



Chapter 3

Image processing

3.1	Point operators	87
3.1.1	Pixel transforms	89
3.1.2	Color transforms	90
3.1.3	Compositing and matting	91
3.1.4	Histogram equalization	92
3.1.5	<i>Application: Tonal adjustment</i>	95
3.2	Linear filtering	95
3.2.1	Separable filtering	99
3.2.2	Examples of linear filtering	100
3.2.3	Band-pass and steerable filters	101
3.3	More neighborhood operators	105
3.3.1	Non-linear filtering	105
3.3.2	Bilateral filtering	107
3.3.3	Binary image processing	110
3.4	Fourier transforms	113
3.4.1	Two-dimensional Fourier transforms	115
3.4.2	<i>Application: Sharpening, blur, and noise removal</i>	118
3.5	Pyramids and wavelets	119
3.5.1	Interpolation	119
3.5.2	Decimation	122
3.5.3	Multi-resolution representations	123
3.5.4	Wavelets	128
3.5.5	<i>Application: Image blending</i>	132
3.6	Geometric transformations	135
3.6.1	Parametric transformations	135
3.6.2	Mesh-based warping	140
3.6.3	<i>Application: Feature-based morphing</i>	142
3.7	Additional reading	143
3.8	Exercises	144

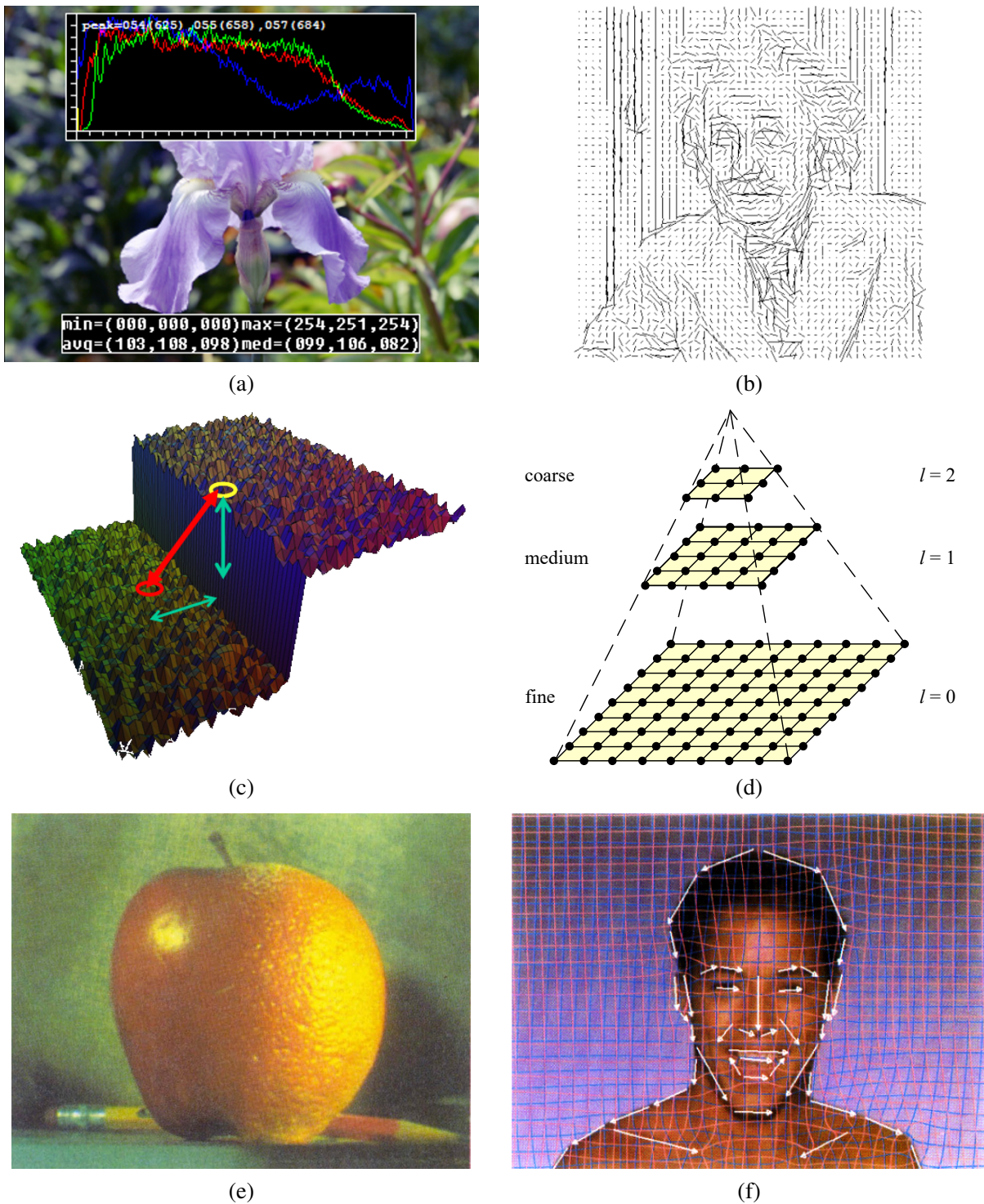


Figure 3.1 Some common image processing operations: (a) partial histogram equalization; (b) orientation map computed from the second-order steerable filter (Freeman 1992) © 1992 IEEE; (c) bilateral filter (Durand and Dorsey 2002) © 2002 ACM; (d) image pyramid; (e) Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM; (f) line-based image warping (Beier and Neely 1992) © 1992 ACM.

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 algorithms, 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, reducing image noise, increasing sharpness, or straightening the image by rotating it. Additional examples include image warping and image blending, which are often used for visual effects (Figures 3.1 and Section 3.6.3). 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 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 Schaffer 1996; Oppenheim, Schaffer, and Buck 1999). There are several popular textbooks for image processing, including Gomes and Velho (1997), Jähne (1997), Pratt (2007), Burger and Burge (2009), and Gonzalez and Woods (2017).

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).

While this chapter covers *classical* image processing techniques that consist mostly of linear and non-linear filtering operations, the next two chapters introduce energy-based and Bayesian graphical models, i.e., *Markov random fields* (Chapter 4), and then deep convolutional networks (Chapter 5), both of which are now widely used in image processing applications.

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).¹

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.

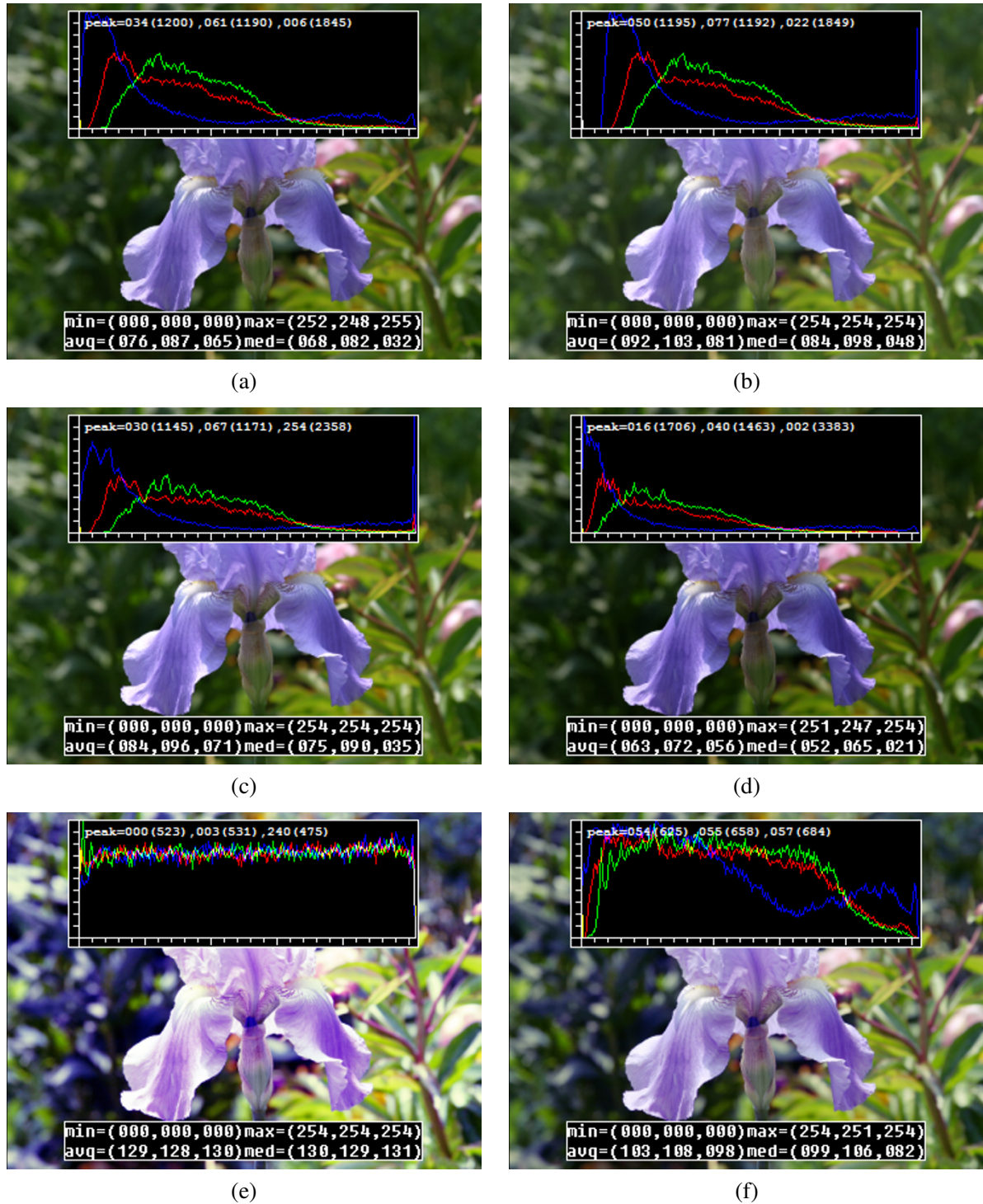


Figure 3.2 Some local image processing operations: (a) original image along with its three color (per-channel) histograms; (b) brightness increased (additive offset, $b = 16$); (c) contrast increased (multiplicative gain, $a = 1.1$); (d) gamma (partially) linearized ($\gamma = 1.2$); (e) full histogram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

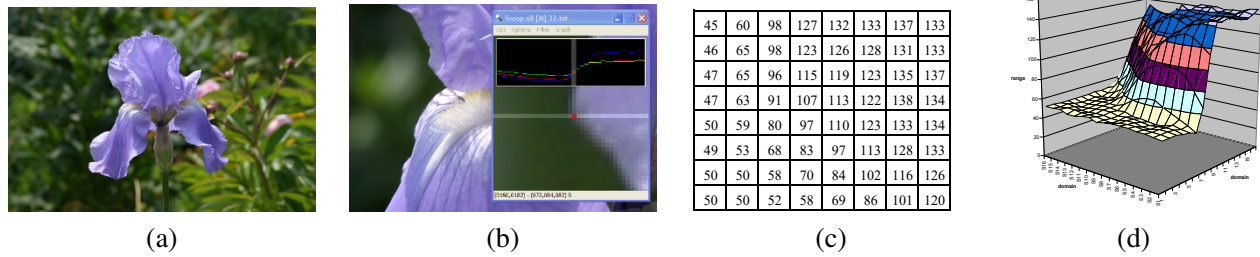


Figure 3.3 Visualizing image data: (a) original image; (b) cropped portion and scanline plot using an image inspection tool; (c) grid of numbers; (d) surface plot. For figures (c)–(d), the image was first converted to grayscale.

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

$$g(\mathbf{x}) = h(f(\mathbf{x})) \quad \text{or} \quad g(\mathbf{x}) = h(f_0(\mathbf{x}), \dots, f_n(\mathbf{x})), \quad (3.1)$$

where \mathbf{x} is in the D -dimensional (usually $D = 2$ for images) *domain* of the input and output functions f and g , which 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*, $\mathbf{x} = (i, j)$, and we can write

$$g(i, j) = h(f(i, j)). \quad (3.2)$$

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,

$$g(\mathbf{x}) = af(\mathbf{x}) + b. \quad (3.3)$$

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).² The bias and gain parameters can also be spatially varying,

$$g(\mathbf{x}) = a(\mathbf{x})f(\mathbf{x}) + b(\mathbf{x}), \quad (3.4)$$

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, as it obeys the *superposition principle*,

$$h(f_0 + f_1) = h(f_0) + h(f_1). \quad (3.5)$$

(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.

¹In convolutional neural networks (Section 5.4), such operations are sometimes called 1×1 convolutions.

²An image's luminance characteristics can also be summarized by its *key* (average luminance) and *range* (Kopf, Uyttendaele *et al.* 2007).



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

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

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

By varying α 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

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

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 (2.105) or even simpler color ratios (2.117) 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. Figures 2.33f–h show 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 3×3 *color twist* transform matrix. Exercises 2.8 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).

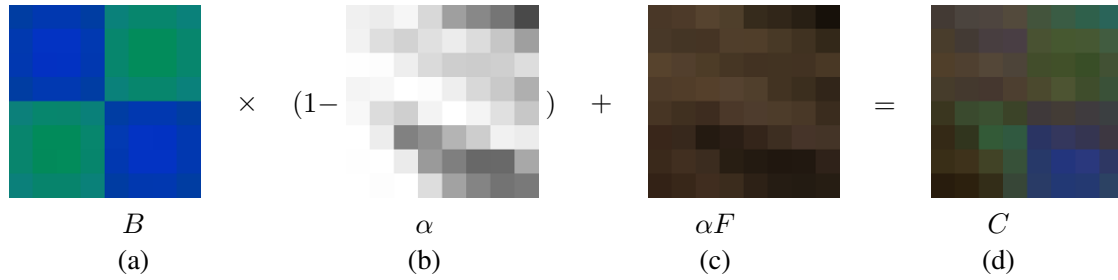


Figure 3.5 Compositing equation $C = (1 - \alpha)B + \alpha F$. The images are taken from a close-up of the region of the hair in the upper right part of the lion in Figure 3.4.

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 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 α (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:

$$C = (1 - \alpha)B + \alpha F. \quad (3.8)$$

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 αF 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 *et al.* 2001), the pure un-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 3.3).

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 9.4.2).



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.

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 *et al.* (1999) review *difference matting*. Since then, there has been a lot of activity in computational photography relating to *natural image matting* (Ruzon and Tomasi 2000; Chuang, Curless *et al.* 2001; Wang and Cohen 2009; Xu, Price *et al.* 2017), which attempts to extract the mattes from a single natural image (Figure 3.4a) or from extended video sequences (Chuang, Agarwala *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 *et al.* 2007).

How can we visualize the set of lightness values in an image to test some of these heuristics? The answer is to plot the *histogram* of the individual color channels and luminance values, as shown in Figure 3.7b.³ 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.

³The 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.

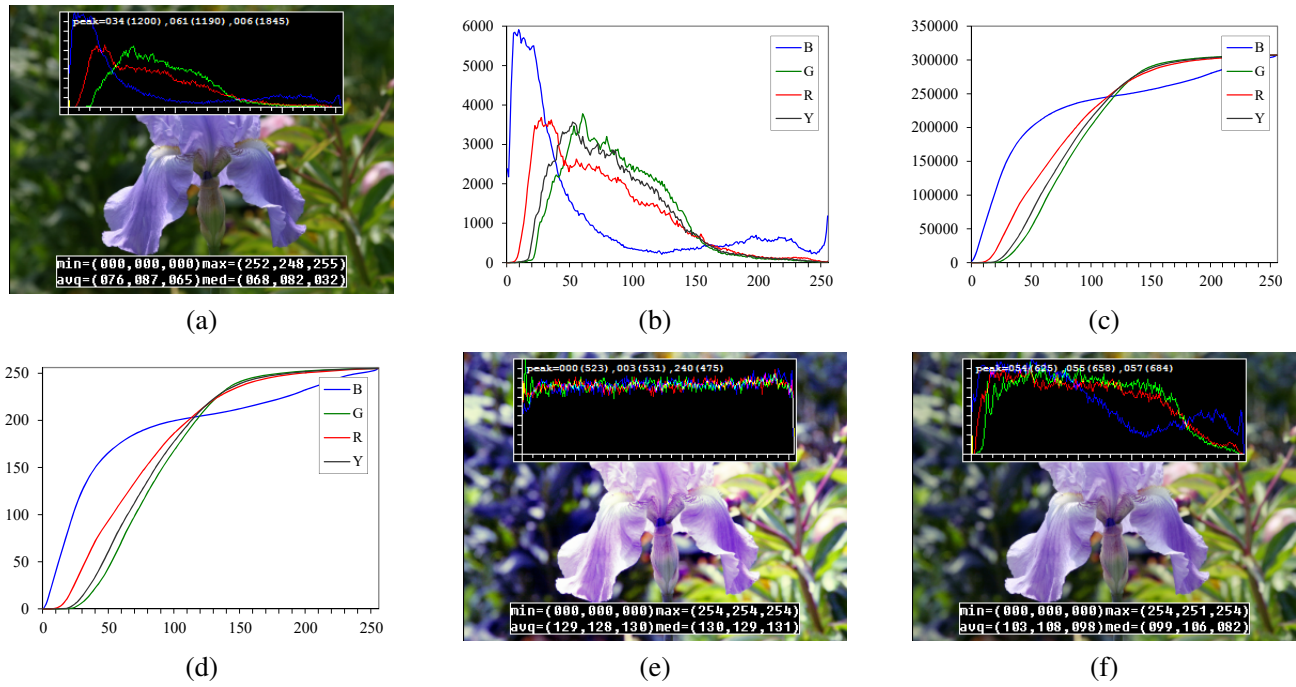


Figure 3.7 Histogram analysis and equalization: (a) original image; (b) color channel and intensity (luminance) histograms; (c) cumulative distribution functions; (d) equalization (transfer) functions; (e) full histogram equalization; (f) partial histogram equalization.

