use Wiener filtering to deblur some images.

1. Modify the Wiener filter derivation (3.66–3.74) to incorporate blur (3.75).

2. Discuss the resulting Wiener filter in terms of its noise suppression and frequency boosting characteristics.

3. Assuming that the blur kernel is Gaussian and the image spectrum follows an inverse frequency law, compute the frequency response of the Wiener filter, and compare it to the unsharp mask.

4. Synthetically blur two of your sample images with Gaussian blur kernels of different radii, add noise, and then perform Wiener filtering.

5. Repeat the above experiment with a "pillbox" (disc) blurring kernel, which is characteristic of a finite aperture lens (Section 2.2.3). Compare these results to Gaussian blur kernels (be sure to inspect your frequency plots).

6. It has been suggested that regular apertures are anathema to de-blurring because they introduce zeros in the sensed frequency spectrum (Veeraraghavan, Raskar, Agrawal *et al.* 2007). Show that this is indeed an issue if no prior model is assumed for the signal, i.e., $P_s^{-1}\boldsymbol{l}1$. If a reasonable power spectrum is assumed, is this still a problem (do we still get banding or ringing artifacts)?

**Ex 3.18: High-quality image resampling** Implement several of the low-pass filters presented in Section 3.5.2 and also the discussion of the windowed sinc shown in Table 3.2 and Figure 3.29. Feel free to implement other filters (Wolberg 1990; Unser 1999).

Apply your filters to continuously resize an image, both magnifying (interpolating) and minifying (decimating) it; compare the resulting animations for several filters. Use both a synthetic chirp image (Figure 3.65a) and natural images with lots of high-frequency detail (Figure 3.65b-c).[27]

You may find it helpful to write a simple visualization program that continuously plays the animations for two or more filters at once and that let you "blink" between different results.

Discuss the merits and deficiencies of each filter, as well as its tradeoff between speed and quality.

**Ex 3.19: Pyramids** Construct an image pyramid. The inputs should be a grayscale or color image, a separable filter kernel, and the number of desired levels. Implement at least the following kernels:

[27] These particular images are available on the book's Web site.

(a) (b) (c)

**Figure 3.65** Sample images for testing the quality of resampling algorithms: (a) a synthetic chirp; (b) and (c) some high-frequency images from the image compression community.

- $2 \times 2$ block filtering;
- Burt and Adelson's binomial kernel $^1/_{16}(1, 4, 6, 4, 1)$ (Burt and Adelson 1983a);
- a high-quality seven- or nine-tap filter.

Compare the visual quality of the various decimation filters. Also, shift your input image by 1 to 4 pixels and compare the resulting decimated (quarter size) image sequence.

**Ex 3.20: Pyramid blending** Write a program that takes as input two color images and a binary mask image and produces the Laplacian pyramid blend of the two images.

1. Construct the Laplacian pyramid for each image.
2. Construct the Gaussian pyramid for the two mask images (the input image and its complement).
3. Multiply each Laplacian image by its corresponding mask and sum the images (see Figure 3.43).
4. Reconstruct the final image from the blended Laplacian pyramid.

Generalize your algorithm to input $n$ images and a label image with values $1 \ldots n$ (the value 0 can be reserved for "no input"). Discuss whether the weighted summation stage (step 3) needs to keep track of the total weight for renormalization, or whether the math just works out. Use your algorithm either to blend two differently exposed image (to avoid under- and over-exposed regions) or to make a creative blend of two different scenes.

**Ex 3.21: Wavelet construction and applications** Implement one of the wavelet families described in Section 3.5.4 or by Simoncelli and Adelson (1990b), as well as the basic Laplacian pyramid (Exercise 3.19). Apply the resulting representations to one of the following two tasks:

- **Compression:** Compute the entropy in each band for the different wavelet implementations, assuming a given quantization level (say, ¼ gray level, to keep the rounding error acceptable). Quantize the wavelet coefficients and reconstruct the original images. Which technique performs better? (See (Simoncelli and Adelson 1990b) or any of the multitude of wavelet compression papers for some typical results.)
- **Denoising.** After computing the wavelets, suppress small values using *coring*, i.e., set small values to zero using a piecewise linear or other $C^0$ function. Compare the results of your denoising using different wavelet and pyramid representations.

**Ex 3.22: Parametric image warping** Write the code to do affine and perspective image warps (optionally bilinear as well). Try a variety of interpolants and report on their visual quality. In particular, discuss the following:

- In a MIP-map, selecting only the coarser level adjacent to the computed fractional level will produce a blurrier image, while selecting the finer level will lead to aliasing. Explain why this is so and discuss whether blending an aliased and a blurred image (tri-linear MIP-mapping) is a good idea.
- When the ratio of the horizontal and vertical resampling rates becomes very different (anisotropic), the MIP-map performs even worse. Suggest some approaches to reduce such problems.

**Ex 3.23: Local image warping** Open an image and deform its appearance in one of the following ways:

1. Click on a number of pixels and move (drag) them to new locations. Interpolate the resulting sparse displacement field to obtain a dense motion field (Sections 3.6.2 and 3.5.1).
2. Draw a number of lines in the image. Move the endpoints of the lines to specify their new positions and use the Beier–Neely interpolation algorithm (Beier and Neely 1992), discussed in Section 3.6.2, to get a dense motion field.
3. Overlay a spline control grid and move one grid point at a time (optionally select the level of the deformation).
4. Have a dense per-pixel flow field and use a soft "paintbrush" to design a horizontal and vertical velocity field.
5. (Optional): Prove whether the Beier–Neely warp does or does not reduce to a sparse point-based deformation as the line segments become shorter (reduce to points).