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 $c(I)$,

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

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 the pixel should take. When working with eight-bit pixel values, the I and c axes are rescaled from $[0, 255]$.

Figure 3.7e 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 3.7f, 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 3.7 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).

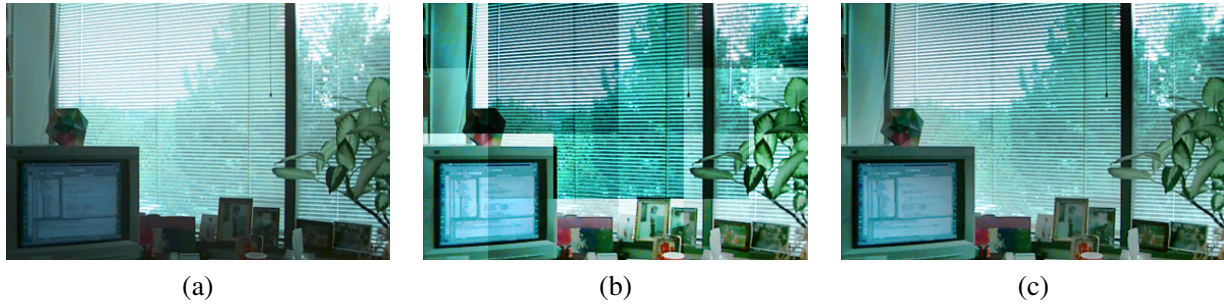


Figure 3.8 Locally adaptive histogram equalization: (a) original image; (b) block histogram equalization; (c) full locally adaptive equalization.

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-limited (gain-limited) version is known as CLAHE (Pizer, Amburn *et al.* 1987).⁴ 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 $\{f_{00}, \dots, f_{11}\}$, a *bilinear* blending function,

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

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 3.8). 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.,

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

where $w(i, j, k, l)$ is the bilinear weighting function between pixel (i, j) and lookup table (k, l) .

⁴The CLAHE algorithm is part of OpenCV.

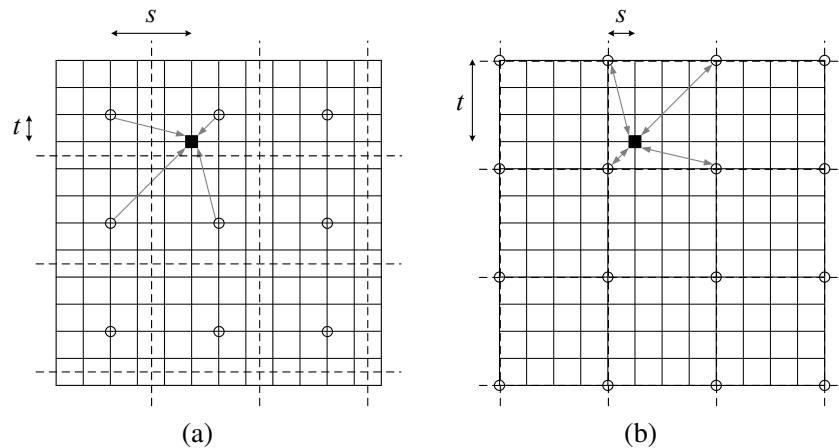


Figure 3.9 Local histogram interpolation using relative (s, t) coordinates: (a) block-based histograms, with block centers shown as circles; (b) corner-based “spline” histograms. Pixels are located on grid intersections. The black square pixel’s transfer function is interpolated from the four adjacent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

This is an example of *soft histogramming*, which is used in a variety of other applications, including the construction of SIFT feature descriptors (Section 7.1.3) and vocabulary trees (Section 7.1.4).

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 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, 3.6, and 3.7 have you implement some of these operations, to become familiar with basic image processing operators. More sophisticated techniques for tonal adjustment (Bae, Paris, and Durand 2006; Reinhard, Heidrich *et al.* 2010) 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 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 fixed 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 widely used type of neighborhood operator is a *linear filter*, where an output pixel’s value is a weighted sum of pixel values within a small neighborhood \mathcal{N} (Figure 3.10),

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

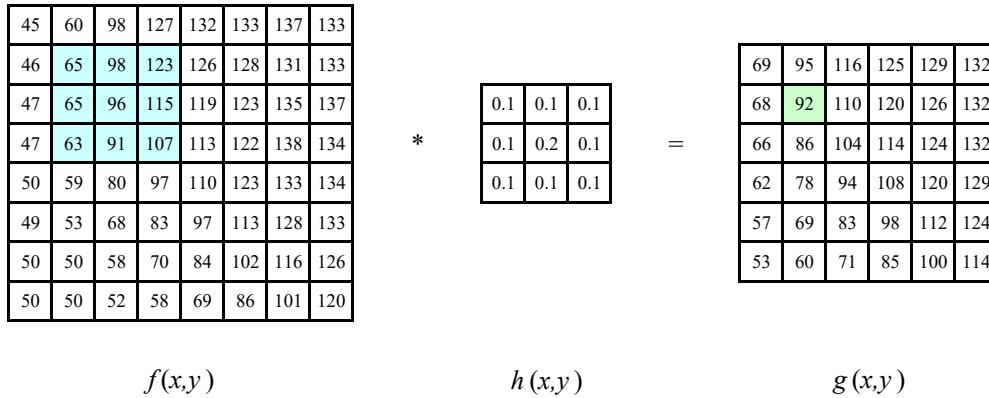


Figure 3.10 Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.

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

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

A common variant on this formula is

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

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

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

and h is then called the *impulse response function*.⁵ The reason for this name is that the kernel function, h , convolved with an impulse signal, $\delta(i, j)$ (an image that is 0 everywhere except at the origin) reproduces itself, $h * \delta = h$, whereas correlation produces the reflected signal. (Try this yourself to verify that it is so.)

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

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

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

and the *shift invariance* principle,

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

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

⁵The continuous version of convolution can be written as $g(\mathbf{x}) = \int f(\mathbf{x} - \mathbf{u})h(\mathbf{u})d\mathbf{u}$.



Figure 3.11 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 (3.44–3.48).

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

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

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

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

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

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

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

where the (sparse) \mathbf{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 correlation shown in Figure 3.10 produces a result that is smaller than the original image, which may not be desirable in many applications.⁶ This is because the neighborhoods of typical correlation and convolution operations extend beyond the image boundaries near the edges, and so the filtered images suffer from *boundary effects*

To deal with this, a number of different *padding* or extension modes have been developed for neighborhood operations (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.

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 these modes are left to the reader (Exercise 3.9).

⁶Note, however, that early convolutional networks such as LeNet (LeCun, Bottou *et al.* 1998) adopted this structure.

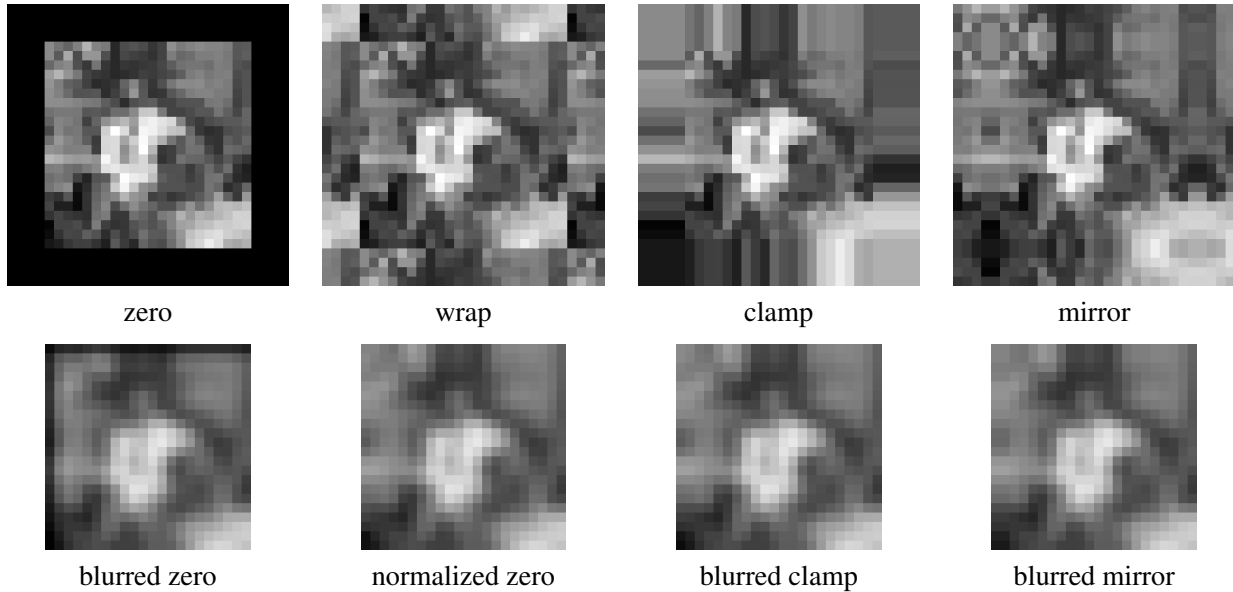


Figure 3.13 Border padding (top row) and the results of blurring the padded image (bottom row). The normalized zero image is the result of dividing (normalizing) the blurred zero-padded RGBA image by its corresponding soft alpha value.

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

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

(see Figure 3.14 for some examples). Because of the increased efficiency, the design of convolution kernels for computer vision applications is often influenced by their separability.

How can we tell if a given kernel \mathbf{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 \mathbf{K} and to take its singular value decomposition (SVD),

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

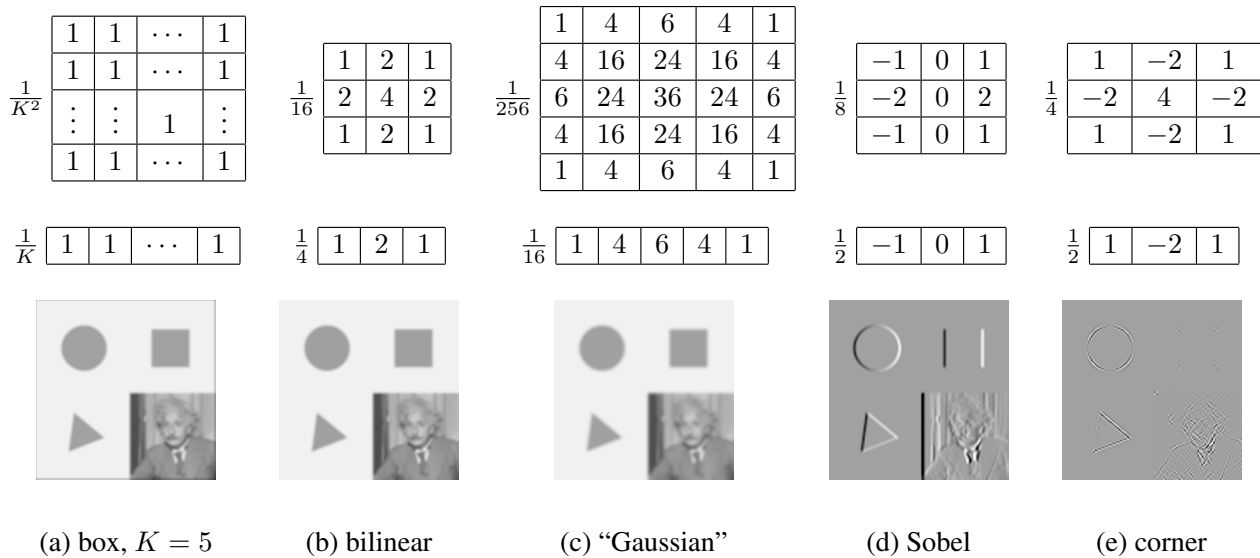


Figure 3.14 Separable linear filters: For each image (a)–(e), we show the 2D filter kernel (top), the corresponding horizontal 1D kernel (middle), and the filtered image (bottom). The filtered Sobel and corner images are signed, scaled up by $2\times$ and $4\times$, respectively, and added to a gray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

(see Appendix A.1.1 for the definition of the SVD). If only the first singular value σ_0 is non-zero, the kernel is separable and $\sqrt{\sigma_0}\mathbf{u}_0$ and $\sqrt{\sigma_0}\mathbf{v}_0^T$ provide the vertical and horizontal kernels (Perona 1995). For example, the Laplacian of Gaussian kernel (3.26 and 7.23) can be implemented as the sum of two separable filters (7.24) (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 3×3 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 3.11).

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 of smoothing to remove noise later (see Sections 3.3.1, 3.4, and as well as Chapters 4 and 5).

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

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

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

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

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

Linear filtering can also be used as a pre-processing stage to edge extraction (Section 7.2) and interest point detection (Section 7.1) algorithms. Figure 3.14d shows a simple 3×3 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 vertical 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 7.1.

3.2.3 Band-pass and steerable filters

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

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

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

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

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

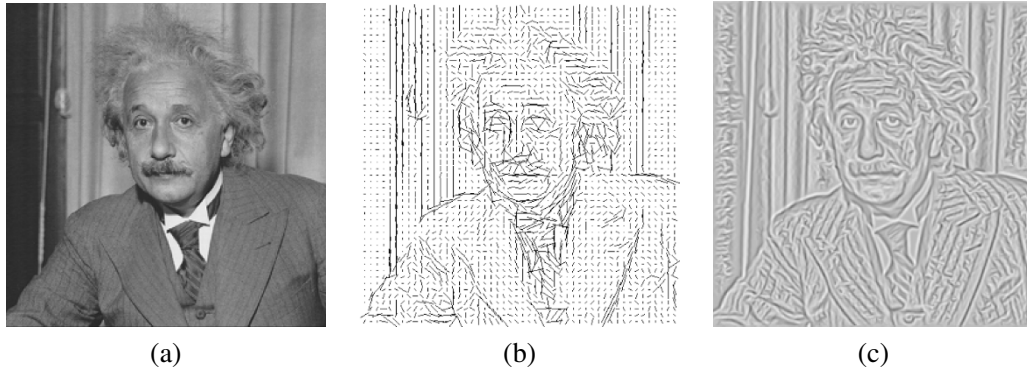


Figure 3.15 Second-order steerable filter (Freeman 1992) © 1992 IEEE: (a) original image of Einstein; (b) orientation map computed from the second-order oriented energy; (c) original image with oriented structures enhanced.

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

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

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

Likewise, the Sobel operator is a simple approximation to a *directional* or *oriented* filter, which can be obtained by smoothing with a Gaussian (or some other filter) and then taking a *directional derivative* $\nabla_{\hat{\mathbf{u}}} = \frac{\partial}{\partial \hat{\mathbf{u}}}$, which is obtained by taking the dot product between the gradient field ∇ and a unit direction $\hat{\mathbf{u}} = (\cos \theta, \sin \theta)$,

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

The smoothed directional derivative filter,

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

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

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

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

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

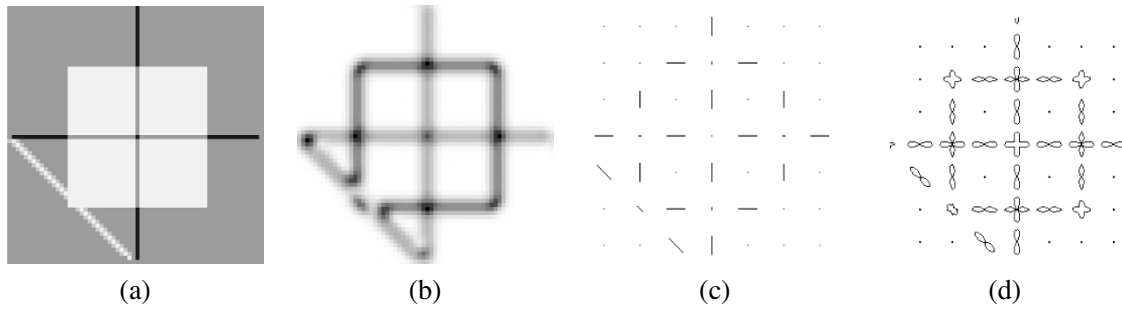


Figure 3.16 Fourth-order steerable filter (Freeman and Adelson 1991) © 1991 IEEE: (a) test image containing bars (lines) and step edges at different orientations; (b) average oriented energy; (c) dominant orientation; (d) oriented energy as a function of angle (polar plot).

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 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 7.1.3) and edge detectors (Section 7.2). 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 2×2 boundary tensor can be used to encode both edge and junction (“corner”) features. Exercise 3.13 has you implement such steerable filters and apply them to finding both edge and corner features.

Summed area table (integral image)

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

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

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

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

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

$$S(i_0 \dots i_1, j_0 \dots j_1) = s(i_1, j_1) - s(i_1, j_0 - 1) - s(i_0 - 1, j_1) + s(i_0 - 1, j_0 - 1). \quad (3.32)$$

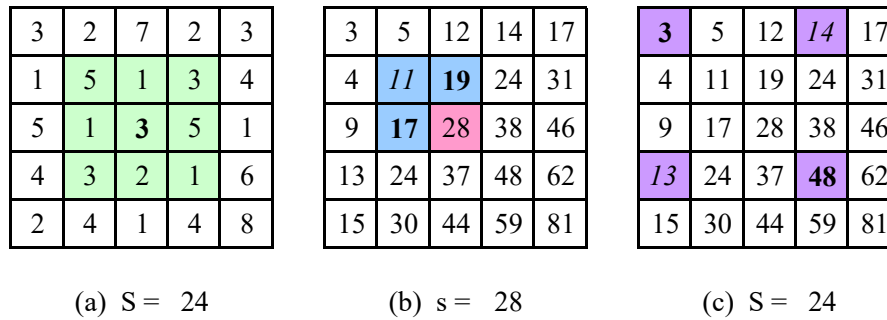


Figure 3.17 Summed area tables: (a) original image; (b) summed area table; (c) computation of area sum. Each value in the summed area table $s(i, j)$ (red) is computed recursively from its three adjacent (blue) neighbors (3.31). Area sums S (green) are computed by combining the four values at the rectangle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

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 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 *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 12.4). In practice, separable moving average filters are usually preferred (Kanade, Yoshida *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 3.3.3).

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

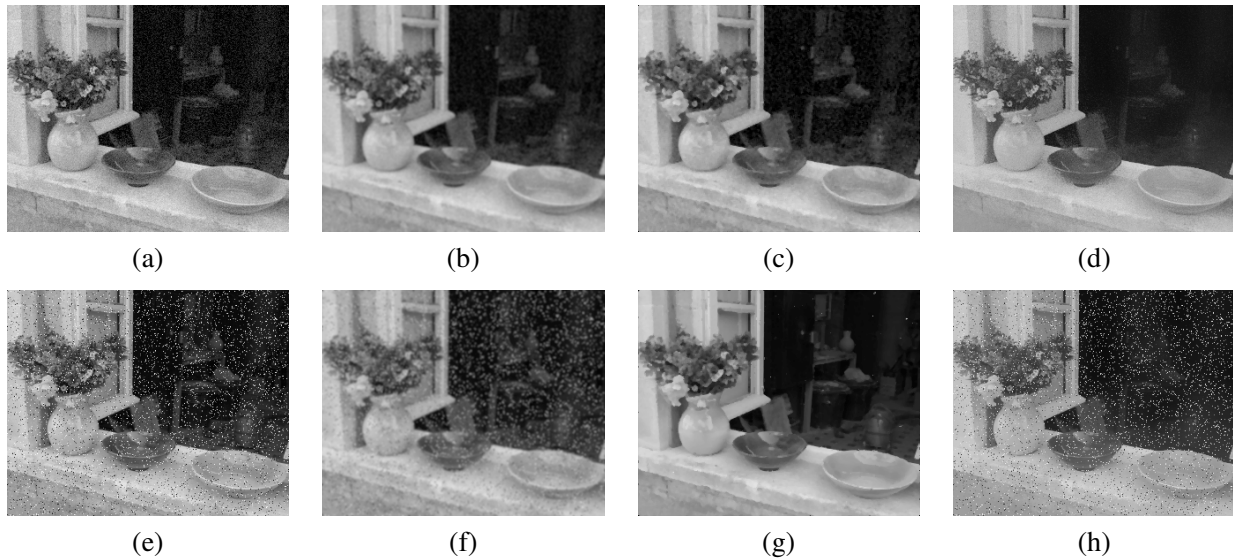


Figure 3.18 Median and bilateral filtering: (a) original image with Gaussian noise; (b) Gaussian filtered; (c) median filtered; (d) bilaterally filtered; (e) original image with shot noise; (f) Gaussian filtered; (g) median filtered; (h) bilaterally filtered. Note that the bilateral filter fails to remove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

3.3 More neighborhood operators

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

3.3.1 Non-linear filtering

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

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

Median filtering

A better filter to use in this case is the *median* filter, which selects the median value from each pixel's neighborhood (Figure 3.19a). Median values can be computed in expected linear time using a randomized select algorithm (Cormen 2001) and incremental variants have also been developed (Tomasi and Manduchi 1998; Bovik 2000, Section 3.2), as well as a constant time algorithm that

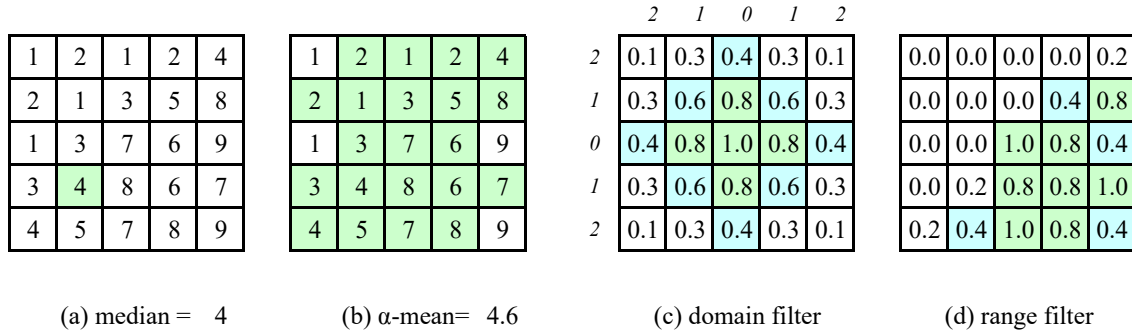


Figure 3.19 Median and bilateral filtering: (a) median pixel (green); (b) selected α -trimmed mean pixels; (c) domain filter (numbers along edge are pixel distances); (d) range filter.

is independent of window size (Perreault and Hébert 2007). 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 3.18g).

One downside of the median filter, in addition to its moderate computational cost, is that because 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 *et al.* 1986; Stewart 1999). A better choice may be the α -trimmed mean (Lee and Redner 1990; Crane 1997, p. 109), which averages together all of the pixels except for the α 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 (Haralick and Shapiro 1992, Section 7.2.6; Bovik 2000, Section 3.2). 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 *et al.* 1986).

Non-linear smoothing has another, perhaps even more important property, especially as 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 well 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 α -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.

3.3.2 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 α , 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), although it had been proposed earlier by Aurich and Weule (1995) and Smith and Brady (1997). Paris, Kornprobst *et al.* (2008) provide a nice review of work in this area as well as myriad applications in computer vision, graphics, and computational photography.

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

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

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

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

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

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

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

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

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

Notice that for color images, 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.⁷

Since bilateral filtering is quite slow compared to regular separable filtering, a number of acceleration techniques have been developed, as discussed in Durand and Dorsey (2002), Paris and Durand (2009), Chen, Paris, and Durand (2007), and Paris, Kornprobst *et al.* (2008). In particular, the *bilateral grid* (Chen, Paris, and Durand 2007), which subsamples the higher-dimensional color/position space on a uniform grid, continues to be widely used, including the application of the *bilateral solver* (Section 4.2.3 and Barron and Poole (2016)). An even faster implementation of bilateral filtering can be obtained using the *permutohedral lattice* approach developed by Adams, Baek, and Davis (2010).

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.

⁷Tomasi and Manduchi (1998) show that using the vector distance (as opposed to filtering each color band separately) reduces color fringing effects. They also recommend taking the color difference in the more perceptually uniform CIELAB color space (see Section 2.3.2).

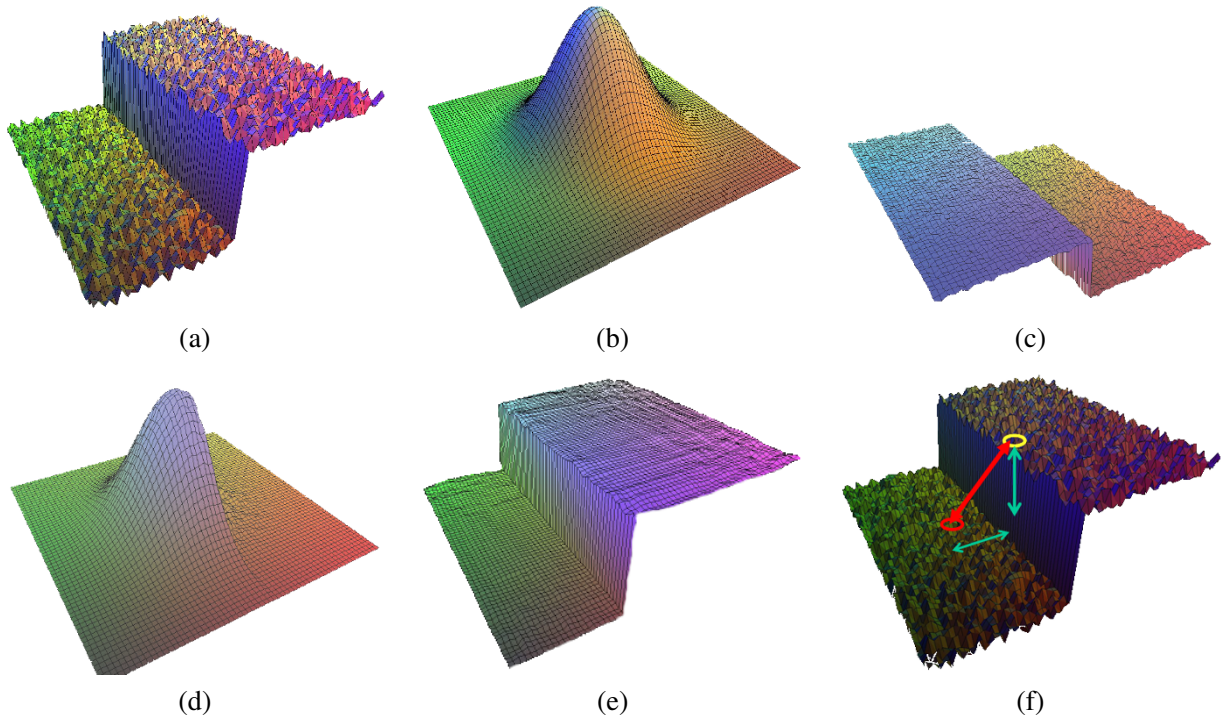


Figure 3.20 Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.

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

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

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

We can thus re-write (3.34) as

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

where $R = \sum_{(k, l)} r(i, j, k, l)$, (k, l) are the \mathcal{N}_4 (nearest four) 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).⁸ Since its original introduction, anisotropic diffusion has been extended and applied to a wide range of problems (Nielsen, Florack, and Deriche 1997; Black, Sapiro *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 1998; Weickert 1998). It has also

⁸The $1/(1 + \eta R)$ factor is not present in anisotropic diffusion but becomes negligible as $\eta \rightarrow 0$.

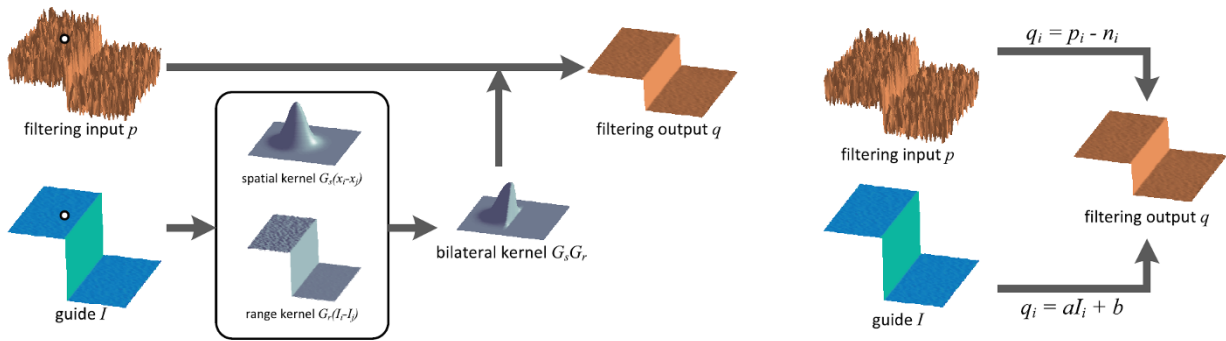


Figure 3.21 Guided image filtering (He, Sun, and Tang 2013) © 2013 IEEE. Unlike joint bilateral filtering, shown on the left, which computes a per pixel weight mask from the guide image (shown as I in the figure, but \mathbf{h} in the text), the guided image filter models the output value (shown as q_i in the figure, but denoted as $\mathbf{g}(i, j)$ in the text) as a local affine transformation of the guide pixels.

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

In its general form, the range kernel $r(i, j, k, l) = r(\|f(i, j) - f(k, l)\|)$, which is usually called the *gain* or *edge-stopping* function, or diffusion coefficient, can be any monotonically increasing function with $r'(x) \rightarrow 0$ as $x \rightarrow \infty$. Black, Sapiro *et al.* (1998) show how anisotropic diffusion is equivalent to minimizing a robust penalty function on the image gradients, which we discuss in Sections 4.2 and 4.3. 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 4.2 and 4.3 and by Scharr, Black, and Haussecker (2003), which introduce such a bias in a principled manner.

Guided image filtering

While so far we have discussed techniques for filtering an image to obtain an improved version, e.g., one with less noise or sharper edges, it is also possible to use a different *guide* image to adaptively filter a noisy input (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004; He, Sun, and Tang 2013). An example of this is using a flash image, which has strong edges but poor color, to adaptively filter a low-light non-flash color image, which has large amounts of noise, as described in Section 10.2.2. In their papers, where they apply the range filter (3.36) to a different guide image $\mathbf{h}()$, Eisemann and Durand (2004) call their approach a *cross-bilateral filter*, while Petschnigg, Agrawala *et al.* (2004) call it *joint bilateral filtering*.

He, Sun, and Tang (2013) point out that these papers are just two examples of the more general concept of *guided image filtering*, where the guide image $\mathbf{h}()$ is used to compute the locally adapted

inter-pixel weights $w(i, j, k, l)$, i.e.,

$$\mathbf{g}(i, j) = \sum_{k,l} w(\mathbf{h}; i, j, k, l) \mathbf{f}(k, l). \quad (3.41)$$

In their paper, the authors suggest modeling the relationship between the guide and input images using a local affine transformation,

$$\mathbf{g}(i, j) = \mathbf{A}_{k,l} \mathbf{h}(i, j) + \mathbf{b}_{k,l}, \quad (3.42)$$

where the estimates for $\mathbf{A}_{k,l}$ and $\mathbf{b}_{k,l}$ are obtained from a regularized least squares fit over a square neighborhood centered around pixel (k, l) , i.e., minimizing

$$\sum_{(i,j) \in \mathcal{N}_{k,l}} \|\mathbf{A}_{k,l} \mathbf{h}(i, j) + \mathbf{b}_{k,l} - \mathbf{f}(i, j)\|^2 + \lambda \|\mathbf{A}\|^2. \quad (3.43)$$

These kinds of regularized least squares problems are called *ridge regression* (Section 4.1). The concept behind this algorithm is illustrated in Figure 3.21.

Instead of just taking the predicted value of the filtered pixel $\mathbf{g}(i, j)$ from the window centered on that pixel, an average across all windows that cover the pixel is used. The resulting algorithm (He, Sun, and Tang 2013, Algorithm 1) consists of a series of local mean image and image moment filters, a per-pixel linear system solve (which reduces to a division if the guide image is scalar), and another set of filtering steps. The authors describe how this fast and simple process has been applied to a wide variety of computer vision problems, including image matting (Section 10.4.3), high dynamic range image tone mapping (Section 10.2.1), stereo matching (Hosni, Rhemann *et al.* 2013), and image denoising.

3.3.3 Binary image processing

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

$$\theta(f, t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else,} \end{cases} \quad (3.44)$$

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

Morphology

The most common binary image operations are called *morphological operations*, because 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 3×3 box filter, to more complicated disc structures. It can even correspond to a particular shape that is being sought for in the image.