**Ex 3.24: Forward warping** Given a displacement field from the previous exercise, write a forward warping algorithm:

1. Write a forward warper using splatting, either nearest neighbor or soft accumulation (Section 3.6.1).
2. Write a two-pass algorithm, which forward warps the displacement field, fills in small holes, and then uses inverse warping (Shade, Gortler, He *et al.* 1998).

3. Compare the quality of these two algorithms.

**Ex 3.25: Feature-based morphing** Extend the warping code you wrote in Exercise 3.23 to import two different images and specify correspondences (point, line, or mesh-based) between the two images.

1. Create a morph by partially warping the images towards each other and cross-dissolving (Section 3.6.3).

2. Try using your morphing algorithm to perform an image rotation and discuss whether it behaves the way you want it to.

**Ex 3.26: 2D image editor** Extend the program you wrote in Exercise 2.2 to import images and let you create a "collage" of pictures. You should implement the following steps:

1. Open up a new image (in a separate window).

2. Shift drag (rubber-band) to crop a subregion (or select whole image).

3. Paste into the current canvas.

4. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.

5. Drag any corner of the outline to change its transformation.

6. (Optional) Change the relative ordering of the images and which image is currently being manipulated.

The user should see the composition of the various images' pieces on top of each other.

This exercise should be built on the image transformation classes supported in the software library. Persistence of the created representation (save and load) should also be supported (for each image, save its transformation).

**Ex 3.27: 3D texture-mapped viewer** Extend the viewer you created in Exercise 2.3 to include texture-mapped polygon rendering. Augment each polygon with $(u, v, w)$ coordinates into an image.

**Ex 3.28: Image denoising** Implement at least two of the various image denoising techniques described in this chapter and compare them on both synthetically noised image sequences and real-world (low-light) sequences. Does the performance of the algorithm depend on the correct choice of noise level estimate? Can you draw any conclusions as to which techniques work better?

**Ex 3.29: Rainbow enhancer—challenging** Take a picture containing a rainbow, such as Figure 3.66, and enhance the strength (saturation) of the rainbow.

1. Draw an arc in the image delineating the extent of the rainbow.

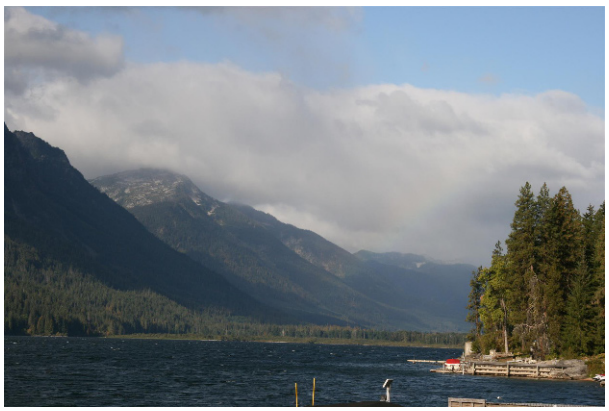


**Figure 3.66** There is a faint image of a rainbow visible in the right hand side of this picture. Can you think of a way to enhance it (Exercise 3.29)?

2. Fit an *additive* rainbow function (explain why it is additive) to this arc (it is best to work with linearized pixel values), using the spectrum as the cross section, and estimating the width of the arc and the amount of color being added. This is the trickiest part of the problem, as you need to tease apart the (low-frequency) rainbow pattern and the natural image hiding behind it.

3. Amplify the rainbow signal and add it back into the image, re-applying the gamma function if necessary to produce the final image.

**Ex 3.30: Image deblocking—challenging** Now that you have some good techniques to distinguish signal from noise, develop a technique to remove the *blocking artifacts* that occur with JPEG at high compression settings (Section 2.3.3). Your technique can be as simple as looking for unexpected edges along block boundaries, to looking at the quantization step as a projection of a convex region of the transform coefficient space onto the corresponding quantized values.

1. Does the knowledge of the compression factor, which is available in the JPEG header information, help you perform better deblocking?

2. Because the quantization occurs in the DCT transformed YCbCr space (2.115), it may be preferable to perform the analysis in this space. On the other hand, image priors make more sense in an RGB space (or do they?). Decide how you will approach this dichotomy and discuss your choice.

3. While you are at it, since the YCbCr conversion is followed by a chrominance subsampling stage (before the DCT), see if you can restore some of the lost high-frequency chrominance signal using one of the better restoration techniques discussed in this chapter.

4. If your camera has a RAW + JPEG mode, how close can you come to the noise-free true pixel values? (This suggestion may not be that useful, since cameras generally use reasonably high quality settings for their RAW + JPEG models.)

**Ex 3.31: Inference in de-blurring—challenging** Write down the graphical model corresponding to Figure 3.59 for a non-blind image deblurring problem, i.e., one where the blur kernel is known ahead of time.

What kind of efficient inference (optimization) algorithms can you think of for solving such problems?