Figure 3.22 shows a close-up of the convolution of a binary image f with a 3×3 structuring element s and the resulting images for the operations described below. Let

$$c = f \otimes s \quad (3.45)$$

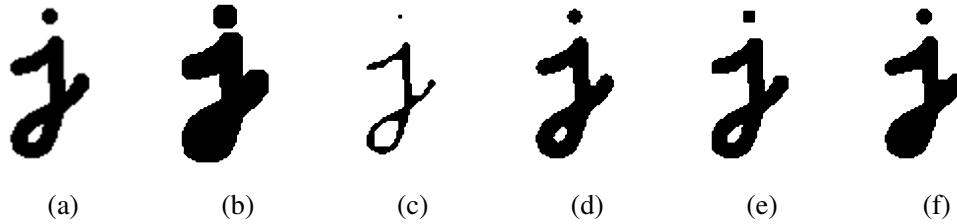


Figure 3.22 Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing. The structuring element for all examples is a 5×5 square. The effects of majority are a subtle rounding of sharp corners. Opening fails to eliminate the dot, as it is not wide enough.

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:** $\text{dilate}(f, s) = \theta(c, 1)$;
- **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 3.22, 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).

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 *et al.* 1995; Felzenszwalb and Huttenlocher 2012; Fabbri, Costa *et al.* 2008). It has many applications, including level sets (Section 7.3.2), fast *chamfer matching* (binary image alignment) (Huttenlocher, Klanderman, and Rucklidge 1993), feathering in image stitching and blending (Section 8.4.2), and nearest point alignment (Section 13.2.1).

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

$$d_1(k, l) = |k| + |l| \quad (3.46)$$

and the *Euclidean* distance

$$d_2(k, l) = \sqrt{k^2 + l^2}. \quad (3.47)$$

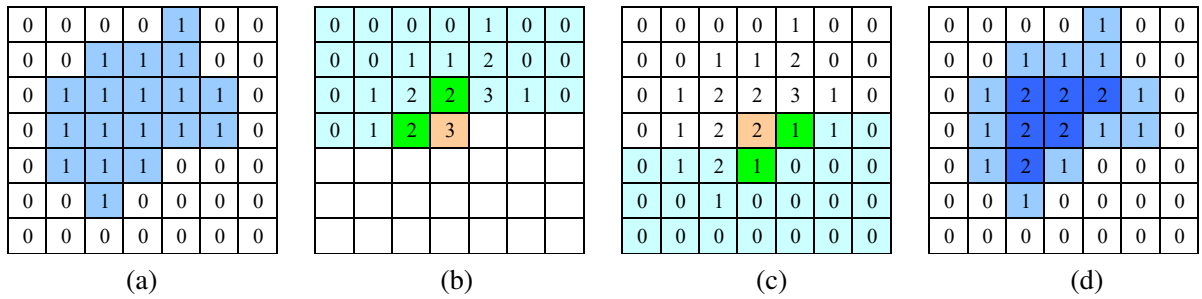


Figure 3.23 City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange value; (d) final distance transform.

The distance transform is then defined as

$$D(i, j) = \min_{k, l: b(k, l) = 0} d(i - k, j - l), \quad (3.48)$$

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 3.23. 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 3.23).

Efficiently computing the Euclidean distance transform is more complicated (Danielsson 1980; Borgefors 1986). 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.

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 (Lavallée and Szeliski 1995; Szeliski and Lavallée 1996; Frisken, Perry *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 7.3.2), where they are called *characteristic functions*.

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. Pixels are said to be \mathcal{N}_4 adjacent if they are immediately horizontally or vertically adjacent, and \mathcal{N}_8 if they can also be diagonally adjacent. Both variants of connected components are widely 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. Over the years, a wide variety of efficient algorithms have been developed to find such components, including the ones described in Haralick and Shapiro (1992, Section 2.3) and He, Ren *et al.* (2017). Such algorithms are usually included in image processing libraries such as OpenCV.

Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region \mathcal{R} . Such statistics include:

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

$$\mathbf{M} = \sum_{(x,y) \in \mathcal{R}} \begin{bmatrix} x - \bar{x} \\ y - \bar{y} \end{bmatrix} \begin{bmatrix} x - \bar{x} & y - \bar{y} \end{bmatrix}, \quad (3.49)$$

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

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 (i.e., the frequency *content* of an image) and how using the Fast Fourier Transform (FFT) lets us perform large-kernel convolutions in time that is independent of the kernel's size. More comprehensive introductions to Fourier transforms are provided by Bracewell (1986), Glassner (1995), Oppenheim and Schaffer (1996), and Oppenheim, Schaffer, and Buck (1999).

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

$$s(x) = \sin(2\pi f x + \phi_i) = \sin(\omega x + \phi_i) \quad (3.50)$$

be the input sinusoid whose *frequency* is f , *angular frequency* is $\omega = 2\pi f$, and *phase* is ϕ_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

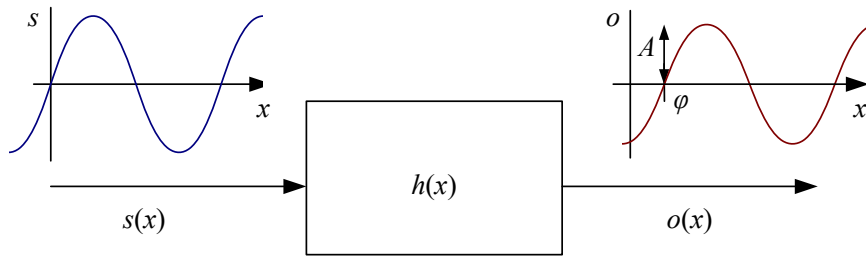


Figure 3.24 The Fourier Transform as the response of a filter $h(x)$ to an input sinusoid $s(x) = e^{j\omega x}$ yielding an output sinusoid $o(x) = h(x) * s(x) = Ae^{j(\omega x + \phi)}$.

engineering literature) and because it is clearer how to distinguish the horizontal (x) and vertical (y) components in frequency space. In this section, we use the letter j for the imaginary number, since that is the form more commonly found in the signal processing literature (Bracewell 1986; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999).

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 ϕ_o ,

$$o(x) = h(x) * s(x) = A \sin(\omega x + \phi_o), \quad (3.51)$$

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.⁹ The new magnitude A is called the *gain* or *magnitude* of the filter, while the phase difference $\Delta\phi = \phi_o - \phi_i$ is called the *shift* or *phase*.

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

$$s(x) = e^{j\omega x} = \cos \omega x + j \sin \omega x. \quad (3.52)$$

In that case, we can simply write,

$$o(x) = h(x) * s(x) = Ae^{j(\omega x + \phi)}. \quad (3.53)$$

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

$$H(\omega) = \mathcal{F}\{h(x)\} = Ae^{j\phi}, \quad (3.54)$$

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

$$h(x) \xleftrightarrow{\mathcal{F}} H(\omega). \quad (3.55)$$

Unfortunately, (3.54) 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

⁹If 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”.

shift, repeat. Fortunately, closed form equations for the Fourier transform exist both in the continuous domain,

$$H(\omega) = \int_{-\infty}^{\infty} h(x)e^{-j\omega x} dx, \quad (3.56)$$

and in the discrete domain,

$$H(k) = \frac{1}{N} \sum_{x=0}^{N-1} h(x)e^{-j\frac{2\pi kx}{N}}, \quad (3.57)$$

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 (3.57) is known as the *Discrete Fourier Transform* (DFT). Note that while (3.57) can be evaluated for any value of k , it only makes sense for values in the range $k \in [-\frac{N}{2}, \frac{N}{2}]$. This is because larger values of k alias with lower frequencies and hence provide no additional information, as explained in the discussion on aliasing in Section 2.3.1.

At face value, the DFT takes $O(N^2)$ operations (multiply-adds) to evaluate. Fortunately, there exists a faster algorithm called the *Fast Fourier Transform* (FFT), which requires only $O(N \log_2 N)$ operations (Bracewell 1986; Oppenheim, Schaffer, and Buck 1999). We do not explain the details of the algorithm here, except to say that it involves a series of $\log_2 N$ stages, where each stage performs small 2×2 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.

The Fourier transform comes with a set of extremely useful properties relating original signals and their Fourier transforms, including superposition, shifting, reversal, convolution, correlation, multiplication, differentiation, domain scaling (stretching), and energy preservation (Parseval's Theorem). To make room for all of the new material in this second edition, I have removed all of these details, as well as a discussion of commonly used Fourier transform pairs. Interested readers should refer to (Szeliski 2010, Section 3.1, Tables 3.1–3.3) or standard textbooks on signal processing and Fourier transforms (Bracewell 1986; Glassner 1995; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999).

We can also compute the Fourier transforms for the small discrete kernels shown in Figure 3.14 (see Table 3.1). 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 prefiltering before decimation (size reduction).

3.4.1 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 ω_x or ω_y , we can create an oriented sinusoid of frequency (ω_x, ω_y) ,

$$s(x, y) = \sin(\omega_x x + \omega_y y). \quad (3.58)$$

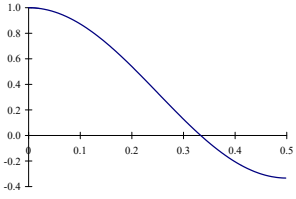
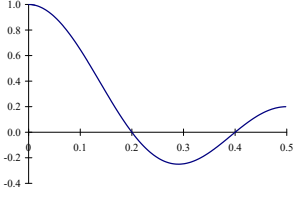
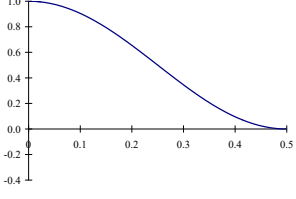
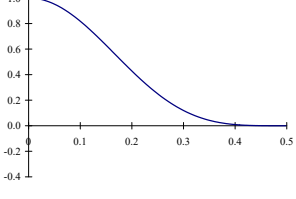
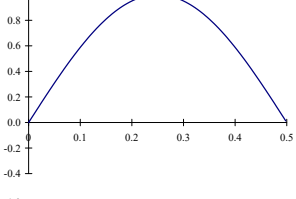
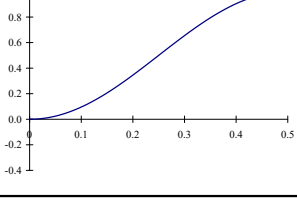
Name	Kernel	Transform	Plot
box-3	$\frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$	$\frac{1}{3}(1 + 2 \cos \omega)$	
box-5	$\frac{1}{5} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$	$\frac{1}{5}(1 + 2 \cos \omega + 2 \cos 2\omega)$	
linear	$\frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$	$\frac{1}{2}(1 + \cos \omega)$	
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 3.1 Fourier transforms of the separable kernels shown in Figure 3.14, obtained by evaluating $\sum_k h(k)e^{-jk\omega}$.

The corresponding two-dimensional Fourier transforms are then

$$H(\omega_x, \omega_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x, y) e^{-j(\omega_x x + \omega_y y)} dx dy, \quad (3.59)$$

and in the discrete domain,

$$H(k_x, k_y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-j2\pi(k_x x/M + k_y y/N)} \quad (3.60)$$

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

All of the Fourier transform properties from 1D carry over to two dimensions if we replace the scalar variables x , ω , x_0 and a , with their 2D vector counterparts $\mathbf{x} = (x, y)$, $\boldsymbol{\omega} = (\omega_x, \omega_y)$, $\mathbf{x}_0 = (x_0, y_0)$, and $\mathbf{a} = (a_x, a_y)$, and use vector inner products instead of multiplications.

Wiener filtering

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

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

$$\langle [S(\omega_x, \omega_y)]^2 \rangle = P_s(\omega_x, \omega_y), \quad (3.61)$$

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

The first edition of this book contains a derivation of the Wiener filter (Szeliski 2010, Section 3.4.3), but I’ve decided to remove this from the current edition, since it is almost never used in practice any more, having been replaced with better-performing non-linear filters.

Discrete cosine transform

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

$$F(k) = \sum_{i=0}^{N-1} \cos\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)k\right) f(i), \quad (3.62)$$

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 3.25. 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.

¹⁰The notation $E[\cdot]$ is also commonly used.

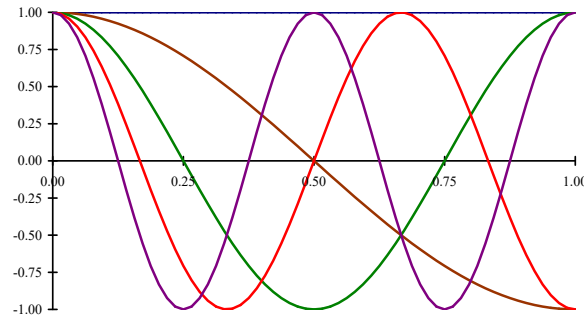


Figure 3.25 Discrete cosine transform (DCT) basis functions: The first DC (i.e., constant) basis is the horizontal blue line, the second is the brown half-cycle waveform, etc. These bases are widely used in image and video compression standards such as JPEG.

It turns out that the DCT is a good approximation to the optimal Karhunen–Loève decomposition of natural image statistics over small patches, which can be obtained by performing a principal component analysis (PCA) of images, as described in Section 5.2.3. The KL-transform decorrelates the signal optimally (assuming the signal is described by its spectrum) and thus, theoretically, leads to optimal compression.

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

$$F(k, l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)k\right) \cos\left(\frac{\pi}{N}\left(j + \frac{1}{2}\right)l\right) f(i, j). \quad (3.63)$$

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 alternatives such as wavelet transforms (Simoncelli and Adelson 1990b; Taubman and Marcellin 2002), discussed in Section 3.5.4, and overlapped variants of the DCT (Malvar 1990, 1998, 2000), are used in the JPEG2000 and JPEG XR standards. These newer algorithms suffer less from the *blocking artifacts* (visible edge-aligned discontinuities) that result from the pixels in each block (typically 8×8) being transformed and quantized independently. See Exercise 4.3 for ideas on how to remove blocking artifacts from compressed JPEG images.

3.4.2 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 operations were performed using linear filtering (see Sections 3.2 and Section 3.4.1). 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 4.2), especially those using non-quadratic (robust) norms such as the L_1 norm (which is called *total variation*), are also often used. Most recently, deep neural networks have taken over the denoising community (Section 10.3). 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 (2.120), where $I(\mathbf{x})$ is the original

(noise-free) image and $\hat{I}(\mathbf{x})$ 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 *et al.* 2004; Wang, Bovik, and Simoncelli 2005) or FLIP image difference evaluator (Andersson, Nilsson *et al.* 2020). More recently, people have started measuring similarity using neural “perceptual” similarity metrics (Johnson, Alahi, and Fei-Fei 2016; Dosovitskiy and Brox 2016; Zhang, Isola *et al.* 2018; Tariq, Tursun *et al.* 2020; Czolbe, Krause *et al.* 2020), which, unlike L_2 (PSNR) or L_1 metrics, which encourage smooth or flat average results, prefer images with similar amounts of texture (Cho, Joshi *et al.* 2012). When the clean image is not available, it is also possible to assess the quality of an image using *no-reference image quality assessment* (Mittal, Moorthy, and Bovik 2012; Talebi and Milanfar 2018).

Exercises 3.12, 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 6.3.1). 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 9.1.1). Finally, image pyramids are extremely useful for performing multi-scale editing operations such as blending images while maintaining details.

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

3.5.1 Interpolation

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

$$g(i, j) = \sum_{k, l} f(k, l)h(i - rk, j - rl). \quad (3.64)$$

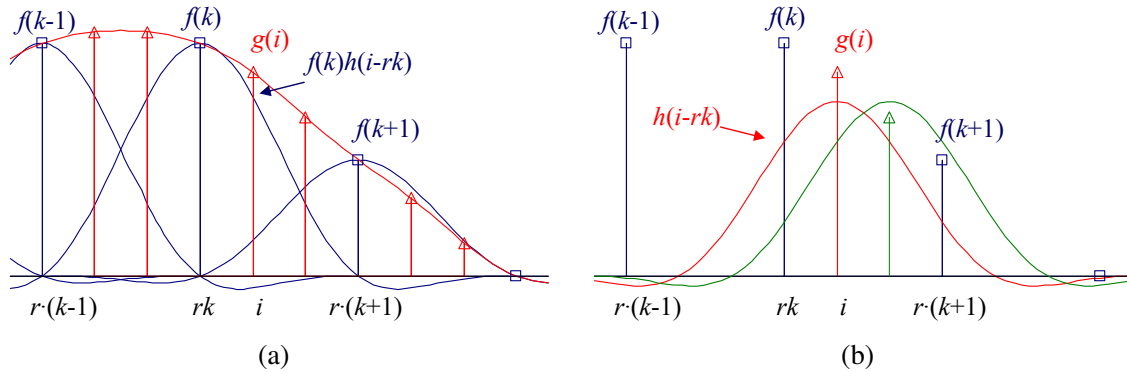


Figure 3.26 Signal interpolation, $g(i) = \sum_k f(k)h(i - rk)$: (a) weighted summation of input values; (b) polyphase filter interpretation.

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 3.26a 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 3.26b, 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.

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 Table 3.1 can be used after appropriate re-scaling.¹¹ The *linear* interpolator (corresponding to the tent kernel) produces interpolating piecewise linear curves, which result in unappealing *creases* when applied to images (Figure 3.27a). The cubic B-spline, whose discrete $1/2$ -pixel sampling appears as the *binomial kernel* in Table 3.1, 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”)¹² whose equation is

$$h(x) = \begin{cases} 1 - (a + 3)x^2 + (a + 2)|x|^3 & \text{if } |x| < 1 \\ a(|x| - 1)(|x| - 2)^2 & \text{if } 1 \leq |x| < 2 \\ 0 & \text{otherwise,} \end{cases} \quad (3.65)$$

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 3.28). 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

¹¹The smoothing kernels in Table 3.1 have a unit area. To turn them into interpolating kernels, we simply scale them up by the interpolation rate r .

¹²The term “spline” comes from the draughtsman’s workshop, where it was the name of a flexible piece of wood or metal used to draw smooth curves.

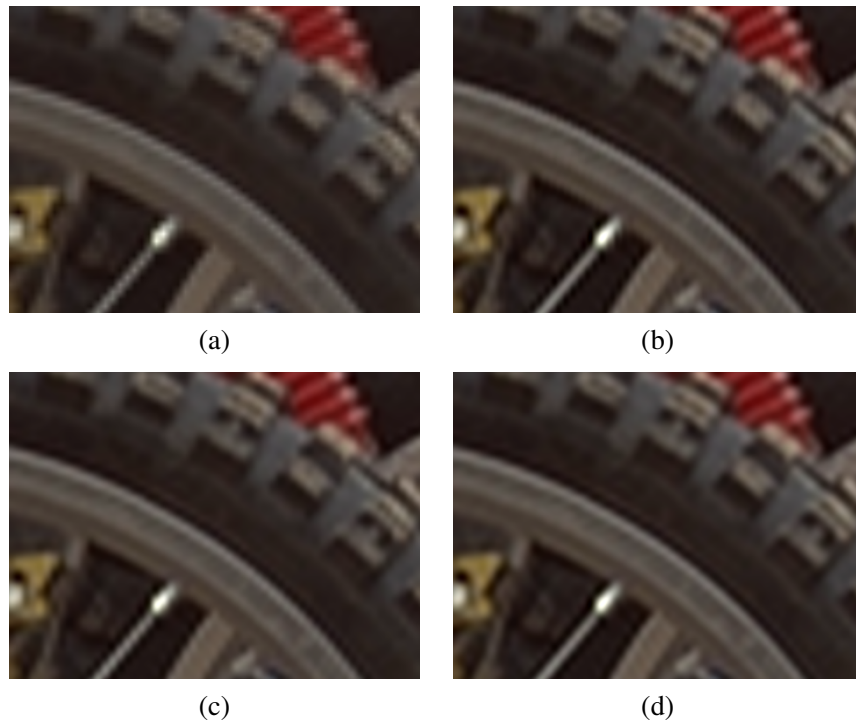


Figure 3.27 Two-dimensional image interpolation: (a) bilinear; (b) bicubic ($a = -1$); (c) bicubic ($a = -0.5$); (d) windowed sinc (nine taps).

quadratic reproducing spline; it interpolates linear and quadratic functions exactly (Wolberg 1990, Section 5.4.3). Figure 3.28 shows the $a = -1$ and $a = -0.5$ cubic interpolating kernel along with their Fourier transforms; Figure 3.27b and c shows them being applied to two-dimensional interpolation.

Splines have long been used for function and data value interpolation because of the ability to precisely specify derivatives at control points and efficient *incremental* algorithms for their evaluation (Bartels, Beatty, and Barsky 1987; Farin 1992, 2002). 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), scattered data interpolation (Section 4.1), motion estimation (Section 9.2.2), and surface interpolation (Section 13.3). 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; Nehab and Hoppe 2014).

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 3.27d). For this reason, some photographers prefer to repeatedly interpolate images by a small fractional amount (this tends to decorrelate the original pixel grid with the final image). Additional possibilities include using the bilateral filter as an interpolator (Kopf, Cohen *et al.* 2007), using global optimization (Section 3.6) or hallucinating details (Section 10.3).

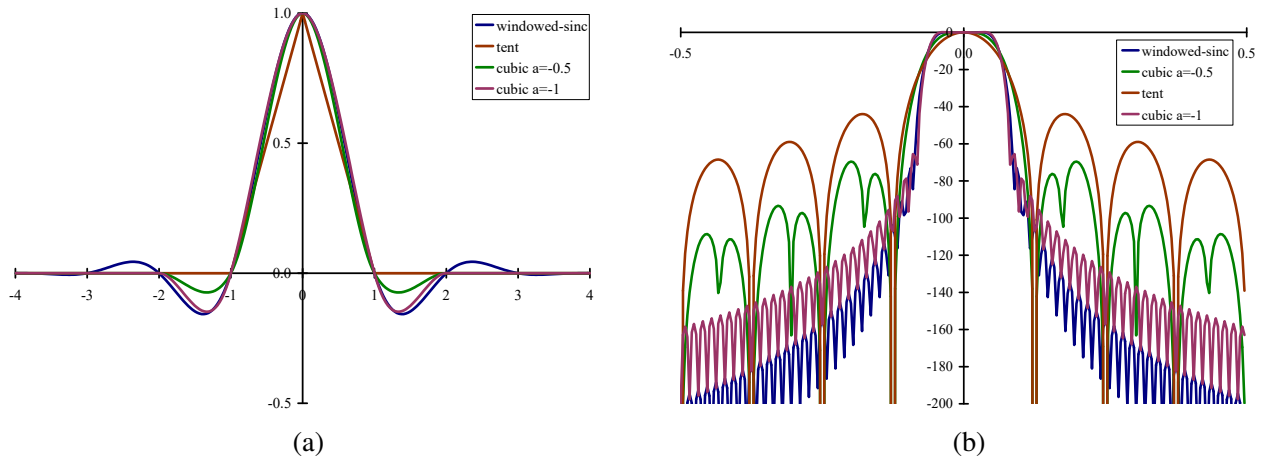


Figure 3.28 (a) Some windowed sinc functions and (b) their log Fourier transforms: raised-cosine windowed sinc in blue, cubic interpolators ($a = -1$ and $a = -0.5$) in green and purple, and tent function in brown. They are often used to perform high-accuracy low-pass filtering operations.

3.5.2 Decimation

While interpolation can be used to increase the resolution of an image, decimation (downsampling) is required to reduce the resolution.¹³ To perform decimation, we first (conceptually) convolve the image with a low-pass filter (to avoid aliasing) and then keep every r th sample. In practice, we usually only evaluate the convolution at every r th sample,

$$g(i, j) = \sum_{k, l} f(k, l) h(ri - k, rj - l), \quad (3.66)$$

as shown in Figure 3.29. Note that the smoothing kernel $h(k, l)$, in this case, is often a stretched and re-scaled version of an interpolation kernel. Alternatively, we can write

$$g(i, j) = \frac{1}{r} \sum_{k, l} f(k, l) h(i - k/r, j - l/r) \quad (3.67)$$

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 3.1, 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 desired. For high downsampling rates, the windowed sinc prefiler is a good choice (Figure 3.28). However, for small downsampling rates, e.g., $r = 2$, more careful filter design is required.

Table 3.2 shows a number of commonly used $r = 2$ downsampling filters, while Figure 3.30 shows their corresponding frequency responses. These filters include:

- the linear $[1, 2, 1]$ filter gives a relatively poor response;

¹³The term “decimation” has a gruesome etymology relating to the practice of killing every tenth soldier in a Roman unit guilty of cowardice. It is generally used in signal processing to mean any downsampling or rate reduction operation.

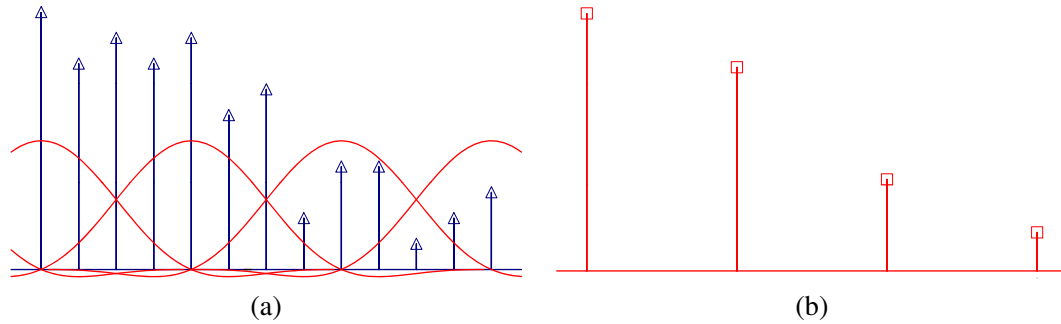


Figure 3.29 Signal decimation: (a) the original samples are (b) convolved with a low-pass filter before being downsampled.

- 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 (3.65); the $a = -1$ filter has a sharper fall-off than the $a = -0.5$ filter (Figure 3.30);
- a cosine-windowed sinc function;
- 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.

3.5.3 Multi-resolution representations

Now that we have described interpolation and decimation algorithms, we can build a complete image pyramid (Figure 3.31). 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 discuss in Section 3.6.

$ 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 3.2 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 3.30 for their associated frequency responses.

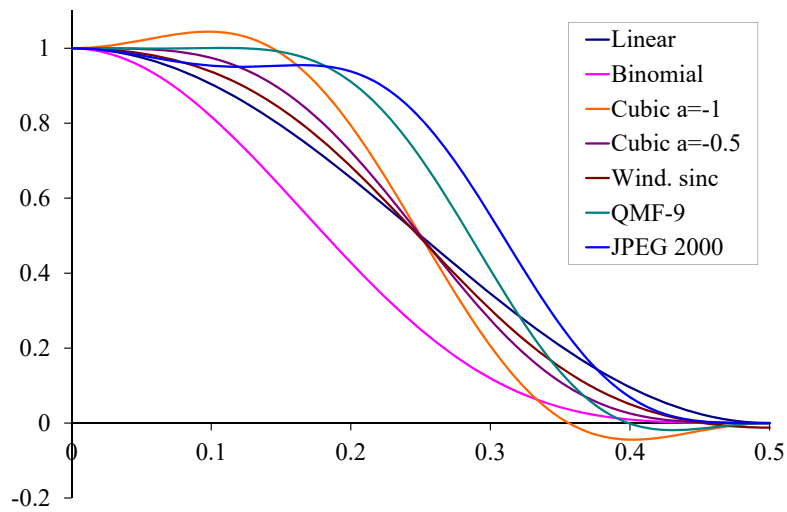


Figure 3.30 Frequency response for some $2 \times$ decimation filters. The cubic $a = -1$ filter has the sharpest fall-off but also a bit of ringing; the wavelet analysis filters (QMF-9 and JPEG 2000), while useful for compression, have more aliasing.

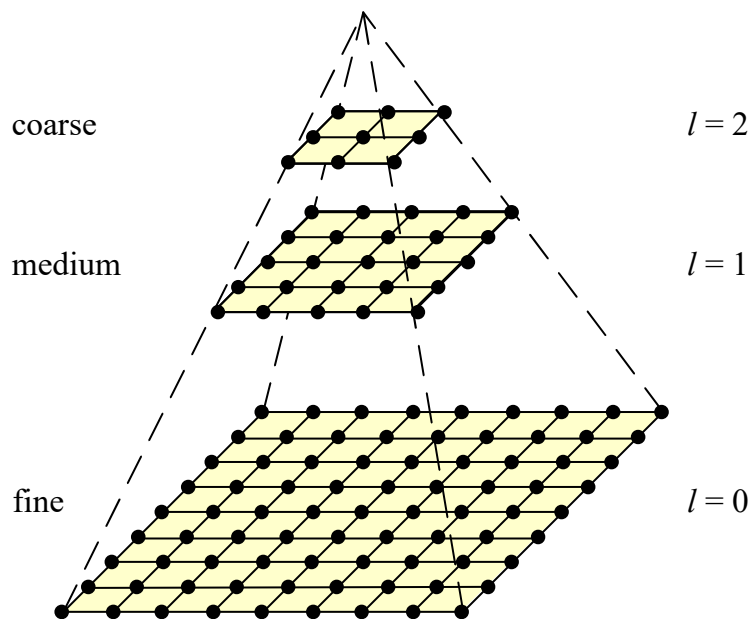


Figure 3.31 A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level.

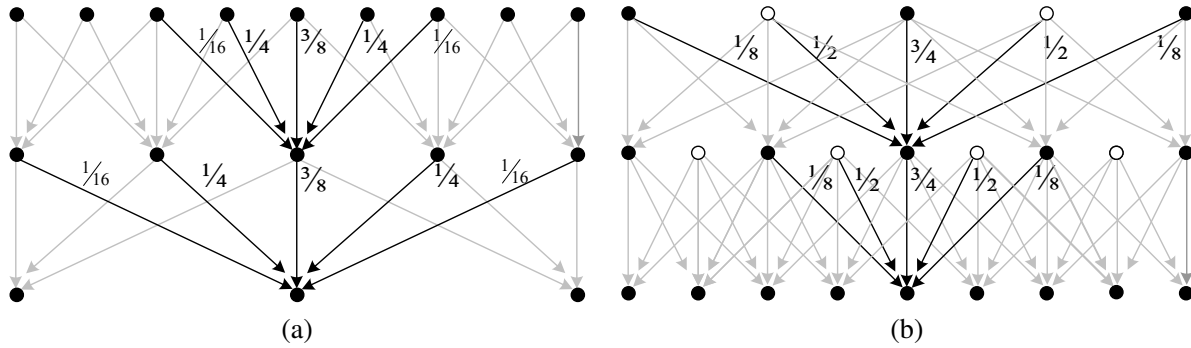


Figure 3.32 The Gaussian pyramid shown as a signal processing diagram: The (a) analysis and (b) re-synthesis stages are shown as using similar computations. The white circles indicate zero values inserted by the $\uparrow 2$ upsampling operation. Notice how the reconstruction filter coefficients are 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*.

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 (Figures 3.31 and 3.32). 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

$$\boxed{c \quad b \quad a \quad b \quad c}, \tag{3.68}$$

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

$$\frac{1}{16} \boxed{1 \quad 4 \quad 6 \quad 4 \quad 1}, \tag{3.69}$$

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.¹⁴

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 3.33). They then subtract this low-pass version from the original to yield the band-pass “Laplacian” 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 3.33) are sufficient to exactly reconstruct the original image. Figure 3.32 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 of 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 3.33). 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

¹⁴Then again, this is true for any smoothing kernel (Wells 1986).

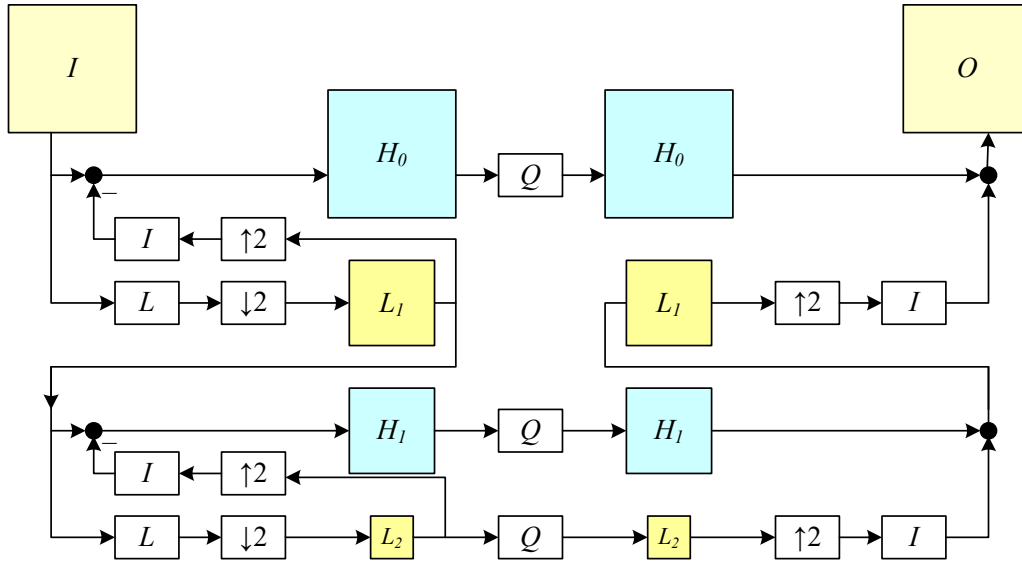


Figure 3.33 The Laplacian pyramid. The yellow images form the *Gaussian* pyramid, which is obtained by successively low-pass filtering and downsampling the input image. The blue images, together with the smallest low-pass image, which is needed for reconstruction, form the *Laplacian* pyramid. Each band-pass (blue) image is computed by upsampling and interpolating the lower-resolution Gaussian pyramid image, resulting in a blurred version of that level's low-pass image, which is subtracted from the low-pass to yield the blue band-pass image. During reconstruction, the interpolated images and the (optionally filtered) high-pass images are added back together starting with the coarsest level. The Q box indicates quantization or some other pyramid processing, e.g., noise removal by *coring* (setting small wavelet values to 0).

images are really differences of (approximate) Gaussians, or DoGs,

$$\text{DoG}\{I; \sigma_1, \sigma_2\} = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I. \quad (3.70)$$

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

$$\text{LoG}\{I; \sigma\} = \nabla^2(G_{\sigma} * I) = (\nabla^2 G_{\sigma}) * I, \quad (3.71)$$

where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \quad (3.72)$$

is the Laplacian (operator) of a function. Figure 3.34 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).

One particularly useful application of the Laplacian pyramid is in the manipulation of local contrast as well as the tone mapping of high dynamic range images (Section 10.2.1). Paris, Hasinoff, and Kautz (2011) present a technique they call *local Laplacian filters*, which uses local range clipping in the construction of a modified Laplacian pyramid, as well as different accentuation and attenuation curves for small and large details, to implement edge-preserving filtering and tone mapping. Aubry,

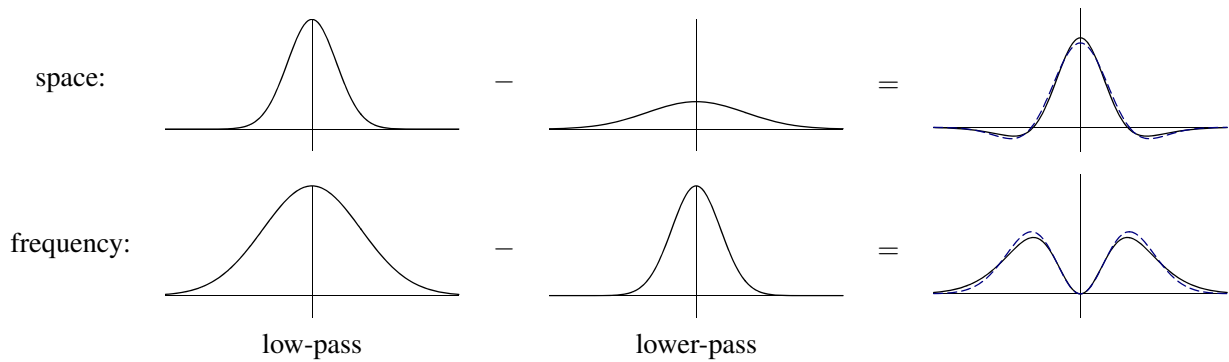


Figure 3.34 The difference of two low-pass filters results in a band-pass filter. The dashed blue lines show the close fit to a half-octave Laplacian of Gaussian.

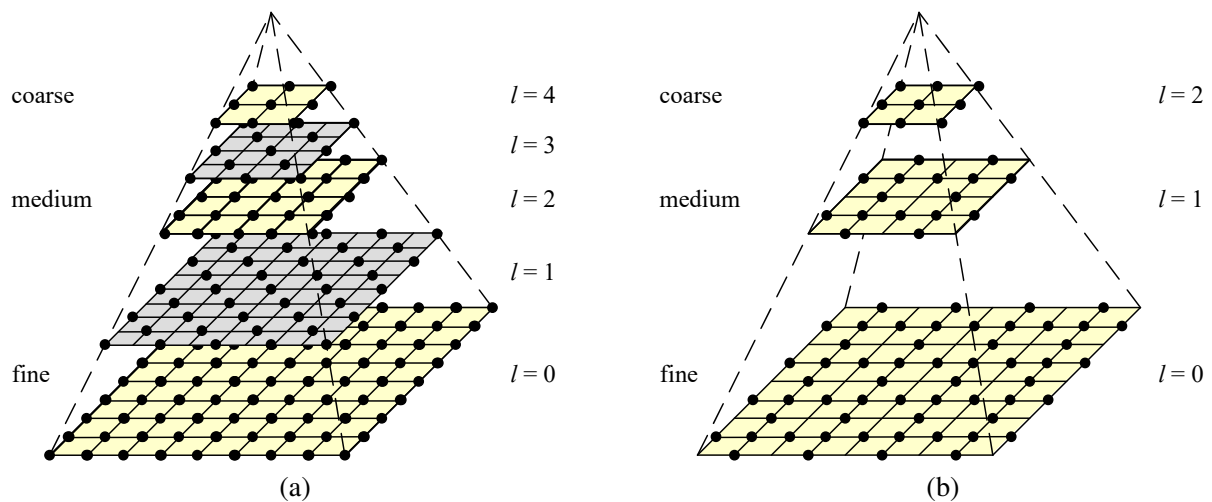


Figure 3.35 Multiresolution pyramids: (a) pyramid with half-octave (*quincunx*) sampling (odd levels are colored gray for clarity). (b) wavelet pyramid—each wavelet level stores $3/4$ of the original pixels (usually the horizontal, vertical, and mixed gradients), so that the total number of wavelet coefficients and original pixels is the same.

Paris *et al.* (2014) discuss how to accelerate this processing for monotone (single channel) images and also show style transfer applications.

A less widely used variant is *half-octave pyramids*, shown in Figure 3.35a. 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 adjacent 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 7.1.1), 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 7.11).

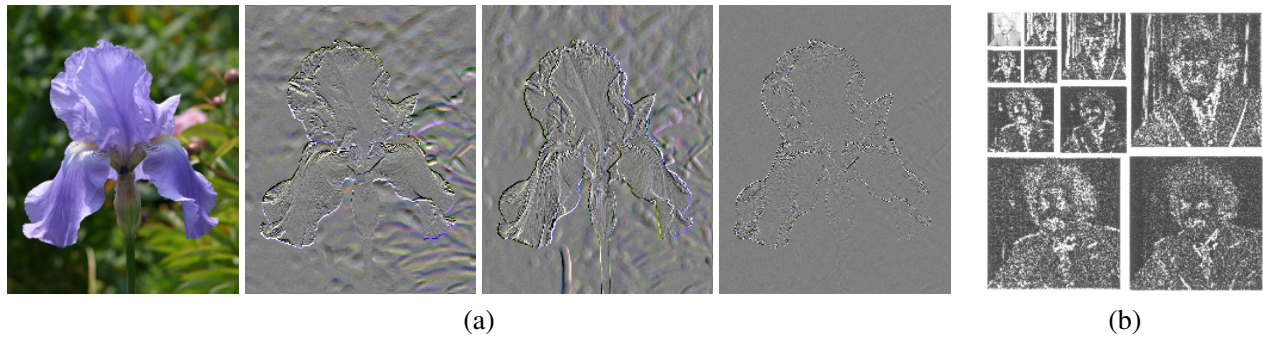


Figure 3.36 A wavelet decomposition of an image: (a) single level decomposition with horizontal, vertical, and diagonal detail wavelets constructed using PyWavelet code (<https://pywavelets.readthedocs.io>); (b) coefficient magnitudes of a multi-level decomposition, with the high–high components in the lower right corner and the base in the upper left (Buccigrossi and Simoncelli 1999) © 1999 IEEE. Notice how the low–high and high–low components accentuate horizontal and vertical edges and gradients, while the high-high components store the less frequent mixed derivatives.

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) 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), and 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 *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; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as well as for multi-scale oriented filtering (Simoncelli, Freeman *et al.* 1992) and denoising (Portilla, Strela *et al.* 2003).

As 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 3.35b). However, some wavelet families *are*, in fact, overcomplete in order to provide better shiftability or steering in orientation (Simoncelli, Freeman *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 $\frac{3}{4}$ of the original pixels, while $\frac{1}{4}$ of the low-frequency coefficients are passed on to the next

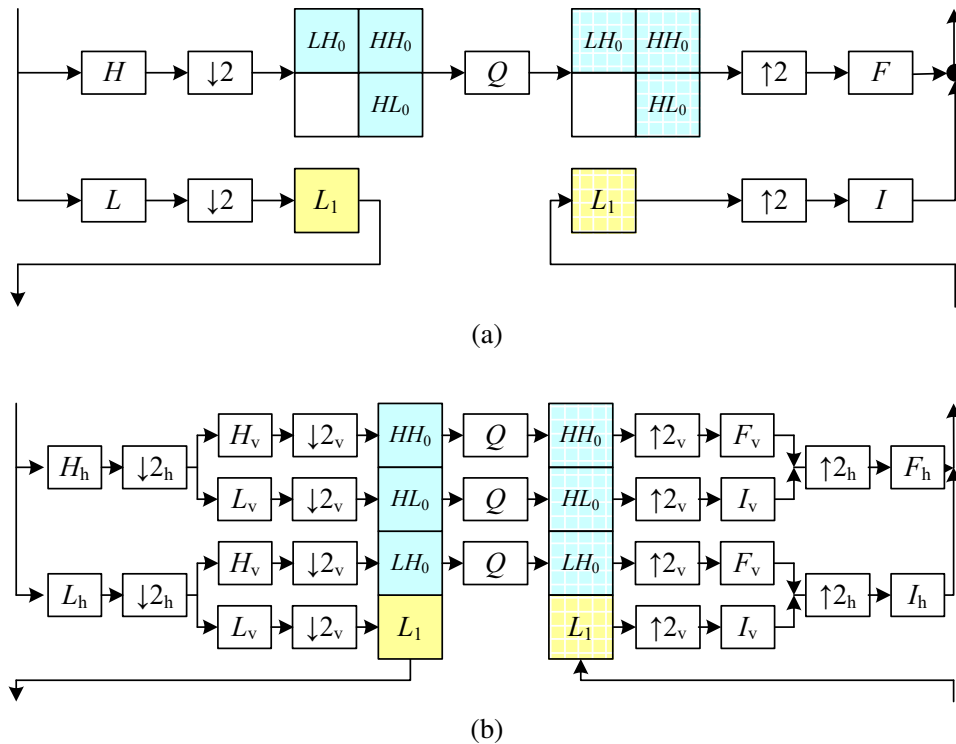


Figure 3.37 Two-dimensional wavelet decomposition: (a) high-level diagram showing the low-pass and high-pass transforms as single boxes; (b) separable implementation, which involves first performing the wavelet transform horizontally and then vertically. The I and F boxes are the interpolation and filtering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

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 (Figure 3.36).

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

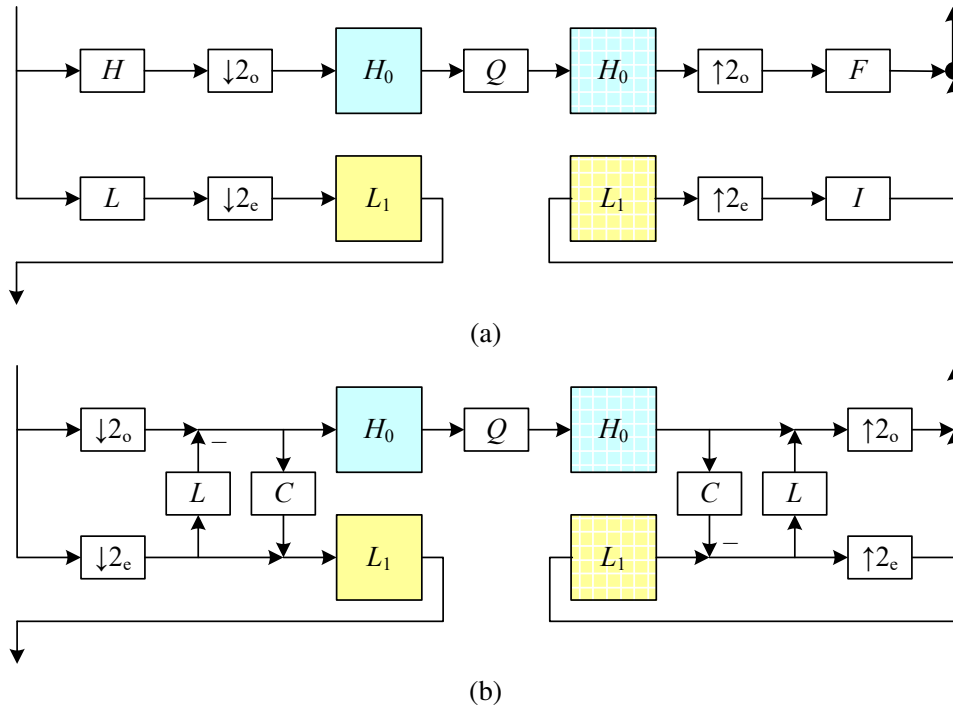


Figure 3.38 One-dimensional wavelet transform: (a) usual high-pass + low-pass filters followed by odd ($\downarrow 2_o$) and even ($\downarrow 2_e$) downsampling; (b) lifted version, which first selects the odd and even subsequences and then applies a low-pass prediction stage L and a high-pass correction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

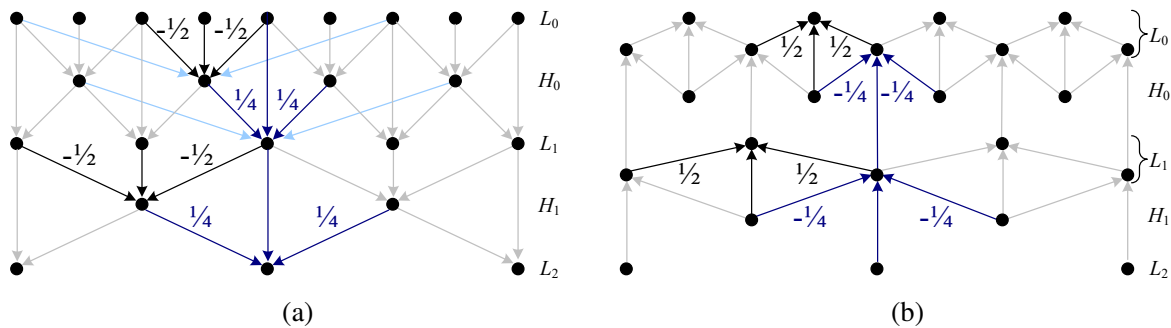


Figure 3.39 Lifted transform shown as a signal processing diagram: (a) The analysis stage first predicts the odd value from its even neighbors, stores the difference wavelet, and then compensates the coarser even value by adding in a fraction of the wavelet. (b) The synthesis stage simply reverses the 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.

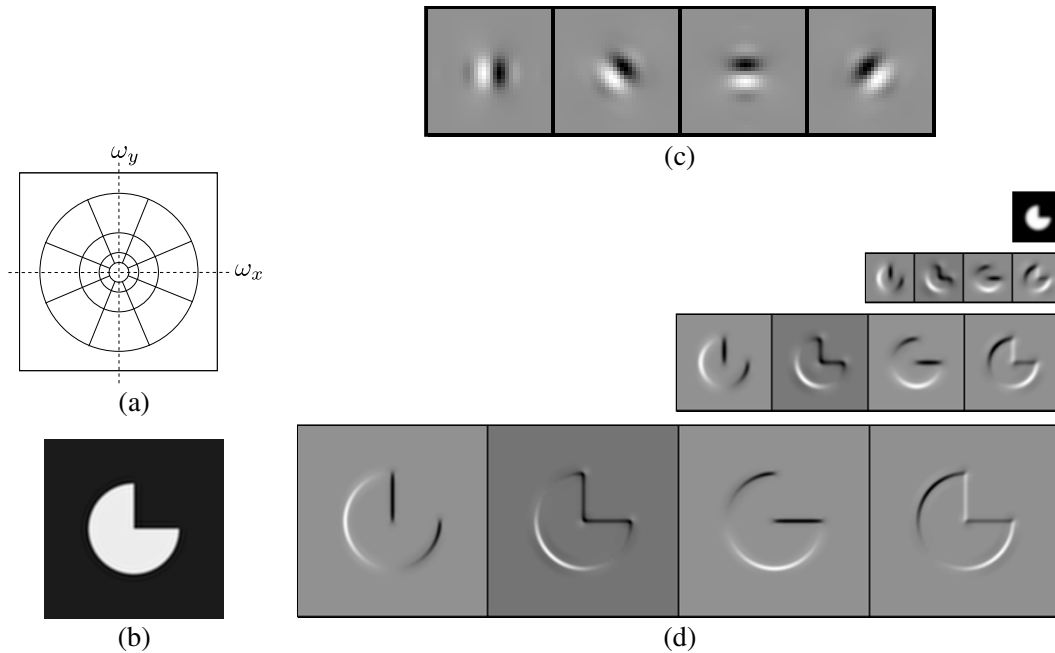


Figure 3.40 Steerable shiftable multiscale transforms (Simoncelli, Freeman *et al.* 1992) © 1992 IEEE: (a) radial multi-scale frequency domain decomposition; (b) original image; (c) a set of four steerable filters; (d) the radial multi-scale wavelet decomposition.

low-pass filter L and subtracting the result from the odd 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 difference signal 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.

This process can 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 precondi-

tioning the kinds of sparse linear systems that arise in the optimization-based approaches to vision algorithms that we discuss in Chapter 4 (Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013).

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.¹⁵ Simoncelli, Freeman *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*. 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 2×2 squares, which results in checkerboard high frequencies, radial arcs are used instead. Figure 3.40d illustrates the resulting pyramid sub-bands. Even though 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 *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 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 3.41g) 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 3.41e–h 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 3.18). This is particularly useful in image stitching and compositing applications, where the exposures may vary between different images, as described in Section 8.4.4, where we also present more recent variants such as Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet *et al.* 2004).

¹⁵Such aliasing can often be seen as the signal content moving between bands as the original signal is slowly shifted.

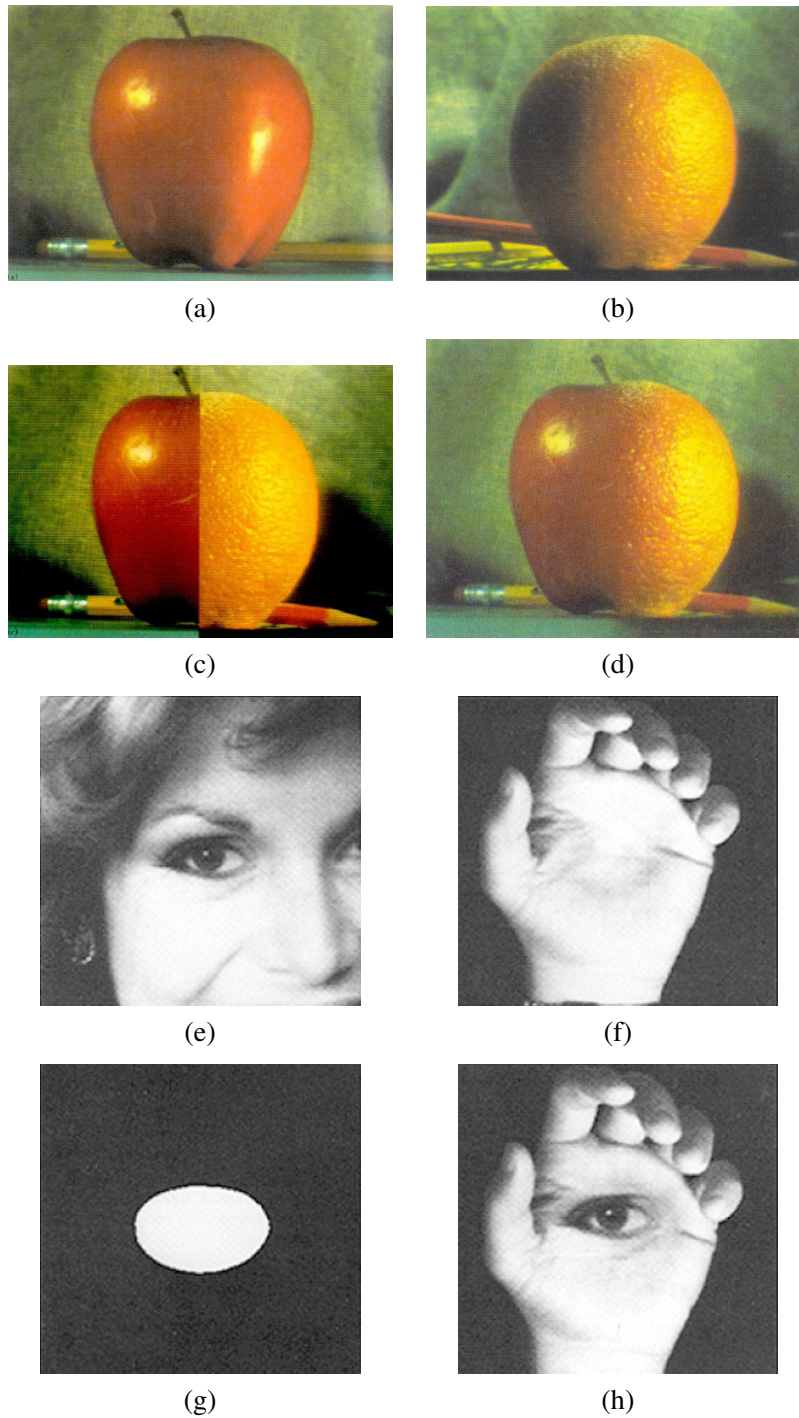


Figure 3.41 Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM: (a) original image of apple, (b) original image of orange, (c) regular splice, (d) pyramid blend. A masked blend of two images: (e) first input image, (f) second input image, (g) region mask, (h) blended image.

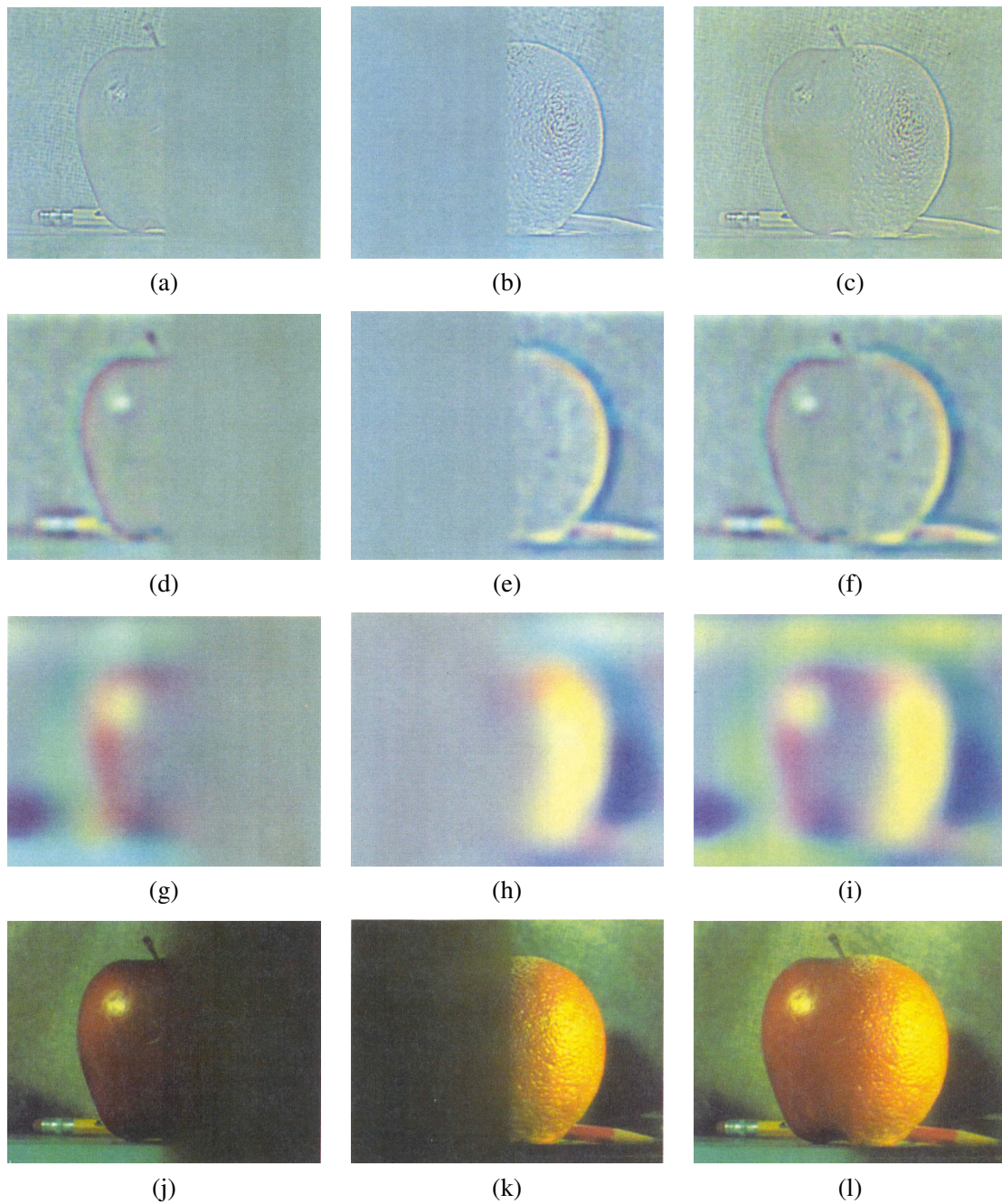


Figure 3.42 Laplacian pyramid blending details (Burt and Adelson 1983b) © 1983 ACM. The first three rows show the high, medium, and low-frequency parts of the Laplacian pyramid (taken from levels 0, 2, and 4). The left and middle columns show the original apple and orange images weighted by the smooth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

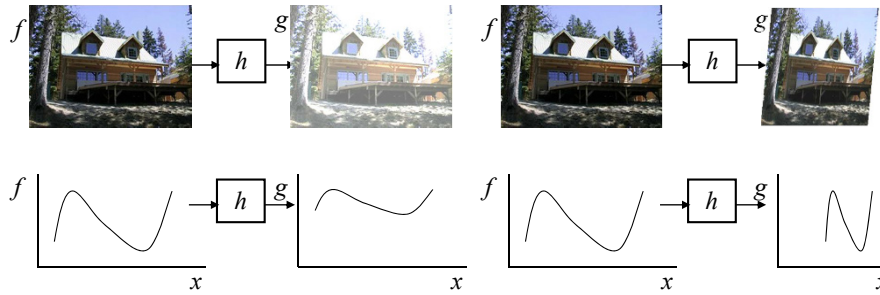


Figure 3.43 Image warping involves modifying the *domain* of an image function rather than its *range*.

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,

$$g(\mathbf{x}) = h(f(\mathbf{x})), \quad (3.73)$$

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

$$g(\mathbf{x}) = f(\mathbf{h}(\mathbf{x})), \quad (3.74)$$

as shown in Figure 3.43.

We begin by studying the global *parametric* 2D transformation first introduced in Section 2.1.1. (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 in Section 3.6.3 how image warps can be combined with cross-dissolves to create interesting *morphs* (in-between animations). 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 *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.

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 3.44 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 3.3 for ease of reference.

In general, given a transformation specified by a formula $\mathbf{x}' = \mathbf{h}(\mathbf{x})$ and a source image $f(\mathbf{x})$, how do we compute the values of the pixels in the new image $g(\mathbf{x})$, as given in (3.74)? 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 3.45a. Can you think of any problems with this approach?

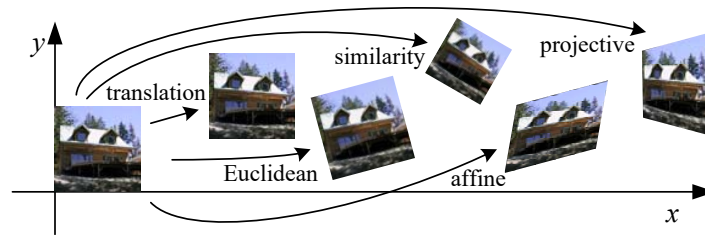


Figure 3.44 Basic set of 2D geometric image transformations.






Transformation	Matrix	# DoF	Preserves	Icon
translation	$[\mathbf{I} \ \mathbf{t}]_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$[\mathbf{R} \ \mathbf{t}]_{2 \times 3}$	3	lengths	
similarity	$[s\mathbf{R} \ \mathbf{t}]_{2 \times 3}$	4	angles	
affine	$[\mathbf{A}]_{2 \times 3}$	6	parallelism	
projective	$[\tilde{\mathbf{H}}]_{3 \times 3}$	8	straight lines	

Table 3.3 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 2×3 matrices are extended with a third $[\mathbf{0}^T \ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.

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

You can round the value of \mathbf{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).

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 $g(\mathbf{x}')$ is sampled from the original image $f(\mathbf{x})$ (Figure 3.46).

How does this differ from the forward warping algorithm? For one thing, since $\hat{\mathbf{h}}(\mathbf{x}')$ is (presumably) defined for all pixels in $g(\mathbf{x}')$, 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.

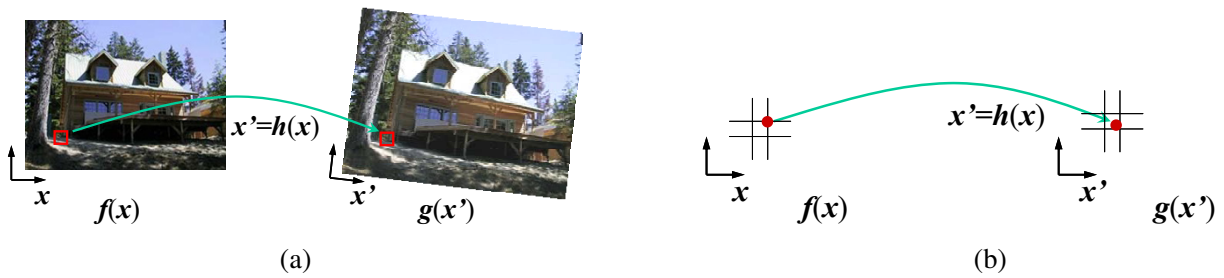


Figure 3.45 Forward warping algorithm: (a) a pixel $f(x)$ is copied to its corresponding location $x' = h(x)$ in image $g(x')$; (b) detail of the source and destination pixel locations.

```

procedure forwardWarp( $f, h$ , out  $g$ ):
    For every pixel  $x$  in  $f(x)$ 
        1. Compute the destination location  $x' = h(x)$ .
        2. Copy the pixel  $f(x)$  to  $g(x')$ .
    
```

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



Figure 3.46 Inverse warping algorithm: (a) a pixel $g(x')$ is sampled from its corresponding location $x = \hat{h}(x')$ in image $f(x)$; (b) detail of the source and destination pixel locations.

```

procedure inverseWarp( $f, h$ , out  $g$ ):
    For every pixel  $x'$  in  $g(x')$ 
        1. Compute the source location  $x = \hat{h}(x')$ 
        2. Resample  $f(x)$  at location  $x$  and copy to  $g(x')$ 
    
```

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

Where does the function $\hat{\mathbf{h}}(\mathbf{x}')$ come from? Quite often, it can simply be computed as the inverse of $\mathbf{h}(\mathbf{x})$. In fact, all of the parametric transforms listed in Table 3.3 have closed form solutions for the inverse transform: simply take the inverse of the 3×3 matrix specifying the transform.

In other cases, it is preferable to formulate the problem of image warping as that of resampling a source image $f(\mathbf{x})$ given a mapping $\mathbf{x} = \hat{\mathbf{h}}(\mathbf{x}')$ from destination pixels \mathbf{x}' to source pixels \mathbf{x} . For example, in optical flow (Section 9.3), we estimate the flow field as the location of the *source* pixel that 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 2.1.5), 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 9.1.3), bicubic, and windowed sinc are preferable where visual quality is important.

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

$$g(x, y) = \sum_{k,l} f(k, l)h(x - k, y - l), \quad (3.75)$$

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 is

$$g(\mathbf{A}\mathbf{x}) \Leftrightarrow |\mathbf{A}|^{-1}G(\mathbf{A}^{-T}\mathbf{f}). \quad (3.76)$$

For all of the transforms in Table 3.3 except perspective, the matrix \mathbf{A} is already defined. For perspective transformations, the matrix \mathbf{A} is the linearized *derivative* of the perspective transformation (Figure 3.47a), i.e., the local affine approximation to the stretching induced by the projection (Heckbert 1986; Wolberg 1990; Gomes, Darsa *et al.* 1999; Akenine-Möller and Haines 2002).

To prevent aliasing, we need to prefilter the image $f(\mathbf{x})$ with a filter whose frequency response is the projection of the final desired spectrum through the \mathbf{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 prefilter images being used for *texture mapping* in computer graphics. A MIP-map¹⁶ is a standard image pyramid

¹⁶The term “MIP” stands for *multi in parvo*, meaning “many in one”.

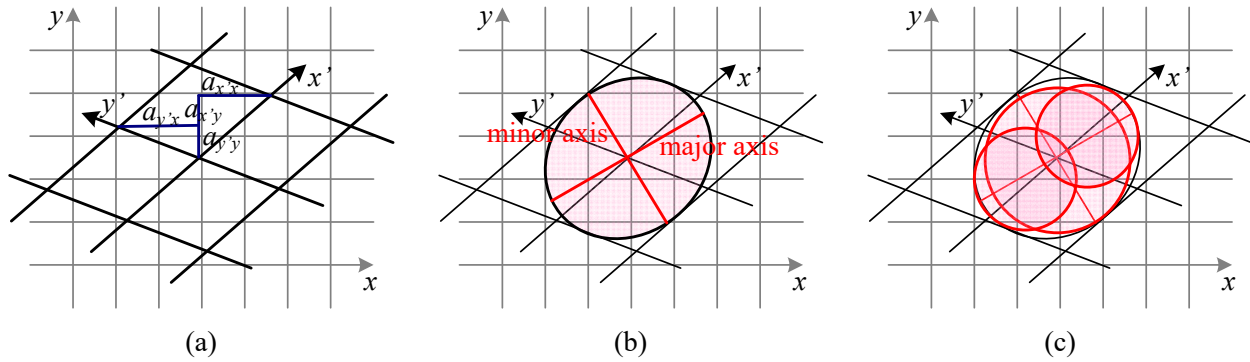


Figure 3.47 Anisotropic texture filtering: (a) Jacobian of transform \mathbf{A} and the induced horizontal and vertical resampling rates $\{a_{x'x}, a_{x'y}, a_{y'x}, a_{y'y}\}$; (b) elliptical footprint of an EWA smoothing kernel; (c) anisotropic filtering using multiple samples along the major axis. Image pixels lie at line intersections.

(Figure 3.31), where each level is prefiltered 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 \mathbf{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,

$$l = \log_2 r. \quad (3.77)$$

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 $\mathbf{x} = \mathbf{A}\mathbf{x}'$ defines a skewed two-dimensional coordinate system in the vicinity of each source pixel \mathbf{x} (Figure 3.47a). For every destination pixel \mathbf{x}' , the ellipsoidal projection of a small pixel grid in \mathbf{x}' onto \mathbf{x} is computed (Figure 3.47b). This is then used to filter the source image $g(\mathbf{x})$ 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, as described in (Szeliski, Winder, and Uyttendaele 2010).

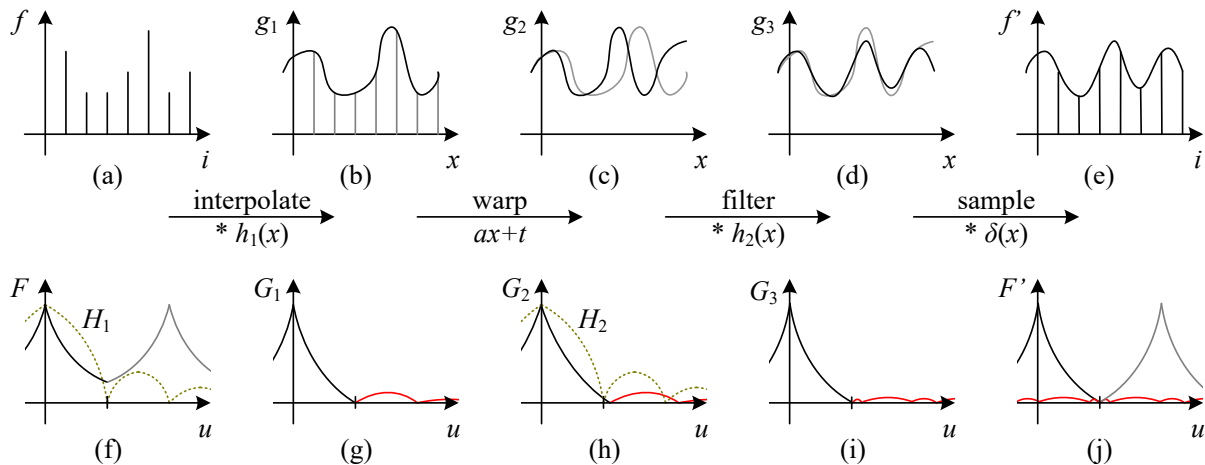


Figure 3.48 One-dimensional signal resampling (Szeliski, Winder, and Uyttendaele 2010): (a) original sampled signal $f(i)$; (b) interpolated signal $g_1(x)$; (c) warped signal $g_2(x)$; (d) filtered signal $g_3(x)$; (e) sampled signal $f'(i)$. The corresponding spectra are shown below the signals, with the aliased portions shown in red.

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 3.47c).

Multi-pass transforms

The optimal approach to warping images without excessive blurring or aliasing is to adaptively prefilter 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 3.47a). Figure 3.48 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 *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).

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.

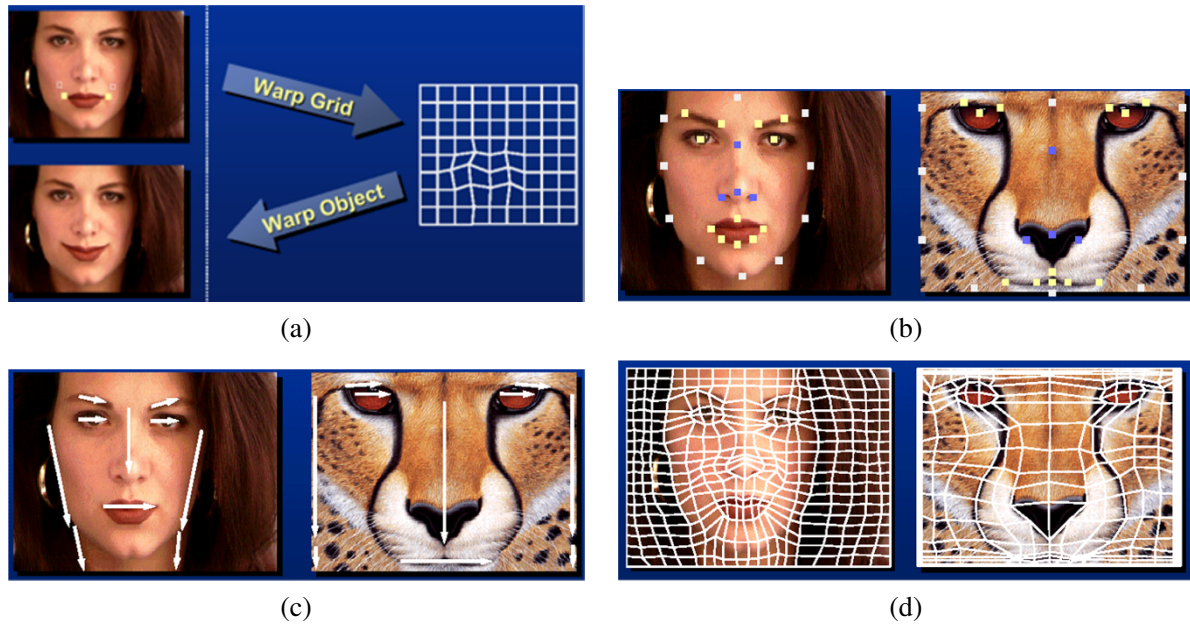


Figure 3.49 Image warping alternatives (Gomes, Darsa *et al.* 1999) © 1999 Morgan Kaufmann: (a) sparse control points \rightarrow deformation grid; (b) denser set of control point correspondences; (c) oriented line correspondences; (d) uniform quadrilateral grid.

Consider, for example, changing the appearance of a face from a frown to a smile (Figure 3.49a). What is needed in this case is to curve the corners of the mouth upwards while leaving the rest of the face intact.¹⁷ To perform such a transformation, different amounts of motion are required in different parts of the image. Figure 3.49 shows some of the commonly used approaches.

The first approach, shown in Figure 3.49a–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 9) using a variety of techniques, which are described in more detail in Section 4.1 on scattered data interpolation. One possibility is to *triangulate* the set of points in one image (de Berg, Cheong *et al.* 2006; Litwinowicz and Williams 1994; Buck, Finkelstein *et al.* 2000) and to use an *affine* motion model (Table 3.3), 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 3.46) 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 3.49a, using *variational* (energy minimizing) interpolants such as regularization (Litwinowicz and Williams 1994), see Section 4.2, or using locally weighted (*radial basis function*) combinations of displacements (Section 4.1.1). (See Section 4.1 for additional *scattered data interpolation* techniques.) If quadrilateral 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 2002; Lee, Wolberg *et al.* 1996).

In some cases, e.g., if a dense depth map has been estimated for an image (Shade, Gortler *et al.* 1998), we only know the forward displacement for each pixel. As mentioned before, drawing

¹⁷See Section 6.2.4 on active appearance models for more sophisticated examples of changing facial expression and appearance.

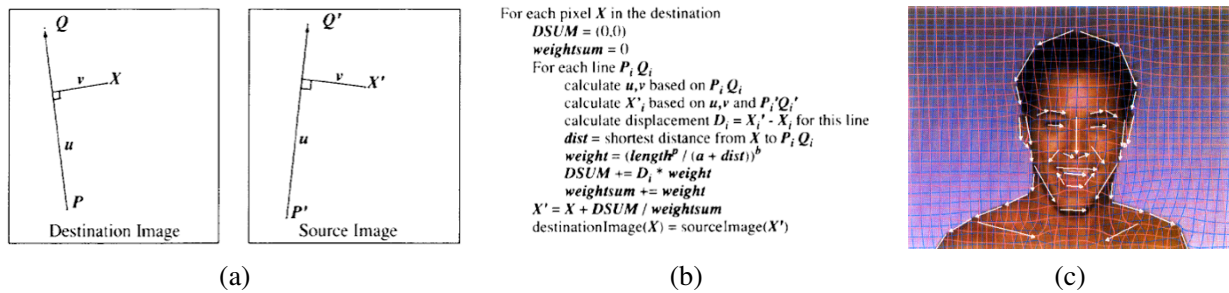


Figure 3.50 Line-based image warping (Beier and Neely 1992) © 1992 ACM: (a) distance computation and position transfer; (b) rendering algorithm; (c) two intermediate warps used for morphing.

source pixels at their destination location, i.e., forward warping (Figure 3.45), 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 *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 3.49c and 3.50. 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 3.3), for pixels in its vicinity, as shown in Figure 3.50a. 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 3.50a if $u \in [0, 1]$, else the shorter of the two distances to P and Q).

One final possibility for specifying displacement fields is to use a mesh specifically *adapted* to the underlying image content, as shown in Figure 3.49d. Specifying such meshes by hand can involve a fair amount of work; Gomes, Darsa *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 *et al.* 1996; Gomes, Darsa *et al.* 1999).

Figure 3.51 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 3.49), 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 the time parameter that describes the sequence of interpolated frames. The weighting functions for the two

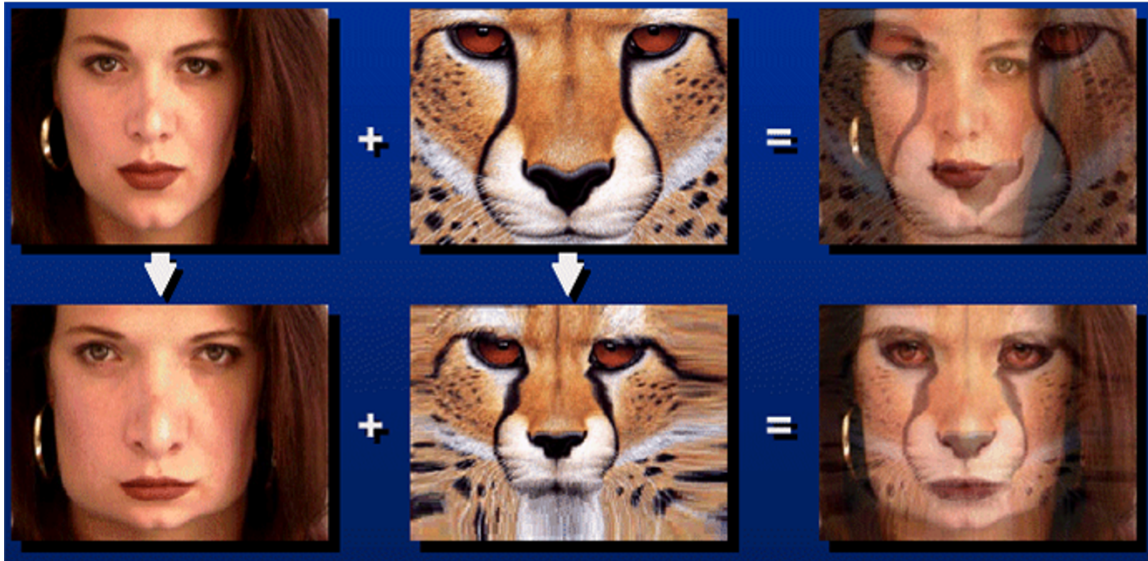


Figure 3.51 Image morphing (Gomes, Darsa *et al.* 1999) © 1999 Morgan Kaufmann. Top row: if the two images are just blended, visible ghosting results. Bottom row: both images are first warped to the same intermediate location (e.g., halfway towards the other image) and the resulting warped images are then blended resulting in a seamless morph.

warped images in the blend are $(1 - t)$ and t and the movements of the pixels specified by the correspondences are also linearly interpolated. 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 *et al.* 1993). Exercise 3.25 has you implement a morphing algorithm and test it out under such challenging conditions.

3.7 Additional reading

If you are interested in exploring the topic of image processing in more depth, some popular textbooks have been written by Gomes and Velho (1997), Jähne (1997), Pratt (2007), Burger and Burge (2009), and Gonzalez and Woods (2017). The pre-eminent conference and journal in this field are the IEEE International Conference on Image Processing 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 (2009) provide a good 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; Chen, Paris, and Durand 2007; Paris and Durand 2009; Paris, Kornprobst *et al.* 2008), related iterative algorithms (Saint-Marc, Chen, and Medioni 1991; Nielsen, Florack, and Deriche 1997; Black, Sapiro *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 1998; Weickert 1998; Barash 2002; Schar, 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), and guided filtering (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004; He, Sun, and Tang 2013).

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), and 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; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as well as for multi-scale oriented filtering (Simoncelli, Freeman *et al.* 1992) and denoising (Portilla, Strela *et al.* 2003).

While image pyramids (Section 3.5.3) are usually constructed using linear filtering operators, more recent work uses 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 *et al.* (2008), and Fattal (2009).

High-quality algorithms for image warping and resampling are covered both in the image processing literature (Wolberg 1990; Dodgson 1992; Gomes, Darsa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010) and in computer graphics (Williams 1983; Heckbert 1986; Barkans 1997; Weinhaus and Devarajan 1997; Akenine-Möller and Haines 2002), where they go under the name of *texture mapping*. Combinations 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 *et al.* 1999).

3.8 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. 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 2.8 for some related ideas.

Ex 3.2: Demosaicing. If you have access to the RAW image for the camera, perform the demosaicing yourself (Section 10.3.1). If not, just subsample an RGB image in a Bayer mosaic pattern.

Instead of just bilinear interpolation, try one of the more advanced techniques described in Section 10.3.1. Compare your result to the one produced by the camera. 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.

Ex 3.3: 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 3.4: 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 3.5: Difference keying. Implement a difference keying algorithm (see Section 3.1.3) (Toyama, Krumm *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 3.3.3), compute their centroids, and track them from frame to frame. Use this to build a “people counter”.

Ex 3.6: 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 3.7: 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 (2.117).
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 3.8: 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 3.7 (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 3.9: 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),

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

(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 3.10: 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 3.11: 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 1? 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 prefilter 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 3.12: 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 3.4.2.

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 3.13: 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 7.4).

Ex 3.14: Bilateral and guided image filters. Implement or download code for bilateral and/or guided image filtering and use this to implement some image enhancement or processing application, such as those described in Section 3.3.2

Ex 3.15: Fourier transform. Prove the properties of the Fourier transform listed in Szeliski (2010, Table 3.1) and derive the formulas for the Fourier transforms pairs listed in Szeliski (2010, Table 3.2) and Table 3.1. 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: High-quality image resampling. Implement several of the low-pass filters presented in Section 3.5.2 and also the windowed sinc shown in Figure 3.28. 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 3.52a) and natural images with lots of high-frequency detail (Figure 3.52b–c).



Figure 3.52 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.

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 the tradeoff between speed and quality.

Ex 3.17: 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:

- 2×2 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 3.18: 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 3.41).
4. Reconstruct the final image from the blended Laplacian pyramid.

Generalize your algorithm to input n images and a label image with values $1 \dots n$ (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 3.19: Pyramid blending in PyTorch. Re-write your pyramid blending exercise in PyTorch.

1. PyTorch has support for all of the primitives you need, i.e., fixed size convolutions (make sure they filter each channel separately), downsampling, upsampling, and addition, subtraction, and multiplication (although the latter is rarely used).
2. The goal of this exercise is *not* to train the convolution weights, but just to become familiar with the DNN primitives available in PyTorch.
3. Compare your results to the ones using a standard Python or C++ computer vision library. They should be identical.
4. Discuss whether you like this API better or worse for these kinds of fixed pipeline imaging tasks.

Ex 3.20: Local Laplacian—challenging. Implement the local Laplacian contrast manipulation technique (Paris, Hasinoff, and Kautz 2011; Aubry, Paris *et al.* 2014) and use this to implement edge-preserving filtering and tone manipulation.

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 3.17). 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, $1/4$ 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 that forward warps the displacement field, fills in small holes, and then uses inverse warping (Shade, Gortler *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).



Figure 3.53 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)?

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 3.53, and enhance the strength (saturation) of the rainbow.

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